FORM PROCEDURES

by

J. Hogg
O. M. Nierstrasz
D. Tsichritzis

Computer Systems Research Group
University of Toronto
121 St. Joseph St.
Toronto, Ontario
Canada
M5S 1A4

Abstract

This paper outlines an effort to introduce automation into an office forms system (OFS). OFS allows its users to perform a set of operations on electronic forms. Actions are triggered automatically when forms or combinations of forms arrive at particular nodes in the network of stations. The actions deal with operations on forms. This paper discusses the facilities provided for the specification of form-oriented automatic procedures and sketches their implementation.

Form Procedures

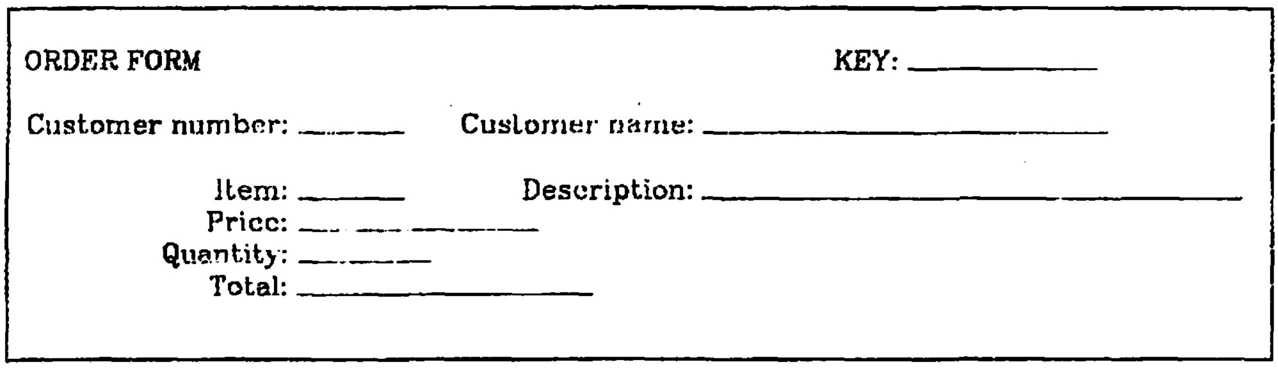
1. Introduction

OFS is an electronic forms management system [Tsichritzis 1980, 1981, Cheung 1979, 1980, Gibbs 1979, 1980]. It provides an interface to MRS, a small relational database system [Hudyma 1978, Kornatowski 1979, Ladd 1979]. OFS and MRS were written in C within the UNIX operating system [Kernighan 1978, Ritchie and Thomson 1978]. They have both been distributed widely to organizations.

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

A distinction is made in OFS 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 filled form represented as a tuple in the database of forms. Its fields may have values assigned to it, and it always has a unique key assigned at creation time by the system. A *form type* is the specification of a form blank and a set of field types (see below). A *form file* is a 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**. Note that some fields of the form instance need not have values associated with them. The key field must have a value which is automatically assigned by the system.

Form fields may be of six different types. Manual fields of type 1 may be inserted or modified at any time, type 2 may be inserted at any time but not modified, and type

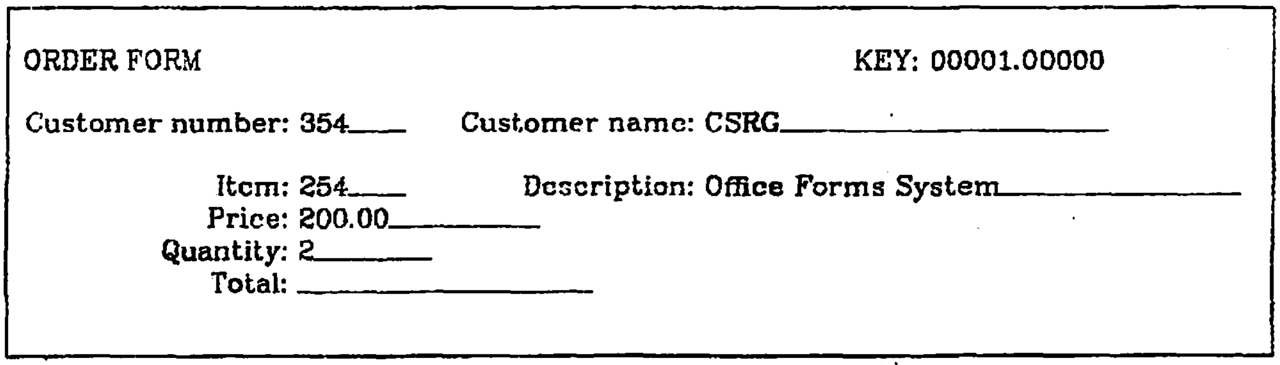


ORDER FORM KEY:

Customer number: Customer name:

Item: Description:
Price:
Quantity:
Total:

An order form blank



ORDER FORM KEY: 00001.00000

Customer number: 354 Customer name: CSRG

Item: 254 Description: Office Forms System
Price: 200.00
Quantity: 2
Total:

An order form instance

Figure 1 *Form blanks and instances*

3 must be inserted at form creation and never modified. Automatic fields of type 1 are key fields, always the first field of a form, type 2 date fields, and type 3 signature fields bearing the station's name if the preceding field is filled in.

Form operations are creation, selection, and modification. Forms may also be attached to *dossiers*. Dossiers are 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 "wastebasket station" which conceptually shreds the electronic form. The wastebasket station may in fact archive rather than erase a form depending upon the needs of a particular application. Form instances are unique, and must always exist at exactly one location in the system. They are either in some form file or waiting in a mail tray. Forms may be mailed from one station to another. 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.

Form files may be accessed as a whole using an MRS interface. However, in this case no protection is provided against illegal operations such as destroying a form or creating a form with a key that is already in use. Therefore, the MRS interface is not meant to be used except by privileged users.

OFS is basically a passive system, i.e., the user has to initiate every action. The only automatic form processing that OFS will do occurs if a form is mailed to a special automatic station. Such a station periodically reads its mail and submits the forms as input to an application program. These programs must be written so as to preserve compatibility with OFS. Consequently, the specification of an OFS automatic procedure requires a great deal of knowledge of the inner workings of OFS. The TLA project was conceived as a tool to introduce automatic form processing into OFS [Hogg 1981, Nierstrasz 1981].

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.

The user interface is presented in terms of objects with which the OFS user is already familiar. Specifying operations within a procedure corresponds closely to performing those operations within a manual system. A user who is editing an automatic forms procedure manipulates "sketches" of forms. Sketches are 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 have a simple syntax.

TLA does not assume any knowledge of the system state 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 a user's own workstation does not concern him. The domain of automation is that of the individual workstation. 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. He can then process the forms and file them or send them on their way. 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 a local library of application programs.

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

There are two types of automatic fields. The first type is filled in only if all its argument fields have values. The other type accepts null values, and is filled in even if some argument fileds are missing. Fields are initially filled in sequence. When an automatic field is reached, an application program written in a conventional programming language (usually C or the UNIX Shell) is executed. The output from this program is assigned to that field. If any argument fields are subsequently modified, the automatic fields which use it are also updated. Typical applications are arithmetic operations such as sales tax calculations, or database queries such as filling in a customer's address.

"Smarter forms" with fields that change value depending upon time conditions, the state of the system, or any other variable, were not implemented. 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". For example, it may never be triggered because missing forms do not arrive. Postconditions may be interpreted in terms of the preconditions of another automatic procedure to which control of the forms is passed.

Automatic procedures run concurrently with the manual functions of the users. Conflicts can arise over the form manipulations. Forms being collected by an automatic procedure could be modified or shipped away manually. They can even be "stolen" by another competing automatic procedure. This implies that when a complete set of forms is gathered for some procedure, it has to be temporarily "removed" from the system. This operation safeguards the forms until they are processed.

2. Interface

The specification of an automatic procedure in TLA bears some resemblence to SBA and OBE [De Jong 1980, Zloof 1980]. The precondition segment of a procedure bears a resemblance to a QBE query with forms instead of tables as the data objects. In the simplest form of a TLA precondition, putting a value in a field of a precondition indicates that a form is to be found with a field matching that value. The action segment of the procedure is similar. The simplest operation is to assign to a field the value specified in an action.

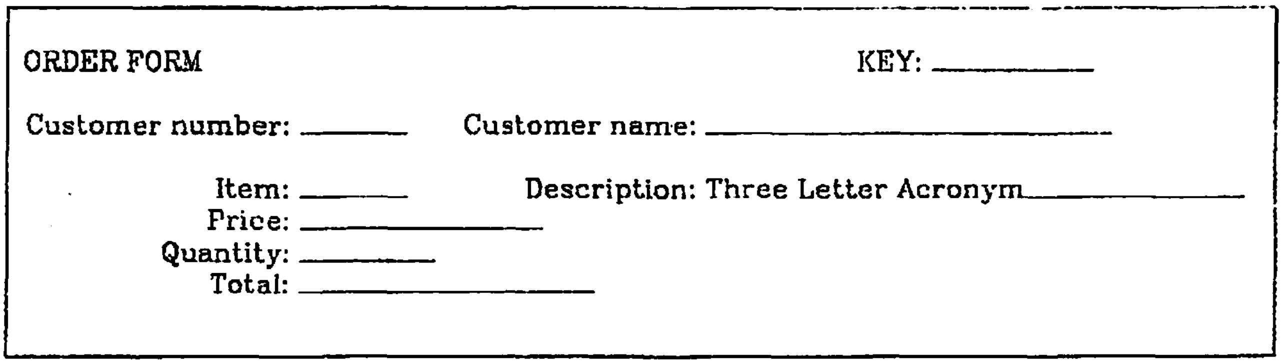
The order in which forms needed by a procedure arrive is not important. The order in which actions are performed is not specified in detail. TLA merely ensures that the procedure be logically consistent. The specification is non-procedural. The user indicates what forms are to be collected, and what is to be done with them. He does not specify how they are to be collected or how the actions are to be performed.

Preconditions in TLA describe what, when and where. For each procedure there is a *working set* of forms. The working set may include forms that come only from certain workstations, forms local to the station specifying the procedure, or forms that have just been processed by another automatic procedure. One may also specify a procedure to run only at certain times or ranges of times.

A TLA procedure is a collection of "sketches". A sketch resembles a form, but is to be distinguished from form blanks, form types or form instances. A *precondition sketch* indicates a request to the system to find "a form that looks like this". An *action sketch* indicates a request to modify a form that has already been obtained. In either case a sketch describes a form instance before or after processing by the procedure. The medium of specification of a sketch is 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 a form are specified by *pseudo-sketches* of "pseudo-forms". For example, the condition that a procedure process only forms coming from user "john" must be indicated on a special *source pseudo-sketch*.

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 a single form in the working set. TLA tries to determine whether a given form satisfies the local restrictions (including the source condition) for some sketch in some automatic procedure. If it does, TLA 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: ____________

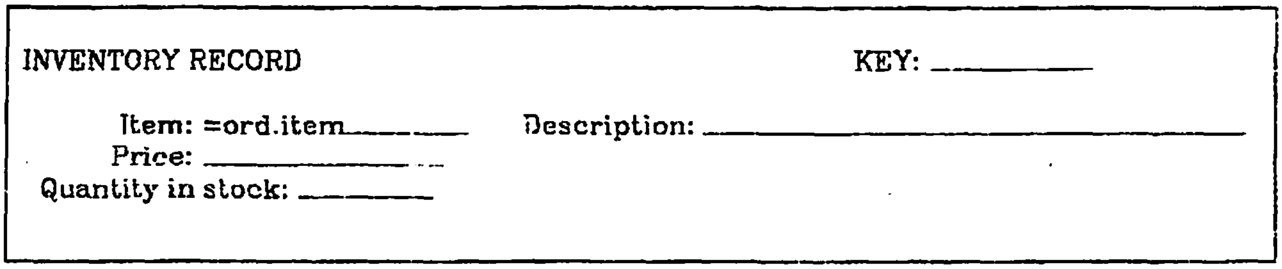
Customer number: ________ Customer name: ______________________________

Item: ________ Description: Three Letter Acronym________________
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. One expects all the forms in a procedure's working set to be linked by certain common field values. Matching field values are therefore probably adequate to model many applications of automatic procedures. However, simple inequality restrictions may also be specified.

Figure 3 shows how a link is made to find an **inv** form for the item requested on an **order** form. Each sketch in a procedure has a name assigned by the user. This name is prepended to the field name. In this way a field of a different sketch can be referenced within a sketch. Note that one could equivalently have placed the restriction "=inv.item" in the item number field of the **order** precondition sketch.

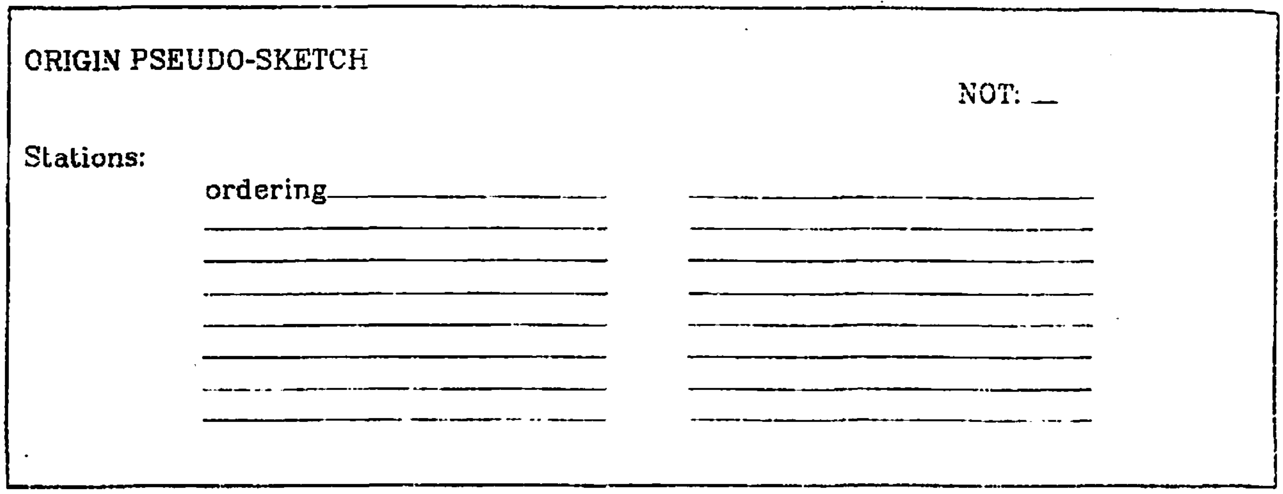

INVENTORY RECORD KEY: ________

Item: =ord.item________ Description: ______________________
Price: ______________
Quantity in stock: ________

Figure 3 *A global (join) precondition*

We can also restrict the source of mail being processed by an automatic procedure. Suppose, for example, that the accounting department receives an order form from the ordering department. This may be interpreted as a request to forward a customer's address to the warehouse so that the order may be filled. If, however, the order form arrives from the warehouse, that may indicate that the order has gone through, and that an invoice should be mailed out. Figure 4 shows an origin pseudo-form sketch for such an application. Forms may thus be processed differently depending upon their point of origin. Alternatively, the special field **not** may be filled in to indicate that only forms coming from stations not listed in the pseudo-sketch should be processed by the procedure. The pseudo-station *me* is also available to indicate that forms must (or must not) come from within the station's own files.

All form modification actions are indicated on action sketches. Every form manipulated by a forms procedure has a precondition sketch and an action sketch. Actions



ORIGIN PSEUDO-SKETCH

NOT: _

Stations:

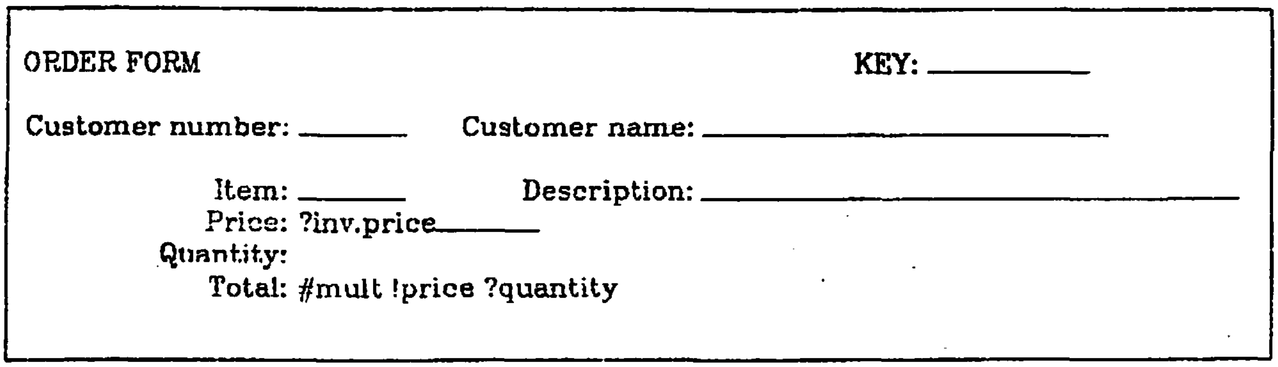
ordering_______________

Figure 4 *An origin pseudo-sketch*

which do not concern themselves with field values must be expressed via pseudo-forms.

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, or possibly function calls to application programs. We distinguish, therefore, 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. In fact, the original value of a field will often be one of the arguments to a function call update 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 **inv** form. A program called "mult" is called to calculate the total. Finally, the original value of **quantity** is accessed whereas the updated value of **price** is used. Note that the symbols "#", "?" and "!" are used to respectively access functions, original and updated field values. If none of these symbols are used, a constant string value is inserted.


ORDER FORM KEY: ________

Customer number: ______ Customer name: ____________________

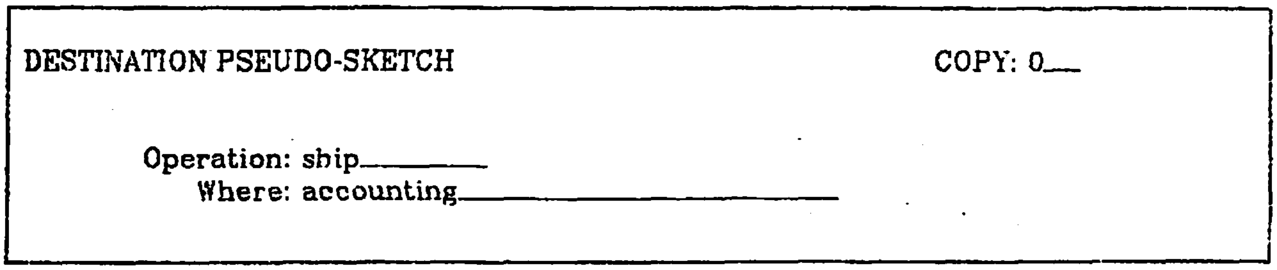
Item: ______ Description: ____________________________
Price: ?inv.price______
Quantity:
Total: #mult !price ?quantity

Figure 5 *An action sketch*

Some analysis is needed 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. Suppose that the price field of the order form were updated to "!inv.price" *and* the price field of the inventory form were updated to "!order.price". No order of execution could make sense of the request.

Field constraints must be obeyed. Procedures that create forms must fill in certain fields. Procedures that modify forms must only modify fields with an appropriate type. Implied actions must also be evaluated if a procedure modifies or inserts a field which is an argument to an automatic field.

After all form modifications are completed, zero or more copies of each form are made. Each form or copy may then be left in the user's files, inserted into a dossier or shipped to another station. The mechanism used to specify these operations is the *destination* pseudo-sketch; an example is shown as figure 6. Copy 0 is the form manipulated by a procedure, and one additional destination pseudo-sketch is filled in for each copy of that form. The operations available are *leave*, *ship* and *dossier*. The first of these requires no where argument, but the others require the name of a station or a dossier respectively. This may be given as a simple constant or a field or function value, just as in action sketches.



DESTINATION PSEUDO-SKETCH COPY: 0__

Operation: ship________

Where: accounting________________________

Figure 6 *Destination pseudo-sketch*

A weak sort of postcondition is available by employing a function call to decide the operation, dossier name or shipping destination. General postconditions can only be acheived by cooperating form procedures which accept different cases of the working set of forms. Suppose, for example, that the processing of an order causes the quantity of an item in stock to dip below a certain acceptable level. We may wish, at this point, to send a memo to the manager initiating an increase in the production of the item. The procedure which processes orders 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 sends the memo.

With this approach individual tasks are clearly identified. Automatic procedures are simple and completely devoid of any control flow. Furthermore, the implementation is simpler because postconditions correspond to separate procedures. The low inventory checker, for example, is only invoked when an inventory form is updated.

3. Implementation

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. The sketch representation is very convenient for the user. This format, however, is wholly unsuitable for implementation. The specification must be analysed and translated for greater run-time efficiency.

We cannot predict when the forms required to trigger a forms procedure may arrive. The processing must, therefore, 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 (so that one form is not used to match two sketches within a single working dossier), and actions. The execution of a forms procedure makes use of these four specifications at different stages. It is convenient to process these specifications at procedure definition time, and translate them into formats that require no further run-time analysis.

Suppose that 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. The local conditions are comprised of the source restriction and the field constraints. If a form does not match the local constraints of any precondition sketch, then TLA assumes that no procedure is prepared to handle it. Suppose that a form does match the local constraints of one or more precondition sketches. That form is then a candidate for a working set for some procedure(s). It is immaterial whether or not a working set including that form is complete. There is always the possibility that at some time the missing forms of the working set could arrive.

The form instance in figure 7 matches the local condition of the precondition sketch, *i.e.* **quantity**>0. There may not necessarily be a global match if there is no order form with the same item number. Even if there is an order form with the same item number, it may not satisfy the other constraints of its precondition sketch. Nevertheless, TLA notes that a local match has been made and waits for the rest of the working set to arrive.

INVENTORY RECORD KEY:

Item: =ord.item______ Description: ____________________
Price: __________
Quantity in stock: >0______

Precondition sketch

INVENTORY RECORD KEY: 00001.00000

Item: 465______ Description: Workstation______________
Price: 16000.00______
Quantity in stock: 12______

Form instance matching local preconditions

Figure 7 *Local matching*

TLA checks the local constraints of a form, records its findings, usually determines that the form does not complete a working set, and then waits for more forms to arrive. Further processing may not occur for some time. All local constraints for forms of the same type are extracted from all procedures and stored in a common file. This file is opened to check the local constraints of a given form for all procedures.

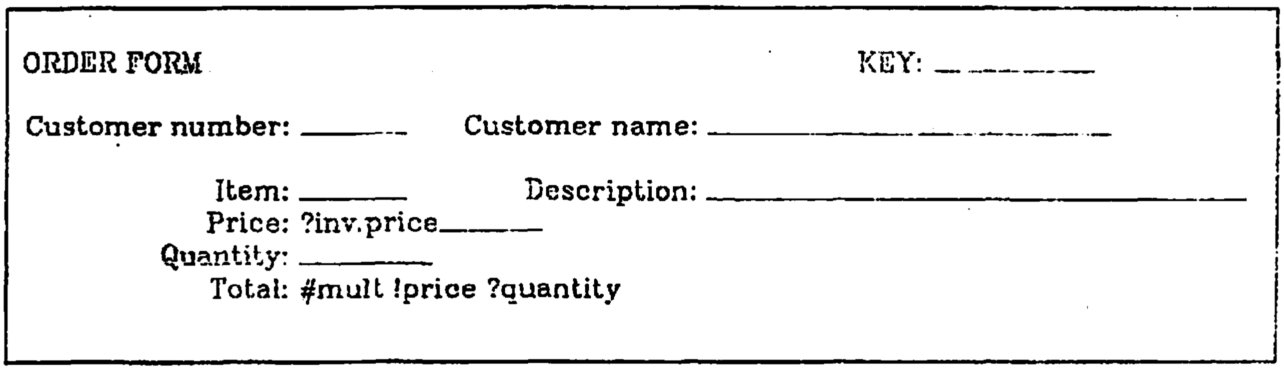
After the local constraints have been matched for a form, TLA checks link conditions between the corresponding sketches of the procedure. The link conditions are stored in files by procedure. Suppose that, in the previous example, TLA found an order for item 0002. It would note that the link between 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.

Even if forms arrive together, the processing of the forms is sequential. TLA treats 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. One can expect a considerable delay between the establishment of local constraints and the evaluation of links between forms.

Actions are performed only once a working set of forms has been compiled. Actions are 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 performed. Actions run to completion.

The example in figure 8 *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.

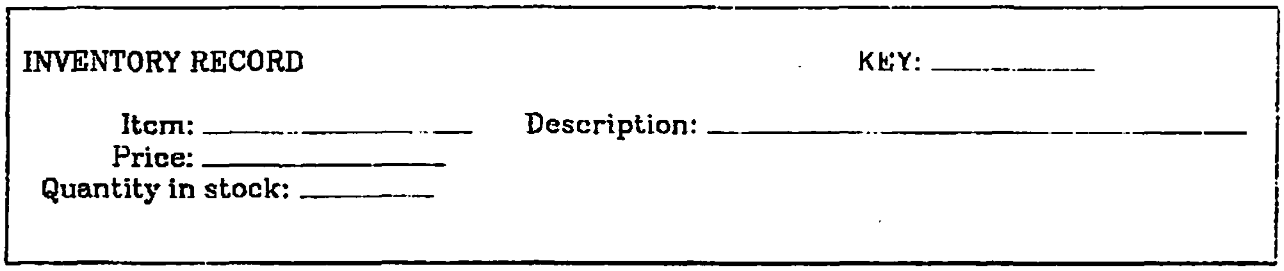
An admittedly unlikely case is captured in figure 9 which is triggered if TLA detects two inventory forms for a single item. Since there are two precondition sketches in the procedure, TLA assumes that they refer to two *different* forms in the working set. Otherwise, any inventory form would trivially satisfy both precondition sketches and thus trigger the procedure. When the procedure is written, TLA notes immediately that two precondition sketches describe forms of the same type. It performs a key comparison of those forms in any working set identified to guarantee that they are not one and the same.


ORDER FORM KEY: ______

Customer number: ______ Customer name: ______

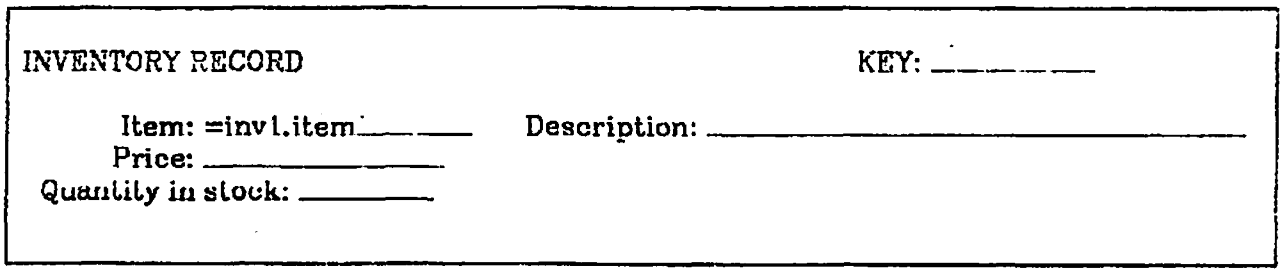
Item: ______ Description: ______
Price: ?inv.price______
Quantity: ______
Total: #mult !price ?quantity

Figure 8 *Ordering of actions*


INVENTORY RECORD KEY: ______

Item: ______ Description: ______
Price: ______
Quantity in stock: ______

Precondition sketch inv1


INVENTORY RECORD KEY: ______

Item: =inv1.item______ Description: ______
Price: ______
Quantity in stock: ______

Precondition sketch inv2

Figure 9 *Duplicate form types in a procedure*

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.

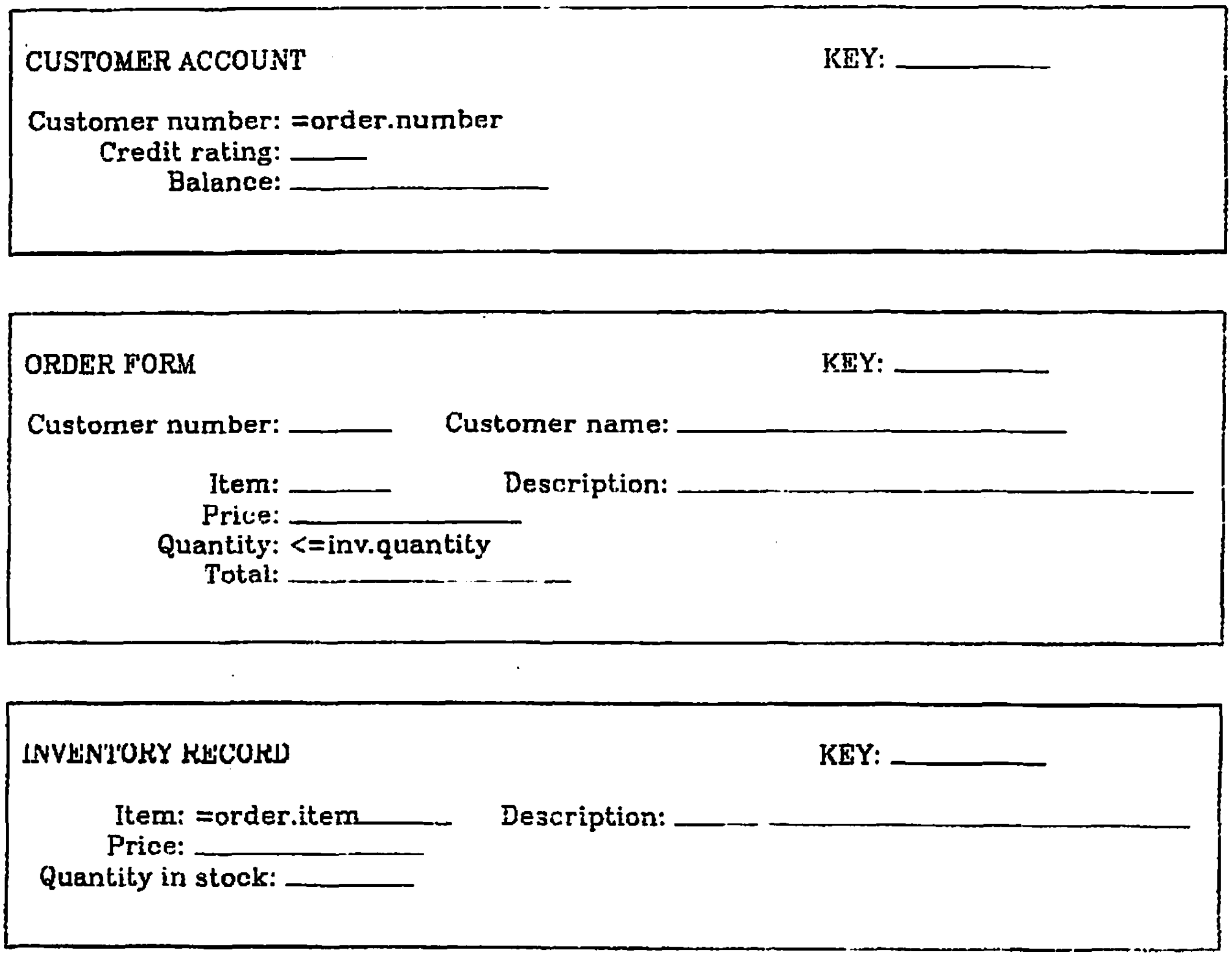
Automatic procedures are meant to run regardless of whether the user to whom the corresponding station belongs ever signs on after the procedure is written. Mail in the system is routed through a host control node. The sending station sends a message to the host consisting of the contents of the form tuple and the name of the station which is to receive the mail. The host then stores the form, updates the receiving station's mail tray and sends a message to the recipient's station. At the recipient's station machine, the interpreting process is started. It 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. It 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. They are not properly forms or copies of forms, but merely *images* of forms.

Mail may arrive while the interpreter is running. It therefore continues to process all mail until it discovers an empty tray in a manner similar to that of the line printer daemon in UNIX. Only one interpreter may run at any time for a given station. In this way we eliminate interference problems between interpreters. A lock is placed on the running of the interpreter for a given station.

4. Sketch and Instance Graphs

The working set of a form procedure is abstracted in terms of a *sketch graph* with the sketches as coloured vertices, and the matching conditions as edges in the graph. The form gathering algorithm must find corresponding forms and satisfy matching conditions of the sketch graph. An *instance graph* is generated associated with the forms retrieved. The interpreter tries to match the sketch graph in the instance graph.

Consider the precondition sketches in figure 10. A link between the account and order forms is established across the customer number. A link between the order and inventory forms is captured by *two* global conditions, one by item number and the other by quantity.



CUSTOMER ACCOUNT KEY: ____________

Customer number: =order.number
Credit rating: ______
Balance: ____________________

ORDER FORM KEY: ____________

Customer number: ________ Customer name: ______________________________
Item: ________ Description: ____
Price: __________________
Quantity: <=inv.quantity
Total: ______________________

INVENTORY RECORD KEY: ____________

Item: =order.item______ Description: _____ _________________________________
Price: __________________
Quantity in stock: __________

Figure 10 *Precondition sketches of a procedure*

The corresponding sketch graph is shown in figure 11. Each sketch is represented by a labelled/coloured node. 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 of the form type, the form key, the procedure and the sketch matched into a 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 relation called EDGE. EDGE records the form keys, types, sketch names and procedure name of every link established.

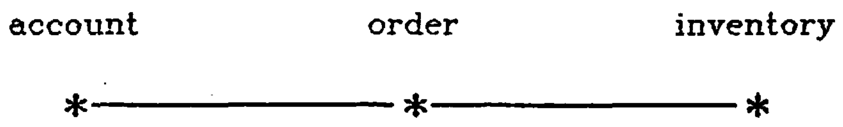


Figure 11 *A sketch graph for a single procedure*

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. 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 12 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. When a working set is found, it is processed and it 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.

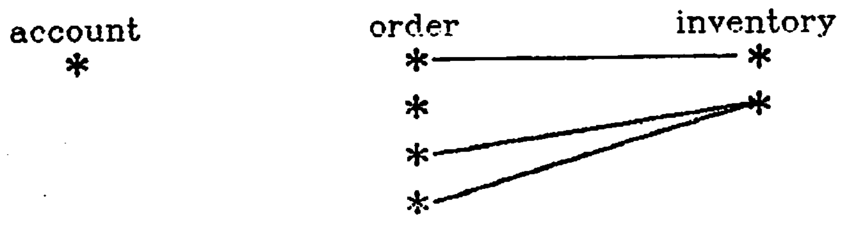


Figure 12 *The instance graph for a procedure*

The relationships between the forms in the working set of a form procedure are usually best expressed in terms of the join conditions. 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 usually be disconnected.

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 precondition sketch may each satisfy a join condition with a third form satisfying a second sketch in the same procedure. This anomaly will occur if the imposed join conditions are "not nice enough". In this case, the connected subgraphs of the instance graph are not as simply related to the sketch graph. Thus, establishing when a complete working set of forms has been compiled requires careful analysis.

When TLA has finished processing a 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. 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.

Join conditions giving rise to sketch *trees* seem natural, 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. In practice, however, things may not be that simple. Join conditions might give rise to cycles, or even disconnected sketch graphs. Suppose that the warehouse, for example, has a single **value** form at its workstation keeping track of the total dollar value of its stock. The 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 13 is therefore disconnected.

5. Graph-chasing

The algorithm which searches the instance graph for a copy of the sketch graph employs a list of *potential working sets*. Initially there exists a single such set contain-

account
order
inventory
value

Figure 13 *A disconnected sketch graph*

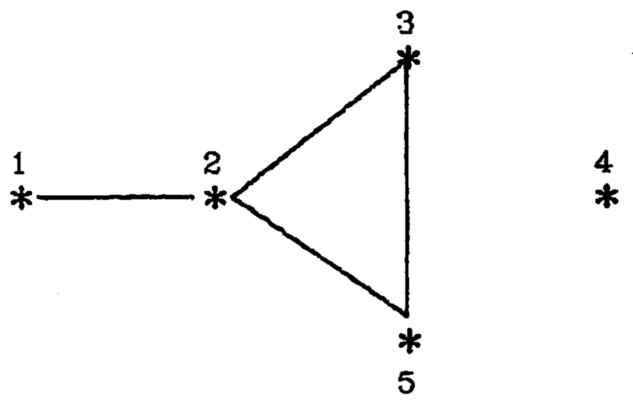
ing only the key of the newly added form. Edges are traversed in the instance graph and keys are added to each set until all the edges and nodes in the sketch graph have been checked.

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 cannot lead from old nodes until an edge is traversed which checks off two new nodes.

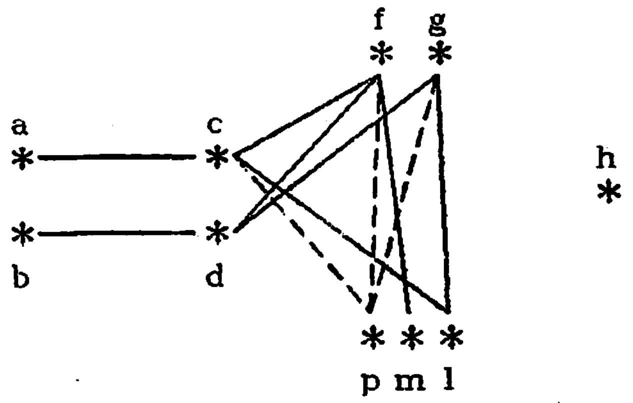
The sketch and instance graphs in figure 14 will be used to illustrate the graph-chasing algorithm. The example contains both cycles and disjoint subgraphs.

Sketches 3 and 5 are sketches for the same form type but represent distinct forms in the procedure. The terms {a, b, c, ...p} 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.

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



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



Instance graph (p is the most recently added node)

Figure 14 *Sample sketch and instance graphs*

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 set in our list. For each success we add a new key to some set, and for each failure, we delete a set. Suppose that several edges may be traversed in the instance graph for a given edge of the sketch graph. We then split the current set and add a new node for each copy. The closing of a cycle in the sketch graph corresponds conceptually to a select on the set list. In this way we ensure that links actually exist in the instance graph for the two relevent forms represented in each set.

Figure 15 describes the steps followed in locating the working set in our example. If at any point all working sets are lost, the algorithm halts with no working set of forms identified.

potential working sets					
1	2	3	4	5	
				p	p is a new form matching sketch 5.
		f g		p p	From node 5 in the sketch graph we can reach node 3 along edge (3,5). The edges ((3,f),(5,p)) and ((3,g),(5,p)) in the instance graph are followed and the potential working set is "split".
	c d d	f f g		p p p	The edge (2,3) is now followed, splitting the first set of the previous step.
a b b	c d d	f f g		p p p	Follow edge (1,2).
a	c	f		p	Edge (2,5) completes a cycle. Perform a select on the sets resulting from the last step. Since ((2,d),(5,p)) is not in the instance graph, two potential working sets are lost.
a	c	f	h	p	All the edges in the sketch graph have been traversed. A form that matches sketch 4 must be added.
a	c	f	h	p	Check that form f differs from form p.

Figure 15 *Finding a working set of forms*

The sketch and instance graphs are described as follows: The sketch graph is G'(N',E') where N' = {1, ... n} is the set of colours and E' is a subset of N' × 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' × F and E is a subset of N × 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},

N = {(1,a), (1,b), ...(5,p)}, and

E = {((1,a),(2,c)), ((1,b),(2,d)), ...((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.

Suppose that 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 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,p)}.

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 checklists, *node* and *edge*, with slots for each element of N' and E' respectively. These record whether or not the edges and nodes have been inspected. All are initially set to false, and a set list, D, is set initially to empty. Each set has n slots to hold all the keys of any working set of forms found by the algorithm:

```
Let x in N represent the newly added form.
Add a set 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
      for each set in D do
         for each (y,z) in E where y' = i and z' = j do
            copy the set
            set slot i to y", slot j to z"
         delete the original set
   else if exactly one of node[i] and node[j] is false then
      /* without loss of generality, node[i] */
      for each set in D do
         for each (y,z) in E where y' = i and z' = j and
            y" is already in slot i of the set do
            copy the set
            set slot j to z"
         delete the original set
   else if node[i] and node[j] are true then
      for each set in D where (y,z) is not in E and
         y" = i, z" = j do
         delete the set
   set edge[e'] to true
   set node[i] to true
   set node[j] to true
Check that forms of the same type are different.
```

If D is empty when the algorithm is finished, then no working sets were found. If D is not empty, then the "first" set containing no duplicate keys is chosen as the working set.

The station's owner may attempt to move some of the forms in the working set while the interpreter is running. Each of the forms must therefore be set aside. 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 retained thus far, and aborts the forms procedure.

If all the forms are successfully obtained, then the interpreter performs 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.

6. Concluding remarks

TLA captures, in some sense, what is meant by an "automatic forms procedure". The context of OFS limits the range of possible actions upon forms. There are also many things that persons can do with OFS which have not been modelled in TLA. Automatic procedures, for example, are not smart enough to expect the timely return of a form which has been shipped away.

Form flow is determined by the particular configuration of procedures across the system. Analytic tools are needed for determining some notion of "correctness" [Tsichritzis 1981]. It is the responsibility of the users and a form administrator to model and analyse that there are no undesirable side effects resulting from some particular combination of automatic procedures. Such analysis should be performed within a reasonable complexity bound and it should be performed mechanically if at all possible.

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. The complexity of identifying a sketch graph within the graph grows if the sketch graph is not merely a subgraph of the instance graph. Obviously, whatever factors contribute to this complexity must be considered in any "good office design". However, exactly what constitutes "good design", and to what extent it is feasible, is not easily established.

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 present forms may also be part of another working set. The missing forms would determine which procedure is to be activated. There is no way of telling which procedure forms are missing until they arrive. Missing forms may never arrive. There is no way of interpreting their absence as an error, except by placing some arbitrary time limit upon form-gathering.

Forms may satisfy partly completed working sets for a number of procedures. There is a need for some convenient way of displaying these sets. Users could interpret what is "missing" and possibly act on this information. Instance graphs could be quite complicated. Several partly completed sets may overlap in a single instance graph. A graphic display would present this information in a much better fashion than lists of form keys.

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. It then 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 between stations in TLA is determined by the interplay of automatic procedures. Flow of execution could be made more explicit by passing control between procedures in different stations. One could then pass working sets of forms between procedures. In this way we could explicitly determine the order of operations. Procedures could then be called from other procedures without the need for form-gathering. Decision points could be modelled by branching rather than by a variety of similar working sets of forms. Which procedure is to be called could be decided by evaluating a function whose arguments are field values from the working set.

Many office automation systems have been strongly influenced by the SBA [deJong 1980] and OBE [Zloof 1980] systems and Officetalk [Ellis & Nutt 1980]. The most noticable exception are SCOOP [Zisman 1979] and BDL [Hammer et al. 1977], which are, however, more office systems programming languages than office worker's languages. TLA follows this trend. It uses forms that are manipulated at workstations, like Officetalk, and the non-procedural interface for defining procedures was in large part inspired by the work of deJong and Zloof. However, TLA takes a somewhat different approach from either.

A major goal of the TLA project was to provide a facility for automating office procedures that could be used by office workers, as opposed to computer professionals, with a minimum of training. As a result, there was an emphasis on providing familiar concepts and a highly uniform interface.

The form is a very familiar concept to all office workers. Therefore, the idea of a sketch is an easy one to teach. By contrast, the SBA notion of boxes is both useful and powerful. However, it has no analog in the office of today, and therefore requires a more expert office worker to use.

In QBE, conditions appear in a separate box from the tables of an application. By contrast, TLA "conditions" (constraints) appear within a form itself. This difference is not quite as minor as it seems; it reflects an underlying philosophy in the TLA project that the user interface should be as uniform as possible. There are no separate condition boxes attached to forms within the underlying manual system, and therefore there are no separate conditions attached to sketches. Information that absolutely cannot be obtained from the form fields (such as the source of the form) is specified using pseudo-sketches that resemble forms as closely as possible.

Another difference between TLA and the IBM systems is that TLA, like its ancestors OFS and MRS, runs on very small computers. Most of the development was done on an LSI-11/23; the remainder was done on a "big machine", a PDP-11/45. This means that the hardware required for TLA is affordable by any office large enough to benefit from automation. At the same time, incremental growth can be easily achieved by adding additional machines of a wide range of sizes to a local net.

Both OFS and TLA have been implemented on PDP-11's and LSI 11/23's running under UNIX. 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..

7. References

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

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

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

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.

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

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

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

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.

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

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

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

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.

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

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

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

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

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

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

Nierstrasz, O.M., "Automatic Coordination and Processing of Electronic Forms in TLA", M.Sc. thesis, Department of Computer Science, University of Toronto, 1981.

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

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.

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

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

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.