Regular Layouts of Butterfly Networks in Three Dimensions

J. Keller

Computer Science/Department of Algorithmics and Architecture

CS-R9352 1993

A User's Guide to the Software Testpilot

M.L. Kersten
CWI, P.O. Box 4079
1009 AB Amsterdam
The Netherlands
{Martin.Kersten@cwi.nl}

F. Kwakkel
CWI, P.O. Box 4079
1009 AB Amsterdam
The Netherland
{Fred.Kwakkel@cwi.nl}

Abstract

The Software Testpilot[1] is a performance assessment tool for software systems, in particular, database management systems. This document is a user's guide to the Testpilot and describes the Testsuite Specification Language concepts in greater detail. Moreover, it provides an introduction to designing a TSL program using the Wisconsin Benchmark as a case study. It concludes with an illustration of a session using the graphical user interface.

1991 CR Categories: Testing and debugging [D.2.5] diagnostics, Database systems [H2.4] performance assessment, Installation management [K.6.2] benchmarks.
Keywords and Phrases: Software testing, quality assurance, benchmarking.

1 Introduction

1.1 Software Testpilot

The Software Testpilot [Kersten92] is a tool designed to aid the DBMS engineer and user to explore a large workload search space to find the slope, top, and knees of performance figures quickly. The approach taken is based on specifying the abstract workload search space, a small interface library with the target system, and a functional description of the expected performance behaviours. Thereafter, the Software Testpilot selects DBMS workload parameter values, constructs an experimentation plan, and performs the experiment, such that the performance characteristics and quality weaknesses are determined at minimal cost (i.e. time).

The user describes the performance study of a target system with a test suite. This test suite is written in a language, called *Test suite Specification Language* (TSL), or built incrementally using a form-based interface. This report starts with a detailed description of the TSL concepts and syntax. It is used subsequently in a case study to illustrate how to design of a TSL program. Thereafter, we illustrate the form-based interface for interaction with interaction with a running experiment.

1. The work reported here is funded by ESPRIT-III Pythagoras project (P7091)

1.2 Running example

Before we delve into the details of specifying a test suite and its execution, we shortly introduce the flavour of TSL using part of our running example concerned with the analysis of relational selections, as illustrated in Figure 1. This TSL program describes a workspace spanned by **factor** and **response** variables. They are identified by a name, an underlying type, and value range constraints. The interface with the target system is captured by the **action** statements, which relates a user-defined action with the factor and response variables. The expected performance behaviour is described with a **hypothesis**, a (linear) model over **factor**, **response**, and **parameter** variables.

Identifiers starting with a capital denote variables, others denote a property type. For instance, "tablesize:Size" means the variable "Size" is of type "tablesize". Property type definitions should be described elsewhere in the test suite. The factor definitions describe their range and underlying type. The factor tablesize has, for instance, an integer type and a range between 1 and 1000K tuples.

```
factor => [     name:       tablesize,
                state:      yes,
                type:       integer,
                interest:   midpoint,
                apply:      optimized,
                unit:       tuples,
                scale:      ratio,
                range:      [0 .. 1000K ] ].

factor => [     name:       predicate,
                state:      no,
                type:       symbolic,
                value:      generate_predicate(P),
                apply:      random,
                scale:      nominal ].

parameter => [  name:       init_sscan,
                type:       integer,
                unit:       us ].

parameter => [  name:       cost_rnext,
                type:       integer,
                unit:       us ].

parameter => [  name:       pred_balance,
                type:       integer,
                unit:       us ].

response => [   name:       scantime,
                unit:       us,
                type:       integer ].
```

Figure 1: Partial Software Testpilot test suite.

The Software Testpilot, after initialization of the target system, selects a point in the workload space spanned by the factors for experimentation. The point chosen depends on the hypothesis given and results of previous experiments. Given a point of interest, it constructs an experiment to bring the target system in the required state and executes the action to obtain the response value. This process is repeated until the confidence level has been attained or the system is stopped by the user.

The cost formula for a sequential scan is expressed in the Software Testpilot test suite with a hypothesis (Figure 2). The factors are the *tablesize* and the selection *predicate*. The measured response metric is the time to scan the table *scantime*. The parameters to be solved are *init_sscan*, *cost_rnext* and *pred_balance*.

```
hypothesis => [  name:       scan_hypothesis,
                 factor:     [ tablesize:Size, predicate:Pred ],
                 response:   [ scantime: Time ],
                 parameter:  [ init_sscan: Init, cost_rnext:Next, pred_balance:B],
                 model:      Time = Init + Next * Size + B * eval(Pred) * Size ].
```

Figure 2: The hypothesis for the sequential scan.

The interface with the target is captured in **action** objects, which describe a vector through the workload search space, such as a state change in the target system (e.g. filling a relation) or a measurement experiment (e.g. tuple selection). Actions also deal with connection, disconnection and, in case of target failure, recovery operations of the DBMS. An action contains a slot to identify a Prolog interface primitive, which contains the peculiarities of the interface between Software Testpilot and target system.

A hypothesis has a related action that describes the measurement of its response variable(s). The following action (Figure 3) describes the measurements of the *scantime* response variable.

```
action => [   name:      scan_select_relation,
              before:    [ tablesize:X, predicate:Predicate ],
              response:  [ scantime:Time ],
              body:      sql_query( select * from tenpct where Predicate, Time) ].
```

Figure 3: An Action object for the sequential scan

The output of the Software Testpilot are the measurement results of all experiments against the target DBMS, the values of the derived model parameters, and the confidence level attained for the hypotheses given. The confidence levels are fed back into the Software Testpilot where they are used to decide whether more experiments are required to proof the hypotheses, whether they are accepted, or should be remodelled. The experiment results are visualized to monitor progress.

1.3 Overview

The remainder of this report is organized as follows. Chapter 2 introduces the language concepts to describe a test suite. Chapter 3 contains the case study based on the Wisconsin benchmark. Chapter 4 describes the features of the graphical interface using a session trace. Appendix A gives the procedure how to install the Software Testpilot. Appendix C contains a full-fledged demonstration program based on the Wisconsin Benchmark to illustrate the system.

2 Test suite Specification Language

2.1 Introduction

TSL is a small object-based programming language to describe a test suite. A TSL program contains all components required by the Software Testpilot to cover the workload search space and to interact with the target system. The program is prepared with a text editor or the form-based interface. In this Chapter, we describe the textual interface, which is also used to document the test suite in a compact format. Figure 4 shows the language syntax in EBNF.

```
TSL-program                     ::= { Object }
Object                          ::= Class => [ { Slot } ].
Slot                            ::= Attribute : Content
Content                         ::= Prolog-Term

Prolog-Term                     ::= Variable
Prolog-Term                     ::= Functor [ ( Prolog-Term { , Prolog-Term } ) ]
Prolog-Term                     ::= Prolog-Term Infix-Op Prolog-Term
Prolog-Term                     ::= Prefix-Op Prolog-Term
Prolog-Term                     ::= Prolog-Term Postfix-Op

Infix-Op | Prefix-Op | Postfix-Op                      ::= Functor
Functor | Attribute | Class                            ::= Identifier

Variable                        ::= UpperCaseCh { Character | Digit | Special }
Identifier                      ::= LowerCaseCh { Character | Digit | Special }
Character                       ::= LowerCaseCh | UpperCaseCh

LowerCaseCh                     ::= a|b|..|z
UpperCaseCh                     ::= A|B|..|Z
Digit                           ::= 0|1|..|9
Special                         ::= _
```

Figure 4: EBNF syntax for a TSL program

A TSL program is made up by instances of the following predefined classes: **factor** [many], **response** [many], **parameter** [many], **action** [many], **experiment** [one], **hypothesis** [many] and **control** [one]. The keyword "one" ("many") indicates that one (more than one) instance of this class may exist in any given test suite. The skeleton of object definitions strongly reflects the flavour of the underlying Prolog implementation, as shown in Table 1. The following notational convention is used in the TSL syntax description.

type	description	example
atom	legal Prolog atom	db_focus
term	legal Prolog term	value(date)
body	a legal Prolog clause body	(X=Y, sql_insert(X))
expression	Prolog term that can be evaluated using ’is’	3+5*X
range list	list with elements of the format A..B or A	[0..2,4]
variable list	list with elements of the format *variable:value*	[time:T, size:2]

Table 1: TSL syntax conventions

2.2 Variables

Factor and **response** variables span the workload search space. Factors denote the target system input for an experiment and response variables denote the output measured of the target system. We distinguish two factor kinds: **state** factors represent an property of the target system state. Non-state or auxiliary factors describe information maintained by the Software Testpilot. They represent properties of the abstract search space or to control its working. For example, the number of tuples in a table is a **state** factor, because it describes part of the underlying target DBMS. The number of tuples selected is a non-state factor, because (in general) selection does not change the target state.

A **factor** has a number of properties to construct meaningful experiments. Each factor has a unique **name** to identify the object in the TSL program and a **type** and **range** to describe its permissible values. In some cases the range may be too large to enumerate then a **value** property can be used to generate the permissible values upon system (re)start[1].

Built-in types are currently: **integer**, **float**, **symbolic**, and **time**. Symbolic factors must enumerate or generate all values. A time value is described lexicographically as [Hr: Min:] Sec [.mSec]. Combined with the factor’s range and scale of measurement most of the required ranges can be created.

Built-in **scale** types are currently: **nominal**, **ordinal**, **interval**, and **ratio**. For example, a nominal scale is [male, female], an ordinal scale is [first, second, third], an interval scale is [10^o C, 20^o C], and a ratio scale is [1,2,3]. The scale type is used for visualization purposes.

The **interest** property is a function to describe the weight of interest in values picked from the factor’s value range. It can be used to direct the Software Testpilot to investigate critical boundary conditions rather than a broad scan experiment. The current version supports three predefined interest functions: **uniform**, **extreme**, and **midpoint**[2]. The **uniform** interest function assigns the same weight to all values. The **extreme** function give preference to values at the range boundaries, while **midpoint** prefers values around the midpoints of the ranges.

Further control over the way the Software Testpilot selects an element from the factor’s range is obtained with the **strategy** property, which takes one of the following values: **cyclic**, **optimize**, **random**. The **optimize** strategy uses hypothesis to generate new points in the abstract workload space. The **random** strategy picks points from the factor range in a uniform distributed manner.

The range and strategy properties can be changed by the user during Software Testpilot flight. All other attributes are fixed for the duration of a session.

attribute	contents	optional	default	example
name	*atom*	no	-	selection_size
type	*atom-restricted*	yes	integer	float
state	*atom-restricted*	yes	no	yes
unit	*atom*	yes	-	tuples
range	*list*	yes	-	[0 .. 100]

Table 2: Factor object properties

1. They should be generated upon request only.
2. This should be generalized to point(Low,High,SelectedPoint).

attribute	contents	optional	default	example
value	*body*	yes	-	odd_number(size)
scale	*atom-restricted*	yes	nominal	ratio
interest	*atom*	yes	uniform	extreme
strategy	*atom-restricted*	yes	optimize	cyclic

Table 2: Factor object properties

Response variables describe the value range of properties measured during experiments. They have, just like factors, a globally unique name, type, value range, and scale. They lack the strategy, unit and interest properties, because these can not (easily) be predicted up front.

Parameters are variables used for hypothesis evaluation and have the same properties as the response variables.

2.3 Experiments

To define which factors and response variables must be investigated an experiment object is used. An experiment is an assignment of values to factors and obtaining the response values, which is an instantiation of an experiment object. Besides **factor** and **response** attributes an experiment object has a **check** attribute which is applied after a new experiment is constructed to eliminate illegal candidate experiment[1]. The **design** property describes the design method for the generation of experiments. Permissible design types are: **random**, **incremental**, **factorial**, and **local**. The latter indicates that the strategy mentioned in the factors should be used instead.

The **candidates** properties describes the minimum number of candidates available for the Software Testpilot to generate and select flight plans. The actual number of candidates may only be less when the total workload space is covered and no candidates can be generated.

The **flightplan** property describes the number of candidates for which flight plans are generated. If the number of flight plans is less than the number of candidates to be generated, only the candidates with the highest **weight** attribute will get a flight plan constructed.

An experiment can be repeated to improve precision of response measured as indicated by the **replicate** factor. The replication factor describes the maximum number of replications of the same experiment. Two experiments are the same if all factor values have the same value.

attribute	contents	optional	default	example
name	atom	no	-	focus_one
factor	factor list	no	-	[selection_size]
response	response list	no	-	[selection_time:T]
check	Prolog query	yes	true	(T < 1000)
design	atom-restricted	yes	local	factorial
candidates	integer	yes	10	20
flightplans	integer	yes	10	15
replicate	integer	yes	0	0

Table 3: The experiment object

2.4 Actions

The interface with the target system is described with a collection of actions. In essence, each **action** object describes a small experiment to measure a response given a fixation of the factors. In addition, it describes a possible state transition in the target system and the **state** properties to be measured. Therefore, three types of action behaviour are distinguished.

- Retrieve the target state, which is used to synchronize Software Testpilot with the target system. The **retrieve** attribute describes the factors to be retrieved at the end of this action.
- Change target state. A target state transition requires a **before** and **after** attribute. They introduce variables for the initial- and final- target state, respectively. The final state can be prescribed with a **step** property. The

1. When experiments are repeated then there is also a need for elimination of extremes.

step size is either fixed, i.e. a constant, or derived from the action properties with an expression. Using the **require** property, required experiment factors values, as picked by the testpilot can be assessed and used for checking legal action usages.

- Measure response values. Response variable that have values assessed during the execution of this action are describe with the **response** attribute.

The call to the target system is describe with a Prolog query in the **body** attribute. If, for some reason, the sub-experiment fails then this predicate should fail too. The Prolog predicate **check** is called when constructing flight plans to assure that only legal flight plans are constructed. Inside the Prolog body variables from the *require*, *before* and *after* properties of the same action object can be used.

attribute	contents	optional	default	example
name	atom	no	-	db_insert
require	factor list	yes	[]	[table_size:R]
before	factor list	yes	[]	[table_size:R]
after	state factor list	yes	[]	[table_size:R1]
step	body	yes	true	R1 is R + 100
response	response list	yes	[]	[insertion_time:T]
retrieve	state factor list	yes	[]	[table_size: R2]
body	body	yes	-	sql_insert(R1-R,T)
check	body	yes	true	(R1>R)
cost	expression	yes	0	(R1-R)
simulate	body	yes	-	(T is 0.3*R+(R1-R))

Table 4: Action object properties

The estimated cost of running an action is described by an expression in property **cost**. It is used by the Software Testpilot to order the possible experiments and, this way, to economize on experimentation time.

To speed up test suite development, the user can run the Software Testpilot in simulation mode. The simulated actions are described by the **simulate** property. It contains a Prolog clause that assigns proper values to the response and target objects.

2.5 Hypothesis object

Hypothesis objects describe relationships between factors and response variables. They describe expected response behaviour of parts of the workload space. The **factor** and **response** variables span the hypothesis space. In the current version of the Software Testpilot a hypothesis may only have one response variable.

The relationships between factor and response variables can have **coefficient**s to be determined by he Software Testpilot and **constant** parameters that take the values from other hypothesis objects. If more hypotheses are given then a topological ordering on these hypothesis can be defined. Hypothesis H_2 is evaluated later than hypothesis H_1 when H_1's constant property attribute contains at least one parameter that appears in the parameter attribute of H_2

The **model** attribute describes the hypothesis equation. The equation should express the relationship between factors, coefficients and constants on the right hand side and the response variable on the left hand side.

attribute	contents	optional	default	example
name	atom	no	-	first_guess
factor	factor list	no	-	[table_size: Size]
response	response list	no	-	[selection_time: Time]
coefficient	parameter list	no	-	[cpu_cost: Cpu]
constant	parameter list	yes	[]	[io_cost: Io]
confidence	real	no	0.95	0.90
model	equation	no	-	Time = Size * (Cpu+Io)
discriminate	parameter list	yes	[]	[index:I,cluster:B]

Table 5: Hypothesis object properties

The **discriminate** attribute describes a list of parameters for which all different parameter values result in a separate determination of the coefficient values. The target confidence level is describe with the **confidence** property.

2.6 Controls object

The controls object describes overall session characteristics. The property **speed** describes the balance between quick execution time and steep increase of confidence level. Synchronization with the target system after each action can be enforced with **selftest**. The **startup** attribute contains a name of an action object to start and initialize the target system. The **shutdown** attribute contains an action name to shut-down a target system. The **recovery** procedures deals with malfunctioning target systems. It is called to synchronize the factors in the Software Testpilot with the state characteristics of the target. The **threshold** property gives the minimal weight an experiment requires to be considered as a candidate experiment. Using the **simulate** property the Testpilot can operate in simulation, herewith the *simulate* attribute of the actions is used for execution instead of the *body* attribute.

attribute	contents	optional	default	example
name	atom	no	-	controls
speed	real	yes	0.5	0.95
simulate	boolean	yes	yes	no
selftest	boolean	yes	no	yes
threshold	float	yes	0.0	0.2
simulate	*boolean*	yes	yes	no
startup	atom	yes	-	dbms_connect
shutdown	atom	yes	-	dbms_disconnect
recovery	atom	yes	-	dbms_recover

Table 6: Control object properties

3 Designing your own TSL program

In this section we describe how a TSL program can be constructed. The context is the Wisconsin database for which we would like to develop a new generation benchmark with the Software Testpilot. However, before the enhanced Wisconsin Benchmark (WB) is defined, we model the benchmark description as closely as possible to the description given in [DeWitt]. In particular, we focus on the selection queries Q1-Q8 and Q10. The step by step presentation reflects considerations made during the development process and illustrates how, ideally, a TSL program comes into being.

3.1 STEP 1: Design the contours of the experiment

The first step in any performance experiment is to give a short narrative description of its objectives and boundary conditions. Such a description often gives a direct clue for the factors and responses required in the TSL program. They are highlighted in parenthesis in the description for WB below.

This experiment investigates the relationship between selection speed (*qrytime*) over one (!) relation of various sizes (*relCard*) and various selectivity factors (*selectivity*) on the Wisconsin database. Moreover, we are interested in the performance impact of secondary indexing (*indexing*) and clustering (*clustering*). The experiment deals with a single relation, called tenktup1, and its attributes unique1 and unique2.

This workload search space can be translated into the following TSL experiment object:

```
experiment => [name:      selections,
               factors:   [ relCard:R, indexing:I, clustering:C, selectivity:P ],
               responses: [ qryTime:T ],
               check:     (R*P>=1.0) ].
```

The *check* property limits the experiment to those cases where at least one tuple is retrieved.

Despite the simple transformation from narrative description into a TSL acceptable program fragment, this experiment is not yet properly defined. For, it does not comply with the Wisconsin benchmark requirements to alternately use two relations to reduce the impact of the DBMS buffer manager. Extension with a factor that describes the possible relations involved is insufficient to obtain this behaviour, because it is up to the Software

Testpilot to decide which relation is the focus for the next experiment. Instead, we let all actions work on two relations and return the result of one to the Software Testpilot.

Moreover, we should be careful in specifying the scope of the experiment. The Wisconsin benchmark deals with both single and multiple relation queries (e.g. joins). Thus, it is tempting to introduce factors that describes all relations (AtenK, BtenK,...) and to combine all queries parameterized with a relation name into a single TSL program. However, this leads to possible performance degradation (single relation queries are executed twice). It is better to develop several TSL programs to analyse orthogonal aspects of the WB[1].

3.2 STEP 2: Design of the hypothesis

Each experiment is related to several hypotheses about the system behaviour. Thus, the next step is to write down the hypothesis object and to iterate between experiment and hypothesis design until you think the situation and performance expectation is described adequately.

The hypothesis is a linear formula to obtain a tractable result. A first attempt leads to a fixation of all but a few factors in the experiment, as follows:

```
hypothesis => [name:        selection_hypo,
               factors:     [ relCard:R, selectivity:S ],
               response:    [ qryTime:T ],
               model:       T = Init + R*S*TupleCost,
               coefficient: [ init:Init, tupleCost:TupleCost] ].
```

This hypothesis ignores the influence of the index and clustering scheme. These schemes are considered non-discriminating, which means that their value is ignored during parameter fitting. A more appropriate definition would read:

```
hypothesis =>[name:        hypo_true_btree,
              factor:      [ relCard:R, indexing:true,clustering:btree, selectivity:S ],
              response:    [ qryTime:T ],
              model:       T = Init + R*S*TupleCost,
              coefficient: [ init:Init, tupleCost:TupleCost] ].
```

However, this approach requires four hypothesis definitions. A more concise description is obtained with a **discriminate** property, which results in a different parameter setting for each value combination in the scope.

```
hypothesis => [name:         hypo_indices,
               factor:       [ relCard:R, selectivity:S ],
               response:     [ qryTime:T ],
               model:        T = Init + R*S*TupleCost,
               discriminate: [ indexing:I, clustering:C],
               coefficient:  [ init:Init, tupleCost:TupleCost]
```

3.3 STEP 3: Design of factors and responses

The next step is to describe the experiment building blocks: **factors** and **responses**. The key factor in the WB is the relation size, which runs from 1K to 100K tuples. During development of the TSL program it pays to reduce this range at run time, otherwise debugging the program would be excessively costly.

```
factor  => [   name:     relCard,
               range:    [ 1000 .. 100000 ],
               state:    yes ].
```

```
factor => [    name:     selectivity,
               type:     float,
               range:    [ 0.001, 0.01,0.1 ] ].
```

1. REMINDER: Maybe we can do it with different experiments, however, the current version of the Software Testpilot can only handle one experiment at the time.

The **state** property is derivable from the rest of the TSL program. It means that there exists an action in which the factor is mentioned in the **after** list or in the **retrieve** property.

WB complicates tuple selection, because it uses 1% 10% and a constant of 1 tuple. Therefore, we included the latter as a percentage (0.1%) too. We eliminate the case of selecting 0.001 of relation size 10 using a validity check in the experiment description later on.

ffactor => [	**name**:	clustering,	**factor** => [	**name**:	indexing,
	type:	symbolic,		**type**:	symbolic,
	state:	yes,		**state**:	yes,
	range:	[heap,btree]]		**range**:	[false,true]].

The WB does not exploit all possible secondary index structures offered by Ingres.We assume a btree. The generalization is to replace the indexing factor with an symbolic enumeration of all cases.

response => [	**name**:	qryTime,
	type:	integer,
	unit:	ms,
	range:	(0 .. 100000)].

Note that the **range** of a response variable is a post-condition on values returned by the target system. It is not a limitation on the running time. The latter should be dealt with in the action part (See Section 3.4), for example:

```
Val is P*R,
get_value(response,qryTime,max,Max),
sql_query(select * from N where Attr<Val, result(_,T,_,_,_),Max)
```

3.4 STEP 4: Design interaction with the target system

We finished the description of the abstract test space. The remainder of the TSL program contains a description of the interface with the target system. We start with the connect and shutdown actions to the target system.

action => [	**name**:	dbconnect,	**action** => [	**name**:	dbdisconnect,
	retrieve:	[relCard:unknown,relName: unknown],		**body**:	sql_disconnect].
	body:	sql_connect(wisc)].			

The **retrieve** property should mention all state factors. They are qualified with the initial value, which can also be derived with the *dbconnect* operation.

The *sql_connect* and *sql_disconnect* primitives have been defined in the Ingres sql-interface library. This library should be specified at the beginning of a TSL program as "`:- use_module(library(sql))`", otherwise proper error checking is not always possible.

The main problem introduced by WB is to deal effectively with relations of different size. Ideally, we would like to perform a delta actions, such as addition or deletion of a few tuples. However, the correlation between *unique1* and *unique2* attributes prohibits a simple (and cheap) implementation. Therefore, each time a relation size changes, we discard the old version and create a relation from scratch.

```
action => [  name:     dbcreate,
             after:    [ relCard:Card ],
             cost:     Card,
             body:     ( sql_query(drop tenktup1),
                         sql_query(drop tenktup2),
                         sql_special('wisc create tenktup1'),
                         sql_special('wisc create tenktup2'),
                         concat_atom(['wisc fill tenktup1',Card],Q1),
                         sql_special(Q1),
                         concat_atom(['wisc fill tenktup2',Card],Q2),
                         sql_special(Q2) ) ].
```

Note that relCard should be included as an **after** property, because it determines the final state of the target system[1]. The action dbselect describes the actual measurement of the *qryTime* response variable. Before the selection time can be measured, *the factors relCard* and *selectivity* should contain the values picked for the experiment. The body of the action describes the SQL queries to select the tuples and measure the response time. JThe WB uses both relations to invalidate the page cache. We simply take the response time of the second subexperiment.

```
action => [ name:        dbselect,
            before:      [ relCard:R, selectivity:P ],
            responses:   [ qryTime:T ],
            simulate:    ( T is R*P ),
            cost:        2* R*P,
            body:        ( Val is P*R,
                           sql_query(select * from tenktup2 where unique2 <Val),
                           sql_query(select * from tenktup1 where unique2 <Val,
                                        result(Cpu, Dio, Bio, Elapse, Response)) ) ].
```

The next two action describe clustering and indexing of the Wisconsin relations. The *clustering* and *indexing* factors are contained in the after attributes because the state before the operation is not important, only the target state.

```
action => [ name:       dbcluster,
            after:      [clustering:I ],
            action:     ( sql_query(modify tenktup2 to I on unique2)
                            sql_query(modify tenktup1 to I on unique2)) ].

action => [ name:       dbindex,
            after:      [indexing:I],
            action:     (I=false -> sql_query(drop index ridx1),
                                    sql_query(drop index ridx2);
                         I=true ->  sql_query(create index ridx2 on tenktup2(unique2) with btree),
                                    sql_query(create index ridx1 on tenktup1(unique1) with btree)) ].
```

Note that the WB does not yet analyse the time for index construction. Before we finalize the TSL program, we should assure ourselves that the action set is indeed complete. A partial answer is to consider the following questions:

- target start-up and shutdown actions are fully specified?
- each response variable is mentioned as a response property in exactly one action?

1. REMINDER: ideally the cost should reflect the most recent knowledge acquired

- all possible state transitions in the target system are covered by an action sequence?

3.5 STEP 6: Design target system interface

Assume that we would like to run the experiments on both Ingres and Oracle. This requires a back-end server and a Prolog library to handle interaction with the back end. Refer to appendix The current version includes a library for interaction with a Ingres server.

```
factor =>[  name:      target,
            type:      symbolic,
            range:     [ingres,oracle] ].
```

The action bodies then can differentiate among the system peculiarities. Experimentation with multiple targets then merely requires a slightly more complex implementation of the action bodies. The last action involves defining the control object, which glues everything together for a successful session with the Software Testpilot.

```
controls => [name:        control_block,
             speed:       0.5,
             candidates:  25,
             flightplans: 5,
             simulate:    no,
             startup:     dbconnect,
             shutdown:    dbdisconnect ].
```

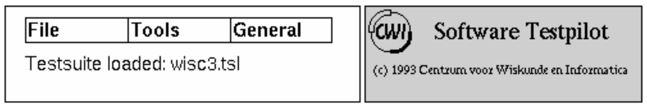
By now the first TSL program has been completed and can be saved in a file for execution later on. The topic of the next section.

4 Using the graphical user interface

This section describes the components of the Software Testpilot user interface and it describes the key steps in running a session. The issues are illustrated with the sample TSL program defined in the previous section and available in the demo directory of the software distribution. The session covers starting the Software Testpilot, loading a TSL program, performing some flight and data analysis, and final shutdown. In passing, we describe the content of the windows displayed and their basic interaction.

4.1 Getting started

Before the Software Testpilot user interface can be started, make sure that the software has been installed in the proper place and that the environment variables are set. (See Appendix A). Once this is done, go to the demo directory of the Software Testpilot and execute the command: *start*. If everything has been installed properly, the top window of the system (Figure 5) becomes visible.



Testsuite loaded: wisc3.tsl

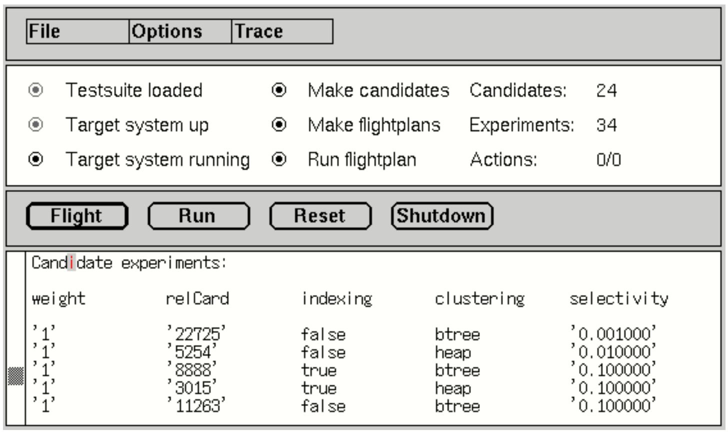
Figure 5: The Software Testpilot main menu

This window is illustrative for the window layout used throughout the system. Its content is based on the widgets provided by the XPCE toolkit, such as predefined layouts for menu bars, button, editing policy. The layout of Software Testpilot windows follow as much as possible the following rules. All windows are decomposed into areas. Selection areas are coloured blue, read-only areas are coloured (grey) and input/output areas (grey)[1].

1. The color setting depends on your *.xrdb* or *.Xdefaults* file. See appendix for adjustments of the color.

Moreover, by convention the selection areas are organized in the left part of a window with a menu bar on the top, a selection column in the middle, and a button list at the bottom. The window title depends on the window manager being used; it may be invisible. We advice you to make them visible. The figures displayed below help you in finding them on the screen.

The top window of the Software Testpilot, recognized by its logo, contains a menu bar **File, Tools, General**. Point and click on a name produces the pop-up menu, denoted as **File:** in the sequel. The menu allows you to load and save TSL programs and results from previous experiments. The option **File:Quit** terminates the session. The menu **Tools:** provide access to the Software Testpilot browsers and editors. The menu **General** contains on-line help and session control commands. Below the menu bar a text area is reserved for essential information about the session, e.g. a welcome and a crash.



```
◉ Testsuite loaded          ◉ Make candidates   Candidates:    24
◉ Target system up          ◉ Make flightplans  Experiments:   34
◉ Target system running     ◉ Run flightplan    Actions:       0/0
```

```
Candidate experiments:

weight      relCard     indexing    clustering    selectivity

'1'         '22725'     false       btree         '0.001000'
'1'         '5254'      false       heap          '0.010000'
'1'         '8888'      true        btree         '0.100000'
'1'         '3015'      true        heap          '0.100000'
'1'         '11263'     false       btree         '0.100000'
```

Figure 6: The Software Testpilot cockpit

The Cockpit window (Figure 6) functions as a console for trace and debugging. The top area contains the menu bar, the middle area contains status indicators, action buttons, and the text console, respectively. The **File:** menu contains facilities for post-session analysis. **File:dum**p dumps the internal Software Testpilot administration. The **Trace:** menu contains toggle buttons to control the trace of the system behaviour at three internal system boundaries.

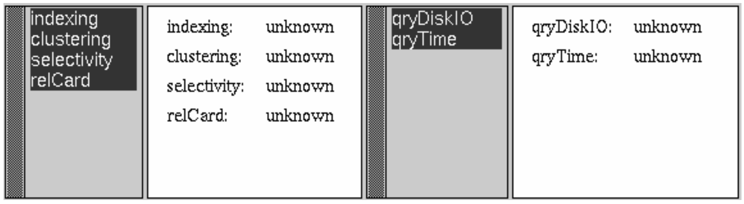
The button **Flight** selects a flight plan and runs it. A continues stream of plans is activated by pressing **Run**. It can be stopped by pressing the same button again. **Reset** destroys all experimental results since system restart (or reset). It should be used whenever major changes have been made in the TSL program during a session. **Shutdown** shuts the target system down by execution bodies of the proper action object(s).

The information provided so far should be sufficient to move around in the user interface. To continue, load the case study by pushing **File:Load Testsuite**. This opens a file selector window. Go to the *specs* directory and select the file *demo.tsl*. Notice the difference in the **Cockpit**, and **Focus** window shortly after. The Cockpit status light for **Testsuite loaded** should be on (If not Consult a local guru). The **Focus** window (Figure 7) provides an overview of the current state of the factors selected (left sub-window) and response values obtained (right sub-window). It contains two selector lists (indexing...relCard) and (qryDiskIO,qryTime). Their content can be manipulated by pointing out elements in their selection column (on the left of the window) as follows.

Point/click on relCard in the left most column. The effect is that all information except for *relCard* disappears from the response values window (i.e. left sub-window) . To view multiple factors at the same time you should hold down the shift key as well. Please try *shift/point/click indexing*, which shows now two lines of information. This convention is used throughout the Software Testpilot. The right column of this sub window displays the value *unknown*. It will be replaced by the last value measured.

Point/click button **Flight** *in the* **Cockpit** *window.* The indicators in this window flash until a stable situation is reached with **Target system up** light on, candidates set to a positive number, and the number of experiments run=1. [1] The **Focus** window has also changed. It now shows the values used/obtained during this flight. For mul-

1. The color of Testsuite Loaded and Target System Up should be identical, otherwise an error has occurred during the flight. This likely means that the target system could not be properly initialized (or an error in the Testsuite was encountered).



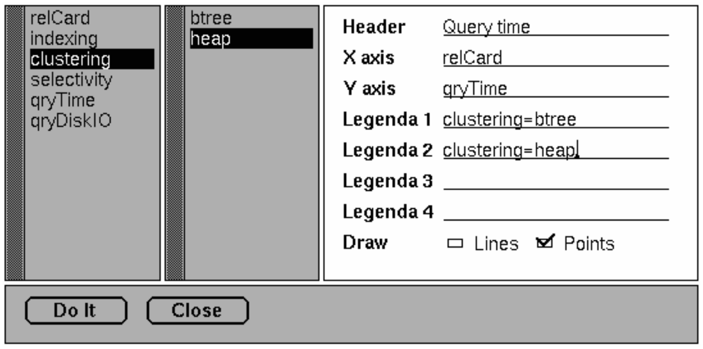
indexing:	unknown
clustering:	unknown
selectivity:	unknown
relCard:	unknown

qryDiskIO:	unknown
qryTime:	unknown

Figure 7: The Software Testpilot focus window

tiple flights press **Cockpit:Run** and after a while press **Cockpit:Stop**. The results have flashed over the screen in the Focus window. You can study the results by selecting the monitor tool in the main menu.

*Point/click **Tools:monitor**.* A new window appears with three selection columns. This window provides an interface to the Xgraph package, which supports display of 2.5 dimensional graphs. The x-axis contains the factors, shown in the left most browser. The y-axes contains a response item, selected also through this browser.



Header	Query time
X axis	relCard
Y axis	qryTime
Legenda 1	clustering=btree
Legenda 2	clustering=heap
Legenda 3	
Legenda 4	
Draw	☐ Lines ☑ Points

Figure 8: Xgraph monitor interface tool

Set the cursor on the X axis field and type 'relCard'. Do the same with the Y axis field and type 'qryTime'[1]. Then *point/click* **Do It**. The effect is the graph shown in Figure 9a, which illustrates the relation measured between relation cardinality and query response time[2]. You may guess that there exists a factor leading to a linear behaviour. This works as follows:

Go back to the monitor window point/click clustering with the left button. You will notice that the second browser is filled with values used so far in the flights. *Set the cursor on Legend 1 and type: clustering=btree*. The Xgraph now shows a subset of all points. Namely, those satisfying the condition that it concerns a btree related experiment. To highlight the measurements, go back to the monitor tool and select line display. Similarly, the seconaxed legend can be filled with clustering=heap, which leads to a more complete presentation of the performance results (Figure 9b). In most cases, this mechanism is sufficient to present and analyse the results. However, the Software Testpilot allows an arbitrary Prolog query to be associated with a legend to filter the elements. A Prolog arithmetic expression can be used in the axes as well. For example, change X-axis into *log(relCard)* and the Y-axis into *qryTime/qryDiskIO*.

Viewing the graph you may wonder what the effect would be if a factor range is changed. For example, what would happen if we extend the relation cardinality range. *Please go to Focus window and point/click the relCard in the blue section*. The effect is that the Editor tool is popped up with the information on relCard displayed. The bottom shows the selection boxes for the properties predefined. The top contains text areas. All, except field name can be edited by placing the cursor at the appropriate place. *Place the cursor on the range field and change it to [10..10000000] followed by a return (!)*[3]. At this point, the TSL program has been changed. Simple errors can be caught by **Testsuite:check** before continuing. Moreover, the experimental results gathered are generally inapplicable now. They are removed with the command **Reset** in the Cockpit window. Note that the target system is not (!) being reset. The Software Testpilot assumes the state reached so far.

Run again a number of flights and inspect the result with the monitor. Get acquainted with the windows and

1. As a short cut: *Point/click relCard with the middle mouse button (!)*, which pastes the item selected at the location indicated by the cursor. *Point/click text field labelled Y (left button) and point/click qryTime (middle button).*
2. For a description of the actions directed to this window read the Xgraph manual, included as Appendix B.
3. Complex edit actions become easier by accessing the build-in editor with the middle button.

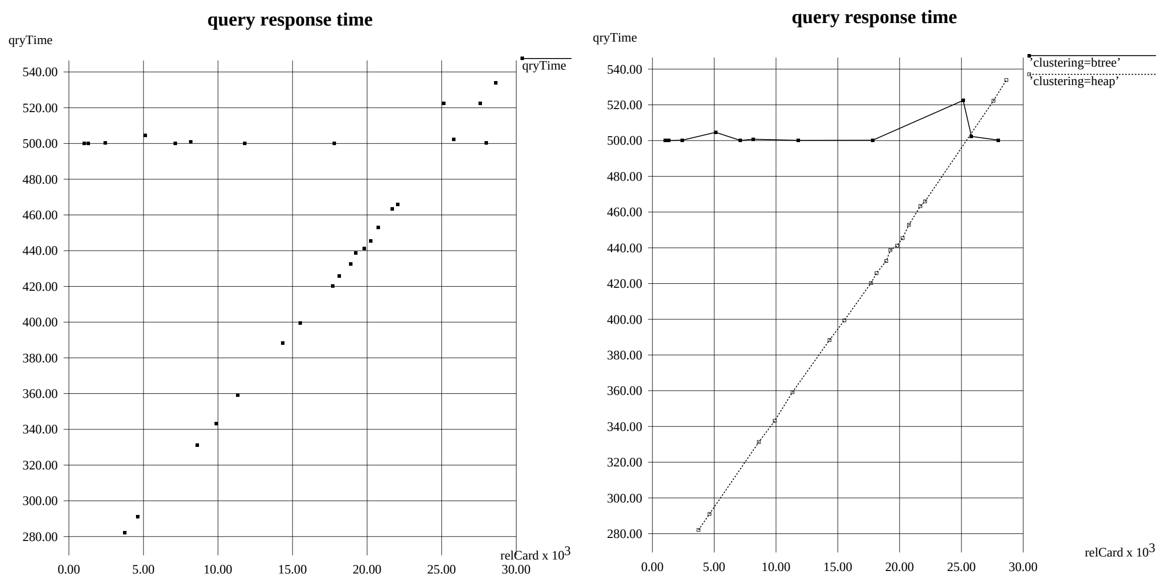
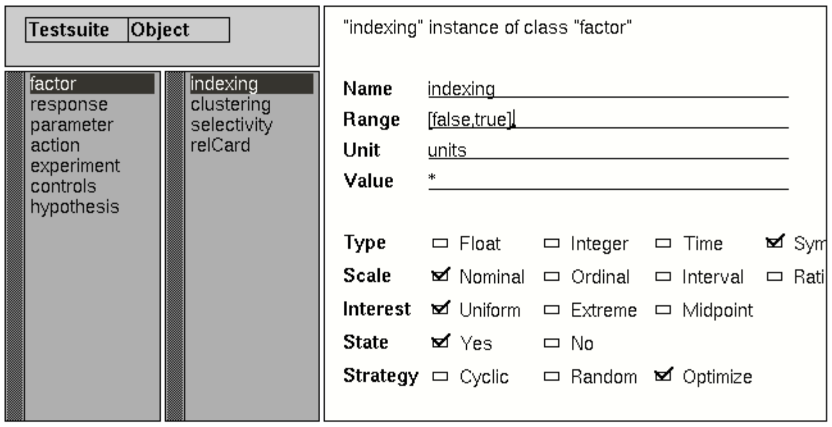


Figure 9: Xgraph windows showing measurements


Testsuite | Object

factor
response
parameter
action
experiment
controls
hypothesis

indexing
clustering
selectivity
relCard

"indexing" instance of class "factor"

Name	indexing
Range	[false,true]
Unit	units
Value	*

Type	Float	Integer	Time	[✓ Sym]
Scale	[✓ Nominal]	Ordinal	Interval	Rati
Interest	[✓ Uniform]	Extreme	Midpoint	
State	[✓ Yes]	No		
Strategy	Cyclic	Random	[✓ Optimize]	

Figure 10: The test suite editor

functionality described. More advanced features are described in a separate document and can be deduced from the XPCE manual.

4.2 End of Session

Likely, you see several windows organized around the screen. You can change and save the organization using the window manager facilities and the menu item **general:savesettings**. The next session will display your preferred ordering of the desktop.

Appendix A

Installation guide for the Software Testpilot

The complete Software Testpilot package is compressed and archived in the file testpilot.tar.Z. To uncompress and un-archive this package use the UNIX command: `zcat testpilot.tar.z | tar xvf -`. This loads software into a new directory called testpilot.

There are currently two version branches of the Software Testpilot. They differ only in the user interface. The first one is the textual based user interface. To use this branch, a copy of SWI-prolog is required. This Prolog version is public domain available from ftp site: *swi.psy.uva.nl* [192.42.96.1]. The second branch is the Testpilot with a graphical user interface. To install this branch the'xpce' toolkit also it required, which can be obtained from the same source but carries a price tag. For more information contact Jan Wielemaker (jan@swi.psy.uva.nl)

The graphical monitor used in the Software Testpilot is based on the public domain program Xgraph.Author: David Harrison, UC Berkeley Electronics Research Lab, (davidh@ic.Berkeley.EDU, ...!ucbvax!ucbcad!davidh) and is included in the archive.

To install the Software Testpilot execute the following commands and tests.

- Set the environment variable TESTPILOT to the location of the Software Testpilot distribution directory.
- Append $TESTPILOT/bin to your PATH environment variable and do a 'rehash' (only for csh users)
- Ensure that the following programs can be launched from the shell:

- pl		
- xpce		* GUI only *
- xgraph	($TESTPILOT/bin/xgraph)	* GUI only *
- fit	($TESTPILOT/bin/fit)	
- bc	(/usr/bin/bc)	
- tmpdir	($TESTPILOT/bin/tmpdir)	
- mknod	(/etc/mknod)	

- Install the Software Testpilot by entering the command: *make* in the directory $TESTPILOT/
- You should now be able to start the software testpilot graphical user interface using the 'gtp' command or the textual user interface using the ttp command.

Directory structure:

$TESTPILOT:	Software Testpilot Root directory
$TESTPILOT/lib:	General library directory
$TESTPILOT/lib/pl:	Prolog library directory
$TESTPILOT/bin:	Binary directory
$TESTPILOT/src:	Prolog source code
$TESTPILOT/specs:	Example specs of TSL programs
$TESTPILOT/doc:	Design documents and manuals
$TESTPILOT/utils:	utilities for the Testpilot (foreign PD code)

Appendix B

TSL program for Wisconsin benchmark

```
:- use_module(library(sql)).
:- use_module(library(simulate)).

:- arithmetic_function(randerror/1).

experiment => [  name:          'q1-q8,q10',
                 factor:        [ relCard:R, indexing:I, clustering:C, selectivity:P],
                 response:      [ qryTime:T, qryDiskIO:D ] ,
                 check:         (R*P>=1.0),
                 design:        random,
                 candidates:    25,
                 flightplans:   5 ].

hypothesis => [  name:          hypo_indices,
                 factor:        [ relCard:R, selectivity:S ],
                 response:      [ qryTime:T ],
                 model:         (T = Init + R*S*TupleCost ),
                 discriminate:  [ indexing:I, clustering:C ],
                 coefficient:   [ init:Init, tupleCost:TupleCost ] ].

factor => [      name:          relCard,
                 type:          integer,
                 range:         [ 1000..100000 ],
                 state:         yes ].

factor => [      name:          selectivity,
                 type:          float,
                 range:         [0.001, 0.01,0.1] ].

factor => [      name:          clustering,
                 type:          symbolic,
                 state:         yes,
                 range:         [heap,btree] ].

factor => [      name:          indexing,
                 type:          symbolic,
                 state:         yes,
                 range:         [false,true] ].

response => [    name:          qryTime,
                 type:          time,
                 unit:          seconds,
                 range:         ( 0 .. 100000 ) ].

response => [    name:          qryDiskIO,
                 type:          integer,
                 unit:          io_requests,
                 range:         ( 0 .. 100000 ) ].

parameter => [   name:          init ].

parameter => [   name:          tupleCost ].

action => [      name:          dbconnect,
                 retrieve:      [ relCard:unknown, indexing:unknown, clustering:unknown],
                 body:          sql_connect(wisc) ].

action => [      name:          dbdisconnect,
                 body:          sql_disconnect ].



action => [      name:          dbcreate,
                after:          [ relCard:Card, indexing:unknown, clustering:unknown ],
                cost:           ( Card ),
                body:           (      sql_query(drop tenktup1),
                                       sql_query(drop tenktup2),
                                       sql_special('wisc create tenktup1'),
                                       sql_special('wisc create tenktup2'),
                                       concat_atom(['wisc fill tenktup1 ',Card],Q1),
                                       sql_special(Q1),
                                       concat_atom(['wisc fill tenktup2 ',Card],Q2),
                                       sql_special(Q2) ) ].

action => [      name:           dbselect,
                 before:         [ clustering:C, relCard:R, selectivity:P ],
                 response:       [ qryTime:T, qryDiskIO:D ],
                 simulate:       (      C=heap -> T is 245 + R*randerror(1)/100 ;
                                        C=btree -> T is 500 + 0.9*R*P*randerror(1)/100 ),
                 cost:           ( R*P ),
                 body:           ( Val is P*R,
                                   sql_query(select * from tenktup2 where unique2 <Val),
                                   sql_query(select * from tenktup1 where unique1 <Val,
                                                               result(_,D,_,T,_))) ].

action => [      name:           dbcluster,
                 require:        [ relCard:R ],
                 before:         [ relCard:R ],
                 after:          [ clustering:I ],
                 cost:           1000,
                 body:           ( sql_query(modify tenktup2 to I on unique2),
                                   sql_query(modify tenktup1 to I on unique2))].

action => [      name:           dbindex,
                 require:        [ relCard:R ],
                 before:         [ relCard:R ],
                 after:          [indexing:I],
                 cost:           1000,
                 body:           ( I=false -> sql_query(drop index ridx1),
                                                sql_query(drop index ridx2);
                                   I=true -> sql_query(create index ridx2 on
                                                        tenktup2(unique1)
                                                        with structure=btree),
                                             sql_query(create index ridx1 on
                                                        tenktup1(unique1)
                                                        with structure=btree)) ].

controls => [    name:           control_block,
                 speed:          0.5,
                 simulate:       no,
                 startup:        dbconnect,
                 shutdown:       dbdisconnect ].
```

Appendix C

Xgraph manual page

NAME

xgraph - Draw a graph on an X11 Display

SYNOPSIS

xgraph [options] [=WxH+X+Y] [-display host:display.screen][file ...]

DESCRIPTION

The xgraph program draws a graph on an X display given data read from either data files or from standard input if no files are specified. It can display up to 64 independent data sets using different colors and/or line styles for each set. It annotates the graph with a title, axis labels, grid lines or tick marks, grid labels, and a legend. There are options to control the appearance of most components of the graph.

After xgraph has readthe data, it will create a new window to graphically display the data. The interface used to specify the size and locationof thiswindow depends on the window manager currently in use. Refer to the reference manual of the window manager for details.

Once the window has been opened, all of the data sets will be displayed graphically (subject to the options explained below) with a legend in the upper right corner of the screen. To zoom in on a portion of the graph, depress a mousebutton in the window and sweep out a region. Xgraph will then open a new window looking at just that portion of the graph. Xgraph also presents three control buttons in the upper left corner of each window: *Close*, *Hardcopy*, and *About*. Windows are closed by depressing a mouse button while the mouse cursor is inside the Close button. Depressing a mouse button while the mouse cursoris in the *Hardcopy* button causes a dialog to appear asking about hardcopy (printout) options. These options are described below:

- **Output Device**: Specifies the type of the outputdevice (e.g. “HPGL”, “Postscript”, etc). An output device is chosen by depressing the mouse inside its name. The default values of other fields will change when you select a different outputdevice.
- **Disposition**: Specifies whether the output should go directly to a device or to a file. Again, the default valuesof other fields will change when you select a different disposition.
- **File or Device Name**: If the disposition is “To Device”, this field specifies the device name. A device name is the same as the name given for the -P command of lpr(1). If the disposition is “To File”, this field specifies the name of the output file.
- **Maximum Dimension**: This specifies the maximum size of the plot on the hardcopy device in centimetres. xgraph takes in account the aspect ratio of the plot on the screen and will scale the plot so that the longer side of the plot is no more than the value of this parameter. If the device supports it, the plot may also be rotated on the page based on thevalue of the maximum dimension.
- **Include in Document**: If selected, this optioncauses xgraph to produce hardcopy output that is suitable for inclusion inother larger documents. As an example, when this option is selected the Postscript output produced by xgraph will have a bounding box suitable for use with psfig.
- **Title Font Family**: This field specifies thename ofa font to use when drawing the graph title. Suitable defaults are initially chosenfor anygiven hardcopy device. The value ofthis field is hardware specific - refer to the device reference manual for details.

- **Title Font Size**: This field specifies thedesiredsize of the title fonts inpoints (1/72 ofan inch). If the device supports scalable fonts, the font will be scaled to this size.

- **Axis Font Family and Axis Font Size**: These fields are like Title FontFamily and Title Font Size except theyspecifyvalues for the font xgraph uses to draw axis labels, and legend descriptions.
- **Control Buttons**: After specifying the parameters for the plot, the “Ok” button causes xgraph to produce a hardcopy. Pressing the “Cancel” button willabort the hardcopy operation. Depressing the About button causes Xgraph to display a window containing the version ofthe program and an electronic mailing address for the author for comments and suggestions.

AUTHOR

David Harrison University of California