Method rule checking in a generic design editing system

by Ray Welland, Stephen Beer and Ian Sommerville

This paper describes a means of incorporating method rule checking in a design editing system intended to support the production of designs expressed in method-specific diagrammatic notations. The novel characteristic of this editing system is the fact that it may be tailored to any notation using a method description language and a graphical tool to define the vocabulary of the notation. Syntactic and semantic rules are expressed in the method description language and are checked, interactively, during an editing session.

1 Introduction

The use of structured methods in the software design process has been advocated for a number of years by practitioners such as DeMarco, Yourdon and Jackson [1–3]. Different practitioners advocate different methods, and a multiplicity of methods [4] have evolved. Before the advent of low-cost personal computers, users of these methods were constrained to use them as purely manual techniques for analysis and design. Now, the availability of so-called CASE workbenches on personal workstations, and a general awareness of the value of software tools have opened up the market for these methods. They are now relatively widely used, particularly in the data-processing systems domain.

The differences between the various methods are not of concern in this paper. Their similarity is that they all make use of diagrammatic representations of a software design, and associate rules and guidelines with these which, it is claimed, result in a 'good' software design. The end-result is an attributed directed graph representing the software design. Although these are not methods in the sense that two designers will necessarily arrive at the same result given the same requirements, they do impose a structure on the design process and ensure that the designer produces a design which is self-consistent.

Of course, without automated checking of the rules and guidelines, consistency cannot be guaranteed. Thus, as well as supporting the drawing of particular classes of diagrams, most CASE workbenches include an element of checking which ensures that some of the rules of the method are enforced.

The work described in this paper was first proposed in the context of a programme to develop an integrated project support environment (IPSE). The distinctions between such an environment and CASE workbenches are as follows:

- An IPSE is intended to provide support facilities for the entire software process, from initial requirements specification to software maintenance and support. It should also incorporate management tools which have access to the products created during software development. By contrast, CASE workbenches are oriented towards the support of design within the software process. Indeed, it may well be the case that the designs developed using these systems are programmed on some other computer.
- An IPSE is particularly concerned with the problems of managing configurations of software components. For large systems with a long lifetime, configuration management (CM) is probably the most significant problem. CASE workbenches are not geared to providing CM support.
- An IPSE is an open environment which can be configured to support a variety of activities. Users can incorporate their own support tools with an IPSE. CASE workbenches are (usually) closed environments where the user has only available the tools supplied by the workbench vendor.

This last point, namely the openness of an IPSE, was a key requirement for our system. We needed a design editing facility comparable to that provided with CASE workbenches, but we could not tie that system to a particular method. We had to define an approach which allowed powerful diagram editing and checking facilities to be configured for any particular design method.

This is relatively straightforward if all that is required is a means of producing diagrams. However, our aim was to produce a system which, as far as possible, checked the user's design interactively. Thus, we had to find some general way of expressing the rules associated with a particular method, and build a general-purpose rule checker which could be incorporated in an editing system.

The approach which we adopted was based on the notion that a design method is simply a language with three components:

- vocabulary; the set of symbols used to represent design.
- syntax; the rules governing the spatial organisation and connectivity of symbols on a diagram.
- semantics; those rules and guidelines which constrain the designs which can be produced, and which should result in 'good' designs.

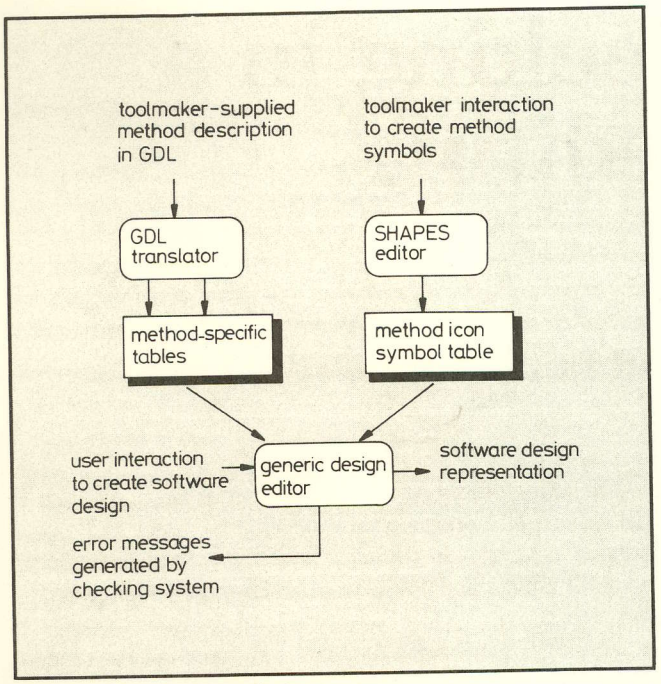

Fig. 1 The design editing system

We thus designed a number of integrated tools which allowed design methods to be described and method editors to be generated. These were

- a language called GDL [5], which allowed the rules associated with a method to be set out;
- a software tool which allowed the symbol vocabulary to be designed;
- a generic editing system which provided basic facilities to create and modify design diagrams;
- a checking system, incorporated with the editor, which checked the given design against the specified method rules.

An overview of this system is shown in Fig. 1, and a broad description is given in Reference 6. The system has now been implemented and is available in various instantiations as part of the ECLIPSE [7] integrated project support environment.

This paper is concerned with how interactive checking can be incorporated in such a system, and we do not discuss features of the editing system here. Initially, we review tools which provide comparable functionality to our own. We briefly introduce the basic structure of GDL, the language which we developed for the description of diagrams, and this is followed by a more detailed discussion of the assertion mechanism which is used to specify design checks. We describe the implementation of the prototype checking system and discuss some of the outstanding problems.

2 Comparable tools

Tools available for software design diagram editing can be categorised in three principal ways:

☐ **method-specific** or **configurable**: there are a large number of tools available which are restricted to use with one method or group of methods. Configurable systems allow tool builders to specify their own methods or local variations on existing methods.

☐ **syntax-driven** or **permissive**: a syntax-driven approach maintains a correct diagram at all times, forcing the user into a rigid interaction style. A permissive approach allows diagrams to go through incomplete or inconsistent states, and there is a choice between **interactive** and **off-line** checking.

☐ **stand-alone** or **integrated** with other tools: a number of tools are available which allow the user to draw diagrams, store them and edit them, but further manipulation of the stored diagram representation is left to the user. Integrated tools allow other types of tools, such as code generators, to manipulate the output from the design editor.

The principal aim of our work was to produce a **configurable** design editing tool which was **permissive** with **interactive** checking facilities. Additionally, the design editor had to be **integrated** within an IPSE, rather than operate as a stand-alone tool.

We do not believe that 'permissiveness' implies that the user should be able to draw any diagram. However, the order of construction is immaterial, and diagrams may be in an incomplete and inconsistent state at various times. The user may choose to ignore error messages during an interactive session, but obviously these errors will need to be resolved at some stage before the diagram is acceptable.

There are a large number of tools available which are generally classified as CASE workbenches. The majority of these are method-specific, and the approach to checking varies widely from a largely syntax-directed approach through to off-line checking, producing long lists of error messages after the diagram has been drawn. These tend to be stand-alone tools, in that they are not integrated with an IPSE, but they often interface to other tools such as a data dictionary. Documentation for these tends to be proprietary, and we have reviewed only two of them, the Analyst and MacCadd, which share some similarities with our tool.

The Analyst [8] is a method-specific tool which provides support for CORE and MASCOT. The syntax rules, specifying valid symbol shapes and connections, are hard-coded in Pascal. However, the checking rules are expressed in Prolog, and these can be extended by the user. Prolog notation is also used to store the design representation. The Analyst provides interactive checking through specific checks invoked from a menu of checks or continuously throughout the design creation. The user has the option of specifying which checks should be applied continuously. Failure of a check is categorised as

- *guidance*, indicating problems of 'style';
- *warning*, usually indicating incomplete specification;
- *error*, a more serious mistake which should be fixed before the diagram can be stored.

The Analyst provides the most sophisticated interactive checking of the tools we have looked at, but it is method-specific and stand-alone.

The nearest tool to ours in philosophy is MacCadd [9], although it is stand-alone. MacCadd allows users to specify their own diagram types using Prolog rules to identify design entities and their connectivity. The vocabulary of symbols from which the user may choose is fixed, and there is no provision for more than one textual annotation to be associated with a particular design entity; a common requirement from many methods. The design representation is also stored as a sequence of Prolog causes which could be

processed by the user. The manual states that further tools could be written to process this output, but we are not aware of any such software being available.

The interactive checking facilities available in MacCadd are more restrictive than those available in our tool. For example, there is no freedom on the positioning of names for nodes; the name is always inside the node symbol at a fixed position. The tool provides connectivity constraints on the way in which nodes are linked and the types of links which are associated with particular nodes. However, there is no way of expressing an interactive check on the number of links associated with a node. A check on the structure of nodes of a particular type can be specified using a 'customised search' to be applied node by node to the diagram. In general, MacCadd concentrates on connectivity constraints for interactive checking and ignores positional constraints and quantifiable constraints.

Both the Analyst and MacCadd use Prolog for their underlying design representation but tight integration with an IPSE database is very difficult with such an approach.

The GRANOT (graphical notations) project at the Open University [10] includes a component called PSN (picture specification notation) which is similar in concept to GDL. PSN is a meta-language which is used to drive design editors within a prototype development environment. The description of the symbol shapes is handled by a separate interactive object editor which is comparable to our Shapes Editor. PSN specifies two levels of rules: syntactic, specifying the combinations of symbols which are allowed, and semantic, concerning the meaning of each syntactically correct diagram. The approach to checking is syntax-directed, where the diagram is as syntactically correct as possible at all times.

3 GDL overview

The graph description language (GDL) was designed to describe those software design diagrams based on directed graphs. Widely used examples of these are Structured Design [2], JSD [3] and SSADM [11]. Our initial design for GDL was described in an earlier paper [5].

A GDL description is compiled into a set of tables which are used to drive the design editor (DE). The language is intended to be used by a tool builder not an end-user of ECLIPSE.

The basic framework of a GDL description is shown in the following simple example:

```
method FSM.simple

with FSM

forward TRANSITION is LINK

type STATE is NODE
  (fromStates : in bag of TRANSITION;
  toStates : out bag of TRANSITION)

for STATE use SYMBOL(FSM.state)
  ++ STATE_NAME (STRING)

type TRANSITION is LINK
  (startState : in STATE;
  endState : out STATE)

for TRANSITION use SYMBOL (FSM.transition)
  ++ EVENT (STRING)
```

This GDL gives the basic definitions for a simple finite state machine diagram, with labelled nodes representing states and labelled arcs for transitions between states. The first three lines are GDL compiler directives: the **method** clause is used for identification and generating names for the tables produced by the compiler; the **with** clause links this GDL description to the shapes library for the method; and the **forward** clause identifies a forward reference.

The **type** statement for STATE defines that an instance of this node will have a collection of input links (fromStates) and a collection of output links (toStates). These collections are identified as bags, since multiple connections between states are allowed. (Both bags and sets are supported in GDL). The type statement for TRANSITION describes it as a directed link between two nodes of type STATE.

The representation expression is introduced by the word **for**. The SYMBOL clause links the type to its symbol which is stored in a shape library. This clause is followed by a concatenated list of other annotations associated with the type. In the above example both STATE and TRANSITION have a single mandatory label of type STRING; optional labels are shown by enclosing them in square brackets.

Each element of the representation expression is mapped onto a **label**, which is a rectangular area containing either a graphical symbol or text. This makes the definition of positional constraints such as 'above', 'below' and 'encloses' simple to define. Therefore, in the above example there will be a label for the symbol for STATE and one for its name (STATE_NAME).

A GDL compiler based on this part of the language was implemented, and some deficiencies were discovered and new requirements became apparent. A major deficiency in our original design concerned the handling of ports and windows in MASCOT 3 [12]; these are nodes which are owned by other nodes and used as connection points for links. We added the concept of **dependent nodes** to handle this relationship between nodes.

We had also encountered problems in the representation of nodes which enclosed other nodes, such as occur in the JSD system implementation diagram [3]. The dependent node idea solved this type of problem as well.

To illustrate the use of dependent nodes, a simple example of a diagram convention which aggregates nodes is shown in Fig. 2. This is a fragment of a structure diagram in which a group of modules, accessing a common data module, have been identified as a 'cluster'.

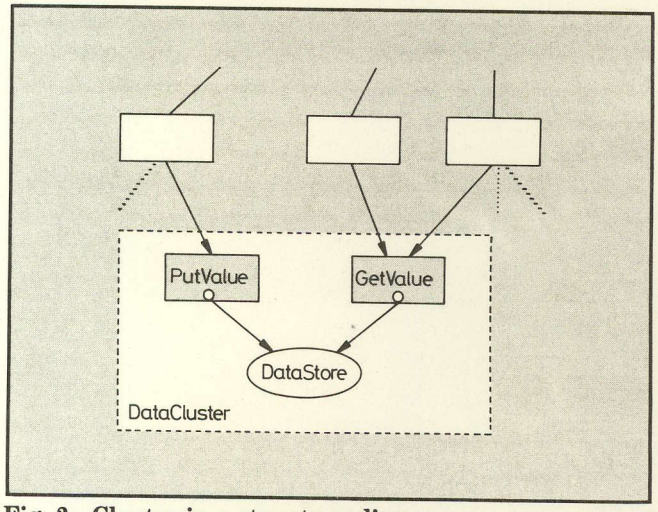

Fig. 2 Cluster in a structure diagram

A small part of the GDL describing this diagram is shown below:

```
type LEAF_MODULE is MODULE
  (clustered : owned by CLUSTER;
  callers : in set of CALL;
  data_link: out DATA_REF)

type DATA_MODULE is MODULE
  (clustered : owned by CLUSTER;
  references : in set of DATA_REF)

type CLUSTER is NODE
  (encloses : owner of set of MODULE)
```

In the above example, CLUSTER is an aggregate node which encloses a collection of modules in a structure diagram. The relationship between owner and owned nodes is one to many, thus an owned node always has a unique parent node.

GDL includes a type hierarchy with inheritance of properties down the hierarchy. Our model of inheritance was deliberately simplified, as we were not interested in investigating inheritance *per se*. Therefore, a subtype can only inherit a complete syntactic unit from its supertype or overwrite that unit; there is no selective overwriting. Syntactic units which may be inherited are the parameter list associated with a type declaration, the representation expression or a complete assertion (see below).

In our early work we spent considerable effort looking at a representation for the expansion of a node into another diagram within a hierarchy of diagrams. However, in the IPSE context, when a design entity is expanded it is not necessarily going to expand to another diagram of the same type or even necessarily another diagram. In the ECLIPSE environment it is just as likely to expand into a textual representation or a program fragment. Therefore, there was no point in GDL describing diagram hierarchies and the DE providing 'zooming' facilities; this was better handled externally. The DE is invoked under the control of a method tool, and when an entity in the current diagram is selected for expansion the DE hands control back to the method tool, passing a reference to the current design entity. The method tool then invokes the appropriate tool to handle the expansion of the entity.

4 Assertions in GDL

Assertions are used in GDL to specify constraints on the construction of a diagram which cannot be expressed via the type hierarchy. The general form of an assertion in GDL is

⟨heading⟩ : ⟨selector expression⟩ : ⟨check expression⟩

The assertion heading is required for two reasons: to tie the assertion to a particular design entity, and more importantly, to allow specific assertions in the type hierarchy to be overwritten. The ⟨selector expression⟩ specifies the particular design objects which are relevant for this assertion; it narrows the search from all possible design entities to those of interest. The ⟨check expression⟩ is a Boolean expression which is applied to those design entities that have been selected. GDL includes a range of built-in functions which can be used in the ⟨check expression⟩; these allow the values or positions of labels to be compared and the size of a set to be enumerated, for example.

The ⟨selector expression⟩ part of the assertion is optional. An example of an assertion which could be associated with the above FSM example is shown below:

```
assertion Connected (STATE):
  not Empty (fromStates)
  and
  not Empty (toStates)
```

In this example the ⟨selector expression⟩ implies the current design entity because the ⟨check expression⟩ refers to the parameters of the type. This assertion ensures that no STATE in the FSM has no links; the assertion will evaluate to false if both the input (fromStates) and output (toStates) bags are empty. Note that the user is only warned if an assertion evaluates to false; a true assertion means that the diagram satisfies the given constraint and processing continues.

There are three selectors used in GDL: 'inst', 'forall' and 'exists'. The **inst** selector gives a handle on the current instance of a design entity in order to place constraints on its labels, for example. The **forall** selector identifies a set of design entities, and for all of these selected design entities the associated ⟨check expression⟩ must evaluate to true for the assertion to be satisfied. The **exists** selector identifies a set of design entities, for at least one of which the ⟨check expression⟩ must evaluate to true for the assertion to hold.

A typical use of the **inst** selector is in label positioning assertions, as shown below:

```
assertion S_name_inside (STATE):
  inst i:
  Encloses (GetLabel (i, SYMBOL),
    GetLabel (i, STATE_NAME))
```

This assertion ensures that, for the current instance of the STATE node, the symbol of the node must completely enclose the string STATE_NAME. The function GETLABEL returns a reference to a label, and the 'Encloses' function compares the areas occupied by two labels and returns true if the first label completely encloses the second.

Another example of an assertion which uses the **inst** selector in conjunction with another selector is shown below:

```
assertion Properly_connected (STATE):
  inst i:
  exists j; Member (toStates, j):
  Destination (j) /= i
```

This assertion is added to the GDL for the FSM diagram to prevent a state satisfying the 'Connected' assertion given above by simply being connected to itself and no other state! The third line of this assertion is a qualified selection, where the variable j is bound to the members of the bag 'toStates', and **exists** is satisfied as soon as the Boolean expression on the fourth line evaluates to true. The function 'Destination' returns a reference to the node which is at the other end of specified link.

A message facility is included in GDL to allow the system to generate meaningful error messages in the event

of assertion failure, i.e. when an assertion evaluates to false, indicating an error. For example

message S_name_inside (STATE):
"Name of state must be inside its symbol"

If the assertion 'S_name_inside' evaluates to false, then the associated message string is displayed to the user by the DE. If no message clause is given, then a default message of the form 'assertion S_name_inside failed' will be displayed.

Some constraints are not expressible in GDL because they require information from outside the design diagram being edited. For example, the validation of the name for a design entity may require checking a table of names held elsewhere in the IPSE database. To allow for this type of constraint, **method dependent functions** may be linked into assertions. These functions will be passed values by the DE and must return the value true or false, like any other function embedded in an assertion. However, once control has been passed to them they can do anything, even open an alternative dialogue with the user outside the DE.

5 Method rules

Rules which are associated with a design method can be classified in an analogous way to those categories which are used in a programming language compilation system. Our classification of rules is

- **Lexical** — concerned with the symbols and labels which appear on a diagram; these are the basic vocabulary of the diagram. Correctness is ensured by restricting the menu options available to the user of the design editor.
- **Type** — GDL is a language which allows type rules to be specified, and these can be used to apply static type checking, again by generating restricted menus in the DE. However, not all type rules can be checked statically.
- **Syntactic** — these rules are concerned with the layout of the diagram. There are connectivity rules which cannot necessarily be expressed using type rules. Most methods have layout constraints concerning symbols and labels or relative positioning of entities.
- **Semantic** — typically these are rules which are concerned with the naming of design entities, either relative to other entities on the same diagram or related to other representations of the design.

To discuss the implementation of the assertion checking mechanism and the categorisation of assertions some examples are presented. These examples are abstracted from the GDL description for MASCOT 3, the largest and most complex GDL description we have so far written and compiled. For the rest of this paper MASCOT 3 will be assumed. Description of the MASCOT method is beyond the scope of this paper; for a comprehensive introduction the reader should see Reference 12. (A detailed description of the method can be found in the MASCOT Handbook [13]). For those familiar with MASCOT, we should emphasise that some simplifications have been made to reduce the amount of detail given in the examples.

The diagram fragment shown in Fig. 3 illustrates a few of the design entities used in MASCOT. There is a *subsystem* (rounded box), which has two *ports* (P1 and P2) and one *window* (W1), and an *activity* (circle), which has two

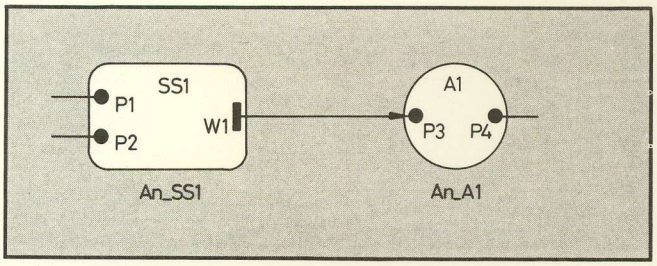

Fig. 3 A MASCOT 3 design fragment

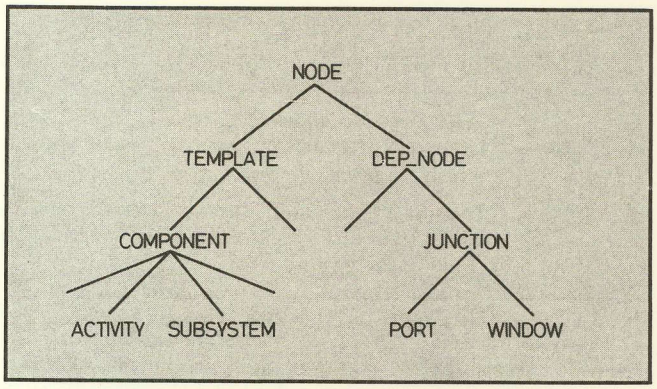

Fig. 4 Part of the MASCOT 3 type hierarchy

```
type TEMPLATE is NODE

type DEP_NODE is NODE

type COMPONENT is TEMPLATE

type SUBSYSTEM is COMPONENT
  (junctions : owner of set of JUNCTION)

for SUBSYSTEM use SYMBOL (MASCOT.subsystem)
  ++ TEMPLATE_NAME (STRING)
  ++ COMPONENT_NAME (STRING)

type ACTIVITY is COMPONENT
  (ports : owner of set of PORT)

for ACTIVITY use SYMBOL (MASCOT.activity)
  ++ TEMPLATE_NAME (STRING)
  ++ COMPONENT_NAME (STRING)

type JUNCTION is DEP_NODE
  (parent : owned by TEMPLATE;
  in_paths : in set of PATH;
  out_paths : out set of PATH)

type PORT is JUNCTION

for PORT use SYMBOL (MASCOT.port)
  ++ JUNCTION_NAME (STRING)

type WINDOW is JUNCTION

for WINDOW use SYMBOL (MASCOT.window)
  ++ JUNCTION_NAME (STRING)

type PATH is LINK
  (one_end : in JUNCTION;
  other_end : out JUNCTION)

for PATH use SYMBOL (MASCOT.path)
```

Fig. 5 GDL for MASCOT

ports (P3 and P4). The window W1 is joined to the port P3 by a *path*. Note that paths in MASCOT join nodes at special junction points; direct links to node boundaries are not permitted. A subsystem has a template name (SS1), which appears inside its symbol, and a component name (An_SS1), which appears outside; activities are similarly named. An

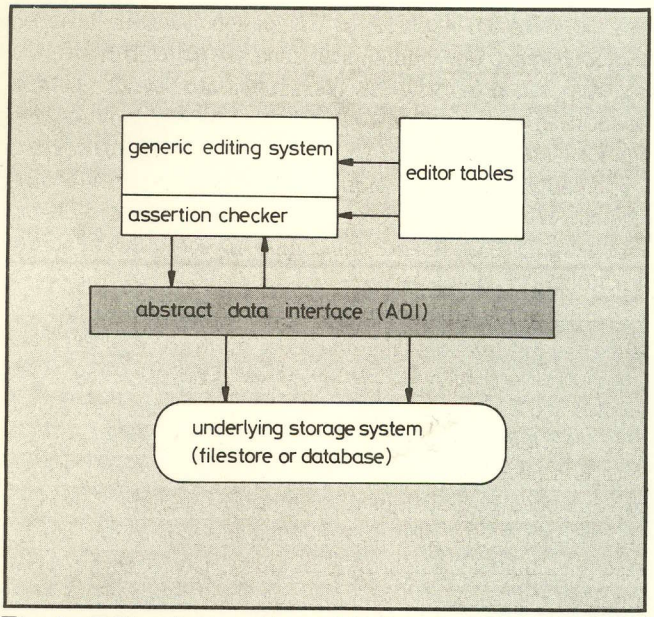

Fig. 6 Editor interaction with the storage system

activity is restricted to having only ports as junctions to paths; windows are not allowed.

In the GDL for MASCOT we define a type hierarchy, part of which is shown in Fig. 4. This allows assertions to be written at various points which can be inherited by lower level nodes in the tree. For example, an ACTIVITY could inherit assertions from TEMPLATE or COMPONENT, as well as having specific assertions of its own. The GDL description for the relevant parts of this hierarchy is given in Fig. 5; note that various compiler directives (e.g. for forward declarations) have been omitted from this GDL description.

Before adding any assertions to the GDL description shown in Fig. 5, we have included both lexical and type checking rules for the design diagram. The **lexical** rules are imposed because a design entity is bound to its symbol type in the **use** clause. The SYMBOL string defines the shape which is associated with this symbol and which is created using a separate tool called the SHAPES editor. Thus, when the user selects a design entity from a DE menu the correct shape is automatically associated with it. Similarly, labels are bound to the design entities.

Type rules are also imposed by this structure. For example, the definition of ACTIVITY states that only ports may be associated with it. The DE will only provide the user with PORT as a possible dependent node to be added to a selected ACTIVITY. Similarly, a PATH can only join two nodes of type JUNCTION; no other node types are allowed. This rule cannot be fully enforced statically. An error in attempting to illegally connect a PATH from a JUNCTION to some other type of node can only be detected when the user attempts to terminate the path at a node other than one of type JUNCTION.

Having established the basic structure of the GDL types, assertions can now be added to enforce further constraints on the diagram. A common **syntactic** constraint, which occurs in many diagrams, is that the name of a design entity should appear within its symbol; an example of this was shown above (S_name_inside). In MASCOT, the component names of all nodes of type COMPONENT must appear *outside* and below the component's symbol. Therefore, an assertion of the following form is added to the GDL description for MASCOT:

```
assertion C_name_outside (COMPONENT):
  inst i:
  not Encloses (GetLabel (i, SYMBOL),
       GetLabel (i, COMPONENT_NAME))
  and
  Below (GetLabel (i, COMPONENT_NAME)
    GetLabel (i,SYMBOL))
```

This assertion states that, for any instance of a component, its symbol must not enclose the component name string associated with that component, and also that the component name must appear below the component's symbol. Since SUBSYSTEM and ACTIVITY are subtypes of COMPONENT, this assertion is inherited by both of these node types.

A common example of a **semantic** constraint in design diagrams concerns the naming rules for groups of design entities. For example, in MASCOT all the junctions owned by a particular instance of a TEMPLATE must have different names, i.e. the JUNCTION_NAME strings must all be different. There are a number of ways of expressing this rule in GDL; one possibility is shown below:

```
assertion J_name_unique (JUNCTION):
  inst i:
  forall j; Member (Dependents (Parent(i)), j)
  and GetType (j) = JUNCTION
  and i /= j:
  GetText (GetLabel (i, JUNCTION_NAME)) /=
  GetText (GetLabel (j, JUNCTION_NAME))
```

This assertion is applied to an instance of a JUNCTION. The assertion specifies that all the siblings of this instance which are also of type JUNCTION should have different names. The qualifier **inst** gives a handle on the current

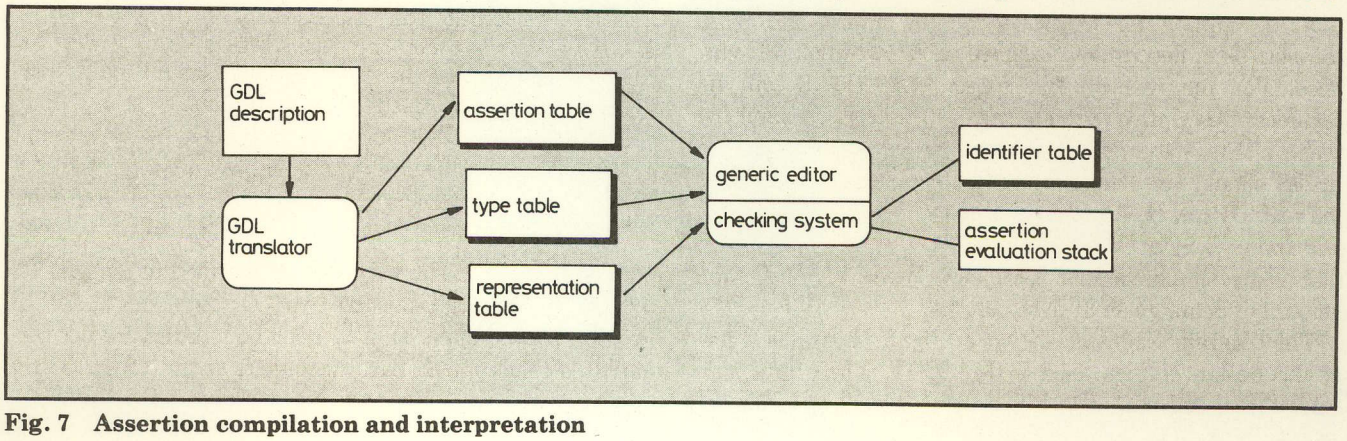

Fig. 7 Assertion compilation and interpretation

Identifier	Type	Current value	Pool of possible values
i	variable	N1	N1, N2, N3, N4, L1, L2, L3
source	parameter	N2	N2

Fig. 8 The assertion checker identifier table structure

```
-- Bind the qualifier name i with the current identifier in the identifier table
-- if the bind operation fails then quit with error message
   i, Get_current_entity, Id_table_enter, bind, if_false goto error
-- Get the label named SYMBOL and stack it
   Eval (i), "SYMBOL", GetLabel
-- Get the label named STATE_NAME and stack it
   Eval (i), "STATE_NAME", GetLabel
-- If no name, check is redundant, exit - no error message
   Dup, NULL, EQ, if_true exit
-- Now check that name label is enclosed by symbol. If it is - success
   Encloses, if_true exit
-- if value on stack is false, print an error message
error: "S_name_inside", Print_error_message, exit.
```

Fig. 9 Code for assertion S_name_inside

node and binds the variable i to it. In the next line of the assertion the function 'Parent' is used to get a handle on the parent of the current instance, and the function 'Dependents' then returns the set of all dependent nodes of this parent node. The variable j iterates over this set of dependent nodes selecting those of type JUNCTION which are not the current instance and checking the values of JUNCTION_NAME.

In MASCOT, there are different constraints on the path-junction connectivity, depending on the type of junction and the type of node with which it is associated. The rule for a PORT owned by a node of type COMPONENT is that there may be exactly one path, either in or out of the port. This rule can be expressed by the following assertion:

assertion Component_ports (PORT):
  inst i; Parent(i) = COMPONENT:
  (Size_of (in_paths) = 1 **and** Size_of (out-paths) = 0)
  or
  (Size_of (in.paths) = 0 **and** Size_of (out_paths) = 1)

Initially, the assertion ensures that the current instance is owned by a node of type COMPONENT. If it is, then there must be either exactly one path into the port or one path out of the port.

This assertion obviously cannot be evaluated as soon as a PORT is added to a MASCOT diagram, otherwise the user will always receive a spurious error message. This is an example of a semantic check which will only be evaluated when the user wishes to initiate checking of the diagram. At this point, if the PORT is incorrectly connected, it can be highlighted and the user can correct the error.

The above assertion (Component_ports) illustrates a problem with classifying when assertions should be evaluated. Ideally, this assertion should be used in two ways: the total number of connected paths should never exceed one, which is a constraint which could be applied whenever a path is connected to an instance of a PORT; and when the diagram is complete the number of paths should be exactly one, since a set size of 0 is not allowed.

There are other completeness checks which are carried out implicitly when the user initiates design checking. For example, in the description of the SUBSYSTEM node type the following appeared:

for SUBSYSTEM **use** SYMBOL (MASCOT.subsystem)
  ++ TEMPLATE_NAME (STRING)
  ++ COMPONENT_NAME (STRING)

The names TEMPLATE_NAME and COMPONENT_NAME are mandatory for all instances of a SUBSYSTEM (Optional names can be specified using square brackets around the definition of the name). Therefore, when the user initiates design checking every SUBSYSTEM instance must be checked to ensure that it has both a TEMPLATE_NAME and a COMPONENT_NAME.

Wherever possible, the design editor attempts to impose the method rules statically by restricting the menu options available to the user. Thus, if an entity can have a single label called NAME, the editor label menu only offers the opportunity of placing that label. Moreover, once a NAME label has been added to an instance of the entity, the menu should not offer it as an option again when that instance is subsequently selected.

However, there are some types of checks which can only be carried out dynamically during the execution of a design editing session. Dynamic checks fall into three classes:

- **immediate** — these should be carried out immediately an editing operation terminates. For example, if an attempt is made to link nodes which should not be linked or a label is placed in the wrong position, the user should be informed of this type of error before doing anything else. Some immediate checks, such as linking nodes illegally, can be enforced simply by refusing to terminate an operation if its termination would cause an assertion to be violated.
- **entity** — these are checks applied to a single node, its labels and its links (or a link and its labels) when all of

these are considered as a single entity. For example, a check may specify some constraint on the number of the links entering or leaving a node.

□ **design** — these apply across the whole design. For example, an assertion may specify that not more than N instances of a particular type should appear on one diagram.

It should be noted that the enforcement of static and immediate checks, as discussed above, is to prevent the user from carrying out operations which are illegal and would lead to a nonsensical diagram. The DE provides a facility for adding free annotations to a design diagram, and this allows the user to put in comments which might otherwise be rejected as illegal.

6 System implementation

In implementing the checking system in the context of the design editor, we were faced with the problem of translating a static GDL assertion into a dynamic check which was initiated appropriately during the editing session. In tools such as MacCadd [9] or Analyst [8] the checking problem was handled by representing both the design and the rules to be checked as a set of Prolog clauses. Checking then was simply carried out by the Prolog system.

We considered this implementation option but rejected it principally because of the need to interface the design editing system to a number of data storage systems, such as the UNIX filestore and the ECLIPSE database. This precluded representing the design in Prolog, and the design representation which was adopted relied on defining an abstract design model and representing this as a set of abstract data types. These types were then interfaced to the underlying data storage system (Fig. 6).

Given the existence of this abstract design representation, the logical implementation of checking was to implement a checking interpreter which acted on this representation and which used assertion information to ensure its consistency. Assertions are thus compiled into reverse Polish form, and the 'compiled code' is held in an editor table, ordered by GDL type. When an editing operation is completed, the appropriate assertions for the types of entity manipulated are interpreted and feedback supplied to the editor user (Fig. 7). Reverse Polish representation was chosen because it is straightforward to convert expressions to this form, and because stack-based interpreters for such a notation are easy to write.

In designing the checking interpreter, the principal problem to be tackled was the design of a mechanism to apply a check to multiple design entities when these were referenced in an assertion. For example, a common form of

```
Qualifier_evaluation
  -- Initialise check counter
  Num_checks := 0
  -- Enter all design entities in identifier table
  Qualified name := 1st design entity from identifier table
  -- The name has been associated with a design entity
  -- now check if that entity should be selected for checking
  -- this involves evaluating the selector expression
  loop
    if Evaluate (<Selector expression>) then
      -- Count checks made - if zero it means design is incomplete
      Num_checks := Num_checks + 1
      -- Entity is a valid selection, does it satisfy assertion
    if Evaluate (<Check expression>) then
      -- Assertion satisfied
      -- if exists qualifier quit evaluation - success!
      exit when qualifier = 'exists'
    elsif
      -- Assertion fails. If forall qualifier do no more checking
      if qualifier = 'forall' then
        Print_error_message (<assertion name>)
        exit
      end if
    end if
    -- if this point is reached, it means that the assertion has not been
    -- completely satisfied and more entities (if they exist) must be checked
    if all associated entities in identifier table have not been processed then
      name := next entity from identifier table
    else
      -- Everything checked and still no success - assertion fails
      -- If no checks made, don't give redundant message
      if Num_checks > 0 then
        Print_error_message (<assertion name>)
      end if;
      exit
    end if
  end loop
```

Fig. 10 Qualifier evaluation algorithm

```
-- Initialise check counter, @ indicates a variable reference
   0, @ Num_checks, :=
-- Bind the qualifier name i with the current identifier in the identifier table
   i, Get_current_entity, Id_table_enter, bind, if_false goto error
-- Associate all design entities with the name j
   j, Get_all_entities, Id_table_enter, bind
-- If all entities have been processed and we get here, failure
exists: if_false goto error
-- Check selection. If it fails, select the next entity
   Eval (i), toStates, Eval (j), Member
   if_false goto next
-- Increment check counter
   Num_checks, 1, +, @ Num_checks, :=
-- Now check assertion on bound values. If it's true, success
   Eval (i), Eval (j), Destination, NEQ
   if_true exit
-- The assertion has failed but qualifier is exists so go on to check next value
-- The bind operation fails if all design entities have been processed
next: j, Id_table_next, bind,
   goto exists
-- Don't print error message if no checks done
error: Num_checks, 0, >, if_false exit
   "Properly connected", Print_error_message, exit.
```

Fig. 11 Code for assertion Properly_connected

assertion selected entities to be checked by specifying a selector expression which these entities satisfy, and, typically, these entities are individually referenced in an assertion by a local identifier such as i or j. We had to make sure that our interpreter applied the check to all selected entities.

In Fig. 7, the checking interpreter is shown to work in conjunction with an identifier table. This identifier table is the means by which names in GDL assertions are bound to design entities. The identifier table has the form shown in Fig. 8.

The effect of the qualifier is to fill in the pool of values which an identifier may take, and values are taken from this pool and assigned to the identifier name in turn. In Fig. 8 this is illustrated by the variable i, which can be bound to a value from N1, N2, N3, N4, L1, L2, L3, and which has a current value N1. The pool of values which can be bound with the parameter source consists of a single value N2.

The pool of values selected by the qualifier **inst** is simply the current design entity. The pool of values selected by the qualifiers **exists** and **forall** includes all entities (nodes and links) in the current design. As the design representation maintains sequences of all design nodes and all design links, it is straightforward to include all design entities in a pool of values.

As a simple illustration of the reverse Polish code and the checking process, consider the assertion on a finite state machine which specifies that the name of a state must lie inside that state.

assertion S_name_inside (STATE):
  inst i:
  Encloses (GetLabel (i, SYMBOL),
    GetLabel (i, STATE_NAME))

This assertion translates into the reverse Polish code shown in Fig. 9. Comments (starting with the symbol --) are included to explain the interpreter actions. Individual commented sequences of instructions are presented on the same line.

Assertions which contain the **forall** and **exists** qualifiers obviously involve more complex code, as they involve iterating through the design entities in the identifier table which may be associated with a qualified name. Rather than describe those with the reverse Polish code, which is a lengthy process, a general evaluation algorithm for these qualifiers is shown in Fig. 10.

To show how this algorithm is realised as reverse Polish code, consider the assertion associated with finite state machines which stated that, for an FSM to be properly connected, a state may not simply be connected to itself.

assertion Properly_connected (STATE):
  inst i:
  exists j; Member (toStates, j):
  Destination (j) /= i

The reverse Polish code for this assertion is shown, with comments, in Fig. 11.

Clearly, when a number of qualified names are used in an assertion the generated code is complex and lengthy, and we do not believe it useful to present a more detailed example here. Rather, we describe the evaluation of one of the MASCOT method assertions which is concerned with checking name uniqueness (Fig. 12).

The qualifiers in this expression are **inst** i, which simply causes the value of the current design entity to be entered into the identifier table and associated with the name i, and **forall** j, which causes all entities in the current design to be entered in the identifier table. The checker then cycles through each of these entities associating its value with j in turn.

The selector expression

Member (Dependents (Parent (i)), j)
  and
  GetType (j) = JUNCTION **and** i /= j:

discovers the 'Parent' node of the current instance, and then, from this, finds all its dependents. The current design entity

```
assertion J_name_unique (JUNCTION)
  -- the selection expression associated with the inst qualifier is implicitly true
  inst i:
  -- the selection expression associated with the forall qualifier selects those
  -- entities in the design which are siblings of type JUNCTION whose
  -- parent is the current design entity
  forall j; Member (Dependents (Parent (i)), j) and GetType (j) = JUNCTION
    and i /= j:
  -- the check expression says that siblings should not have the same name
  GetText (GetLabel (i, JUNCTION_NAME)) /=
  GetText (GetLabel (j, JUNCTION_NAME))
```

Fig. 12 Name uniqueness assertion

bound to j is then compared against the set of dependents, and, if it is a member of that set, its type is checked. If its type is JUNCTION and it is not the 'current' node, the selection criteria are satisfied. Once a particular value has passed through the selection, the check expression

```
GetText (GetLabel (i, JUNCTION_NAME)) /=
GetText (GetLabel (j, JUNCTION_NAME))
```

is applied.

Evaluation continues in this way until all of the entities in the identifier table have been bound to a name and checked.

7 Check classification

In the previous Section, we introduced the notion of classifying checks into implicit checks where the choices available to the user were restricted; immediate checks which were checks made after every editing operation; entity checks which apply when an entity representation is complete; and design checks which were checks across a number of design entities. In implementing the prototype version of the system, we decided to support only implicit and immediate checks. The reasons for this were

- it simplified the implementation;
- it allowed us to avoid problems of check classification;
- it ensured that checking was always 'fail-safe' and that errors were notified to the user at the earliest possible moment.

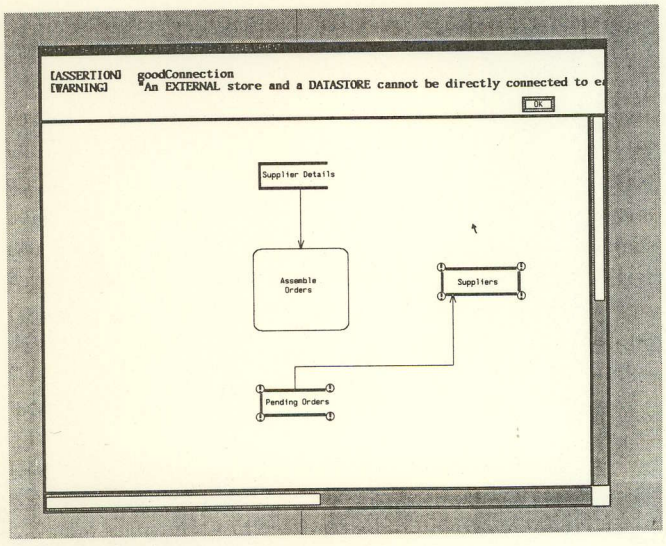

Fig. 13 Checking in the prototype DE

Screen dump provided

Apart from the inevitable runtime overhead involved (which we discovered was not overly significant), the decision to support only immediate checks meant that we had to support a system of 3-valued logic where checks could be true, false or 'not applicable at this point'.

For example, consider the assertion on finite state machine diagrams introduced earlier, which stated that the name of a state should be enclosed by the symbol representing that entity.

```
assertion S_name_inside (STATE):
  inst i:
  Encloses (GetLabel (i, SYMBOL),
    GetLabel (i, STATE_NAME))
```

This is an assertion on type STATE, and so, along with other assertions on this type, is checked whenever a component operation is carried out. However, when the assertion is evaluated following a create operation, it should not evaluate to false because the user has simply not yet added a name. In the example 'assertion evaluation code' above, this is detected by counting the number of actual checks made. If no checks are made, this means that no design entities satisfy the selector expression. The representation is incomplete, so a redundant error message should not be generated.

Thus, when a component representation is incomplete and the assertion refers to elements which are not present, the value returned by the assertion checker is 'not applicable' rather than 'false'. This is considered to be equivalent to 'true' in the current version of the checking system, but a different value is returned so that the tool maker may provide a more directed form of interaction forcing a user to complete the representation of a design entity.

8 Conclusions and further developments

The current checking system has been implemented in the context of a prototype design editing system interfaced to and integrated with the UNIX filestore. An example of this prototype system in use is shown in Fig. 13.

The diagram being drawn is a data flow diagram (DFD), as described by Gane and Sarson [14]. The user has linked a datastore (Pending Orders) directly to an external entity (Suppliers) which is not permitted. The nodes in error are highlighted, and the DE control panel, which contains all

the editing menus etc., is overlaid by an error message. Therefore, the user is unable to continue diagram construction without acknowledging the error, using the 'OK' button. However, after acknowledgment the user may continue diagram construction without being forced to correct the error immediately.

A production-quality version of the editing system with limited checking facilities has been developed and integrated with the database of the ECLIPSE IPSE. Implementation of an editing system with the complete checking facilities discussed here is scheduled for release in a future version of ECLIPSE.

As discussed above, all checking in the prototype system is immediate, and, although this does not have an excessive effect on system performance, it is clear that it involves some unnecessary overheads. Thus, an obvious development of the system is to add more 'intelligence' to the system so that unnecessary checks are not applied.

This might be done in a number of ways:

☐ When a **forall** or **exists** qualifier is used, all design entities are considered to be candidates for selection. This is logically inefficient, although, in fact, our design representation makes this the simplest approach to take. From a checking point of view, a better approach would be to use the type information associated with nodes and links to select only those entities which could conceivably satisfy the selector condition.

☐ Assertions might be automatically classified by the GDL processor to be either immediate, entity or design assertions. This appears to be possible by comparing the assertion to the representation assertion and by examining the qualifiers used. For example, a check which references the number of links associated with a node should be classed as an entity check and only executed after links have been entered.

The general approach which has been adopted in the editing system is a permissive one. The user is allowed to do as he or she pleases, and, if this is incorrect, an error report is generated. The user may choose to ignore this report. An alternative approach would require the user to maintain design consistency at all times. Thus, if an error was detected by the checking system after some operation, the user might not be permitted to move on to another operation until that error was corrected.

In fact, a possible development of the editing system would be to provide the tool builder with facilities to generate syntax-directed editors where the editor drove the interaction with the user. Thus, when a user placed a node, say, the editor would prompt for its name and associated labels. Hooks to provide these facilities exist in the current system.

In conclusion, we believe that the work reported here demonstrates that design editing systems which maintain the correctness and consistency of a design need not be built individually for each method. By reusing knowledge about design methods and editing functions, it is possible to generate a variety of tailored editing systems quickly and cheaply, which have the additional advantage that the user is always presented with a consistent editing interface.

The prototype version of ECLIPSE included configured versions of the DE for MASCOT 3 and the major diagram types of SSADM. We also configured versions of the DE for the principal JSD diagrams, structure diagrams and a few other simple diagram types for our own experimental use. The commercial version of ECLIPSE includes a DE configured for MASCOT 3 which is fully integrated with the database.

Our industrial collaborators were able to build a demonstrable DE for a new design method called HOOD (Hierarchic Object-Oriented Design) [15] in one afternoon, which illustrates the effectiveness of the system. A commercial version of the ECLIPSE toolset including a fully configured DE and other tools for HOOD is now marketed by IPSYS.

9 Acknowledgments

The work described in this paper was carried out as part of the Alvey ECLIPSE project at the University of Strathclyde. We would like to thank collaborators in that project, Software Sciences Ltd., CAP Group Ltd., Learmonth and Burchett Management Systems Ltd., the Universities of Strathclyde and Lancaster, and the University of Wales at Aberystwyth.

10 References

[1] DE MARCO, T.: 'Structured analysis and system specification' (Yourdon Press, 1978)
[2] YOURDON, E., and CONSTANTINE, L.L.: 'Structured design' (Prentice-Hall, 1979)
[3] JACKSON, M.A.: 'System development' (Prentice-Hall, 1983)
[4] BIRRELL, N.D., and OULD, M.A.: 'A practical handbook for software development' (Cambridge University Press, 1985)
[5] SOMMERVILLE, I., WELLAND, R.C., and BEER, S.: 'Describing software design methodologies', *Comput. J.*, 1987, **30,** (2), pp. 128–133
[6] BEER, S., WELLAND, R., and SOMMERVILLE, I.: 'Software design automation in an IPSE'. Proc. ESEC '87, Strasbourg, France, September 1987
[7] BOTT, M.F.: 'ECLIPSE — an integrated project support environment' (Peter Peregrinus, 1989)
[8] STEPHENS, M., and WHITEHEAD, K.: 'The Analyst — a workstation for analysis and design'. Proc. 8th Int. Conf. on Software Engineering, London, UK, August 1985
[9] JONES, J.: 'MacCadd — an enabling software method support tool'. Proc. 2nd Conf. of British Computer Society Human Interaction Specialist Group, Cambridge, UK, September 1986
[10] HEKMATPOUR, S., and WOODMAN, M.: 'Formal specification of graphical notations and graphical software tools'. Proc. ESEC '87, Strasbourg, France, September 1987
[11] CUTTS, G.: 'SSADM — structured systems analysis and design methodology' (Paradigm, 1987)
[12] MASCOT Special Edition of *Softw. Eng. J.*, 1986, **1,** (3)
[13] JIMCOM: 'The official handbook of MASCOT: Version 3.1'. RSRE Malvern, June 1987
[14] GANE, C., and SARSON, T.: 'Structured systems analysis: tools and techniques' (Prentice-Hall, 1979)
[15] 'HOOD Manual: Issue 2.2'. ESTEC, Noordwijk, Netherlands, April 1988

Paper received on 11th November 1988 and in revised form on 11th September 1989.

R.C. Welland is with the Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK; S.J. Beer is with Oracle Corporation (UK), Oracle Park, Bittams Lane, Guildford Road, Chertsey, Surrey KT16 9RG, UK; and Professor I. Sommerville is with the Department of Computing, University of Lancaster LA1 4YR, UK.