Kenneth J. Turner and Ashley McClenaghan. Visual animation of LOTOS using SOLVE (extended version). In Dieter Hogrefe and Stefan Leue, editors, Formal Description Techniques VII, Amsterdam, October 1994. North-Holland.

Visual Animation of Formal Requirements

K. J. Turner*, A. McClenaghan†

1st June 1999

SOLVE (Specification using an Object-based, LOTOS-defined, Visual language) is designed to allow formal requirements capture, particularly for interactive systems. The SOLVE language is object-based, and formally defined using LOTOS (Language Of Temporal Ordering Specification). SOLVE is also a set of software tools that allow direct visual animation of systems specified in this language. Communicating objects control on-screen icons that can be manipulated directly by the user. Animation is supported by translating a SOLVE specification automatically into a LOTOS specification, and then simulating this using standard LOTOS tools. A VCR (Video Cassette Recorder) clock controller is used to illustrate the SOLVE approach. A further application is embodied in the XDILL tool that supports requirements specification and animation of digital logic circuits. The architecture of the SOLVE toolset is described.

1 Introduction

1.1 The SPLICE Project

Requirements capture, analysis and specification are difficult yet extremely important parts of system development. Errors in the top-level specification have a major impact on later refinements. Significant work has been undertaken on most aspects of using LOTOS (Language Of Temporal Ordering Specification [13]) throughout the software engineering life-cycle. However, one phase that has received little attention to date has been the use of LOTOS in requirements capture. This was the background to the project SPLICE I

*Department of Computing Science, University of Stirling, Stirling FK9 4LA, UK (kjt@cs.stir.ac.uk)

†Philips Research Labs., Redhill, Surrey RH1 5HA, UK (ukrmccl@prl.philips.co.uk)

(Specification and Prototyping for a LOTOS Interactive Customer Environment – Phase I), called SPLICE for brevity in the following.

SPLICE explored the conviction that requirements capture and specification would benefit from the use of greater formality – specifically LOTOS. However, it was recognised that the use of a formal language in the requirements phase creates potential problems. Customers and clients are generally not trained in formal methods, and find it hard to understand and relate to formal specifications of requirements. Designers and programmers are more likely to have training in formal methods – at least to the point of being able to read formal specifications – but this cannot be assumed of every member in a development team. Analysts have to bridge the gap between the expectations and background of customers/clients and designers/programmers.

Analysts are also faced with another problem: how to structure and represent requirements, ideally in a formal way. This is essentially an architectural problem, and experience shows that developing a good system architecture is difficult. Another challenge is representing architectural concepts appropriately in a chosen formal language. It is frequently the case that many alternative formal representations are possible for architectural concepts, such that only widespread experience can show the most effective ways of modelling them (e.g. see [24] for an approach to modelling the OSI architecture).

The common thread throughout SPLICE was therefore developing an effective bridge between customers/clients, analysts and designers/programmers. An equally important bridge had to be developed between system requirements, system architecture and formal representations.

Traditional software engineering methods for requirements capture tend to be informal and may not be supported by tools. The SPLICE philosophy was to have the tool user indirectly manipulate the formal representation of the requirements via a familiar interface (e.g. a visual representation of the problem). In this way the user can produce, analyse and interact with formal requirements without having to learn an unfamiliar formal notation.

The objective of SPLICE was to develop methods and prototype software tools to support the use of LOTOS for requirements capture, analysis and specification in a number of selected application domains. The aim was to make the benefits of formality accessible to non-formalists. Visual animation of LOTOS was one of the two major themes in SPLICE; other research was undertaken on formal (LOTOS-defined) requirements capture using object-based methods.

SPLICE has looked at four distinct application domains: OSI services, digital logic specification, neural networks and interactive systems. [21] discusses approaches to architectural specification of OSI services and digital logic; the latter is further elaborated in [23]. Both approaches are supported by tools using the

m4 macro language [14, 22]. [9] reports complementary work on specification of neural networks. Tool support has been developed for this using the Sather language and the SunView windowing environment. The present paper reports on SPLICE work to support specification and prototyping of interactive systems using LOTOS.

1.2 Related Work

Commercially available tools to capture requirements for interactive systems tend to be sophisticated graphics editors. These systems help a user to collect and organise information through visual attributes of the system. However the descriptions that these tools produce are often inadequate models of the behavioural requirements, cannot be used as executable prototypes, and lack the rigour needed for testing and refinement. The production of tractable specifications from requirements is a desirable precursor to formal development. Moreover, in a world ever more cluttered by 'push-button' interactive devices, the importance of generating analysable models during requirements capture becomes particularly significant. [20] demonstrates how building formal models of interactive systems leads to early problem identification and better designs.

The QUICK system [5] inspired and shaped the work reported here. QUICK (Quick User Interface Construction Kit) is a toolkit that allows non-programmers to construct and explore graphical interfaces by direct manipulation. SOLVE also uses graphical presentation and manipulation to convey the meaning of a specification. Unlike QUICK, SOLVE's primary concern is to deal with formal specifications. Other related work includes XIT [11] and STATEMENT [10].

SOLVE specifications can be automatically translated into LOTOS specifications. As will be seen, SOLVE is object-based. However, the aim was to generate standard LOTOS so no attempt was made to introduce object-orientation into the translated specification. The approach thus offers a contrast to work on defining object-oriented LOTOS variants and methods (e.g. [1, 3, 8, 16, 18]).

The object basis of SOLVE confers a natural style of modelling that is appropriate for requirements specifications to be evaluated by customers/clients. It was not an aim to incorporate the full paraphernalia of an object-oriented language and method such as SMALLTALK, C++ or EIFFEL. The advantages claimed for SOLVE stem from its formal basis and animation possibilities rather than its use of objects.

2 The SOLVE Approach

2.1 The Goals of SOLVE

The approach taken to visual animation of LOTOS specifications is called SOLVE (Specification using an Object-based, LOTOS-defined, Visual language). The key concepts in SOLVE are formal specification (via LOTOS), interactive animation and object-based modelling. SOLVE is a language for specifying and animating (prototyping) the requirements of interactive systems. These include human-oriented devices such as VCRs (Video Cassette Recorders) and other domestic appliances. SOLVE is backed up by software tools running under the X window environment for manipulating and animating specifications written in the SOLVE language.

SOLVE is designed to be used by people who are not familiar with formal languages (in particular LOTOS). SOLVE is a system for building formal requirements and for exploring these specifications using interactive animation. The challenge faced by SOLVE is helping users to explore the consequences of formal specifications of requirements. SOLVE meets this challenge by allowing a requirements specification to be written in a straightforward object-based language that can be automatically translated into LOTOS and then visually animated.

The sense in which SOLVE is based on LOTOS is that it has a straightforward denotation in terms of LOTOS; the automatic translation of LOTOS to SOLVE embodies this denotation. As a result, use of SOLVE confers the same benefits as using LOTOS: precision, analysability, equivalences, tools, etc. The equivalence of two SOLVE specifications is determined by the equivalence of their LOTOS denotations. Nonetheless, new relationships might be defined for SOLVE specifications only (e.g. for subclassing). Such relationships would be defined using the underlying LOTOS semantics (e.g. the *cred* and *cext* relations of LOTOS). SOLVE aims to gain the advantage of formality at the price of hiding LOTOS — from non-specialists at least. A major gain is that once the requirements specification in SOLVE has been checked, the corresponding LOTOS specification can be used as the basis for further formal development. This might include use of refinement, correctness-preserving transformation, and formal derivation of tests.

The sense in which SOLVE is object-based is discussed in Section 2.3. It was decided not to build inheritance into the language but rather into the environment. That is, SOLVE specifications would indicate where classes were to be imported. The environment would maintain libraries of classes and their inheritance relationships. The result is a simpler specification language at the expense of a more complex support environment. The SOLVE environment discussed in this paper does not include inheritance mechanisms, so

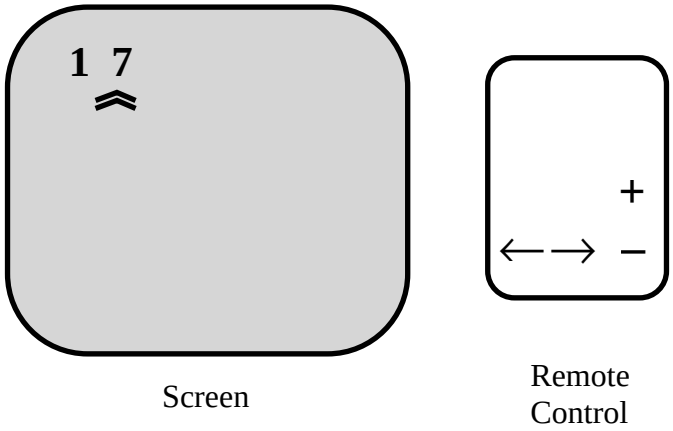

Fig. 1 VCR Clock System

SOLVE presently lacks full object orientation.

The sense in which SOLVE is visual is discussed in more detail later. Once a SOLVE specification has been translated into LOTOS, it can be visually animated. This allows requirements to be investigated by interacting with the specification directly (e.g. by clicking or dragging object icons). The object-based approach of SOLVE leads to animation of requirements in a user-friendly way.

2.2 Applications of SOLVE

SOLVE is suitable for requirements capture in design domains where systems are interactive (producing feedback in response to user input) and can be represented by visual animation. SOLVE has been applied to a number of simple systems including a VCR on-screen clock controller, a database access mechanism, a light switch, and a message passing communications protocol. The definition and analysis of digital logic circuits has been investigated using XDILL — a development of SOLVE for this application domain. A VCR clock and a D-Latch, a simple one-bit memory, are used as examples later in the paper.

The VCR clock allows the user to set an on-screen 24-hour clock using on-screen cursor and control buttons. The cursor can be moved left or right to select one digit, but cannot be moved beyond the digits. For simplicity, the clock shows hours only, represented as on-screen digits for tens and units of hours. Increment and decrement buttons allow the clock digit pointed to by the cursor to be adjusted by one, except that the clock cannot be adjusted outside the range 00 to 24 inclusive. Tens and units of hours are related (e.g. incrementing 09 yields 10). Fig. 1 shows the appearance of the screen while the clock is being set.

2.3 Objects in SOLVE

The SOLVE notion of objects reflects several basic aspects of the object-oriented paradigm. An object is an autonomous entity with well-defined interface methods, and communicates with other objects via blocking or non-blocking message passing. An object can decide its future behaviour dependent on internal values and communication with other objects. A simple object has an icon — a visual representation of the object. Interaction with the environment is also in terms of message passing, e.g. the user clicking on an icon causes a message to be sent to the appropriate method for the object responsible for the icon. A composite object is a set of interworking objects, whether simple or composite.

An important feature of SOLVE is visualisation. Each object is visualised as an icon — in fact a bitmap displayed on the screen. An object is responsible for displaying and modifying its own object icon, representing an abstraction of the state of an object or some part of the total system. The object icons visually inform the SOLVE user what is happening in the system under design.

Object-orientation sometimes supports class or type-based objects [2] as opposed to the prototypical objects [5] supported by SOLVE. (The distinction between these two approaches is clearest when looking at how objects are coded.) The key idea of class-based inheritance is top-down specialisation, whereas the key idea of prototypical objects is bottom-up composition of objects from simpler objects. Each simple SOLVE object is created with all the characteristics of a basic prototypical object. Once an object has been created it may be customised. In fact the user may build up a kit of predefined simple and composite objects. [5] points out that prototypical objects support the two often quoted advantages of inheritance: abstraction and reuse. Prototypical objects may be aggregated, and the aggregate labelled to form a new abstract 'class' or 'type'. Objects may be duplicated, thus supporting reuse.

A SOLVE specification consists of a number of objects that communicate via messages. A message either invokes an object method or returns the results of an invoked method. Objects may execute concurrently. One of the objects in an executing SOLVE specification is called *Interface*. This object is implicit — it is declared and defined automatically by SOLVE. In contrast, all the other objects in a SOLVE specification have to be declared and defined explicitly by the user.

Fig. 2 shows part of the VCR clock system. Objects *GoLeft*, *GoRight* and *Cursor* are declared explicitly in the SOLVE specification. Potential message communication paths are shown as dotted lines in Fig. 2, meaning that all objects may communicate with one another. The user interacts with the animation of a SOLVE specification by communicating with the *Interface* object. *Interface* handles the screen window

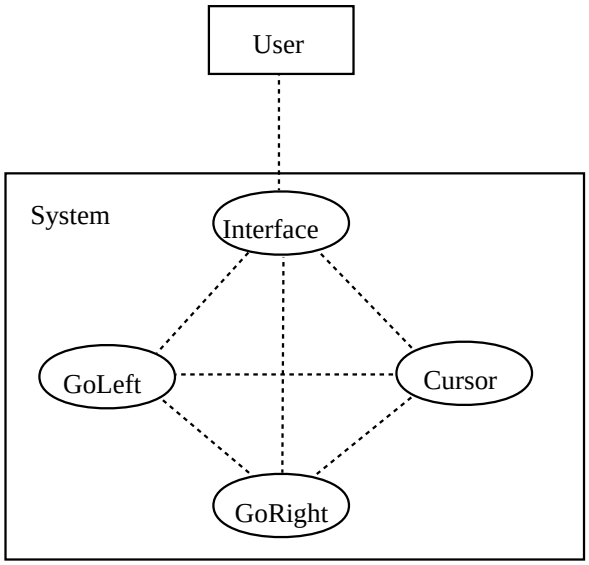

Fig. 2 Communicating Objects in a SOLVE Specification

system, and so is responsible for displaying object icons and for accepting mouse click and drag requests from the user. In an actual interaction the user might click on the *GoLeft* button icon. The mouse click would be handled by a method within *Interface*, which would send an *IconClicked* message to *GoLeft*. A method in *GoLeft* would then send a *Left* message to *Cursor*. *Cursor* in turn would now send a *SetIcon* message to *Interface*, requesting the *Interface* object to redisplay the *Cursor* icon one place to the left. Responding to this request, *Interface* would instruct the window system to display the *Cursor* icon in its new position.

Object declarations provide sort ('type') information used to check correct use of objects with definitions. Each object has a list of instance parameters and a set of methods. The sorts of instance parameters are declared, as are the names and signature of the methods.

Reflecting the visual nature of SOLVE, each object has three implicitly declared instance parameters called *xPos*, *yPos* and *iconPic*, all of sort *Int*. These three parameters are used to represent the object's *(x,y)* co-ordinates and the bitmap picture of its icon. Each object also has three implicitly declared methods called *Initialize* (initial values assigned and icon set up), *IconClicked* (mouse click on icon) and *IconMoveRequest* (mouse drag on icon). Although these methods are implicitly declared, an explicit definition of each must be given by the specifier.

2.4 SOLVE Language Elements

The SOLVE language is object-based and hopefully intuitive. It has been deliberately designed to resemble a programming language, avoiding the algebraic feel of LOTOS. However, SOLVE descriptions can be automatically translated into LOTOS specifications. Some of the mechanics of SOLVE are automatically incorporated in the translation. This makes the LOTOS specification SOLVE-oriented, but makes it suitable for visual animation and for subsequent object-based development.

Although objects may execute concurrently, behaviour within a simple object is sequential. This means that an object may only be executing one method at any time. The method definitions may use any of the following SOLVE language statements:

-- introduces comment text up to the end of line.

Variables *Name : Sort, ...* **EndVariables** allows local variables to be declared with the lifetime of the enclosing method definition, and with a scope confined to the following statements in the method definition.

Assign *(Variable, Value)* binds a variable to a value. The SOLVE language supports sorts *Int* (integer) and *Bool* (boolean) with the usual values and operations. SOLVE pre-defines the four *Int* constants *XMIN*, *XMAX*, *YMIN* and *YMAX* as the bounding coordinates of the visual display.

If *Condition* **Then** *Statements* **Else** *Statements* **EndIf** offers simple conditional branching.

While *Condition* **Do** *Statements* **EndWhile** supports a simple loop.

AskWaitCall *ObjectName.MethodName (ParameterList) (ResultsList)* is used to invoke a particular object with a particular method and a list of parameters. **AskWaitCall** blocks execution in the invoking object until the call returns to assign values to the variables in the results list.

TellCall *ObjectName.MethodName (ParameterList)* is used like **AskWaitCall** except that it does not block execution in the invoking object. Any parameters returned by the invoked method are ignored.

3 SOLVE in Action: A Video Cassette Recorder Clock

3.1 The SOLVE Model

The VRC clock will be used as an example of how SOLVE can be used to capture and visually animate requirements. The mapping between informal requirements and the SOLVE specification is straightforward. Table 1 shows the objects, attributes and methods that can be identified from the informal requirements. Only aspects directly relevant to requirements (and not just artifacts of the SOLVE specification) are shown here.

The manual operation of any of the four push-button objects is represented by the user clicking on its icon. As an example of interactive animation, consider what happens when the user wishes to check the effect of moving the cursor right. The user would click the *GoRight* icon, invoking the *IconClicked* method of *GoRight* and causing a *Right* message to be sent to *Cursor*. This may respond by moving its icon to the right depending on the current position; if it is already under the units digit then it will not move any further right. The visually animated SOLVE specification thus reacts to direct manipulation by the user via the graphical interface.

3.2 The SOLVE Specification

An outline of the SOLVE specification is as follows; ellipses mark where text has been omitted for brevity. Objects, attributes and methods correspond closely to the analysis given in Fig. 1. The full SOLVE specification is about 160 non-comment lines and is given in [17].

```
System VCRclock Is                          -- system name

  PictureDeclarations                       -- icon declarations
    leftArrow, rightArrow, ...              -- arrow icon filenames
    digitZero, digitOne, ...                -- digit icon filenames
    cursorPtr                               -- cursor icon filename
  EndPictureDeclarations

  ObjectDeclarations                        -- object interfaces

    Object GoLeft() Is                      -- left button object
    EndObject
```

```
  Object GoRight() Is                                  −− right button object
  EndObject

  Object Cursor(Bool) Is                               −− cursor display object
    QueryXPos()(Int)                                   −− check cursor x position method
    Left()()                                           −− move cursor left method
    Right()()                                          −− move cursor right method
  EndObject

  Object Tens() Is                                     −− tens display object
    Inc()()                                            −− increment method
    Dec()()                                            −− decrement method
    QueryValue()(Int)                                  −− check value method
  EndObject

  Object Units() Is                                    −− units display object
    Inc()()                                            −− increment method
    Dec()()                                            −− decrement method
  EndObject

EndObjectDeclarations

ObjectDefinitions                                      −− object internals

  Object GoLeft() Is                                   −− left button object
    Method Initialize() Is                             −− initialisation method
      Assign(xPos,6)                                   −− initialise x position
      Assign(yPos,5)                                   −− initialise y position
      Assign(iconPic,leftArrow)                        −− initialise icon image
      TellCall Interface.SetIcon(xPos,yPos,iconPic)    −− asynchronous call to display icon
      Return()
    EndMethod
```

```
  Method IconClicked() Is                          −− icon clicked method
    TellCall Cursor.Left()                         −− asynchronous call to move left
    Return()
  EndMethod
  Method IconMoveRequest(Int:a,Int:b) Is           −− icon move requested method
    Return()                                       −− icon move ignored
  EndMethod
EndObject

Object GoRight() Is ... EndObject                  −− right button object

Object Cursor(Bool:flashingOn) Is                  −− cursor display object
  Method Initialize() Is
    Assign(xPos,2)                                 −− initialise x position
    Assign(yPos,2)                                 −− initialise y position
    Assign(iconPic,cursorPtr)                      −− initialise icon image
    TellCall Interface.SetIcon(xPos,yPos,iconPic)  −− asynchronous call to display icon
    Return()
  EndMethod
  Method IconClicked() Is
    Return()                                       −− icon click ignored
  EndMethod
  Method IconMoveRequest(Int:newXPos,Int:newYPos) Is ... EndMethod
  Method QueryXPos() Is                            −− return cursor x position
    Return(xPos)
  EndMethod
  Method Left() Is                                 −− move cursor left
    If (xPos Nei 1)                                −− position not 1 (i.e. leftmost)?
      Then
        Assign(xPos,xPos Minus 1)                  −− decrement position
        TellCall Interface.SetIcon(xPos,yPos,iconPic) −− asynchronous call to display icon
      Else                                         −− move attempt ignored
    EndIf
```

```
        Return()
      EndMethod
      Method Right() Is ... EndMethod
    EndObject

    Object Tens() Is        ... EndObject        -- tens display object

    Object Units() Is       ... EndObject        -- units display object

  EndObjectDefinitions
```

The icon picture declarations are actually the names of files containing the bitmap images. An object declaration describes the names, instance parameter sorts and method parameter sorts of each object. An object definition describes the inner details of the object. The definition must conform to the object declaration and define the declared methods, including the implicitly declared methods *Initialize*, *IconClicked* and *IconMoveRequest*.

The parentheses after an object name are used to declare the types of the object's instance parameters. The *GoLeft* object has only the three implicitly declared instance parameters: *xPos*, *yPos* and *iconPic*. The *Cursor* object also has one explicitly declared instance parameter of type *Bool*.

The object *GoLeft* has only the implicitly declared methods: *Initialize*, *IconClicked* and *IconMoveRequest*. The object *Cursor* also has three explicitly declared methods: *QueryXPos*, *Left* and *Right*. The two parenthesised lists after each method declaration are used to give the sorts of parameters and results. The method *QueryXPos* has one result of sort *Int*, while methods *Left* and *Right* have neither parameters nor results.

The cursor may be flashing or not. The variable *flashingOn* appears in the instance parameter list for *Cursor*, matching the need for one instance parameter of type *Bool*. All the instance parameters (explicit and implicit) may be referenced or assigned to within the object's methods. An instance parameter maintains its assigned value between method invocations until it is reassigned some other value.

3.3 Animating the Specification

The SOLVE specification of the VCR clock is translated automatically to LOTOS by the *parser* tool. If the specification is syntactically and static semantically correct, the result is a LOTOS specification that can

be animated. Various SOLVE control files are produced as a byproduct of translation. The VCR clock specification in SOLVE is translated into about 1300 non-comment lines of LOTOS – an expansion of roughly eight times, showing the productivity gains possible compared to direct specification in LOTOS.

The analyst will wish to check that the SOLVE specification reflects the informal requirements using the SOLVE toolset. Together the analyst and customer/client may explore and assess the animated behaviour of the specification to establish its correctness and completeness using the *animator* tool. The *animator* displays the state of the system graphically, and allows interaction with it in an intuitive and visual way. More importantly, the animation is intelligible to a customer/client or designer/programmer without knowledge of LOTOS.

When animation begins, the *animator* and *displayer* tools open their windows. The *animator* provides a menu of possible next events. When animation of the VCR clock specification starts, seven internal events are possible. These are **TellCall** invocations of *Initialize* methods in the seven objects to set up their icons. Before these calls occur, *displayer* shows default bitmaps for the object icons at a default location.

The user may choose a view option that shows the event offers from the system and its environment. This results in four windows. The *displayer* shows the graphical view of the system objects, and the events offered by the user via this graphical view. The *animator* shows the events offered by the system via the *hippo* simulator, and the menu of possible events that are acceptable both to the environment and system.

An option in *animator* allows automatic selection of events. Although the user can interact with the specification in a conventional event-by-event simulation, the power of SOLVE lies in the visual animation. Normally the user will set *animator* for automatic selection of events and will then interact directly with the system by mouse clicks and drags on the graphical display.

As an example, consider the animation windows in Fig. 3(a); the windows are numbered from the top in the following description. The user has just clicked the *GoLeft* button in window 2. The result is the event offer *Interface ! IconClicked ! GoLeft* from the system environment, shown in window 4. The *animator* matches this with the events currently offered by the system, shown in window 3, and displays the permissible events in window 1. Only one event is possible, and *animator* selects this since it is in automatic mode. A chain of events is now followed automatically, leading to the situation shown in Fig. 3(b). The environment event window and overall event window are now empty, waiting for the user to invoke another action in the graphical window. The *GoLeft* click has caused the cursor to move one place left to the tens position. Although the operation of all four windows has been discussed, in practice the user may concentrate on the graphical view only.

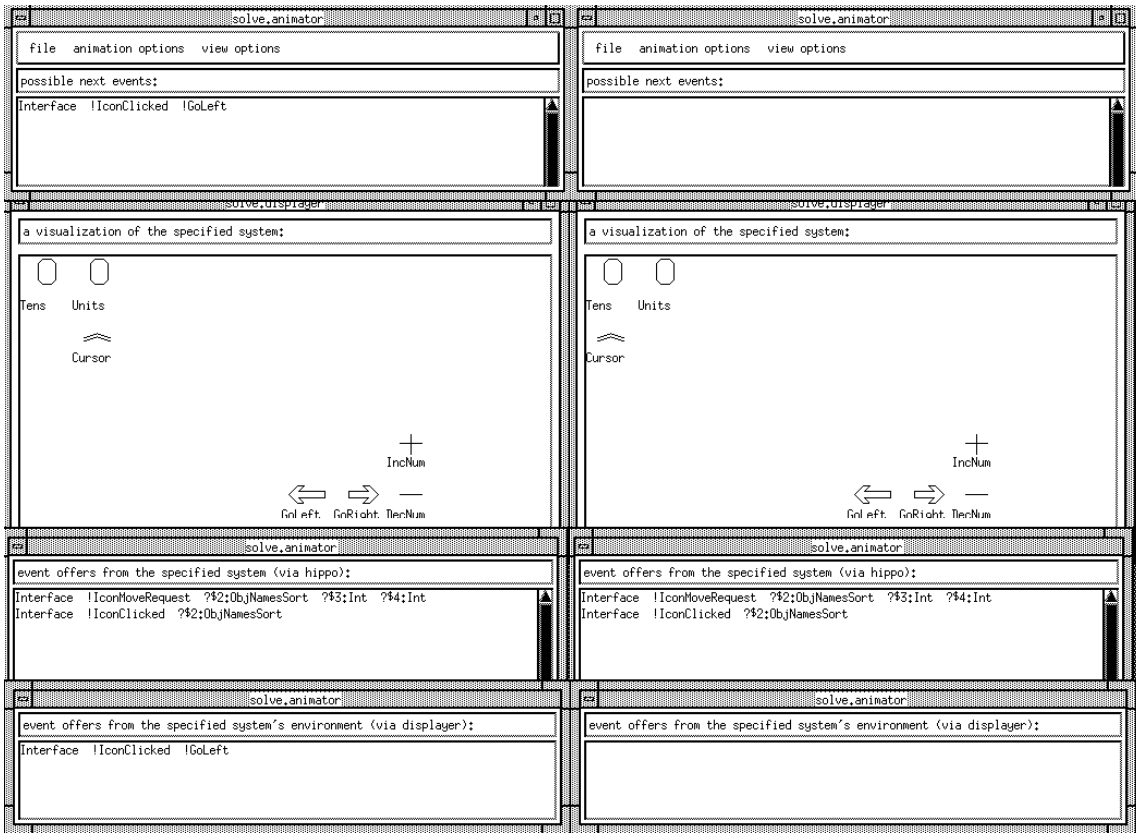

Fig. 3(a) After Clicking *GoLeft* Fig. 3(b) After Processing *GoLeft*

4 SOLVE In Action: A Digital Latch

DILL (Digital Logic in LOTOS) [23] was developed for the specification and validation of digital logic components and circuits using LOTOS. DILL provides a library of pre-defined circuit components — a catalogue of LOTOS specification fragments. A digital circuit specification consists of a number of components combined using a simple macro language and LOTOS operators. A DILL specification is a black-box description of a circuit with input wires and output wires. During animation of a DILL specification, the user may change the logic values on the inputs and observe the resultant logic values on the outputs.

The SOLVE approach has been extended to allow interactive, visual animation of DILL specifications under the X window environment, hence the name XDILL. Given a DILL source file, the *xdill* tool carries out the translation into SOLVE and LOTOS and then animates it. As an example, consider the specification of a D-Latch – a standard hardware memory component, so-called because it latches (stores) one data bit on a clock pulse. A D-Latch has two inputs and two outputs conventionally labelled *D* (input data), *C* (input

clock), *Q* (output) and *Qbar* (output negated). The *xdill* tool needs help to distinguish inputs and outputs: the specifier must begin input gates with ‘i’ and output gates with ‘o’. The DILL specification of a circuit using a single D-Latch is as follows:

```
circuit(                                    # circuit declaration
   ‘DLatch[iD,iC,oQ,oQbar]’,‘               # circuit functionality
       DLatch[iD,iC,oQ,oQbar]               # circuit behaviour
    where DLatch_Decl                       # D-Latch library declaration
’)
```

The *circuit* declaration defines a logic circuit; its parameters are the overall LOTOS functionality and the LOTOS behaviour specification. The conventions of DILL require parameters to be quoted, and comments to be preceded by ‘#’. As usual in LOTOS, subsidiary definitions are introduced following ‘**where**’. A D-Latch is one of the pre-defined library components so it can be declared directly, leading to a rather simple DILL specification.

Invoking *xdill* causes *animator* and *displayer* windows to appear as for SOLVE. When animation of the D-Latch starts, several internal and observable events are possible. The initial internal events correspond to settling down of the circuit. The initial observable events correspond to the first output values of the D-Latch.

The *displayer* window graphically depicts the DILL component as a box. The number and position of the inputs and outputs are automatically calculated. Inputs are placed along the left side of the box depicting the DILL component, and outputs along the right side. Logic values *T* and *F* (True and False) are displayed in arrow-shaped boxes. To the left of each input are two buttons marked *T* and *F* that may be clicked to set a particular logic value.

Animation options for XDILL are the same as for SOLVE. Typically the user chooses automatic selection of events and interacts with the animation via the graphical window. As reported in [23], simulation of digital logic circuits involves substantial numbers of internal events. Selecting these manually is very tedious, so the automatic selection supported by XDILL is a real benefit.

When the user clicks on a *T* or *F* button, it momentarily turns into a zig-zag as shown in Fig. 4(a). In this example, the input data line *iD* has already been set False. The input clock line *iC* has just been set False, so changes are propagated throughout the system. After automatic selection of events the result is the new stable state shown in Fig. 4(b). As expected, the D-Latch has latched the new input value False on the falling edge of the clock pulse; the output and negated output are complementary.

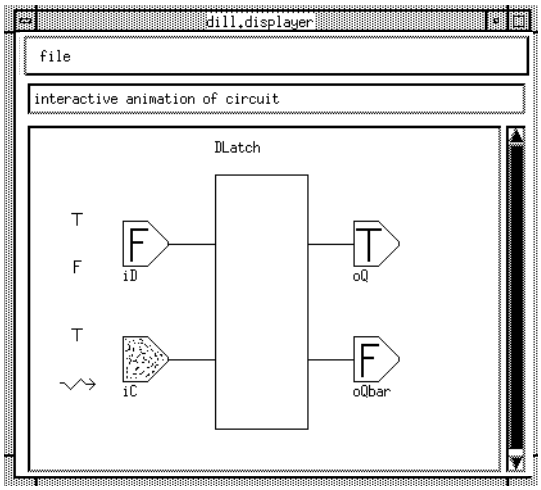

Fig. 4(a) After Falsifying Clock Input

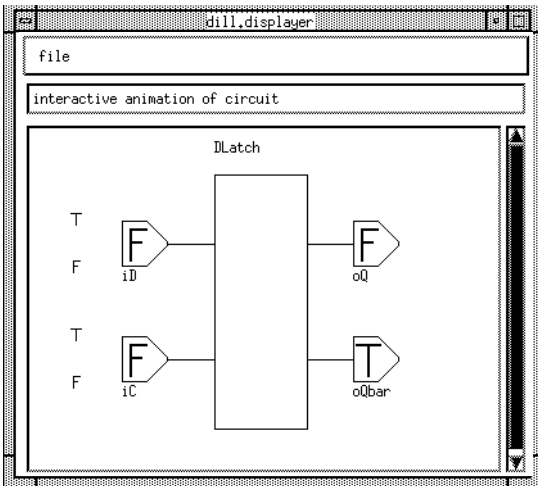

Fig. 4(b) After Processing Clock Input

5 Tool Support for SOLVE

The SOLVE toolset consists of the following main programs: *editor*, *parser*, *displayer*, *animator*, a modified version of the *hippo* simulator [15], *solve* and *xdill*. The tools are built using the X window environment and C, though some of this code is generated by *yacc* and *X-Designer* [12].

5.1 Front-End Tools

SOLVE specifications are purely textual and so can be produced using a standard text editor. However, the syntax-directed editor *syd* [6] was developed with SOLVE in mind. Fig. 5 shows *syd* in use to edit part of the VCR clock specification. *syd* is a novel compromise between a traditional text-editor and a strict syntax-directed editor. Pure syntax-directed editing tends to be rather awkward for the experienced language user, though it can be very helpful for beginners. Syntax-directed editing also tends to be very inconvenient for entry of expressions.

The approach of *syd* is to enforce the syntax of the language down to a specified level. Low-level language elements such as expressions or identifiers can be treated as purely textual: the user may enter them without restriction. The syntactic level at which *syd* operates may be lowered or raised according to requirements. At the highest possible level, *syd* is merely a text editor. In fact, *syd* is completely configurable since it reads the syntax of the language to be edited. The meta-syntax defines the grammar rules of the language in terms of its grammatical elements. Another convenient feature of *syd* is that the meta-syntax may specify the conventional textual layout of the language (e.g. the use of newlines and

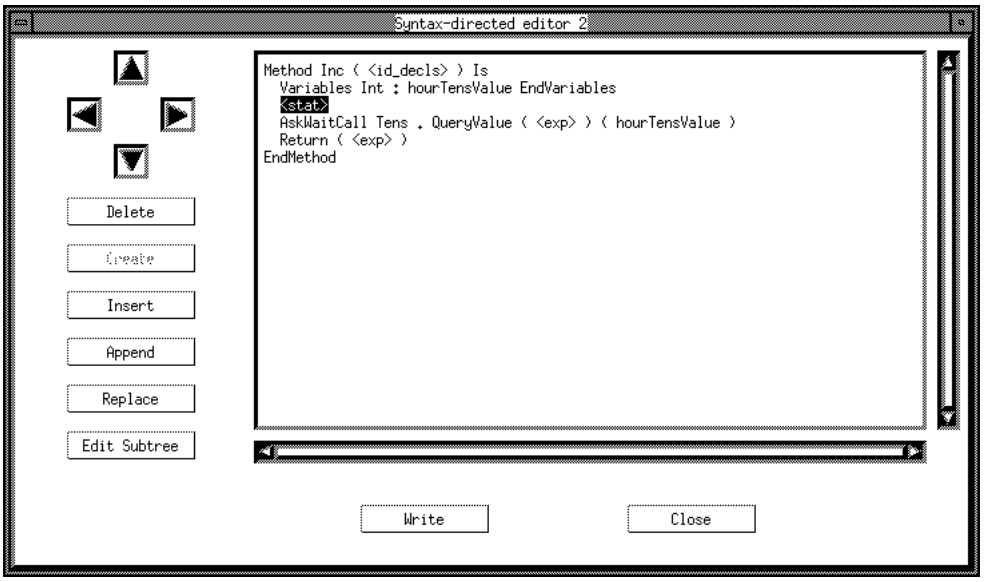

Fig. 5 The *syd* Editor being used on a SOLVE Specification

indentation). Although *syd* was developed for use with SOLVE, it is really a general-purpose syntax-directed editor.

The *parser* tool is built using *yacc*. *parser* accepts a file containing a SOLVE specification, and carries out syntax and static semantics checks. For a valid specification, *parser* produces a LOTOS specification as well as a special control file for the *animator* tool.

The translation from SOLVE to LOTOS is reasonably straightforward; see [17] for the details. Each object corresponds to a LOTOS process that interacts with other via an intermediate process *ObjectComms* as communication medium. Inter-object communication is supported by *ObjectComms* rather than directly for two reasons. The most important reason is that it allows non-blocking **TellCall** method invocations. These are effectively queued within *ObjectComms* until the targeted process (the server object) is ready to receive them. The second reason is to allow flexibility in routing messages. The communication model permits dynamic modification of communication connections, although dynamic creation of objects and their connections are not currently supported by SOLVE.

Fig. 6 outlines part of the LOTOS specification architecture for the SOLVE VCR clock specification. This shows the actual communication paths compared to the logical communication paths in Fig. 2. The hidden gate *Messages* carries inter-object communications. The *Interface* object is implicit in a SOLVE specification, but created explicitly in the LOTOS translation to handle communication with the user via a gate of the same name.

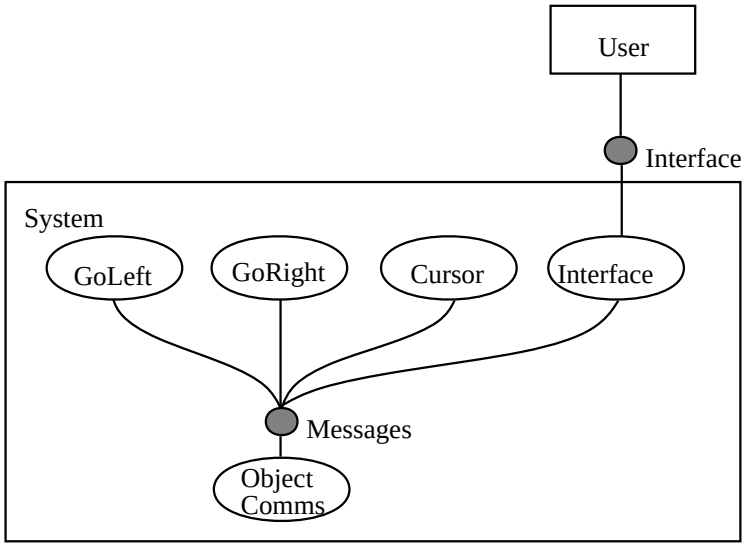

Fig. 6 Translated LOTOS Specification Architecture

5.2 Animation Tools

The *displayer* tool appears in its own window. It displays object icons (bitmaps and identification text) in response to requests from the *animator* tool, and it passes user requests to click or drag object icons to *animator*.

The *animator* tool appears as a set of windows with pull-down menus. Its function is to manage the interactive animation of a SOLVE specification. When it is invoked with a LOTOS specification file and an animation control file, *animator* spawns *displayer* and *hippo* as child processes. *animator* communicates via Unix pipes to/from the standard input/output of *displayer* and *hippo*. This simple means of communication is the reason that *hippo* was used rather than a later simulator such as *smile* [7] or the one supplied with the *topo* toolset [4].

The *hippo* tool is an early LOTOS simulator produced by the SEDOS project [25]. Other tools in the SEDOS toolset are used to process and check the LOTOS specification. The purpose of *hippo* is to simulate the behaviour of the given system, yielding lists of possible events. *displayer* turns user input (from the graphical interface) into event offers. This effectively yields lists of events offered by the environment. To manage an interactive animation, *animator* synchronises the *hippo* events offers and the *displayer* event offers. This is possible because both *parser* and *displayer* generate only (a small number of) predefined LOTOS event structures. The interfaces between *displayer*, *animator* and *hippo* are shown in Fig. 7. The sequence of animation events is shown in Fig. 8.

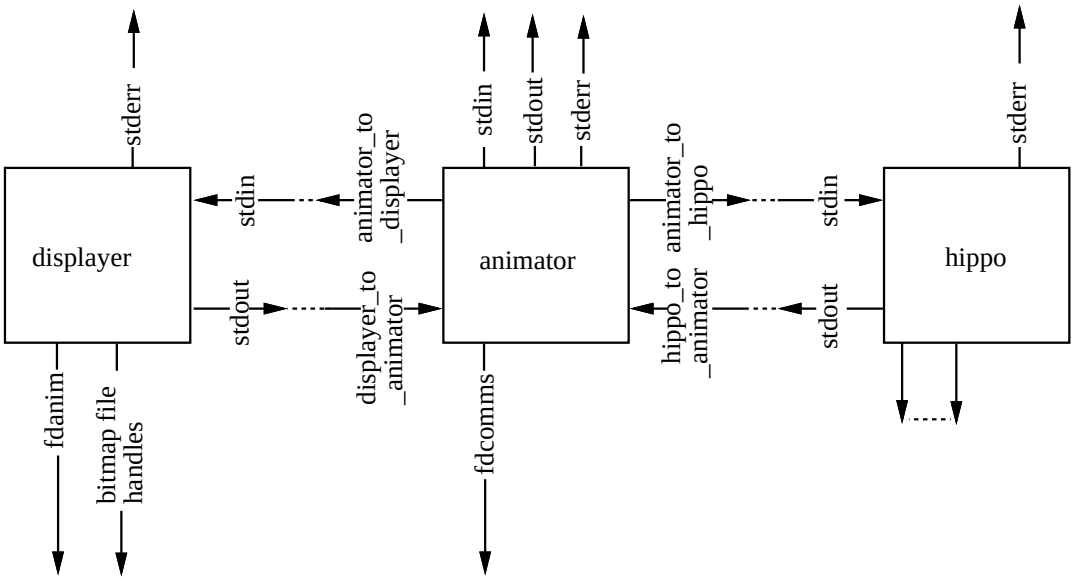

Fig. 7 Interfaces between Animation Tools

The *hippo* simulator used by SOLVE is somewhat old in design. Minor changes were made to facilitate communication via standard input/output. Unfortunately, the version of *hippo* available to the authors continually allocates memory during simulation without freeing it. The consequence is that realistic simulations cannot be taken far without exceeding virtual memory limits.

During an animation there may be a choice of possible events. A menu option allows either *animator* or the user to resolve a choice between possible events. For automatic choice, *animator* gives highest priority to internal events, then events that display icons, and then other observable events in the order in that they are offered. This prioritisation makes sense for the object-based nature of the LOTOS specifications generated by SOLVE.

The command-line interface to the toolset is *solve* or *xdill* — Unix *shell/make* scripts that invoke the tools in the correct order.

7 Conclusions

In designing SOLVE, the aim was to produce a language and set of tools that would allow effective specification and visual animation of requirements. It was also intended that the system be accessible to those without training in formal methods. SOLVE can claim to have gone a long way towards meeting these objectives. The SOLVE language uses familiar object-based modelling concepts in a programming style. Yet it is translated automatically into LOTOS, ensuring a precise basis for any specifications. The

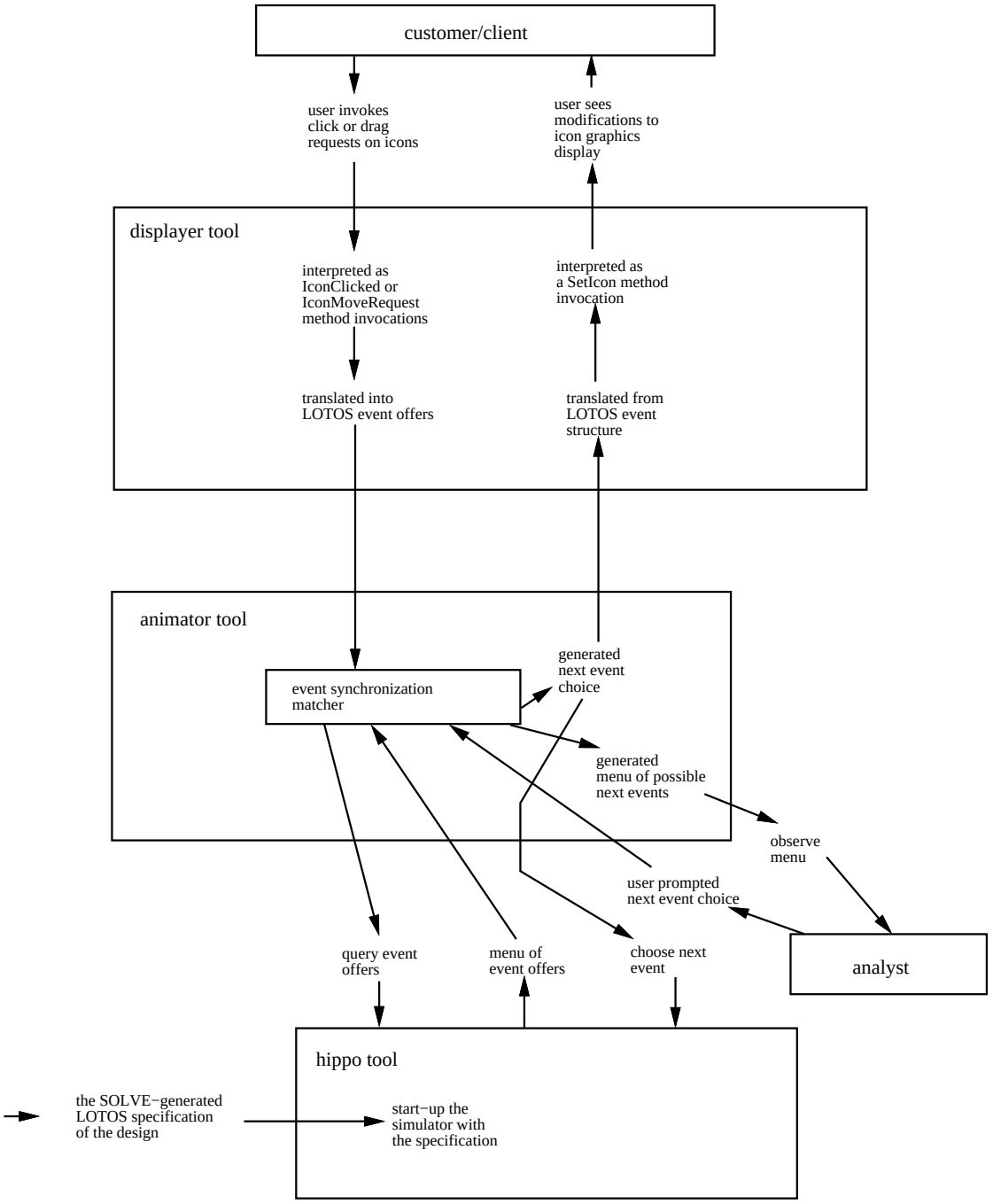

Fig. 8 Sequence of Animation Events

visual animation requires no knowledge of LOTOS, and is equally usable by customers/clients, analysts and designers/programmers. The availability of a SOLVE-specific syntax-directed editor is a boon to novices wishing to use SOLVE.

However, some further investigations are necessary. SOLVE has been partly oriented towards interactive systems that lend themselves to this kind of approach. This is not an intrinsic restriction, as witness the application to digital logic design with XDILL. The basic concepts of SOLVE are quite general, and translate to LOTOS in a natural way. Objects represent interactive graphical entities, and could be used in a variety of other application domains. Many applications have a conventional graphical form that could be animated (e.g. circuit diagrams in electronics, time-sequence diagrams in data communications, entity-relation diagrams in databases).

The SOLVE language would benefit from some extensions. A greater variety of data types, particularly records, would be desirable. Inheritance would permit easier re-use of SOLVE components. Dynamic creation and deletion of objects would be useful, as would dynamic modification of object communication paths. Following the current trend towards multi-media systems, the objects displayed by SOLVE could have other characteristics such as sounds or moving images.

SOLVE has still to be used on serious applications, but it is believed that the approach will scale up satisfactorily. Certainly, SOLVE fills a gap in the software engineering life-cyle that is presently not covered by LOTOS.

Acknowledgements

SOLVE and XDILL were designed and implemented by the second author, Ashley McClenaghan, who was supported by the UK Science and Engineering Research Council on the SPLICE project. SyD-EG was designed and implemented by Philip Eccles (University of Stirling), who was supported by an advanced course studentship from the UK Science and Engineering Research Council. The authors thank Richard Sinnott (University of Stirling) for comments.

References

[1] BLACK, S.: 'Objects and LOTOS', in VUONG, S. T., editor, *Proc. Formal Description Techniques II*, North-Holland, Amsterdam, Netherlands, December 1989

[2] CARDELLI, L. and WEGNER, P.: 'On understanding types, data abstraction, and polymorphism', *ACM Computing Surveys*, 1985, **17**, (4), pp. 471–522

[3] CUSACK, E. L., RUDKIN, S., and SMITH, C.: 'An object-oriented interpretation of LOTOS', in VUONG, S. T., editor, *Proc. Formal Description Techniques II*, North-Holland, Amsterdam, Netherlands, December 1989

[4] DE MIGUEL MORO, T., ROBLES, T., SALVACHUA, J., and AZCORRA, A.: 'The SRTS experience: Using TOPO for LOTOS design and realization', in QUEMADA, J., MAÑAS, J., and VAZQUEZ, E., editors, *Proc. Formal Description Techniques III*, North-Holland, Amsterdam, Netherlands, November 1990

[5] DOUGLAS, S., DOERRY, E., and NOVICK, D.: 'QUICK: A tool for graphical user-interface construction by non-programmers', *The Visual Computer*, 1992, **8**, (2), pp. 117–133

[6] ECCLES, P. A.: 'SyD-EG: A syntax-directed editor generator using standard Unix tools', Master's thesis, Department of Computing Science, University of Stirling, UK, March 1994

[7] EERTINK, H. and WOLZ, D.: 'Symbolic execution of LOTOS specifications', in DIAZ, M. and GROZ, R., editors, *Proc. Formal Description Techniques V*, pp. 295–310, North-Holland, Amsterdam, Netherlands, October 1992

[8] GIBSON, J. P.: 'Formal Object Oriented Development of Software Systems using LOTOS', PhD thesis, Department of Computing Science, University of Stirling, UK, 1993

[9] GIBSON, J. P.: 'A LOTOS-based approach to neural network specification', Technical Report CSM-112, Department of Computing Science, University of Stirling, UK, May 1993

[10] HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., SHTULL-TRAURING, A., and TRAKHTENBROT, M.: 'STATEMENT: A working environment for the development of complex reactive systems', *IEEE Transactions on Software Engineering*, 1990, **16**, (4), pp. 403–414

[11] HERCZEG, J., HOHL, H., and RESSEL, M.: 'A new approach to visual programming in user interface design', in *Proc. of 5th International Conference on Human-Computer Interaction*, Orlando, USA, August 1993

[12] IMPERIAL SOFTWARE TECHNOLOGY LTD.: 'X-Designer', Reading, UK, 1992

[13] ISO/IEC: 'Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour', ISO/IEC 8807 (International Organization for Standardization, 1989)

[14] KERNIGHAN, B. W. and RITCHIE, D. M.: 'The *m4* macro processor', Technical report, Bell Laboratories, Murray Hill, New Jersey, USA, 1977

[15] MARSHALL, A. K.: 'Introduction to LOTOS tools', in VAN EIJK, P., VISSERS, C. A., and DIAZ, M., editors, *The Formal Description Technique* LOTOS, North-Holland, Amsterdam, Netherlands, 1989

[16] MAYR, T.: 'Specification of object-oriented systems in LOTOS', in TURNER, K. J., editor, *Proc. Formal Description Techniques I*, pp. 107–119, North-Holland, Amsterdam, Netherlands, 1989

[17] MCCLENAGHAN, A.: 'SOLVE: Specification using an object-oriented, LOTOS-based, visual language', Technical Report CSM-115, Department of Computing Science, University of Stirling, UK, January 1994

[18] MOREIRA, A. M. D. and CLARK, R. G.: 'Using rigorous object-oriented analysis', Technical Report CSM-111, Department of Computing Science, University of Stirling, UK, August 1993

[19] QUEMADA, J., PAVON, S., and FERNANDEZ, A.: 'Transforming LOTOS specifications with LOLA: The parameterized expansion', in TURNER, K. J., editor, *Proc. Formal Description Techniques I*, pp. 45–54, North-Holland, Amsterdam, Netherlands, 1989

[20] THIMBLEBY, H. W.: 'User Interface Design' (Addison-Wesley, 1990)

[21] TURNER, K. J.: 'An engineering approach to formal methods', in DANTHINE, A. A. S., LEDUC, G., and WOLPER, P., editors, *Proc. Protocol Specification, Testing and Verification XIII*, pp. 357–380, North-Holland, Amsterdam, Netherlands, June 1993, Invited paper

[22] TURNER, K. J.: 'Exploiting the *m4* macro language', Technical Report CSM-126, Department of Computing Science, University of Stirling, UK, September 1994

[23] TURNER, K. J. and SINNOTT, R. O.: 'DILL: Specifying digital logic in LOTOS', in TENNEY, R. L., AMER, P. D., and UYAR, M. Ü., editors, *Proc. Formal Description Techniques VI*, pp. 71–86, North-Holland, Amsterdam, Netherlands, 1994

[24] TURNER, K. J. and VAN SINDEREN, M.: 'LOTOS specification style for OSI', in VAN DE LAGE-MAAT, J. and BOLOGNESI, T., editors, *The LOTOSPHERE Project*, Kluwer, 1994, Forthcoming

[25] VAN EIJK, P. H.: 'Software tools for the Specification Language LOTOS', PhD thesis, Department of Informatics, University of Twente, Enschede, Netherlands, 1988

Captions

Fig. 1 VCR Clock System

Fig. 2 Communicating Objects in a SOLVE Specification

Fig. 3 (a) After Clicking *GoLeft*

Fig. 3 (b) After Processing *GoLeft*

Fig. 4 (a) After Falsifying Clock Input

Fig. 4 (b) After Processing Clock Input

Fig. 5 The *syd* Editor being used on a SOLVE Specification

Fig. 6 Translated LOTOS Specification Architecture

Fig. 7 Interfaces between Animation Tools

Fig. 8 Sequence of Animation Events

Tables

Table 1 Object-Oriented Analysis of VCR Clock

Object	**Category**	**Attributes**	**Methods provided**	**Methods called**
Units	Display	Digit Value	*Inc* *Dec*	*TensDigit.QueryValue*
Tens	Display	Digit Value	*Inc* *Dec* *QueryValue*	
Cursor	Display	X Position	*Left* *Right* *QueryXPos*	
GoLeft	Button		*IconClicked*	*Cursor.Left*
GoRight	Button		*IconClicked*	*Cursor.Right*
IncNum	Button		*IconClicked*	*Cursor.QueryXPos* *Tens.Inc* *Units.Inc*
DecNum	Button		*IconClicked*	*Cursor.QueryXPos* *Tens.Dec* *Units.Dec*