Kenneth J. Turner, Ashley McClenaghan and Colin Chan. Specification and Animation of Reactive Systems. In Volkan Atalay, Ugur Halici, Kemal Inan, Nese Yalabik and Adnan Yazici, editors, Proc. International Symposium on Computer and Information Systems XI, pages 355-364, Antalya, Turkey, November 1996.

Specification and Animation of Reactive Systems

K. J. Turner†, A. McClenaghan‡, C. Chan†

†Computing Science, University of Stirling, Stirling FK9 4LA, UK (kjt|cch@cs.stir.ac.uk)
‡APM Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD, UK (am@ansa.co.uk)

Abstract

SOLVE (Specification using an Object-based, LOTOS-defined, Visual language) is designed to allow formal requirements capture, particularly for reactive systems. The SOLVE language is object-based, and formally defined using LOTOS (Language Of Temporal Ordering Specification). SOLVE is supported by tools that allow direct visual animation of systems specified in this language. Animation is supported by translating a SOLVE specification automatically into a LOTOS specification, and then graphically simulating this. A further application is embodied in the XDILL tool that supports requirements specification and visual animation of digital logic circuits. Several illustrative SOLVE examples are given.

Keywords: animation, formal specification, LOTOS (Language Of Temporal Ordering Specification), requirements capture, visualisation

1 Introduction

Requirements capture, analysis and specification are difficult yet very important parts of system development. The project SPLICE I (Specification and Prototyping for a LOTOS Interactive Customer Environment – Phase I) was planned to show that requirements capture and specification would benefit from the use of greater formality, specifically LOTOS (Language Of Temporal Ordering Specification [6]). SPLICE aimed to develop an effective bridge between clients, analysts, designers and programmers. An equally important bridge had to be developed between system requirements, system architecture and formal representation.

The goal is to have the tool user indirectly manipulate the formal representation of requirements via a familiar interface – a visual representation of the problem. The approach taken to visual animation of LOTOS specifications is called SOLVE (Specification using an Object-based, LOTOS-defined, Visual language). SOLVE uses graphical presentation and manipulation to convey the meaning of a specification. SPLICE aimed 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: OSI services, digital logic specification, neural networks and reactive systems. 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 requirements capture using object-based methods.

SOLVE specifications are automatically translated into LOTOS specifications. The object basis of SOLVE confers a natural style of modelling that is appropriate for requirements specifications to be evaluated by clients. It was not an aim to incorporate the full paraphernalia of an object-oriented method and language. The advantages claimed for SOLVE stem from its formal basis and graphical animation rather than its use of objects.

QUICK (Quick User Interface Construction Kit [3]) inspired the work reported here. QUICK 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. However, unlike QUICK, SOLVE's primary concern is to generate formal specifications.

Perhaps the best known work on formal modelling of reactive systems is that embodied in the STATEMATE tool [5]. The visual formalism of statecharts allows compact, hierarchical descriptions based on a finite state machine model. The graphical model itself can be directly executed. Animation and formal analysis are possible with SOLVE, but less directly than in the case of statecharts. SOLVE deliberately distances itself from the underlying formalism of LOTOS, and takes an object-based view of the system. A number of object-oriented methods including O-MATE [4] use variants of statecharts to describe behaviour.

A graphical syntax for LOTOS has been standardised, but merely as an alternative to the textual syntax. GLOTOS is convenient for seeing the structure of a specification, but is not an analysis tool like SOLVE. Some object-oriented styles have been developed for LOTOS (e.g. [8]). However, objects appear in the LOTOS generated by SOLVE only as a result of the translation.

2 The SOLVE Approach

2.1 The Nature of SOLVE

The key concepts in SOLVE are formal specification (via LOTOS), graphical animation and object-based modelling. SOLVE is designed to be used by people who are not familiar with formal languages (in particular LOTOS). SOLVE is a system for specifying requirements formally and for exploring these using interactive animation. SOLVE is a language for specifying and animating (prototyping) the requirements of reactive systems that produce feedback in response to user input. These include human-oriented devices such as domestic appliances. The specification and analysis of digital logic circuits has been investigated using a special version of SOLVE.

SOLVE has a straightforward denotation in terms of LOTOS, embodied in the automatic translation from SOLVE to LOTOS. As a result, SOLVE confers the same benefits of precision and analysability as LOTOS. The SOLVE user is largely unaware of LOTOS as the underlying language. It is therefore possible to develop and validate a SOLVE specification without needing to take advantage of the features of LOTOS. Nonetheless once a SOLVE specification is completed to the satisfaction of the user, the power of LOTOS can be brought into play. The LOTOS specification provides a formal statement of requirements that can be used as the basis of a contract between the client and the developers. The LOTOS specification can be formally refined towards an implementation. Correctness-preserving transformations can be used, or a low-level specification in LOTOS can be developed and shown to satisfy the high-level requirements. Conformance tests can be derived from the high-level specification. Properties of a LOTOS specification can be formally checked. Readily available tools can be used to support these activities [1]. Of course, using the LOTOS specification derived from SOLVE needs someone who is expert in the language. But the important point is that it adds value to SOLVE-based development without forcing formality on the ordinary user.

It was decided to build inheritance into the environment not into the language. 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 yet include inheritance mechanisms, so SOLVE presently lacks full object orientation. 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 SOLVE notion of objects reflects several basic aspects of the object-oriented paradigm. A simple object has an icon – a visual representation of the object. Interaction with the environment is in terms of message passing, e.g. clicking on an icon causes a message to be sent to the appropriate method of the object responsible for the icon. One of the objects in an executing SOLVE specification is *Interface* that is automatically declared and defined.

An important feature of SOLVE is visualisation. Each object is displayed as an icon – in fact a bitmap displayed on the screen. An object is responsible for displaying and modifying its own icon, representing an abstraction of its state or the state of some part of the total system. The object icons visually inform the SOLVE user what is happening in the system under consideration. Most objects are declared explicitly in the SOLVE specification. The user interacts with the animation of a SOLVE specification by communicating with the *Interface* object. This handles the screen window system, and so is responsible for displaying object icons and for accepting mouse clicks and drags.

2.2 SOLVE Language Elements

Object declarations provide type information to check correct use of definitions by objects. Each object has a list of instance parameters and a set of methods. Reflecting the visual nature of SOLVE, objects have 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* (set up instance variables and icon), *IconClicked* (handle mouse click on icon) and *IconMoveRequest* (handle mouse drag on icon). Although these methods are implicitly declared, an explicit definition of each must be given by the specifier.

The SOLVE language is object-based and intentionally resembles a programming language. However, SOLVE specifications can be automatically translated into LOTOS specifications. Although objects may execute concurrently, behaviour within a simple object is sequential. This means that an object may execute only one method at a time.

The following is an extract from the SOLVE specification of the microwave oven discussed in section 3.1. Portions of SOLVE specification are followed by their explanation. Ellipses (...) show parts of the specification that have been omitted for brevity.

```
System Microwave Is                          −− declare system name

  PicDecls                                   −− declare icons
    beep, no_beep,                           −− audible alarm status
    ...                                      −− declare other icons
  EndPicDecls
```

The icon picture declarations are actually the names of files containing the bitmap images.

```
  ObjectDeclarations                         −− declare objects
    Object Timer() Is                        −− declare timer object
      QueryTimer()(Int)                      −− check timer method
      AddMin()()                             −− add minute method
      Reset()()                              −− reset method
    EndObject
    ...                                                   —— declare other objects
  EndObjectDeclarations
```

An object declaration gives its name and instance parameters. The name, parameter sorts and result sorts of each method are also given. For example, *QueryTimer* has an empty parameter list and a result list with just one integer.

```
  ObjectDefinitions                                       —— define objects
    Object Timer () Is                                    —— define timer object
      Method Initialize () Is                             —— initialize timer
        Assign (xPos, 1)                                  —— set icon x coord. to 1
        Assign (yPos, 1)                                  —— set icon y coord. to 1
        Assign (iconPic, digit_0)                         —— set icon to 0
        TellCall Interface.SetIcon (xPos, yPos, iconPic)  —— display new icon
        Return ()                                         —— no return result
      EndMethod
      Method IconClicked () Is                            —— handle click on timer
        If (iconPic Eqi digit_0)                          —— is timer zero?
          Then                                            —— timer 0, so no action
          Else                                            —— timer not 0
            Assign (iconPic, iconPic Minus 1)             —— reduce timer, display
            TellCall Interface.SetIcon (xPos, yPos, iconPic)
            If (iconPic Eqi digit_0)                      —— is timer now zero?
              Then TellCall PowerTube.DeEnergise ()       —— time 0, stop cooking
              Else                                        —— not 0, keep cooking
            EndIf
        EndIf
        Return ()                                         —— no return result
      EndMethod
      Method IconMoveRequest (Int : a, Int : b) Is        —— handle timer move
        Return ()                                         —— no move, no result
      EndMethod
      Method QueryTimer () Is                             —— check timer
        Return (iconPic)                                  —— return current value
      EndMethod
      Method AddMin () Is ...                             —— increase timer
      Method Reset () Is ...                              —— zeroise timer
    EndObject
    ...                                                   —— define other objects
  EndObjectDefinitions

EndSystem
```

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 *Timer* object has only the three implicitly declared instance parameters: *xPos, yPos* and *iconPic*. 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. 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 conditional branching.

While *Condition* Do *Statements* EndWhile supports a conditional 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 like **AskWaitCall** except that it does not block execution in the invoking object. Any results returned by the invoked method are ignored.

2.3 Tool Support for SOLVE

The SOLVE toolset consists of the following main programs: *editor*, *parser*, *displayer*, *animator*, a modified version of the *hippo* simulator, *solve* and *xdill*. The tools are built using the X windows environment and C, though some of this code is generated by *yacc* and *X-Designer*. The command-line interface to the toolset is *solve* or *xdill* – Unix *shell*/*make* scripts that invoke the tools in the correct order.

Together the analyst and the client may explore and assess the animated behaviour of the specification to establish its correctness and completeness using the *animator* tool. *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 user without a (deep) knowledge of LOTOS.

SOLVE specifications can be produced using a standard text editor. However, the syntax-directed editor *syd* was developed with SOLVE in mind. 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 the preferred granularity of editing.

The *parser* tool makes use of *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 *animator*. The translation from SOLVE to

LOTOS is reasonably straightforward; see [7] for the details. Each object corresponds to a LOTOS process that interacts with others via an intermediate process *ObjectComms* as communication medium. The communication model permits dynamic modification of communication connections, although dynamic creation of objects and their connections are not currently supported by SOLVE.

The *displayer* tool appears in its own window. It displays object icons (bitmaps and identification text) in response to requests from *animator*, and it passes on mouse clicks and drags. *animator* appears as a set of windows with pull-down menus that 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*.

The *hippo* tool is an early LOTOS simulator produced by the project SEDOS (Software Environment for the Design of Open Distributed Systems). Other tools in the SEDOS toolset are used to process and check the LOTOS specification. Simple communication via pipes is the reason that *hippo* was used rather than a later simulator such as *smile* or the one supplied with the *topo* toolset. The purpose of *hippo* is to simulate the behaviour of the given system, yielding lists of possible events. *displayer* turns user input (i.e. mouse actions) into event offers. This effectively yields lists of events offered by the environment. To manage an interactive animation, *animator* synchronises *hippo* event offers with *displayer* event offers.

When animation begins, *animator* and *displayer* open their windows. *animator* provides a menu of possible next events. The user may choose a view option that shows the event offers from the system and its environment. *displayer* shows the graphical view of the system objects and handles events offered by the user via this graphical view. *animator* shows the events offered by the system via *hippo*, and the menu of possible events that are acceptable to both the environment and the 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 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.

3 Some SOLVE Examples

3.1 Reactive Systems: A Microwave Oven

A microwave oven is a typical reactive system. The user interacts with the oven through a control panel containing three buttons. The door close button is used to close the oven prior to cooking. The cooking control button is used to begin cooking. By default the cooking period is two minutes, but an additional minute is added for each press of the cooking control button. At any one time the cooking period is limited to a maximum of nine minutes, but the cooking control button can be used again later during cooking. When the period is up, cooking ceases. The door open button is normally used to open the oven after cooking. If the door is opened during cooking, the timer is reset and cooking stops. The oven also has various output devices. A timer shows the remaining time to cook. A light is switched on while the door is open or during cooking. A power tube is used to heat the oven. When cooking finishes, an audible ‘beep’ is sounded.

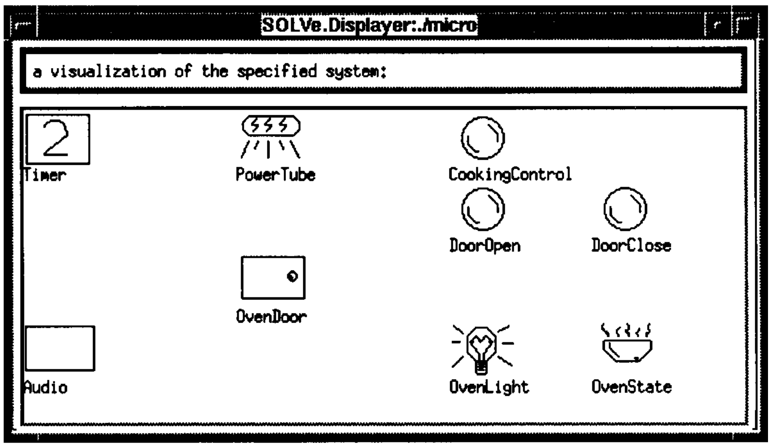


Figure 1: SOLVE View of Microwave Oven during Cooking

The user's view of the SOLVE model during animation is given in figure 1. To reach this state the user has clicked on the door close and cooking control buttons. The oven door icon is in the closed state. The power tube and oven light are shown to be on, and the oven state is shown to be cooking. The timer is currently set at two minutes. The user simulates the passage of time by clicking on the timer icon, reducing its count by one minute for each click. When the count reaches zero, the SOLVE display will show that the power tube and light are switched off and that the oven state is no longer cooking. In addition the audio device will show a 'beep' signal. The full SOLVE specification is about 310 non-comment lines and is given in [2].

3.2 Simulation: A Petri Net

The example chosen for a Petri net describes a fan heater. The heater has a fan and a heating element, both controlled by switches with an interlock. The fan alone may be switched on (just to blow cool air). If the fan is on, the heater may also be switched on (to blow hot air). The heating element may not be switched on alone (since it would overheat). If both fan and heating element are on, they may be switched off together. The user inputs are thus the settings of the two switches. The outputs correspond to operation of the fan and the heating element.

The Petri net model of the heater is given in figure 2. The SOLVE model shows icons for only the fan, heating element, places and transitions. SOLVE is currently able to draw only icons, not to join them up. The intermediate lines in the figure have been added manually to make sense of the net. The SOLVE user clicks on transitions to cause them to fire. As usual, if the places leading to a transition have a token then the transition may fire and cause tokens to be transferred to the output places. The SOLVE animation takes care of the firing rules and the graphical display of tokens. The approach also allows transitions with inhibitions, for example to prevent the fan from being switched off while the heating element is on, and to prevent the heating element being switched on while the fan is off. In the situation of figure 2, both the fan and heating element are on. The *turnheateroff* transition is possible, but the *turnfanoff* transition is inhibited by the token in the *HeaterOn* place. The *turnbothoff* transition is also possible. The full SOLVE specification is about 350 non-comment lines and is given in [2].

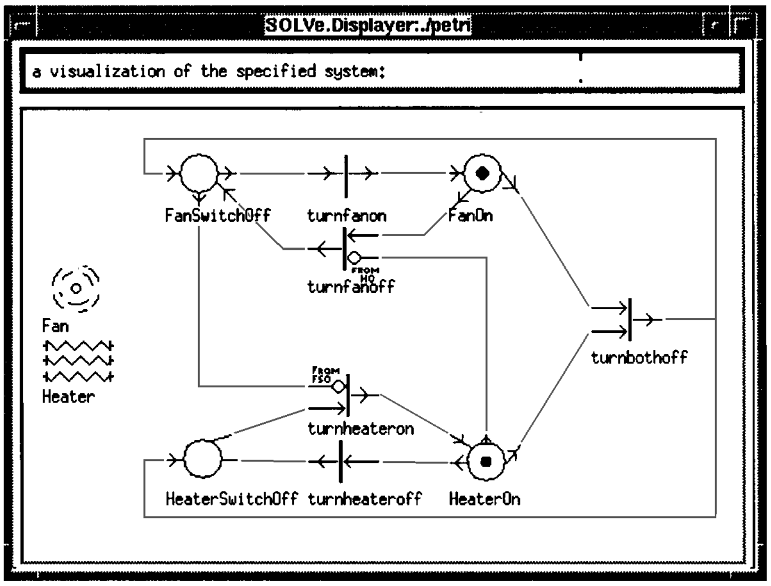


Figure 2: SOLVE View of Heater with Fan and Heater On

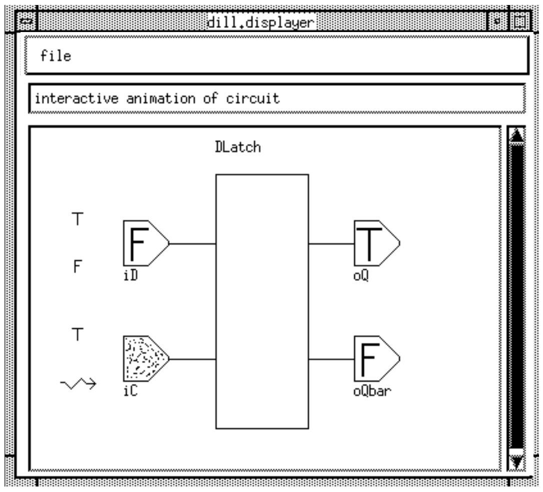
3.3 Digital Logic: A Data Latch

DILL (Digital Logic in LOTOS [10]) was developed for the specification and validation of digital logic components and circuits using LOTOS. The SOLVE approach has been extended to allow interactive visual animation of DILL specifications using X windows, hence the name XDILL. Given a DILL source file, the *xdill* tool carries out the translation into SOLVE and LOTOS and then animates it. 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. Animation options for XDILL are the same as for SOLVE.

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, so the names of input gates must begin with 'i' and those of output gates with 'o'. A D-Latch is one of the pre-defined DILL 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, and outputs along the right side. Logic values *T* and *F* (True and False) are displayed in arrow-shaped buttons that may be clicked to set a particular logic value.

When the user clicks on a *T* or *F* button, it momentarily turns into a zig-zag as shown in figure 3(a). In this example, the input data line *iD* is False. The input clock line *iC* has just been clicked to set it False, so changes are propagated throughout the system. After automatic selection of events the result is the new stable state shown in figure 3(b). As expected, the



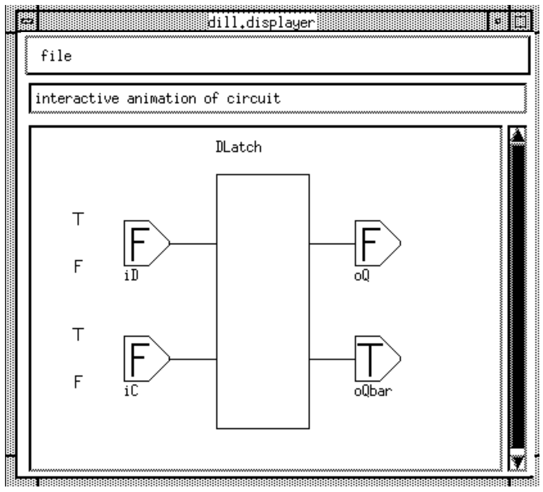


Figure 3(a) After Falsifying Clock Input Figure 3(b) After Processing Clock Input

D-Latch has stored the new input value False on the falling edge of the clock pulse; the output *oQ* is False and the negated output *oQbar* is True.

4 Conclusions

In designing SOLVE, the aim was to produce a language and a 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 specification. LOTOS specialists can then use the specification to carry out formal verification, validation and refinment. The visual animation requires no knowledge of LOTOS, and is thus suitable for non-specialists such as end users. The availability of a SOLVE-specific syntax-directed editor is an advantage for novices.

However, some further investigations are necessary. SOLVE has been partly oriented towards reactive 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 reactive graphical entities with which the user may interact. Many applications have a conventional graphical form that could be animated (e.g. circuit diagrams in electronics, time-sequence diagrams in data communications, mimic diagrams in process control).

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 largely not covered by existing LOTOS-based methods.

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. The third author, Colin Chan, updated the implementation and carried out further case studies. The authors thank Richard Sinnott (University of Stirling) for comments. This paper expands and extends the preliminary work reported in [9].

References

[1] Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris A. Vissers, editors. *The LOTOSPHERE Project*. Kluwer Academic Publishers, London, UK, 1995.

[2] Colin Chan. SOLVE. B.Sc. Dissertation, Department of Computing Science and Mathematics, University of Stirling, UK, April 1996.

[3] Sarah Douglas, Eckehard Doerry, and David Novick. QUICK: A tool for graphical user-interface construction by non-programmers. *The Visual Computer*, 8(2):117–133, 1992.

[4] David Harel and Eran Gery. Executable object modeling with Statecharts. In *Proc. 18th International Conference on Software Engineering*, pages 246–257. Institution of Electrical and Electronic Engineers Press, New York, USA, 1996.

[5] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[6] 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, Geneva, Switzerland, 1989.

[7] Ashley McClenaghan. SOLVE: Specification using an object-oriented, LOTOS-based, visual language. Technical Report CSM-115, Department of Computing Science and Mathematics, University of Stirling, UK, January 1994.

[8] Ana M. D. Moreira and Robert G. Clark. Using rigorous object-oriented analysis. Technical Report CSM-111, Department of Computing Science and Mathematics, University of Stirling, UK, August 1993.

[9] Kenneth J .Turner and Ashley McClenaghan. Visual animation of LOTOS using SOLVE. In FORTE-VI-Ed, editor, *Proc. Formal Description Techniques VI*, pages 283–285. Chapman-Hall, London, UK, 1995.

[10] Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 71–86. North-Holland, Amsterdam, Netherlands, 1994.