Making Changes to Formal Specifications: Requirements and an Example

David W. Bustard and Adam C. Winstanley

Abstract—**Formal methods have had little impact on software engineering practice, despite the fact that most software engineering practioners readily acknowledge the potential benefits to be gained from the mathematical modeling involved. One reason is that existing modeling techniques tend not to address basic software engineering concerns. In particular, while considerable attention has been paid to the construction of formal models, less attractive maintenance issues have largely been ignored. The purpose of this paper is to clarify those issues and examine the underlying requirements for change support. The discussion is illustrated with a description of a change technique and tool developed for the formal notation LOTOS. This work was undertaken as part of the SCAFFOLD project, which was concerned with providing broad support for the construction and analysis of formal specifications of concurrent systems. Most of the discussion is applicable to other process-oriented notations such as CCS and CSP.**

Index Terms— **Change control, formal specification, process algebra, and LOTOS.**

I. Introduction

THERE are essentially two main ways to use formal models in software development, as summarized in Fig.1. In a *subsidiary support* role, a formal model (or models) helps to clarify requirements that are specified informally and provides a reference base for software design and implementation. In a *central construction* role, a formal model is refined in stages towards an implementation. Each refinement extends the preceding model by dropping down to some lower-level, less abstract description, which may bring in additional detail from the informal specification.

These two roles for formal models are quite distinct but the types of change involved in each case are similar. In both cases, a model is built initially and adjusted until it matches the corresponding informal description, which itself may be changed in the process. Refinement changes are obviously an important part of the central construction approach but similar changes are also made when a formal model in a subsidiary role is expanded to explore requirements in greater detail. Where the two approaches differ significantly is in the way that requirement changes are handled after models have been completed. In the subsidiary support case, such changes are no different from those made during initial model construction. However, in the central construction approach it is necessary to ripple changes through the refinement sequence. Even here, however, although the means of change may be different, the changed models can be evaluated in much the same way.

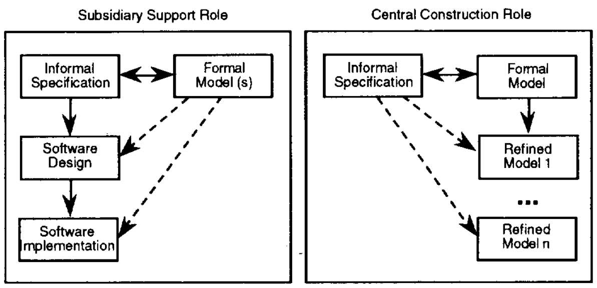


Fig. 1. Roles of formal models in system development.

Very little research has been done on supporting change to formal system models. Some of the few examples of relevant work in this area include those contained in [1] and [2]. Indeed, it is often suggested [3], [4] that the term 'formal methods' is misleading because, as yet, users have been offered little more than formal notations. Clearly it is highly desirable to also have guidance on how such notations can be applied effectively in software production [5] and also to have tool support for the process involved [6]. This paper considers both issues with respect to making changes to formal models. More specifically, the paper concentrates on the basic question of how changes can be made to an individual formal model and how such a procedure might be supported. It can be assumed that change occurs under configuration management control [7]. This means that each formal model is an explicit configuration item, any changes made to it are agreed by a change control board, and a full change history is maintained. At this level, each change is made with respect to some particular version of the model—the *baseline*. The change is specified in advance and gives details of the requirement change. It may also include a definition of some of the necessary physical changes to the model, with remaining details recorded when the modified model is placed back under configuration control.

As a new model is developed it will go through a succession of intermediate, transient changes that are neither planned nor recorded. For all changes, however, the underlying steps involved are the same, namely: understand the need for change, implement it and evaluate the results. However, the precise means of change will vary considerably with the style

Manuscript received August, 1993; revised May, 1994. This work was supported by SERC Grant GR/G 03700. Recommended for acceptance by I. Sommerville.

D. W. Bustard is with the Department of Computing Science, University of Ulster, Cromore Road, Coleraine, BT52 1SA, Northern Ireland; e-mail: dwb@ucvax.ulster.ac.uk. A. C. Winstanley is with the Department of Computer Science, Queen's University, Belfast, Northern Ireland, U.K., BT7 1QP; e-mail: a.winstanley@qub.ac.uk.

IEEE Log Number 9403574.

of formal model used. The variety of styles [4] includes model-oriented specifications, algebraic specifications, modal logics and process algebras. This paper considers just this last group which are typically used to specify concurrent systems. The particular language used is LOTOS [8], [9], [10], but much of what is said is also applicable to CCS [11] and CSP [12], the notations on which the behavioral component of LOTOS is based.

The next section gives a brief overview of LOTOS and illustrates its form and use with a simple example. This is then followed by a section that identifies how changes might be made to such models and a section that describes tool support that has been developed in the SCAFFOLD project [13].

II. LOTOS Specifications

LOTOS (Language of Temporal Specification) is used to define the behavior of concurrent systems. Behavior is described in terms of the significant events (or actions) in a system and the constraints on their order of occurrence. A LOTOS specification is structured as a hierarchy of communicating processes and the overall specification itself is also a process. As an example, consider the specification of a very simple automated bank teller (adapted from [14]). The teller accepts a cash card and PIN (Personal Identification Number) typed on a keypad and, if valid, returns £30; otherwise the transaction is rejected. In both cases the cash card is returned as the final action. For simplicity, it is assumed that there is sufficient money available in the account identified and in the teller machine itself.

From this description, the following events, representing communication between the teller and its user, can be identified: *AcceptCard, ReturnCard, RequestPIN, AcceptPIN, SupplyMoney* and *DisplayRejection.* In addition, there are internal teller events associated with the examination of the card and the PIN, and the subsequent actions taken: *IdentifyValidCard, IdentifyInvalidCard, IdentifyValidPIN, IdentifyInvalidPIN.* The behavior of the teller can be described by the set of event sequences that can occur, namely:

1) a card is rejected because it cannot be read: *AcceptCard; IdentifyInvalidCard; DisplayRejection; ReturnCard.*
2) a transaction is rejected because the PIN is faulty: *AcceptCard; IdentifyValidCard; RequestPIN; AcceptPIN; IdentifyInvalidPIN; DisplayRejection; ReturnCard.*
3) a transaction is completed successfully: *AcceptCard; IdentifyValidCard; RequestPIN; AcceptPIN; IdentifyValidPIN; SupplyMoney; ReturnCard.*

As these sequences have common components, it is more informative to combine them. Fig. 2, for example, shows the permitted sequences in the form of an *action tree* [9]. The nodes in the tree represent unnamed system states and the arcs represent events. Each event is a transition from one system state to another. Branches indicate where there is a choice of event. Note that the repeated transaction behavior has been suppressed. This is essential in cases where repetition occurs an unspecified number of times. A square leaf node, in general, represents a suppressed component. The elaboration of such

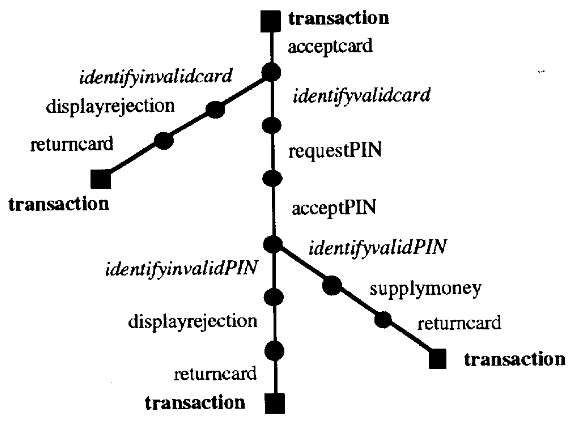


Fig. 2. Action tree for teller machine.

a component is determined by locating an identically named node within the tree.

The corresponding LOTOS description might take the following form (the numbers have been added to aid the explanation given subsequently):

```
1)  specification teller [AcceptCard, ReturnCard,
            RequestPIN, AcceptPIN, SupplyMoney,
            DisplayRejection]: noexit
    behavior
    transaction [AcceptCard, ReturnCard, RequestPIN,
            AcceptPIN, SupplyMoney, DisplayRejection]
    where
2)    process transaction [AcceptCard, ReturnCard,
            RequestPIN, AcceptPIN, SupplyMoney,
            DisplayRejection]: noexit :=
3)      hide IdentifyInvalidCard, IdentifyValidCard,
            IdentifyInvalidPIN, IdentifyValidPIN in
4)      (AcceptCard;
          (IdentifyInvalidCard; DisplayRejection;
            ReturnCard; exit
5)        []
          IdentifyValidCard; RequestPIN; AcceptPIN;
            (IdentifyInvalidPIN; DisplayRejection;
                ReturnCard; exit
6)        []
          IdentifyValidPIN; SupplyMoney;
            ReturnCard; exit)))
7)      >> transaction [AcceptCard, ReturnCard, RequestPIN,
            AcceptPIN, SupplyMoney, DisplayRejection]
      endproc
    endspec
```

The events that represent communication between the teller and the user are listed as specification parameters 1). The behavior of the teller is described by a single process *transaction* 2) which takes all of the external events as parameters. Internal events are 'hidden' within the transaction process 3). The behavior specified for transaction 4) follows the shape of the action tree in Fig. 2. A choice expression defines a branch in the tree, from which one of several behaviors may follow

5), 6). Once a particular sequence has been completed the transaction process is reinstantiated recursively 7) to indicate a return to the original teller state.

For larger specifications, and when refining a specification towards an implementation, it is usually desirable to structure a specification as a collection of interacting processes. For example, a LOTOS description of a card reader process might take the following form, indicating that the card reader repeatedly accepts and returns cards, distinguishing between cards that are valid and those that are not.

```
process cardreader [AcceptCard, ReturnCard,
  IdentifyValidCard, IdentifyInvalidCard]: noexit:=
  AcceptCard;
  (IdentifyValidCard; ReturnCard;
   cardreader [AcceptCard, ReturnCard, IdentifyValidCard,
     IdentifyInvalidCard])
  []
  (IdentifyInvalidCard; ReturnCard;
   cardreader [AcceptCard, ReturnCard, IdentifyValidCard,
     IdentifyInvalidCard])
endproc
```

The behavior expression for the teller specification then becomes:

```
behavior
 hide IdentifyValidCard, IdentifyInvalidCard in
 (transaction [AcceptCard, ReturnCard, RequestPIN,
 AcceptPIN, SupplyMoney,DisplayRejection,
 IdentifyValidCard, IdentifyInvalidCard
 |[AcceptCard, ReturnCard,
 IdentifyValidCard,IdentifyInvalidCard] |
 cardreader [AcceptCard, ReturnCard, IdentifyValidCard,
IdentifyInvalidCard])
```

This indicates that the *transaction* and *cardreader* processes run in parallel and synchronise on the *AcceptCard, ReturnCard, IdentifyValidCard* and *IdentifyInvalidCard* events. Note that the declaration of the *IdentifyValidCard* and *IdentifyInvalidCard* events has been brought outside the *transaction* process.

This discussion presented so far has illustrated the form of a LOTOS specification and two alternative ways of describing the information it contains: 1) as a set of event sequences; and 2) as an action tree. The order in which these representations have been presented also suggests a plausible means of using them in combination to help develop the LOTOS description, namely: define possible sequences, combine them in tree form and then build the LOTOS model to describe the resulting behavior. Even if designers prefer to build LOTOS descriptions directly, the other representations are still of benefit in evaluating the LOTOS model produced. That is, they can be used as specifications of the model and verified automatically against the model. The same approach can be used when modifying a LOTOS specification. Details of how this approach has been implemented in practice are given in a later section. Before that, however, the next section takes a more detailed view of the general requirements for supporting change to formal models and the particular support needed for event-based models.

III. Change Requirements

The introduction described the basic process of model change in terms of three stages of activity: understanding the need for change, implementing the change and evaluating the change. This section considers the needs of each of these activities in turn and briefly identifies some of the existing tool support for LOTOS in this area.

A. Understanding the Need for Change

Understanding the need for change involves, in general, an investigation of the requirements for the system being modelled and an investigation of the existing model. To fix ideas, assume that in the case of the teller model it has been discovered that users are tending to pick up their money and leave without taking their card. It has therefore been decided that a card should be returned before the money is dispensed. This is the change requirement expressed informally. To produce a precise specification of the change it is then necessary to understand the model and describe the change in terms of modifications to the event sequences permitted.

At present, the most common way of examining a LOTOS model is through the use of a *simulator*, which effectively translates the model into an action tree and allows it to be explored. Typically, however, the tree is not presented in its entirety but is explored one event at a time. For each state in the tree the set of possible next events is computed, from which an observer then makes a selection. This allows the simulator to derive the next state. In this way, a trace of events representing one path down the action tree is built up step-by-step. A simulator only constructs those parts of the tree that are needed for this path. Examples of simulators include HIPPO, developed as part of the Esprit SEDOS (Software Environment for the Design of Open Distributed Systems) project [10]; SMILE, a development from HIPPO by the Esprit Lotosphere project [15] which is incorporated into LITE (Lotosphere Integrated Tool Environment) [16]; EXPOSE [17]; and the University of Ottowa LOTOS Toolset [18]. These simulators include various features to reduce the tedium of single step examination of the tree, such as an ability to use predefined sequences or to recognise when equivalent states have been encountered [15].

Examination of the teller specification indicates that the *SupplyMoney* and *ReturnCard* events need to be reversed, implying that the money is supplied only when the card has been retrieved. There are no existing tools that allow such a change to be specified any more formally than this, except perhaps in terms of the new event sequence that a revised model should permit. Of course, a satisfactory specification should also indicate that the old sequence is no longer acceptable, that all other sequences should remain unchanged and that no new sequences should be introduced. This is the same as defining the new tree.

B. Implementing the Change

Implementing change generally means editing the model. However, refinement, through the use of correctness preserving transformations, is possible for some types of change. This is the most difficult and least investigated aspect of LOTOS support. In fact, much work in his area has primarily been directed at transforming specifications into forms suitable for use in other tools (for example, verification tools that only act on subsets of LOTOS syntax) rather than for system development *per se*. ASDE (Advanced System Design Environment) [19] provides an interactive environment for defining transformations in a suitably extended version of LOTOS and for applying them to parts of LOTOS specifications. Application of a rule is performed by selecting its template and the behavior expression (or part of one) to which it is to be applied. The system checks that the template contained in the rule matches the selected expression and that any conditions that are applicable are met. It then produces a new version of the specification as output.

The Lotosphere tool-set LITE [16] supports transformational refinement in three main ways:

1) the facilities provided by a structure editor allow the interactive transformation of parts of a specification using the analysis tools to ensure correctness;
2) the *bipartition of functionality* divides a process into two communicating sub-processes; and
3) *re-grouping of parallel processes* re-arranges the topology of processes—for example to allow for the separation of implementation concerns.

In general, however, most changes will require direct adjustment of the model.

C. Evaluating the Change

Evaluating a change means 1) verifying that the change has been implemented as intended; and 2) validating that the requirement for change was appropriate, by comparing the new model with the real world. Verification will involve the comparison of the new model with either the existing model or a specification of the required behavior of the new model.

Overall, there are four main reasons for changing a model:

1) a *corrective change*: to repair a mismatch between a model and the system it represents;
2) an *adaptive change*: to mirror actual changes to a system that have occurred or to define proposed changes;
3) a *refinement change*: to extend a model with lower level detail; and
4) a *presentation change*: to modify the appearance of a model; presentation changes typically include adjusting the layout of the model, rearranging the presentation order of components and adding comments.

The semantics of the model are changed in the first three cases but not the fourth. For example, the restructuring of the LOTOS teller model to include a cardreader process, as described earlier, is a presentation change and should not effect the behavior of the model. This can be confirmed by exhaustive comparison of the action trees for the two specifications concerned. More precisely, this means proving that the new form is *strongly equivalent* [20] to the original, i.e., that the two specifications produce the same set of event traces and, in each state, offer the same events.

An example of a refinement change would be the introduction of further internal events such as one to represent a database enquiry to determine if a PIN number was registered. This change produces a description that is *observationally* or *weakly* equivalent to the original [20]. Again the correctness of the change involved can be verified automatically although this would not guarantee the preservation of the original order of internal events. Thus, the designer might also need to see how the new tree differs from the old to ensure that the new event has been located correctly. This might be achieved by displaying the trees and highlighting their differences. Such an approach would be the main technique when dealing with corrective and adaptive changes that modify the external behavior of the model.

As mentioned earlier, another approach would be to specify a change fully in advance and then compare the new and specified models for strong equivalence. Again, however, a mechanism would be needed for reporting differences to help locate faults when a model has been modified incorrectly.

Two specifications can be compared for strong, weak and other equivalences using algorithms such as those presented in [21] and [22]. A substantial amount of memory is needed to hold an action tree for most practical specifications and this tends to limit the size of specification that can be handled. Such state limitations can be alleviated to some extent, however, by performing an equivalence comparison "on the fly" as the action tree is being constructed [23]. Aldébaran [24] verifies specifications with respect to several equivalences (strong, observational, and safety) using the Paige and Tarjan algorithm. It has also been used to prototype the "on the fly" algorithm [23]. Other verification tools include Squiggles [25], part of the SEDOS tool-set, and AUTO [26], which is now integrated as part of the LITE tool-set [16].

In summary, to facilitate change to formal models, in general, there appears to be a need to provide support for:

1) the specification of a model prior to its construction
2) the analysis of an existing model prior to its modification
3) the specification of a model change
4) the transformation of a model, preserving its semantics
5) the verification that a new model meets its specification or is equivalent, in some defined sense, to the model from which it has been derived
6) the investigation of unexpected differences between models, because of a faulty change or faulty requirement.

The next section describes an approach to providing such support.

IV. An Approach to Supporting Change for LOTOS Models

Tool design and development for LOTOS was undertaken as part of the SCAFFOLD project (Support for the Construction and Animation of Formal Language Descriptions) [13]. Its broad concern was to investigate ways of making formal descriptions more accessible and thereby encourage the wider use of formal modeling as a standard software development technique. It used LOTOS as a specific example notation. This section outlines some of the facilities that have been developed to support change to such models.

SCAFFOLD allows event-based descriptions of behavior to be expressed in the three equivalent forms discussed in earlier sections: 1) a set of event sequences; 2) an action tree; 3) a LOTOS specification. The action tree is the common conceptual representation of these descriptions but from a development point of view the event sequences and action tree are there in support of LOTOS specification construction and modification. A tool has been developed that will input pairs of descriptions in any of these forms, compare them with respect to strong or weak equivalence and report any differences that are found. For convenience in this experimental work, trees are currently represented textually. For example, the tree for the teller specification, shown in Fig. 2, would have the following form:

```
 0   AcceptCard
 1     +i:IdentifyInvalidCard
 2       DisplayRejection
 3         ReturnCard
 4           i:exit → 0
 5     +i:IdentifyValidCard
 6       RequestPIN
 7         AcceptPIN
 8           +i:IdentifyInvalidPIN
 9             DisplayRejection
10               ReturnCard
11                 i:exit → 0
12           +i:IdentifyValidPIN
13             SupplyMoney
14               ReturnCard
15                 i:exit → 0
```

The numbers down the left hand side identify the nodes in the tree. Sequences of events are indented successively to the right. Internal events have an "*i*:" prefix. A "+" before an event indicates that it is at a fork in the tree. Other branches from the same fork can be determined by looking down the same column. Any subsequent indentation to the left indicates the end of a branch. Looping behavior is marked by an arrow followed by the number of the node at the beginning of the loop.

Such trees can be constructed or modified directly using a text editor and can also be generated from a LOTOS model. The LOTOS analyser recognises simple tail recursion but other more complex forms of looping have to be identified and reported by the user. This is achieved interactively by generating the tree to some specified branch limit and then inviting the user to name any pairs of nodes between which loop connections should be made. Even with such adjustments, however, trees may be very large and so the option has been provided to prune them if necessary by indicating that further events exist along a branch but should be ignored. Where this is done the preceding event is followed by the marker ">>>". Such partial trees can then only be used as a test of a LOTOS specification rather than a full verification.

When two equivalent representations are compared the analysis will simply confirm this equivalence. If they differ, two trees are generated to explain the difference. For example, in the case of the teller machine where the *SupplyMoney* and *ReturnCard* events were reversed, the following two trees would be produced on comparing the models found at the bottom of the page.

An asterisk in one tree indicates where it differs from the other tree. This has occurred at node 13 in both cases. The following '>>>' symbol shows that there are subsequent events on each branch that have been ignored. In general, there may be several such branches identified in this way.

The basic recursive algorithm for comparing two action trees T1 and T2 for strong equivalence is as follows:

```
function Equivalent action trees (T1, T2: action tree):
  Boolean;
Determine sets of first level events(initials), I1 and I2,
  for action trees T1 and T2
{ Each event has a: name (name);
    current equivalence status (matched) - set initially
        to false;
    reference to the subtree (if any) following that
        event (subtree) }
```

Original Specification

```
 0   AcceptCard
 1     +i:IdentifyInvalidCard
 2       DisplayRejection
 3         ReturnCard
 4           i:exit → 0
 5     +i:IdentifyValidCard
 6       RequestPIN
 7         AcceptPIN
 8           +i:IdentifyInvalidPIN
 9             DisplayRejection
10             ReturnCard
11                 i:exit → 0
12           +i:IdentifyValidPIN
13             SupplyMoney * >>>
```

Modified Specification

```
 0   AcceptCard
 1     +i: IdentifyInvalidCard
 2       DisplayRejection
 3         ReturnCard
 4           i:exit → 0
 5     +i:IdentifyValidCard
 6       RequestPIN
 7         AcceptPIN
 8           +i: IdentifyInvalidPIN
 9             DisplayRejection
10               ReturnCard
11                 i:exit → 0
12           +i:IdentifyValidPIN
13             ReturnCard * >>>
```

```
for each event E1 in I1 do
  for each event E2 in I2 do
      if E1.name = E2.name then (* event names match *)
        if Equivalent action trees
         (E1.subtree, E2.subtree) then
            begin E1.matched := true;
            E2.matched := true; end;
  Compress (E1.subtree); Compress (E2.subtree);
  Equivalent action trees := matched true for every event
  in I1 and I2
```

This algorithm is a modified version of the "on the fly" algorithm described by Fernandez and Mounier [23]. Versions to compare specifications for strong, safety and observational equivalence have been developed but only that for strong equivalence is described here. Whereas, in the interests of efficient use of memory, the original "on the fly" algorithm explicitly uses stacks, the SCAFFOLD comparison tool uses the calling mechanism of the recursive procedure to store the equivalence results for successor states during a depth-first exploration of the two action trees. Testing for strong equivalence between two specifications consists of comparing each event sequence and the choices offered by one specification with those of the other. At the end of the analysis, skeleton trees for each specification will have been constructed, subject to any truncation imposed. To save memory space, state information at each node is discarded as each node is checked. Equivalent states in the two trees are marked during the analysis and so any difference can be determined by performing a further traversal of the trees, examining each state in turn.

There are several ways in which the facilities provided might be used. It is possible, for example, to work mainly with the LOTOS descriptions and use the tree representations as an evaluation or debugging aid to help understand differences between two models. Alternatively, models might be built and modified by trying to define the desired event sequences, then define a matching action tree and finally build or modify a LOTOS description. These are two extreme approaches and there are many possibilities in between. The choice may well depend on the nature of the system described. If the tree is complex, for example, then it would be preferable to first build the LOTOS model and then examine the tree it produces (It may be useful to at least prepare a few expected event sequences as tests of the developed model). Thereafter, however, it is beneficial to save the tree, after dealing with looping behavior, and have it available for editing when future modifications are required. This then would be a full specification of an intended change and make verification straightforward.

V. Conclusion

This paper has discussed the general issue of providing support for change to formal models. Requirements for such support were identified and an example of how that support might be realized discussed for the particular case of models expressed in LOTOS. Details of specific facilities developed within the SCAFFOLD project were also presented. This is a research area that has been given little attention generally and yet it is a fundamental concern for those who wish to see formal methods become an integral part of an acceptable software engineering process.

The ideas presented here will undoubtedly be refined as further experience is gained with the approach advocated. In addition, there are other aspects of the research and tool development work that need further investigation. In particular, it would be desirable to:

1) provide hypertext links between the various representations to show how they interconnect; this is a particular difficulty because of the basic mismatch between the process-based structure of a LOTOS description and the flat behavior tree;
2) provide a graphical representation for an action tree;
3) examine the approach with respect to other types of formal specification; of immediate concern is the incorporation of the data type component of LOTOS although the state explosion problems here are considerable; and
4) examine how support might be provided for changes rippling through a refinement sequence.

This last issue is a particularly difficult problem. The strategy suggested would result in the need to manage evolving versions of a specification, each made up of a refinement sequence. Changes may be started at different points in each sequence depending on the level of concern so the resulting version network is relatively complex. Fortunately, such relationships can be handled with existing configuration management techniques and research in this area into merging versions of program modules may be adaptable for use with LOTOS descriptions and action trees. A more fundamental problem, however, is identifying user needs for refinement, since currently there are no well established refinement procedures for LOTOS.

In conclusion, the facilities developed so far through SCAFFOLD, and described in this paper, seem useful and the approach advocated promises to be a significant aid to improving the efficiency and effectiveness of formal process-based modeling.

Acknowledgment

Both authors have benefited from discussions with colleagues on the SCAFFOLD project at York University and British Aerospace, namely M. Harrison, R. Took, D. Nuttall, and J. Song. Thanks are also extended to M. Norris and R. Orr at British Telecom who were involved with the initial research that led to the SCAFFOLD project.

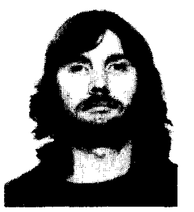
References

[1] D. R. Kuhn, "A technique for analyzing the effects of changes in formal specifications," *The Comput. J.*, vol. 35, pp. 574–578, Dec. 1992.
[2] A. M. L. de Vasconcelos and J. A. McDermid, "Incremental processing of *Z* specifications," in *Formal Description Techniques V*, M. Diaz and R. Groz, Eds. Amsterdam, The Netherlands: North-Holland, 1993, pp. 65–80.
[3] E. Brinksma, "What is the method in formal methods," in *Formal Description Techniques IV*, K. Parker and G. Rose Eds. Amsterdam, The Netherlands: North-Holland, 1992, pp. 33–50.
[4] J. Woodcock and M. Loomes, *Software Engineering Mathematics*. New York: Pitman, 1988.

[5] D. W. Bustard, M. T. Norris, R.A. Orr, and A.C. Winstanley, "An exercise in formalising the description of a concurrent system," *Software Practice & Experience*, vol. 22, pp. 1069–1098, Dec. 1992.
[6] S. Patel, R. A. Orr, M. T. Norris, and D. W. Bustard, "Tools to support formal methods," in *Proc. 11th Int. Conf. Software Eng.*, Pittsburgh, PA, May 1989, pp. 123–132.
[7] D. Whitgift, *Methods and Tools for Software Configuration Management*. New York: Wiley, 1991.
[8] I.S.O., "LOTOS—A formal description technique based on the temporal ordering of observational behavior," ISO8807, 1989.
[9] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Comput. Networks and ISDN Syst.*, vol. 14, pp. 25–59, Jan. 1987.
[10] P. H. J. van Eijk, C. A. Vissers, and M. Diaz, *The Formal Description Technique LOTOS*. Amsterdam, The Netherlands: Elsevier, 1989.
[11] R. Milner, "A calculus of communicating systems," *Lecture Notes in Computer Science*, vol. 9. New York: Springer-Verlag, 1980.
[12] C. A. R. Hoare, *Communicating Sequential Processes*, Englewood Cliffs, NJ: Prentice-Hall, 1985.
[13] D. W. Bustard and M. D. Harrison, "Animating process-oriented specifications: experiences and lessons," in *Proc. IEE Colloquium Automating Formal Methods for Comput. Assisted Prototyping*, London, January 1992.
[14] H. Alexander, "Structuring dialogues using CSP," in *Formal Methods in Human-Computer Interaction*, M. Harrison and H. Thimbleby, Eds. Cambridge: Cambridge Univ. Press, 1991.
[15] P. H. J. van Eijk and H. Eertink, "Design of the Lotosphere symbolic LOTOS simulator," in *Formal Description Techniques III*, J. Quemada, J. Mañas, and E. Vazquez Eds. Amsterdam, The Netherlands: North-Holland, 1991, pp. 709–712.
[16] P. H. J. van Eijk, "The Lotosphere Integrated Tool Environment Lite," in *Formal Description Techniques IV*, K. Parker and G. Rose Eds. Amsterdam, The Netherlands: North-Holland, 1992, pp. 473–476.
[17] A. C. Winstanley and D. W. Bustard: "EXPOSE: An animation tool for process-oriented formal descriptions," *Software Eng. J.*, vol. 6, pp. 463–475, Nov. 1991.
[18] L. Logrippo, "The University of Ottawa LOTOS toolkit," in *Formal Description Techniques III*, J. Quemada, J. Mañas, and E. Vazquez Eds Amsterdam, The Netherlands: North-Holland, 1992, pp. 689–692.
[19] G. León, C. D. Kloos, G. González, M. A. Ruz, S. Marchena, L. Santos, and J. Navarro, "ASDE: Design of a transformational environment for LOTOS," in *Formal Description Techniques II*, S.T. Vuong Ed. Amsterdam, The Netherlands: North-Holland, pp. 501–516, 1990.
[20] R. Milner, *Communication and Concurrency*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
[21] R. Paige and R. E. Tarjan, "Three Partition Refinement Algorithms," *SIAM J. Computing*, vol. 16, pp. 973–989, 1987.
[22] K. G. Larsen, "Context-Dependent bisimulation between processes," Univ.. of Edinburgh, Tech. Rep. CST-37-86, 1986.
[23] J. Fernández and L. Mounier, "Verifying bisimulations on the fly," in *Formal Description Techniques III*, J. Quemada, J. Mañas, and E. Vazquez, Eds. Amsterdam, The Netherlands: North-Holland, 1991, pp. 91–107.
[24] J. C. Fernández, "Aldébaran: A tool for the verification of communicating processes," Tech. Rep. SPECTRE c14, LGI-IMAG, 1989.
[25] T. Bolognesi and M. Caneve, "Squiggles: A tool for the analysis of LOTOS specifications," in *Formal Description Techniques*, K. J. Turner, Ed. Amsterdam, The Netherlands: North-Holland, 1989, pp. 201–216.
[26] E. Madelaine and D. Vergamini, "Tools for process algebras," in *Formal Description Techniques IV*, K. Parker and G. Rose, Eds. Amsterdam, The Netherlands: North-Holland, 1992, pp. 463–466.

David W. Bustard received the B.Sc. degree in physics and M.Sc. and Ph.D. degrees in computer science from Queens University, Belfast in 1971, 1973, and 1980, respectively.

He was a Visiting Research Fellow at the British Telecommunications Research Laboratories at Martlesham, England in 1989 and a Visiting Scientist at the Software Engineering Institute at Carnegie Mellon in 1990. He held various posts at Queens University from 1974 to 1989 before taking up his current post of Professor of Computing Science at the University of Ulster in 1990. His current main area of interest is requirements engineering. He also leads the Requirements Definition Research Group at the University of Ulster.



Adam C. Winstanley received the B.A. degree in archaeology from the University of Cambridge in 1978. After several years working as an archaeologist and cartographer with the Ordnance Survey of Northern Ireland, he received the M.Sc. degree in computer science from Queen's University, Belfast in 1987, followed by a Ph.D. degree in 1992.

He was a research officer on the *Scaffold* project in the Department of Computing Science, University of Ulster between 1990 and 1993 before returning to Queen's University as a temporary lecturer in Computer Science. He is currently a research fellow in the Artificial Intelligence Research Group at Queen's University with interests in AI applications for geographical information systems and quantitative problem solving.