Kenneth J. Turner. Incremental Requirements Specification with LOTOS. Requirements Engineering Journal, 2:132-151, Springer-Verlag, London, UK, November 1997.

Incremental Requirements Specification with LOTOS

Kenneth J. Turner
Department of Computing Science and Mathematics
University of Stirling, Stirling FK9 4LA, UK
Email kjt@cs.stir.ac.uk

1st June 1999

Abstract

The importance of formally and incrementally specifying requirements is discussed. An approach based on LOTOS (Language Of Temporal Ordering Specification) is proposed that exploits desirable characteristics of the constraint-oriented style. The nature of constraint-oriented specification is discussed at some length, and guidelines for how to use it effectively with LOTOS are presented. Small introductory examples lead to the incremental specification of a file access system using the approach in the paper. It is shown how the requirements for the file access system can be gradually formalised, leading to a complete system specification.

Keywords: constraint, formal method, LOTOS (Language Of Temporal Ordering Specification), requirements, specification

1 Introduction

1.1 Requirements Specification

Requirements specification is a crucial activity in system development, following on from requirements capture and analysis. This is the point at which the client's requirements are defined as the basis for system specification and later development. Experience teaches that it is very desirable to specify requirements accurately and completely [1]. If an error in requirements is allowed to propagate through development, it may cost a hundred to a thousand times more to correct the error if it is discovered only at a late stage [2, 3]. A requirements specification may also be the basis of a contract between the client and the developer. Failure to meet an inadequate specification may therefore have legal as well as financial implications; it may not be clear whether the client or the developer is at fault.

It is therefore hardly surprising that formal methods have been put forward as a valuable tool in requirements specification. Formal methods offer precision of expression and the possibility of rigorous analysis. Whatever the real (or perceived) cost of formal methods, the benefits of catching errors early can easily outweigh the cost of applying formal methods [4, 5]. For safety-critical or quality-critical applications, the cost of formality may be only a minor factor. Indeed, for certain types of application (e.g. in defence [6, 7]), the use of formal methods may be mandatory.

Unfortunately, formal specification of requirements suffers from two major problems: the incomprehensibility of a formal specification to a typical client (or developer!), and the need to balance the rigour of a formal method against the inherently informal process of requirements capture and analysis. This paper describes a possible solution using LOTOS (Language Of Temporal Ordering Specification [8]) to formally specify functional requirements. Non-functional requirements such as performance are also important, but they can be dealt with separately (e.g. [9]) and are not part of the method described here. Typically, non-functional aspects influence the design process (e.g. a time constraint might affect the choice of algorithm) [10]. Although much of the complexity in current system design deals with functionality rather than qualitative aspects, only some designs that are functionally correct may meet the non-functional requirements.

Notation	Meaning
(* text *)	a comment
stop	a behaviour that does nothing (no further action)
exit	a behaviour that immediately terminates successfully
exit (results)	successful termination with result values
gate	a 'port' at which event offers may synchronise
gate ! value	an offer to synchronise on a given value
gate ? variable : sort	an offer to synchronise on any value of the given sort, binding the actual value to the given variable name
gate ! ... ? ... [predicate]	an event offer with a predicate on values synchronised
process name [gates] (parameters) : **noexit** := behaviour	a named process abstraction with given gates and value parameters, but no termination (e.g. it repeats indefinitely)
process name [gates] (parameters) : **exit** (results) := behaviour	a process that terminates successfully with the given result sorts
name [gates] (parameters)	an instantiation of a named process
offer ; behaviour	prefixes an event offer to some behaviour ('followed by')
[guard] $\Rightarrow$ behaviour	offers behaviour only if the guard condition is satisfied ('if')
behaviour1 $\Box$ behaviour2	offers a choice between two behaviours ('or')
behaviour1 $\gg$ behaviour2	allows the second behaviour to occur if the first behaviour terminates successfully ('enables')
exit (results) $\gg$ **accept** declarations **in** behaviour	successful termination with export of result values
behaviour1 $\rhd$ behaviour2	allows the second behaviour to disrupt the first behaviour unless this terminates successfully first ('disabled by')
behaviour1 || behaviour2	allows two behaviours to run in parallel, but fully synchronised on their events ('synchronised with')
behaviour1 ||| behaviour2	allows two behaviours to run in parallel, but with independent occurrence of their events ('interleaved with')
behaviour1 |[gates]| behaviour2	allows two behaviours to run in parallel, synchronising on all events at the given gates ('synchronised on *gates* with')

Table 1: Selected LOTOS Syntax

1.2 LOTOS

LOTOS is an internationally standardised FDT (Formal Description Technique). To describe behaviour, LOTOS uses a process algebra based on CCS (Calculus of Communicating Systems [11]) and CSP (Communicating Sequential Processes [12]). LOTOS has an integrated abstract data type language based on ACT ONE [13]. For readers not familiar with LOTOS, a summary of selected syntax is provided in table 1. Tutorials on LOTOS are found in [14, 8, 15].

The LOTOS notation for processes involves a lot of syntax so that proper type-checking can be carried out. To avoid such syntactic details getting in the way, a simplified notation is used in this paper. The simplified syntax is incomplete, but the translation to full LOTOS syntax is straightforward. As an example, consider a process that multiplies natural numbers (non-negative integers) by a fixed factor given as a parameter. The process repeatedly reads a number and then outputs that number multiplied by the factor. The simplified syntax:

```
Multiplier (factor) :=
  read ? number : Nat;
  write ! number * factor;
  Multiplier (factor)
```

translates into full LOTOS syntax as:

process Multiplier [read, write] (factor : Nat) : **noexit** :=

```
    read ? number : Nat;
    write ! number * factor;
    Multiplier [read, write] (factor)
endproc
```

LOTOS is a flexible language that can be used in many different ways. Considerable effort has been expended on developing specification styles with well-defined characteristics (e.g. [16]). Since LOTOS contains a data sub-language and a process sub-language, either of these may be emphasised according to the nature of a specification (e.g. data for an information model, or processes for a behavioural model). Behavioural styles are often characterised by the set of operators used to combine behaviour expressions. An object-based analysis precedes the approach reported in this paper. One of the more popular and successful LOTOS styles is constraint-oriented. This is at the heart of the method reported here, and is discussed in some detail in section 2.1. Other behavioural styles include those termed resource-oriented, state-oriented and monolithic.

1.3 Related Work

Requirements capture and specification methods range from the informal (but systematic) to the formal (or rigorous). Requirements capture techniques have existed for some time in informal analysis methods such as CORE (Controlled Requirements Expression [17]) and SADT (Systems Analysis and Design Technique [18]). Because requirements capture is essentially an intuitive task, there has generally been little formality. Instead, efforts have been made to systematise the process. Use cases (e.g. [19]), scenarios (e.g. [20, 21]) and viewpoints (e.g. [22, 23, 24]) are all popular methods for organising requirements capture.

In the field of formal methods there is surprisingly little literature that explicitly mentions formalising requirements. This is probably because formal specification languages are normally at a high level of abstraction. Formal specifications are therefore usually suitable as requirements specifications. Popular formal methods such as B (e.g. [25]), RAISE [26], VDM (e.g. [27]) and Z (e.g. [28]) would no doubt claim to tackle requirements specification. However as noted in [29], there is a general lack of guidance for producing an initial formal specification from requirements. [30] advocates formalising critical requirements first, then gradually refining the specification until a high-level language description can be written. The idea of scenarios has been used to help formalise requirements [31, 32]. A particular issue is how formal specifications can be kept up to date with changes in requirements [33, 34].

Little work appears to have been undertaken specifically on requirements capture and specification with LOTOS. A rigorous object-oriented approach using LOTOS is reported in [35]. The work on animation of LOTOS specifications [36, 37] is useful for evaluating formalised requirements and agreeing them with clients. The work in [38] is complementary to this paper since it emphasises tracking the evolution of requirements. A range of specification styles has been defined for LOTOS [16]. As argued in this paper, the constraint-oriented style appears to be particularly suitable for requirements specification.

1.4 Structure of the Paper

Since writing a requirements specification needs guidance and experience, the style of this paper is an expository 'how to' guide rather than a theoretical treatment. Section 2 describes and discusses the approach proposed in this paper. Initially, the two most relevant specification styles (object-based and constraint-oriented) are described, then the overall approach is presented. Section 3 illustrates the approach by progressively and incrementally developing a LOTOS specification. The example describes a file access system [39] such as might be found in an operating system. The inspiration for this example came from the EDS (European Declarative System) project. In [39], temporal logic is used to describe the example and offers an interesting comparison with the approach reported here. An informal, object-based description is given first in this paper, then the LOTOS specification is developed using the proposed approach. Although the example is relatively small, it nonetheless embodies many of the issues that arise when specifying more realistic systems.

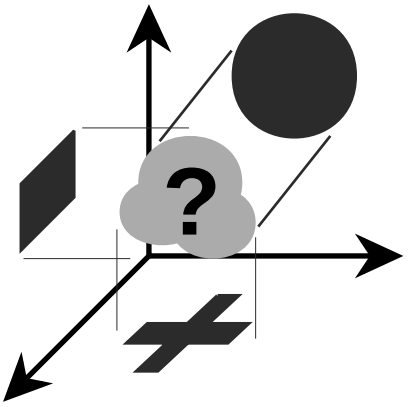

Figure 1: Constraints as Projections onto Planes of Abstraction

2 Specification Approach

2.1 Constraint-Oriented Specification in LOTOS

2.1.1 Constraints in General

The constraint-oriented style in LOTOS [16] has been widely used in the specification of standards. This is largely because the style permits fairly abstract specifications that focus on externally observable behaviour. The constraint-oriented style has perhaps been used too obsessively, which is unfortunate since it is rather distant from an implementation. Nonetheless the style is particularly suitable for specifying requirements in an incremental and compositional way. Although the constraint-oriented style is well known to practitioners, its philosophy has never been fully elaborated. The literature contains *examples* of the constraint-oriented style, but there is a lack of published *method*; the paper attempts to fill this gap. Newcomers to the style often find it strange. Some space is therefore taken here to explain the thinking behind the constraint-oriented style.

The high-level behaviour of a system can often be described from a number of largely independent viewpoints. For example, the Reference Model for ODP (Open Distributed Processing [40, 41, 42]) adopts five viewpoints for the description of a distributed system. At a high level, system behaviour in a viewpoint may be characterised by the set of rules that define it. Frequently these rules can be interpreted as constraints. A constraint may be visualised as the projection of system behaviour onto some plane of abstraction. The overall system behaviour is specified implicitly by its individual projections. Figure 1 illustrates this in a geometrical sense; is there a solid object that has these projections?[1] A low-level view of behaviour is usually rather operational and does not lend itself so well to definition by constraints; instead a state-oriented or monolithic description may be more appropriate.

A set of constraints can be imagined as placing restrictions on otherwise free behaviour. In fact, a constraint often takes the form:

> **if** some condition or state applies
> **then** behaviour is restricted according to the constraint
> **else** behaviour is unconstrained

Section 2.1.3 discusses how constraints may be expressed in LOTOS.

A constraint often delimits valid behaviour within the space of all permissible behaviours. Constraints can thus narrow the range of behaviour within their own particular scope. Since constraints represent different aspects of behaviour, the intersection of these valid behaviours represents the allowed overall behaviour. This idea is represented in the alternative graphical view of constraints given by figure 2. Of

[1] There is indeed a solid object that projects onto three orthogonal planes as a cross, a square and a circle. If necessary, see [43, Puzzle 177] for the answer.

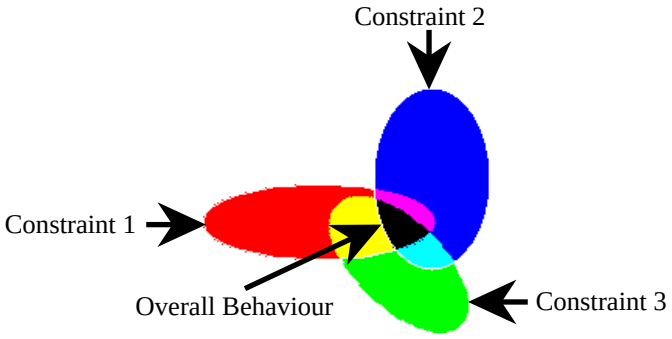

Figure 2: Overall System Behaviour as the Intersection of Constraints

course if constraints were mutually contradictory, then the intersection would be empty and no system behaviour could satisfy the constraints. Constraints can prohibit behaviour by explicitly excluding it, or can permit behaviour by explicitly including it. Note that a constraint cannot by itself force behaviour since another constraint may prohibit it. Behaviour becomes an obligation only when the overall constraints leave no alternative.

An important issue is that constraints must synchronise and should therefore share a common event structure. This is not essential, but a regular event structure makes it easier to add in further constraints and reduces the risk of inadvertent deadlock. The event structure to use may not be evident at first, and may in fact evolve as the specification is made more elaborate. The approach of the paper makes this explicit, allowing parameters to be added to events as the need for finer distinctions grows during elaboration of the specification.

Specifying a system through parallel composition of constraints may introduce artificial or illusory parallelism. Certainly there is parallelism among the constraints, but the overall behaviour that is specified may actually be sequential. (This is true, for example, of the specification in the next subsection.) For this reason, a constraint-oriented specification needs considerable refinement before a specification suitable for implementation can be derived. However, the key issue is capturing requirements effectively in a specification, not producing an implementation structure. Refining constraint-oriented specifications towards an implementation has been studied elsewhere [44, 16].

Although constraints are often conjoined (intersected, composed in parallel), they are also commonly disjoined (unioned, composed through choice) where there are alternative behaviours. Whereas conjunction may be used to progressively narrow down the required behaviour, disjunction may be used to extend existing behaviour. The constraint-oriented style thus mainly makes uses of ‘and’ and ‘or’ combinations.

The composition of constraints does not guarantee global properties of the specification such as freedom from deadlock. It is necessary to analyse a specification to determine whether such properties hold. However, no specification style can ensure that only correct specifications are written, so it should not be surprising that such analysis is required. A more subtle problem is that the behaviour of a constraint-oriented specification is in a sense the largest possible, i.e. the least constrained. As a result, undesirable behaviour may accidentally be left in the specification through not ruling it out. However, in the author’s experience it is more likely that the specification be made overly restrictive rather than overly loose. Constraining behaviour too far usually leads to deadlocks that are quickly detected during analysis.

2.1.2 A Simple Constraint-Oriented Example

An everyday example is given below to illustrate the constraint-oriented style. Consider the conventional behaviour associated with eating meals. Natural language statements can be misleading since they may hide other assumptions. For example, the statement ‘breakfast is followed by lunch’ implicitly refers to all such meals and not just one of each. A specifier cannot therefore naively formalise natural language requirements; it is necessary to go beyond the surface meaning.

Separate constraints will be given for parts of the overall behaviour of eating meals. Simple LOTOS events such as *breakfast* will correspond to eating a meal. The constraints and their formalisation in LOTOS are given in the following. Since the natural language formulation implies repeated behaviour, the constraints are recursive. In fact constraints are normally defined recursively since they usually apply indefinitely. In each case below, a statement in English is followed by its equivalent in LOTOS.

Constraint1 constraints on eating:

Constraint1A breakfast is followed by lunch:

Constraint1A := breakfast; lunch; Constraint1A

Constraint1B lunch is followed by dinner or (high) tea:

Constraint1B := lunch; (dinner; Constraint1B [] tea; Constraint1B)

Constraint2 constraints on eating in relation to being awake:

Constraint2A waking is followed by breakfast:

Constraint2A := waking; breakfast; Constraint2A

Constraint2B dinner or (high) tea is followed by sleeping:[2]

Constraint2B := dinner; sleeping; Constraint2B [] tea; sleeping; Constraint2B

Constraint3 constraints on being awake: waking is followed by sleeping:

Constraint3 := waking; sleeping; Constraint3

To anticipate the fuller use of this approach later in the paper, a mildly hierarchic breakdown of constraints is given in the description above. Each of the constraints is a simple and separate statement. However, the events that they refer to are not independent – there is partial overlap between the constraints. Constraints may be put together in LOTOS by one of the parallel operators: || for constraints that have to agree fully; ||| for constraints that are completely independent; and |[...]| for constraints that have to agree on the common events (gates) listed in the brackets. Composing constraints is a relatively straightforward matter of choosing the correct parallel operator. For the constraints above, the overall behaviour is given by:

```
  (Constraint1A |[lunch]| Constraint1B)
|[breakfast, dinner, tea]|
  (Constraint2A ||| Constraint2B)
|[waking, sleeping]|
  Constraint3
```

The constraint-oriented style has the strange property that specifications are mainly developed top-down but understood bottom-up. The top-down approach follows from progressively decomposing the constraints on the system until they are self-contained and manageable. However, this may result in the definition of behaviour being spread out across the whole specification text. In general it is not possible to take just one piece of such a specification and learn all there is to know about a specific aspect of behaviour. (Of course, if that aspect is independent of others then this is possible with the constraint-oriented style.) Bottom-up understanding comes from taking the specification piecewise and confirming the meaning of each piece. If there is understanding of each piece and how the pieces fit together, then there is understanding of the whole. A holistic view is necessary because constraints might limit each other by placing additional restrictions.

At first this may seem a difficult and obscure way to write specifications. However, this very approach is well accepted in the engineering disciplines. The idea might be termed component-based and property-based specification: a sound system can be structured from known components combined in known ways. This idea is elaborated in [45], where it is shown how a component-based method for specification can be

[2] Alternatively, the English description could be reflected more directly in LOTOS as:

Constraint2B := (dinner; **exit** [] tea; **exit**) >> sleeping; Constraint2B

made to work at an abstract level (for telecommunications services) and at a concrete level (for digital logic design).

Specification components are re-usable fragments of specification that embody domain-specific knowledge. For telecommunications, specification components could include services, service access points, multiplexers and protocols. For digital logic, specification components could include basic logic gates, flip-flops, code converters and adders. Specification components can be specialised according to the required properties. For example, a telecommunications service might be required to offer connections, reliable delivery and an expedited data facility. A digital logic component might have to exhibit certain timing characteristics or fan-out.

Note that the effect of given components may be modified by their combination with other constraints. This may be used as a deliberate mechanism to specialise (i.e. restrict) the behaviour of given components or to extend their behaviour. The given components still work as expected, but are modified in the overall context of the system. This might be regarded as undesirable since supposedly known components cease to behave as expected. In fact this situation arises frequently in engineering. For example, an electronics engineer may place a resistor and a capacitor in parallel. Each component continues to fulfill its function, but the combination is intentionally far from that of pure resistance or pure capacitance.

The constraint-oriented style is reminiscent of axiomatic specification: each constraint is like a logical statement about the system. Constraints are combined using the LOTOS equivalent of logical 'and' (the parallel operators) and logical 'or' (the choice operator). The effect of logical 'not' is achieved using guards that forbid certain behaviour. The constraint-oriented style is compositional: the overall specification is the hierarchical composition of progressively smaller constraints. The high-level nature of the constraint-oriented style makes it very suitable for specifying requirements; the compositional nature makes it very appropriate for incremental specification. As a matter of specification philosophy, it is desirable to avoid overlaps among constraints. This requires each logical constraint to appear just once in a specification. To achieve this requires discipline on the part of the specifier – each part of a specification should deal with a distinct aspect of the problem.

The constraint-oriented style is able to accommodate a number of other needs in requirements specification. Clients are usually not accustomed to formulating their requirements in a complete and well-defined manner. The systems analyst frequently has to tease out the requirements during a number of discussions with the client. The intention of the approach described in this paper is that as a specification evolves it can be explored with the client. Since the specifications resulting from the approach are executable, exploration means that the analyst uses the specification as a prototype and 'walks' the client through its behaviour. In particular, execution scenarios (e.g. use cases [19]) are useful to confirm that the system behaves as the client might expect.

Even once the requirements have apparently been specified, later requests for enhancements may lead to changes to the specification. These considerations mean that requirements are often built up gradually and incrementally, for which the constraint-oriented style is suitable. Client requirements can often be broken down in a hierarchical way as in Structured Analysis (e.g. [46]), again in sympathy with this style. It is also common to find that requirements can be be framed in situational terms: 'in this situation, that must (not) happen', or 'when this happens, it is followed by that'. The constraint-oriented style offers a natural way to reflect such characteristics.

2.1.3 Expressing Constraints in LOTOS

As in the example of eating meals, constraints may simply restrict the order of event offers using only gates. More typically, constraints apply to structured events and restrict the values associated with these.

As an example, suppose that a computer register holds a fixed point integer as a scale factor (a power of two) and a signed value. Operations on the register include *scale* (to set the scale factor), *add* (to increase the value) and *sub* (to decrease the value). For example purposes only, it is required that the scale factor and the amount that is added/subtracted be non-zero.

Operations on the register could be modelled as interactions at separate gates:

```
scale ! 1                                   (* set scale to 1 *)
add ! x                                     (* add x to register *)
```

There are two difficulties with this approach. If the specification has to be extended to allow multiple registers, each would require its own set of gates: *scale1*, *add1*, *scale2*, *add2*, etc. Also in order to decompose the specification, it would be desirable to give the constraints on each operation separately. In such a case, each constraint would have to allow other operations to take place without restriction. This would require an explicit list of irrelevant events. For example, the constraint on setting the scale factor might have to say:

```
RegScale :=
  scale ? value : Int [scale ne 0]; RegScale          (* constrains non-zero scale *)
[]
  add ? value: Int; RegScale                                  (* unconstrained *)
[]
  ...
```

If all the register constraints are to be synchronised in the final specification, *RegScale* must be prepared to allow events unconnected with scaling. This would rapidly become unworkable as the number of operations increased.

Instead it is better to treat the operation as a parameter of events, simply being names in some sort *Op*. All register operations can then take place at some gate *Reg*. A register number can easily be added at the same time, giving events of the form:

```
Reg ! 2 ! scale ! 1                          (* register 2 has scale set to 1 *)
Reg ! 5 ! add ! x                              (* register 5 has x added *)
```

The 'else' part of a constraint can now be wrapped up in a single condition:

```
RegScale :=
  Reg ? reg : Nat ! scale ? value : Int [value ne 0]; RegScale          (* non-zero scale *)
[]
  Reg ? reg : Nat ? op : Op ? value : Int [op ne scale]; RegScale       (* unconstrained *)
```

Such a specification is somewhat clumsy because the 'else' part needs to be given explicitly and does not add much to the readability. A neater solution is to take advantage of the boolean implication operation. This evaluates to *true* if its first condition is *false* and so places no restriction on the event offer:

```
RegScale :=
  Reg ? reg : Nat ? op : Op ? value : Int [(op eq scale) implies (value ne 0)]; RegScale
```

Unfortunately this simplification is not always possible, particularly where the behaviour following the event offer depends on which operation occurs.

Since constraints are usually synchronised, it is highly desirable to adopt a uniform structure for events. This may force the introduction of dummy parameters for some operations. A common solution, not required in this example, is to combine an operation with its parameters:

```
Reg ! 2 ! scale (1)                          (* register 2, scale 1 operation *)
Reg ! 5 ! add (x)                            (* register 5, add x operation *)
```

The constraints so far were designed to be used in a context where they are fully synchronised:

```
Reg := RegScale || RegAdd || ...
```

However, this requires every constraint to synchronise on every event offer – even those that are 'don't care' offers as far as the constraint is concerned. Where possible, it is better to decompose the constraints such that they can be interleaved:

```
Reg := RegScale ||| RegAdd ||| ...
```

The freedom to use pure interleaving is not so common in practice because constraints are often interdependent. The ideal of interleaving allows each constraint to describe only one operation without worrying about others, for example:

```
RegScale :=
  Reg ? reg : Nat ! scale ? value : Int [value ne 0]; RegScale
```

Achieving this depends on a clean breakdown of the constraints. This may not be possible if a particular decomposition strategy is desired. There may also be interdependencies among constraints that prevent interleaving.

Difficulties can arise when constraints are associated with state variables. So far only the constraints on operations have been given, but to specify the full behaviour requires state variables. For the computer register these are the current scale factor and the current value. Since these are independent, separate constraints can be given for the *scale* and *add* operations:

```
Reg := RegScale (0) ||| RegAdd (0)                          (* interleaved constraints *)

RegScale (factor) :=                                        (* non-zero scale *)
  Reg ? reg : Nat ! scale ? value : Int [value ne 0]; RegScale (value)

RegAdd (amount) :=                                          (* non-zero addition *)
  Reg ? reg : Nat ! add ? value : Int [value ne 0]; RegAdd (amount + value)
```

If a *sub* operation were now introduced, this approach would no longer be viable since both *add* and *sub* operate on the same state variable (the register value). To solve this problem needs constraints as terminating processes that exit with their current value. These have to be combined using [] by a controlling process:

```
Reg := RegScale (0) ||| RegValue (0)                        (* interleaved constraints *)

RegScale (factor) :=                                        (* non-zero scale *)
  Reg ? reg : Nat ! scale ? value : Int [value ne 0]; RegScale (value)

RegValue (amount) :=                                        (* disjoined constraints *)
  (RegAdd (amount) [] RegSub (amount)) >> accept new_amt : Int in RegValue (new_amt)

RegAdd (amount) :=                                          (* non-zero addition *)
  Reg ? reg : Nat ! add ? value : Int [value ne 0]; exit (amount + value)

RegSub (amount) :=                                          (* non-zero subtraction *)
  Reg ? reg : Nat ! sub ? value : Int [value ne 0]; exit (amount - value)
```

2.2 An Approach to Incremental Requirements Specification with LOTOS

The approach is intentionally quite general so that it can be used for a variety of applications. For example it is relevant to the specification of communications systems [45], distributed systems [47], Intelligent Network services [48] and digital logic [49].

The work on requirements specification to be described is better termed an *approach* rather than a *method*, since a set of principles rather than a definitive procedure is proposed. The expression of constraints in LOTOS should follow the suggestions given in section 2.1.3. Logical steps are given below, though these are not applied in a strictly sequential way. Of course this is to be expected since requirements capture is not a linear process. Steps may have to be re-applied as new requirements are discovered. The steps are numbered (S_n) for later reference.

A high-level description of the steps is given in figure 3. The client is involved during most stages. Initially, the analyst uses client input to build an object model (step S_1). Information from the client about how the system should work allows behavioural constraints to be defined (steps S_2 and S_3). Completion of the specification (steps S_4 and S_5) is a purely technical activity that probably does not require the client. The complete specification is evaluated by the client and analyst, using simulation or prototyping. This is likely to identify deficiencies in the way the requirements have been captured (step S_6). These will cause the object model or the specification to be modified (step S_7).

(S_1) The first step is to create an object model of the system. Virtually any object-based or object-oriented analysis can be used, provided that it identifies objects, methods and communication between objects.

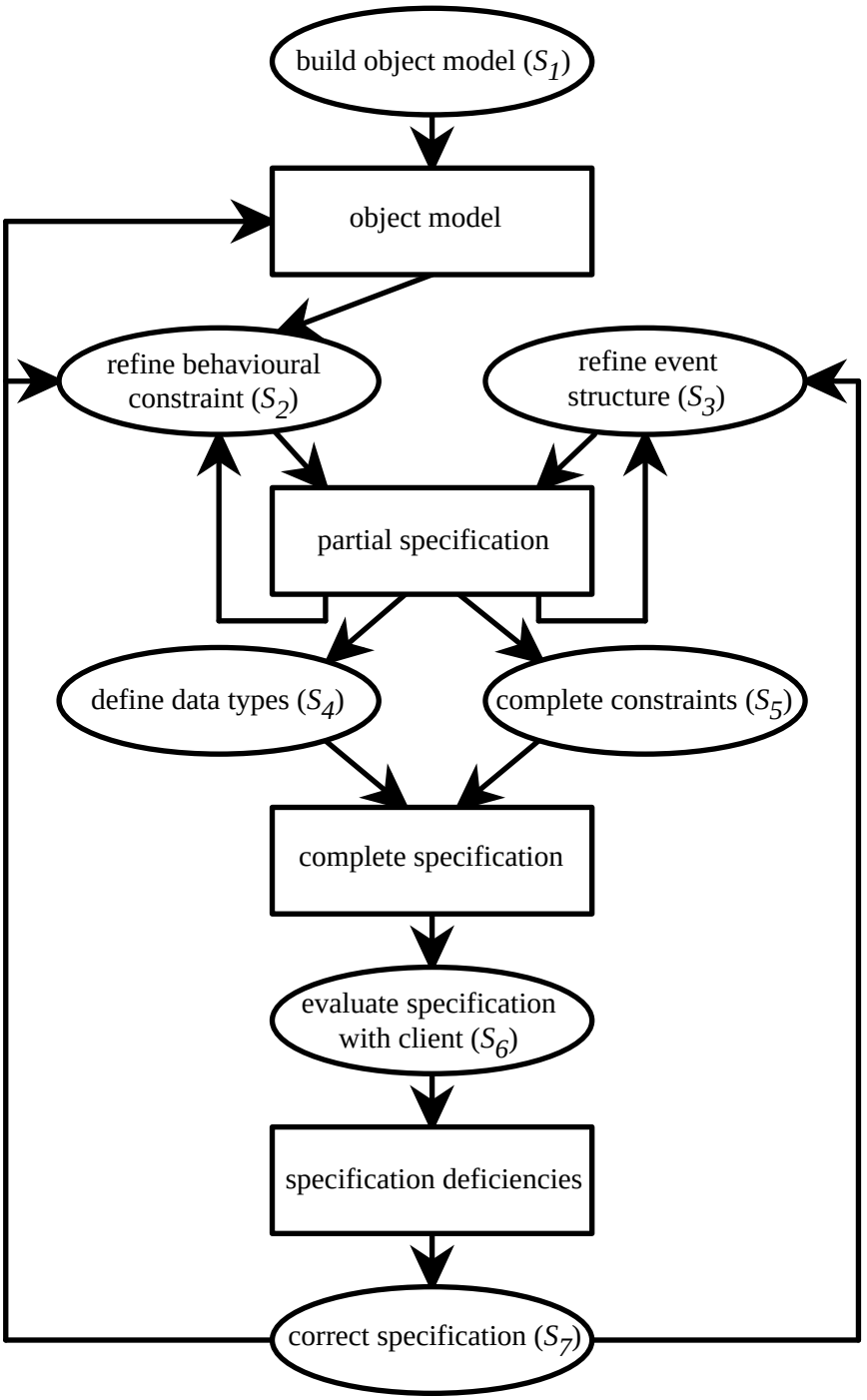

Figure 3: Steps in Requirements Capture and Specification

The analysis technique is not defined by the approach, and could be any standard one such as OMT (Object Modelling Technique [50]).

(S_2) The behaviour of the system is the combined behaviour of its objects as they communicate. The rules for this communication should be thought of as a collection of constraints. The aim should be to limit all permissible behaviours to those that are valid for the constraint under consideration.

Constraints should be identified in a top-down fashion, progressively refining them until they can be specified in a self-contained manner. Note that decomposition may not mean strict partitioning, i.e. separation into fully independent concerns. Lower-level constraints may each give only part of a higher-level constraint. The different types of constraint described below suggest different refinement strategies. A single strategy is often used at any one level, but the strategy may change from level to level. For example, a viewpoint constraint may develop into functional constraints and then into temporal constraints as the level of detail increases.

In step S_2, the constraints to be considered depend on the nature of the system. The following suggests some types of constraint that might be introduced during analysis of requirements. The categories of constraints are not sharply defined and may overlap; the types of constraint are lettered for later reference.

(a) Assertional: Such constraints make logical statements about the system in an axiomatic style. For example, it might be convenient to think of a sorting algorithm as respecting the assertions 'output is a permutation of input' and 'output values are in ascending order'.

(f) Functional: Such constraints reflect the behaviour of individual functions within a decomposition of the system. A compiler, for example, might be defined by constraints on functions such as lexical analysis, parsing, code generation and optimisation.

(i) Informational: Such constraints reflect the operations on information structures and their invariants. A database, for example, might have constraints on the integrity of its data.

(m) Morphological: Such constraints reflect structural relationships among parts of the system. These might resemble an object model, but other kinds of structure may be given. For example, a digital logic design might have constraints on fan-in and fan-out of gates.

(s) Situational: Such constraints make rule-based statements about system behaviour in particular states or under particular conditions. A bounded buffer, for example, might be constrained when full to reject new input until some space has become available.

(t) Temporal: Such constraints reflect phases of behaviour or relative ordering of events. For example, separate constraints could be given for the phases of a communications connection: establishment, maintenance and release.

(v) Viewpoint: Such constraints reflect different perspectives on the system. For example they might deal with business aspects (how a manager sees the system), distributed computation (how the system is networked) and technology issues (how the system is implemented) [51].

(S_3) Event structures in the specification should be allowed to evolve as increasing detail dictates. In practice this means that more parameters may be required in events as the level of magnification is increased. Adopting this approach means that a simplified view of the system can be taken initially. Such a view is likely to emphasise the major features of the system; dealing too early with detail and deviations from the ideal is likely to divert attention from the key issues.

In step S_3, event structures might be refined to deal with the following aspects; the refinements are lettered for later reference.

(e) Event Ordering: This defines behaviour by giving a partial ordering of events. The event gate and selected event parameter values appear in events. As an example, addition to a register would require behaviour that adds the destination register to the source register.

(g) Gate Introduction: This introduces new gates to deal with constraints among processes (objects). The gates allow the processes to synchronise (communicate). The analogy here is with identifying the interfaces between components.

(p) Parameter Introduction: This introduces new event parameters. The permitted values for an event parameter are constrained irrespective of other behavioural issues. For example, a register number might always have to be in the range 0 to 15 for any register operation.

(r) Relationship between Parameters: This defines interdependencies among permitted values for event parameters. A register transfer operation, for example, may not be allowed if the source and destination registers are the same.

(S_4) Data specification should be deferred as far as possible in order to concentrate on behavioural aspects. If constraint decomposition identifies the need for some data item, it should be specified in outline terms initially. For LOTOS this means giving a type definition with operation signatures only. The equational definitions for operations can be delayed until the rest of the behavioural specification is complete. The only exception would be a complex operation that it would be wise to specify before proceeding further.

(S_5) As requirements are progressively identified in constraints, it may be necessary to modify the definition of constraints that have already been identified. Usually such retrospective changes are straightforward and do not significantly affect the previous formulation. There is a choice of updating the specification as it evolves or applying all such changes as a final step. The following kinds of change may be required; they are lettered for later reference.

(c) Combination of Constraints: If the final set of constraints were specified individually, this would lead to a rather flat structure for the specification. Instead it may be desirable to combine related constraints. This effectively composes constraints in the opposite order to which they were identified during decomposition of requirements. All constraints for a particular register operation, for example, may be combined as a matter of good specification structure.

(n) New Event Parameters: Adding new event parameters means that the partial specification written so far will have to be reworked. Fortunately the changes usually involve straightforward addition of 'don't care' event parameters that are irrelevant at the higher levels. As an example, recognising that operations on registers need to specify the register number would cause this parameter to be added to register events. This would affect higher-level constraints through addition of a register number that was not initially identified.

(w) Write out Syntax: To make the specification syntactically correct, a certain amount of rubric has to be added. For convenience, a simplified process notation may have been used as suggested in section 1.2. A fairly mechanical step is then required to write out the full syntax of the specification.

(S_6) Once the specification of requirements has been completed, it can be subjected to the usual checks and can be evaluated by the client. Since LOTOS is used to create an executable specification, this can be used as a high-level prototype. Simulation may be interactive, or automatic to a certain search depth (or until recursion or termination is detected). Using standard simulation techniques and scenarios (e.g. use cases [19]), the analyst can take the client interactively through sequences of operations. The goal of this activity is to check that the client's requirements have been properly specified. The outcome is a list of deficiencies in the requirements specification.

Simulation is a fairly effective way of communicating a requirements specification to a client (or developer). It is useful to individually check the constraints captured in the specification before the overall system behaviour is evaluated. It is often convenient to set up a 'test harness' – an environment specified in LOTOS that exercises the rest of the system. With a little care, simulating tests can be virtually automatic, producing a simple pass/fail verdict.

A disadvantage of standard LOTOS simulators is that they are usually oriented towards the specifier and assume detailed language knowledge. Work has been undertaken by the author and his colleagues on tools for visual animation of requirements translated into LOTOS [52, 53, 54, 36]. There is also other work by others on animation of LOTOS [55, 37]. The uniform event structure in constraint-oriented specifications makes (visual) animation particularly easy.

The requirements specification is, of course, open to all the standard techniques and tools for formal manipulation and analysis of LOTOS (e.g. see the collected papers in [56]). LOTOS is rich in verification theory and tools, much of which is shared with other process algebras. Checking specification equivalences and relations is particularly relevant when the specification is to be transformed into some more implementation-oriented form. This may use correctness-preserving transformations,

architecturally-driven transformations that preserve some implementation relation, or pre-defined implementation constructs. Specification properties may be formulated in some modal logic so that a model-checker can be used to check their validity.

(S_7) Any deficiencies found in the formalised requirements are used to correct the specification. This may require the object model and/or the behavioural constraints to be changed. Since the specification is highly structured, changes will hopefully be localised to the constraints that are incorrect. Even if a more serious problem is found, e.g. a missing object or interface, it should be possible to preserve the constraints on indirectly related parts of the specification.

3 An Example of Incremental Specification

3.1 Information Problem Description

As a realistic example of how requirements can be incrementally formalised in LOTOS, the file system described by Sa and Warboys in [57, 39] will be used. [58] provides more information on the methodology used in [39]. In fact this example more accurately concerns file access methods; the description and terminology of the original will be followed though it is not ideal. The specification structure here will also faithfully reflect the original, though the analysis is not always consonant with the constraint-oriented style. A cleaner decomposition of requirements would have been preferable. Still, Sa and Warboys claim that their requirements are as might be stated by a client. If the analyst re-organises requirements in a better way, it may become difficult for the client to relate to them. There is some merit in staying close to the client's statement of requirements even if it complicates the specification structure.

Following the description in [57], it is supposed that application programmers wish to access files using a collection of pre-defined methods. Figure 4 shows the relationship between the file system objects graphically. The file handler *FH* supports the basic methods of *open*, *read*, *write*, *save* and *close*. However, two further tasks (composite access methods) are provided for more convenient use by the programmers. Task *T1* presents the programmer with three higher-level operations: *op1* opens a file; *op2* supports a read-after-write operation for reliability, checking that the file contents after writing are the previous contents with the write data appended; *op3* closes a file. Task *T2* offers a single operation *op0* that merely writes to the file. Although this second task adds little to the example, it is presumed to be a useful abstraction. An application program is free to call the operations of *FH*, *T1* and *T2*. Since concurrent execution of these operations is possible, the system must protect against interference between them. Section 3.5 describes these requirements in detail.

The object model for the file access system is obvious: the application program, the file handler and the two tasks are all objects, and the operations correspond to invocation of object methods. For simplicity, the specification deals with only single instances of objects: there is one application program and one file, and the file is opened just once in a single session. The extension to multiple programs, files and sessions is straightforward even in the presence of concurrency. See, for example, the LOTOS specifications of the OSI (Open Systems Interconnection) Session Service and Transport Service [59, 60].

In the following sections, the specification of the file access system will be developed incrementally. Constraints on its behaviour will be added gradually, the aim being to construct the specification progressively. Each new constraint is intended to be a simple and hopefully obvious statement, though its formalisation may sometimes be tricky. As in [39], it is claimed that this style of specification reflects how a client might elaborate the requirements for a system. Indeed as the specification unfolds, one might imagine how the dialogue between the client and the analyst leads to the addition of such constraints.

The application program is external to the system. The primary breakdown of the specification therefore reflects the three main objects identified in the system: the file handler and the two tasks. These will be specified in isolation. Finally, system-level constraints will bind the parts of the specification to yield the required overall behaviour. Most requirements will introduce new constraints, but some will require new parameters to be introduced into events. A simple English statement of each requirement as a constraint will be given (C_n), followed by its formulation in LOTOS. Where appropriate, a discussion of this will also follow.

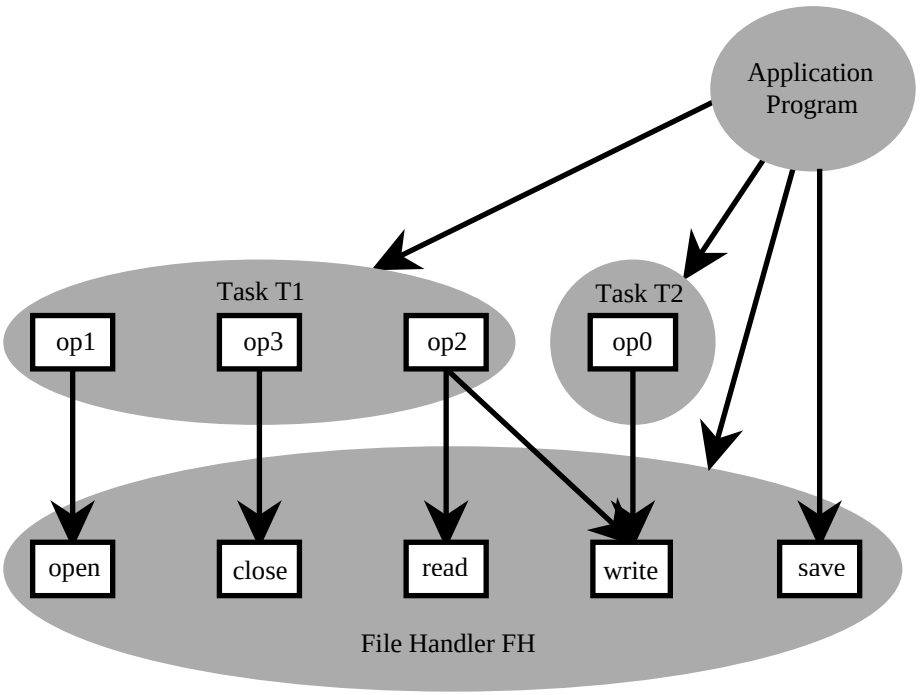

Figure 4: Object Model of File Access System

The file access system studied in this section is artificial in that it did not reflect requirements specification for an actual client. However, the requirements were taken as informal statements from the separate work of Sa and Warboys, who claim that the statements are typical of what might be encountered in practice. Figure 5 provides a 'route map' describing how constraints are introduced in the following subsections. In several places the step S_{3p} introduces a new event parameter. This necessitates step S_4 (define data type for parameter) and step S_{5n} (retrospectively add new event parameter). S_4 and S_{5n} may be taken into account immediately during specification or at the end in order to complete the specification. To simplify the figure, steps S_4 and S_5 are shown only once as the last stage of specification.

3.2 Incremental Specification of the File Handler

3.2.1 Communication Constraints

(C_1) The file handler repeatedly communicates with other objects:

FHComm := fh; FHComm

The file handler requires to communicate with the application program and with the other tasks. An event gate *fh* is therefore introduced. The above constraint is almost vacuous, but it does at least limit communication to one gate. More importantly, it lays the foundation for what follows; event parameters will later be added to the structure of events at *fh*.

3.2.2 Operation Constraints

(C_2) The file handler needs to be told which operation to invoke. An operation name is therefore added as a parameter to events:

fh ! op

Operation names will simply be constants of an enumerated type (in the programming language sense), with equality *eq* and inequality *ne*. This could be quickly specified now, but is deferred since it is so obvious.

(C_3) The file handler engages in *open*, *read*, *write*, *save* and *close* operations:

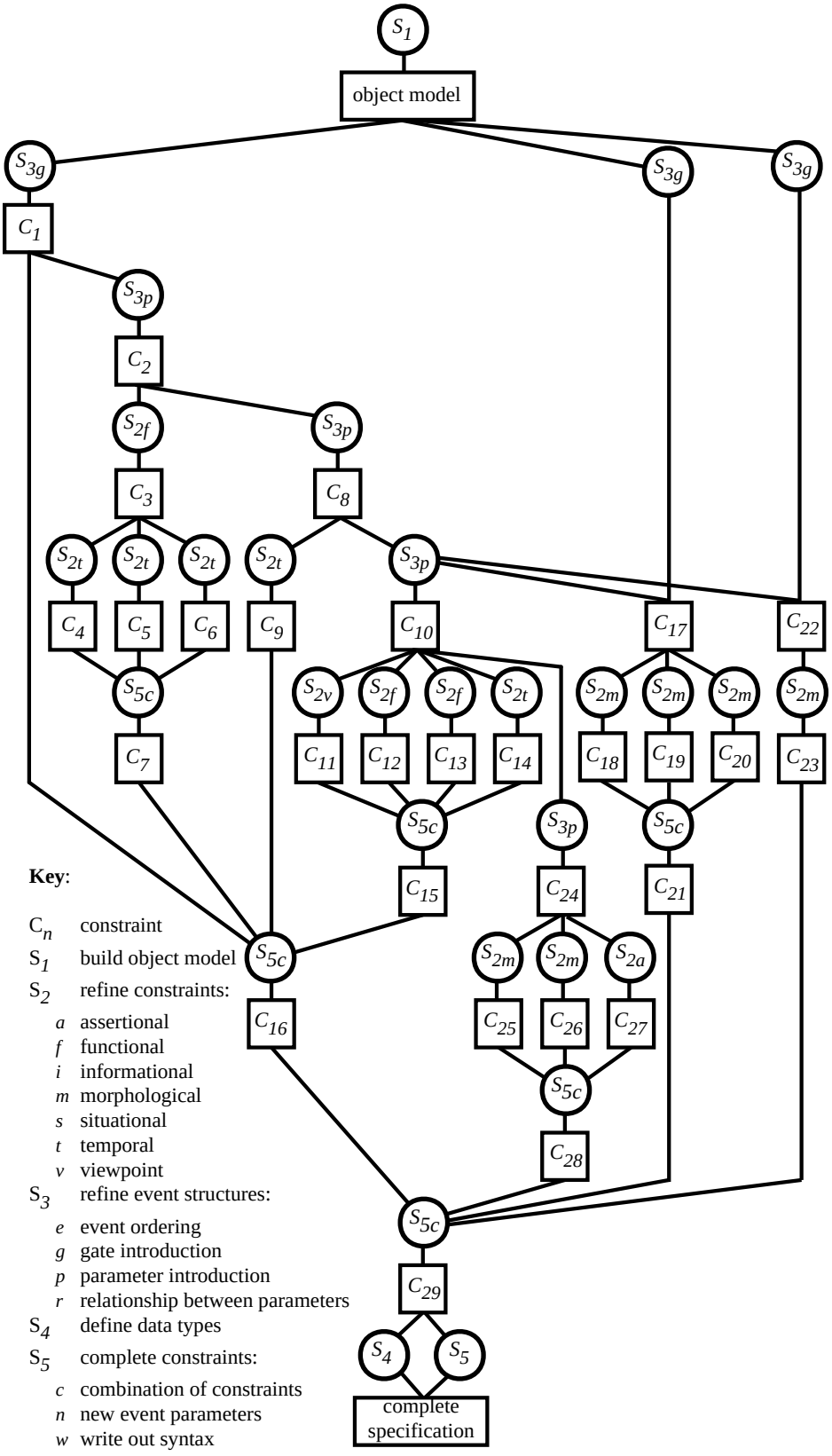

Figure 5: Route Map for Development of File Access System Specification

```
FHOpname := fh ? op : Op; FHOpname
```

The operations that the file handler supports must be limited by a constraint. The above does this implicitly since all operations must belong to the sort *Op*. It is now necessary to introduce constraints on the ordering of these operations.

(C_4) The first operation must be *open*, and this is performed only once:

```
FHOpen := fh ! open; FHOpen1

FHOpen1 := fh ? op : Op [op ne open]; FHOpen1
```

As with a number of other constraints, behaviour that depends on state is specified using a separate process to represent state implicitly. This is an auxiliary process *FHOpen1* to indicate that the file has been opened, so *open* is no longer allowed.

(C_5) The last operation must be *close*:

```
FHClose :=
   fh ! close; stop
[]
   fh ? op : Op [op ne close]; FHClose
```

In other words, the occurrence of *close* prohibits other operations from following.

(C_6) A *write* must be followed by *save* before *close* (to ensure that changes to a file are not lost):

```
FHSave :=
   fh ! write; FHSave1
[]
   fh ? op : Op [op ne write]; FHSave

FHSave1 :=
   fh ! save; FHSave
[]
   fh ? op : Op [(op ne save) and (op ne close)]; FHSave1
```

In [39], the operation constraints on the file handler say that *save* must be issued just before *close*; however, this is too strong (e.g. it should be permissible to execute *save*/*read*/*close*). Instead, [39] imposes the requirement that *save* be preceded by *write* as part of the overall system constraints. This correctly means that a file should not be saved unless it has been written to. However, this is misplaced as a *system* constraint. The specification above correctly handles this constraint as part of the file handler; after a *write*, *close* is forbidden until *save* occurs. The specification above is somewhat indirect since it is not possible to use the simplifications for constraints suggested in section 2.1.3.

(C_7) The operation constraints work together:

```
FHOper := FHOpname || FHOpen || FHClose || FHSave
```

3.2.3 Call Constraints

(C_8) The file handler needs to distinguish between the invocation of an operation and its return. A direction parameter for *inv(ocation)* or *ret(urn)* is therefore added to events:

```
fh ! op ! dir
```

This refines events, which are now split into two lower-level events correspond to a single operation at a higher level. For *dir(ection)* a simple enumerated type suffices and need not be considered further at the moment.

(C_9) An invocation is followed by a return for the same operation:

```
FHCall := fh ? op : Op ! inv; fh ! op ! ret; FHCall
```

This implies that file handler operations cannot be executed concurrently, though it will be found later that task operations might execute concurrently with file handler operations.

3.2.4 Data Constraints

(C_{10}) Some operations are associated with a data value. The structure of an event is therefore extended to carry a data parameter:

fh ! op ! dir ! octs

For simplicity and generality, the data value can be taken as a string of octets (bytes). This can make direct use of library type *OctetString* and so does not require further specification. Where an operation does not make use of data, the null string <> is used to represent a dummy or missing value. As suggested in section 2.1.3, a better alternative would be to combine operations with their parameters. This would give a simpler event structure at the expense of more complex data types. In the cases where a dummy parameter is used in this specification, a future use could be found for such a parameter (e.g. specifying a file name on *open*). For simplicity, dummy parameters have been used.

(C_{11}) Other constraints are respected for operations that do not concern input-output:

FHNotIO :=
  fh ? op : Op ? dir : Dir ? octs : OctetString [(op ne read) and (op ne write)]; FHNotIO

The data constraints concern only *read* and *write* operations. It is therefore preferable to deal with these in isolation. However, other constraints must be respected through synchronisation. The process above allows these to happen without restriction.

(C_{12}) A *read* is invoked with no data parameter, and returns some string of octets:

FHRead :=
  fh ! read ! inv ! <>; fh ! read ! ret ? octs : OctetString; FHRead

(C_{13}) A *write* is invoked with some non-empty data parameter, and returns no data result:

FHWrite :=
  fh ! write ! inv ? octs : OctetString [octs ne <>]; fh ! write ! ret ! <>; FHWrite

(C_{14}) A *read* returns the characters previously stored by *write*:

FHFile (octs) :=
  fh ! write ! inv ? octs1 : OctetString; FHFile (octs ++ octs1)
[]
  fh ! read ! ret ! octs; FHFile (octs)

In accordance with [39], a *read* is defined as retrieving the entire contents of a file; blocks and records are unimportant at this level of specification. Note that this is the first time that a state variable (the data written so far) has had to be introduced into the specification. The ++ operation above is string concatenation, used to append newly written data to the current file contents.

(C_{15}) The data constraints work together:

FHData := FHNotIO ||| ((FHRead ||| FHWrite) || FHFile (<>))

This combination reflects the extent to which the constraints handle only certain kinds of operation. The <> parameter means that the file starts out empty.

3.2.5 Overall Constraints

(C_{16}) The communication, operation, call and data constraints work together:

FH:= FHComm || FHOper || FHCall || FHData

Constraints on the file handler have been separately built up on communication, operations, operation calls and operation data.

3.3 Incremental Specification of Task 1

(C_{17}) Task 1 requires to communicate separately with the application program. An event gate *t1* is therefore introduced:

t1 ! op ! dir ! octs

Communication with task 1 takes the form shown above for consistency with the event structures introduced so far. Now the relationship between task 1 and the file handler can be considered.

(C_{18}) Operation *op1* of task 1 calls *open* in the file handler:

```
T1Op1 :=
  t1 ! op1 ! inv ! <>;
  fh ! open ! inv ? octs : OctetString; fh ! open ! ret ? octs : OctetString;
  t1 ! op1 ! ret ! <>;
  T1Op1
```

Notice that there are no constraints here on the data parameters for *open*; these are properly defined by the file handler (though it does not actually care).

(C_{19}) Invoking *op2* of task 1 takes a string, and returns the file contents after appending the data using the file handler:

```
T1Op2 :=
  t1 ! op2 ! inv ? octs1 : OctetString;
  fh ! read ! inv ? octs : OctetString; fh ! read ! ret ? octs2 : OctetString;
  fh ! write ! inv ! octs1; fh ! write ! ret ? octs : OctetString;
  fh ! read ! inv ? octs : OctetString; fh ! read ! ret ? octs3 : OctetString;
  t1 ! op2 ! ret ! octs3 −− octs2;
  T1Op2
```

Again, the constraints on data parameters for *read* and *write* are left to the file handler and are not part of task 1. The constraint imposed above is that *op2* returns the file contents after writing, having removed the prefix of what was already there. The result, which should be the string written, is returned to the caller of *op2* for checking. The string subtraction operation $--$ used above is an addition to the standard library and should be specified later.

(C_{20}) Operation *op3* of task 1 calls *close* in the file handler:

```
T1Op3 :=
  t1 ! op3 ! inv ! <>;
  fh ! close ! inv ? octs : OctetString; fh ! close ! ret ? octs : OctetString;
  t1 ! op3 ! ret ! <>;
  T1Op3
```

Any constraint on the data parameter of *close* must be part of the file handler (though it does not actually care).

(C_{21}) The constraints on task 1 work together:

T1 := T1Op1 ||| T1Op2 ||| T1Op3

3.4 Incremental Specification of Task 2

(C_{22}) Task 2 requires to communicate separately with the application program. An event gate *t2* is therefore introduced:

t2 ! op ! dir ! octs

Communication with task 2 takes the form shown above for consistency with the event structures introduced so far. Now the relationship between task 2 and the file handler can be considered.

(C_{23}) Operation *op0* of task 2 takes some character string and calls *write*:

```
T2 :=
  t2 ! op0 ! inv ? octs : OctetString;
  fh ! write ! inv ! octs; fh ! write ! ret ? octs : OctetString
  t2 ! op0 ! ret ! <>;
  T2
```

The relationship between task 2 and the file handler concerns just one operation, so no final combination of constraints is necessary.

3.5 Overall System Specification

(C_{24}) The originator of a request to the file handler must be known. An additional originator parameter is therefore added to events since an application program can call the file handler directly or via one of the tasks:

```
fh ! op ! dir ! octs ! orig
t1 ! op ! dir ! octs ! orig
t2 ! op ! dir ! octs ! orig
```

Once again, *orig(inator)* is a simple enumerated type and can be left for later specification. The new (and final) event structure can be used to express the relationship between the file handler and the two tasks.

(C_{25}) The *open* operation in the file handler is called only by task 1:

```
SysOpen :=
  fh ? op : Op ? dir : Dir ? octs : OctetString ? orig : Orig [(op eq open) implies (orig eq t1)];
  SysOpen
```

(C_{26}) The *close* operation in the file handler is called only by task 1:

```
SysClose :=
  fh ? op : Op ? dir : Dir ? octs : OctetString ? orig : Orig [(op eq close) implies (orig eq t1)];
  SysClose
```

(C_{27}) The *read/write/read* invoked by *op2* in task 1 is atomic:

```
SysWrite :=
  fh ! read ! inv ? octs : OctetString ! t1; fh ? op : Op ? dir : Dir ? octs : OctetString ! t1;
  SysWrite1
[]
  fh ? op : Op ? dir : Dir ? octs : OctetString ? orig : Orig
    [(op ne read) or (dir ne inv) or (orig ne t1)];
  SysWrite

SysWrite1 :=
  fh ! read ! ret ? octs : OctetString ! t1;
  SysWrite
[]
  fh ? op : Op ? dir : Dir ? octs : OctetString ! t1 [(op ne read) or (dir ne ret)];
  SysWrite1
```

This is the most difficult constraint of all to formalise because it makes a fairly global statement. The constraint is formulated by saying that the invocation of task 1's first *read* is followed by one or more operations from task 1 until the return of task 1's second *read*. Operations invoked by other sources are not permitted during this period. Note that it is deliberately not said that *op2* does *read*/*write*/*read*. This is to avoid overlap with *T1Op2* which states exactly this constraint.

(C_{28}) The overall system constraints work together:

```
SysGen := SysOpen || SysClose || SysWrite
```

(C_{29}) The application program works within the complete set of constraints (overall system, file handler and tasks):

```
Sys := SysGen |[fh]| FH |[fh]| (Appl |[t1, t2]| (T1 ||| T2))
```

Notice that the application program (which is not specified here) has to be embedded within the final composition. This is a consequence of the way it synchronises with the file handler and two tasks. In particular, synchronisation between the application and the file handler is distinct from synchronisation between a task and the file handler.

3.6 Completing Requirements Capture

The essential details of the specification have now been worked out. The remaining steps, S_4 (define data types) and S_5 (complete constraints), can be performed in either order. For the file system example, the (trivial) specification of four data types is added in step S_4. In step S_5 any remaining tidying up is done. In the example, constraints were already combined after they had been identified (step S_{5c}) though this might have been deferred. Event parameters introduced during the refinement of constraints have to be added to earlier constraints (step S_{5n}). Finally, in step S_{5w} any syntactic abbreviations are written out in full and the necessary specification rubric is added. The complete specification of data and behaviour is not given here in the interests of brevity, but it is available on-line [61].

The specification is evaluated during step S_6 using standard simulation, analysis and verification techniques. The file access system specification is highly constraint-oriented, with over 20 synchronised processes. However, standard LOTOS tools can cope without problems. For this example, a range of application program behaviour can be specified in test scenarios. A practical difficulty in dealing with constraint-oriented specifications is that it is fairly easy to introduce deadlocks inadvertently. Sometimes this is due to an inappropriate parallel operator, but often the problem arises because a constraint is too strict. Because virtually all processes synchronise on events, a flaw like this usually leads directly to deadlock. As a result, errors in the specification are generally very noticeable during checking. LOTOS simulators vary in the help given to track down the source of deadlock, but at least they should indicate which event offers are failing to synchronise.

In the final step, S_7, any deficiencies noted in the requirements specification are used to modify the object model or the behavioural constraints as appropriate.

3.7 Comparison with the Work of Sa and Warboys

The intention of the approach in this paper is broadly similar to that of Sa and Warboys. Yet the differences in approach have a profound influence. For this paper, the notation is that of LOTOS used in a constraint-oriented style. Sa and Warboys use pre/post-conditions in VDM fashion to define operations, coupled with temporal logic to relate the execution of operations. Both approaches develop a formal specification incrementally with an outwardly similar structure. Although [39] does not talk about constraints, the predicates it uses act in much the same way as LOTOS constraints.

A major difference between the two approaches is that a LOTOS specification can take advantage of the built-in language constructs for communication, synchronisation, sequencing, choice and concurrency. LOTOS also offers an integrated framework for specification of data and behaviour. In contrast, the work of Sa and Warboys:

- uses a combination of two notations (derived from VDM and temporal logic), which results in a more complex specification language
- gives a more complex definition of operations as an accept event, a call pattern and a return event
- introduces history variables to keep track of operation invocations, parameter values and return values
- uses temporal ordering operators that are defined using a more basic temporal logic
- defines parallel operators using state variables, since synchronisation is not primitive to the specification language used
- does not have the ability to define arbitrary data types

The notation of [39] is too extensive to give an example here, but as a rough comparison it takes about 20 lines of this notation to express a constraint like C_{12}.

Although these differences favour LOTOS, the use of temporal logic in [39] confers an important advantage. When it comes to assessing specification properties such as safety and liveness, these can be formulated and evaluated within the same temporal logic framework as used for specification. LOTOS *per se* deals only with specification aspects. A separate (but well-known) theory is used to formulate and

evaluate properties of LOTOS specifications. From the specifier's point of view, it is a matter of taste whether verification is performed within the language framework or using a separate theory. It is claimed that the approach of this paper makes it easier to write a requirements specification, and that a client can relate more easily to such a specification. As the examples have hopefully demonstrated, it is practicable to develop specifications incrementally. In particular, it is reasonably straightforward to turn individual requirements into formal statements.

4 Conclusion

The importance of requirements specification and the desirability of formal, incremental specification have been noted. The intent of this work is broadly similar to that of Sa and Warboys, though details of the approach and the language differ considerably. The integrated features of LOTOS for describing data, communication and behaviour are very convenient for requirements specification.

An approach that exploits the constraint-oriented style for LOTOS has been presented, using a file access system as the main example. Although this example is relatively small, it has highlighted the key features of the approach:

- It was remarked that engineering practice builds on known components and known combinations. In the proposed approach for capturing requirements, the constraints act like components that are built into the overall specification structure. Even if some requirements subsequently change, many of the components (constraints) should be re-usable. Although the file access system did not illustrate it, some components and their combinations can be common to specifications of the same class of system. For example, LOTOS specifications of OSI services commonly share specification components and combinations such as service primitives, service access points, connections, expedited data and multiplexing.

- The approach yields a formal specification in LOTOS, and so is amenable to various forms of analysis like simulation, verification of equivalences and proof of properties.

- The file access system has shown how requirements can be incrementally added to a specification, moving from a partial specification to the complete specification. Requirements may be progressively introduced from the continuing dialogue between the analyst and the client. For the file access system, the requirements were drawn one at a time from the source document.

- Specifications resulting from this approach are compositional, easing understanding. In particular, each constraint on the file access system can be evaluated in its own right for completeness and correctness. The intention is that the client be involved in this evaluation.

- A categorisation of types of constraint has been presented. Many constraints on the file access system take the form 'in this situation the system should do that'. It is felt that this style of constraint is appropriate for capturing client requirements.

- One a specification has been developed using the approach of the paper, there are possibilities for developing it further. For example, it can be transformed into a lower-level specification and thence into an implementation. This could exploit separate work on style transformation such as [44, 16]. Implementation could make use of LOTOS refinement and compilation tools like CADP (Cæsar Aldébaran Development Package [62]) or LITE (LOTOS Integrated Tool Environment [63]).

It was noted in section 1.1 that formal specifications are incomprehensible to a typical client. The approach of the paper tries to solve this problem by relating the structure and behaviour of the formal specification closely to how the client might understand requirements informally. Since the resulting specification is executable, it is possible for the analyst to animate the requirements and thus to demonstrate specification behaviour to a client without requiring knowledge of the formal language.

It was also noted in section 1.1 that there is a need to balance the rigour of a formal specification against the inherently informal nature of requirements capture. The paper presents a systematic (though

not mechanical) approach that nonetheless leads to a formal specification. Since the approach captures informal requirements as constraints that are then formalised, it is believed that it strikes an appropriate balance. Through animation, the client can be made aware of what has been specified.

In summary, the approach is formal, incremental and suitable for specifying requirements. It is hoped that it will prove of value in realistic situations.

Acknowledgements

The author is grateful to Dr. Robert G. Clark and Daren A. Reed, University of Stirling, for their perceptive comments on a draft of the paper. The constructive criticisms of the anonymous reviewers were also very valuable. Hubert Garavel, INRIA Rhône-Alpes, kindly undertook independent checks on the LOTOS specification of the file access system.

References

[1] Barry W. Boehm. Verifying and validating software requirements and design specification. *IEEE Transactions on Software Engineering*, 1(1):75–88, January 1984.

[2] Barry W. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, December 1976.

[3] M. E. Fagan. Design and code inspections to reduce errors in program. *IBM Systems Journal*, 15(3):7/1–7/26, August 1979.

[4] Jonathan P. Bowen and Victoria Stavridou. The industrial take-up of formal methods in safety-critical and other areas: A perspective. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 183–195, Berlin, Germany, 1993. Springer-Verlag.

[5] B. P. Collins, John E. Nicholls, and Ib H. Sorensen. Introducing formal methods: The CICS experience with Z. In Neumann, Simpson, and Slater, editors, *Mathematical Structures for Software Engineering*, pages 153–164. IBM UK Laboratories, Winchester, 1991.

[6] Ministry of Defence. The procurement of safety-critical software in defence equipment — Part 1: Requirements. Technical Report Defence Standard 00-55, Issue 1, Ministry of Defence, Glasgow, UK, April 1991.

[7] Ministry of Defence. The procurement of safety-critical software in defence equipment — Part 2: Guidance. Technical Report Defence Standard 00-55, Issue 1, Ministry of Defence, Glasgow, UK, April 1991.

[8] 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.

[9] Howard Bowman, Gordon S. Blair, Lynne Blair, and Amanda G. Chetwynd. Time versus abstraction in formal description. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 467–482. North-Holland, Amsterdam, Netherlands, 1994.

[10] Kees Bogaards. LOTOS supported system development. In Kenneth J. Turner, editor, *Proc. Formal Description Techniques I*, pages 279–294. North-Holland, Amsterdam, Netherlands, 1989.

[11] A. J. Robin G. Milner. *Communication and Concurrency.* Addison-Wesley, Reading, Massachusetts, USA, 1989.

[12] C. Anthony R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1985.

[13] Hartmut Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, Germany, 1985.

[14] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), January 1988.

[15] Kenneth J. Turner, editor. *Using Formal Description Techniques — An Introduction to ESTELLE, LOTOS and SDL*. Wiley, New York, January 1993.

[16] Chris A. Vissers, Giuseppe Scollo, and Marten van Sinderen. Architecture and specification style in formal descriptions of distributed systems. *Theoretical Computer Science*, 89:179–206, 1991.

[17] G. P. Mullery. *CORE – A Method for Controlled Requirements Specification*. Institution of Electrical and Electronic Engineers Press, New York, USA, 1979.

[18] D. T. Ross. Applications and extensions of SADT. *Computer*, 18(4):25–34, 1985.

[19] I. Jacobson. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison-Wesley, Reading, Massachusetts, USA, 1992.

[20] K. Benner, M. Feather, W. L. Johnson, and L. Zorman. Utilizing scenarios in the software development process. In *Proc. Working Conference on Information Systems Development Process*, Amsterdam, Netherlands, 1993. North-Holland.

[21] C. Potts and K. Takahashi. *An Active Hypertext Model for System Requirements*. Institution of Electrical and Electronic Engineers Press, New York, USA, 1993.

[22] Julio Cesar Sampaio do Prado Leite and Peter A. Freeman. Requirements validation through viewpoint resolution. *IEEE Transactions on Software Engineering*, 17(12):1253–1269, December 1991.

[23] Anthony Finkelstein, J. Kramer, and J. K. Goedicke. *Viewpoint-Oriented Software Development*. Institution of Electrical and Electronic Engineers Press, New York, USA, December 1990.

[24] Gerald Kotonya and Ian Sommerville. Viewpoints for requirements definition. *Software Engineering Journal*, 7(6):175–187, November 1992.

[25] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, UK, 1996.

[26] C. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1992.

[27] Clifford B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1990.

[28] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1992.

[29] Nico Plat, Jan van Katwijk, and Hans Toetenel. Application and benefits of formal methods in software development. *Software Engineering Journal*, 7(5):335–347, September 1992.

[30] Richard A. Kemmerer. Integrating formal methods in the development process. *IEEE Software*, 7(5):37–50, September 1990.

[31] V. Kelly and U. Nonnenmann. *Reducing the Complexity of Formal Specification Acquisition*, pages 41–64. MIT Press, Boston, USA, 1991.

[32] Stéphane Somé, Rachida Dssouli, and Jean Vaucher. Towards an automation of requirements engineering using scenarios. *Journal of Computing and Information*, 2(1):1070–1092, 1996.

[33] Alexandre M. L. de Vasconcelos and John A. McDermid. *A Technique for analyzing the Effects of Changes in Formal Specifications*, pages 65–80. North-Holland, Amsterdam, Netherlands, 1993.

[34] D. R. Kuhn. A technique for analyzing the effects of changes in formal specification. *The Computing Journal*, 35:574–578, December 1992.

[35] Robert G. Clark and Ana M. D. Moreira. Constructing formal specifications from informal requirements. In *Proc. Software Technology and Engineering Practice 97*, London, UK, July 1997. Institution of Electrical and Electronic Engineers Press.

[36] Kenneth J. Turner, Ashley McClenaghan, and Colin Chan. Specification and animation of reactive systems. In Volkan Atalay, Uğur Halici, Kemal İnan, Neşe Yalabik, and Adnan Yazici, editors, *Proc. International Symposium on Computer and Information Systems XI*, pages 355–364. Middle-East Technical University, Ankara, Turkey, November 1996. ISBN 975-429-103-9.

[37] Adam C. Winstanley and David W. Bustard. EXPOSE: An animation tool for process-oriented specifications. *Software Engineering Journal*, 6(6):114–118, November 1991.

[38] David W. Bustard and Adam C. Winstanley. Making changes to formal specifications: Requirements and an example. *IEEE Transactions on Software Engineering*, 20(8):562–568, August 1994.

[39] Jin Sa and Brian C. Warboys. Specifying concurrent object-based systems using combined specification notations. Technical Report UMCS-91-7-2, Department of Computer Science, University of Manchester, Manchester, UK, July 1991.

[40] ISO/IEC. *Open Distributed Processing – Basic Reference Model – Part 1: Overview and Guide to the Use of the Reference Model*. ISO/IEC 10746-1. International Organization for Standardization, Geneva, Switzerland, 1995.

[41] ISO/IEC. *Open Distributed Processing – Basic Reference Model – Part 2: Foundations*. ISO/IEC 10746-2. International Organization for Standardization, Geneva, Switzerland, 1995.

[42] ISO/IEC. *Open Distributed Processing – Basic Reference Model – Part 3: Architecture.* ISO/IEC 10746-3. International Organization for Standardization, Geneva, Switzerland, 1995.

[43] Henry E. Dudeney. *A Puzzle Mine*. Thomas Nelson, London, 1959.

[44] Kenneth J. Turner. A LOTOS-based development strategy. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*, pages 157–174. North-Holland, Amsterdam, Netherlands, 1990. Invited paper.

[45] Kenneth J. Turner. An engineering approach to formal methods. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *Proc. Protocol Specification, Testing and Verification XIII*, pages 357–380. North-Holland, Amsterdam, Netherlands, June 1993. Invited paper.

[46] Edward Yourdon. *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1989.

[47] Richard O. Sinnott and Kenneth J. Turner. Applying the architectural semantics of ODP to develop a trader specification. *Computer Networks and ISDN Systems*, 29(4):457–471, March 1997.

[48] Kenneth J. Turner. An architectural foundation for relating features. In Petre Dini, Raouf Boutaba, and Luigi M. S. Logrippo, editors, *Proc. Fourth International Workshop on Feature Interactions in Telecommunication Networks*, pages 226–241. IOS Press, Amsterdam, Netherlands, 1997.

[49] 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.

[50] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991.

[51] Richard O. Sinnott and Kenneth J. Turner. Modelling ODP viewpoints. In B. Cameron, C. Geldrez, A. Hopley, D. Howes, B. Mirek, and M. Plucinska, editors, *Proc. OOPSLA '94 Workshop on Precise Behavioural Specifications in OO Information Modelling*, pages 121–128, Portland, Oregon, USA, October 1994.

[52] 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.

[53] Ashley McClenaghan. XDILL: An X-based simulator tool for DILL. Technical Report CSM-119, Department of Computing Science and Mathematics, University of Stirling, UK, April 1994.

[54] Kenneth J .Turner and Ashley McClenaghan. Visual animation of LOTOS using SOLVE. In Dieter Hogrefe and Stefan Leue, editors, *Proc. Formal Description Techniques VII*, pages 283–285. Chapman-Hall, London, UK, 1995.

[55] David W. Harrison and Michael D. Harrison. Animating process-oriented specifications: Experiences and lessons. In *Proc. Automating Formal Methods for Computer-Assisted Prototyping*, London, UK, January 1992. Institution of Electrical Engineers.

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

[57] Jin Sa and Brian C. Warboys. The EDS specification framework. Technical Report EDS WP.6I.sa990, Department of Computer Science, University of Manchester, UK, September 1990.

[58] J. A. Keane, Jin Sa, and Brian C. Warboys. Applying a concurrent formal framework to process modelling. In *Proc. Formal Methods Europe '94*, volume 873 of *Lecture Notes in Computer Science*, pages 291–305. Springer-Verlag, Berlin, Germany, 1994.

[59] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Formal Description in LOTOS of the Connection-Oriented Session Service.* ISO/IEC TR 9571. International Organization for Standardization, Geneva, Switzerland, 1990.

[60] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Formal Description in LOTOS of the Connection-Oriented Transport Service.* TR 10023. International Organization for Standardization, Geneva, Switzerland, 1990.

[61] Kenneth J. Turner. Constraint-oriented specification of a file access system. http: //www. cs. stir. ac. uk/∼kjt/research/well/fas. html, May 1997.

[62] J. C. Fernández, Hubert Garavel, L. Mounier, A. Rasse, and C. Rodriguez. A toolbox for the verification of LOTOS programs. In *Proc. 14th International Conference on Software Engineering and its Applications*, pages 246–259, May 1992.

[63] Peter H. J. van Eijk. The LOTOSPHERE integrated tool environment LITE. In Kenneth R. Parker and Gordon A. Rose, editors, *Proc. Formal Description Techniques IV*, pages 471–474. North-Holland, Amsterdam, Netherlands, November 1991.