

SINGLE LOCATION OBJECT-ORIENTED SOFTWARE DEVELOPMENT.

M.M. Ross*, D.G. Kourie* and R.J. van den Heever*

*Dept. of Computer Science, University of Pretoria, Lynnwood Road, Pretoria, 0001, South Africa

Abstract: A constructive approach to software development is proposed in which correctness reasoning forms an integral part of the analysis, design and implementation phases. A computational model adapted from UNITY for an object-oriented paradigm simplifies the correctness reasoning process. A small example demonstrates the method's emphasis on correctness properties and the role played by these properties. The object-oriented nature of the method promotes reusability, extensibility and encapsulation. A primary objective is to offer ordinary software practitioners a more rigorous approach towards software development without requiring an in-depth understanding of the underlying mathematics. The specification notation will map to a wide range of OO languages. Here the focus is on Java. Suitable for developing concurrent or distributed systems, the method enables the user to abstract away from the target architectural details until the implementation phase.

Key words: Lightweight formal methods, UNITY, distributed / concurrent software.

1. INTRODUCTION

Formal methods are seldom used in practice, and many regard the theme as rather esoteric [1, 2], mainly because the underlying mathematics is perceived as difficult and tedious to use. The software development method offered here (which, for reasons to be explained below, is called the SLOOP method) is a pragmatic constructive approach that does not require an in-depth knowledge of formal methods. It is "lightweight" in that formal correctness proofs are not mandated. However, it is sufficiently rigorous to support effective informal correctness reasoning while refining a specification into code, thereby uncovering possible instances of ambiguity, incompleteness and inconsistency. SLOOP's constructive nature leads to systems that are correct (faithful to their specification) from the outset, instead of ones that need to be debugged into correctness [3]. The systematic, rather than *ad hoc*, approach to specification and development leads to better structured programs which are believed to be generally easier to verify [4].

SLOOP is intended for developing concurrent or distributed object-oriented (OO) software. It is based on UNITY (Unbounded Nondeterministic Iterative Transformations) – the computational model and proof system devised by Chandy and Misra [5]. A UNITY program is classified as a *Single Location Program* (SLP). Gerth and Pnueli [6] define such a program as having the form

$$I: *[A_1 [] \dots [] A_n] \quad (1)$$

Where:

A_1, \dots, A_n are conditional assignments

And:

I specifies the initial state.

The asterisk indicates that the section in square brackets is executed infinitely often. The "[]" symbols separate the conditional assignments and indicate that any one of them may be executed (atomically) during each iteration, provided the fairness requirement that each conditional assignment is executed infinitely often, is satisfied. The main benefit of SLPs is that location counters can be ignored in correctness arguments. As a consequence, an SLP avoids a major deficiency of specification formalisms that rely on control flow in parallel programs – *i.e.* it avoids the "complexity of reasoning about computation histories" [6].

Here an extension to the SLP computational model is proposed that is consistent with the observation in [6] that: "...the particular form of the actions within the iteration of an SLP-program does not matter. It is the fact that these actions are executed atomically that counts. Accordingly, we can allow arbitrary programs instead of assignments as the atomic actions." SLP is therefore extended to programs that incorporate OO features: instead of restricting the A_i 's in (1) above to multiple assignment statements, these statements may also contain additional OO-related actions, as will later be described. A program that conforms to this extended SLP computational model is designated as a Single Location Object Oriented Program – hence the acronym SLOOP, which is used as an adjective to describe both the language and the particular development method proposed here.

The SLOOP development method allows for analysis, design and implementation in an iterative incremental fashion. In all the phases there is a strong focus on specifying, refining and reasoning about correctness properties. The absence of flow of control in the

underlying computational model significantly simplifies the reasoning process.

The analysis phase starts with a class diagram derived by any conventional approach (*e.g.* Fusion, Rational Unified Process, etc.). The subsequent steps in the analysis phase are particularly important: instead of focusing on sequence, collaboration and state diagrams – dealing primarily with flow of control issues – the focus is on correctness properties of the system. The system's intended dynamic behaviour is described as a set of informally specified safety, liveness and precedence properties. These are systematically derived from a disciplined consideration of a SLOOP checklist of properties.

The properties of the analysis phase are refined during the design phase. A SLOOP program is then refined from the correctness properties. Section 3 illustrates the refinement process, based on a small example. The implementation phase covers the mapping of the SLOOP program to a target architecture and target implementation language. This is briefly discussed in section 4.

Section 5 reviews the differences between SLOOP and rival approaches. Finally, in section 6, an assessment of SLOOP is given, based on experience in its use. The SLOOP method addresses the problem of developing correct concurrent software by encouraging a constructive approach towards software development and by combining OO features such as data encapsulation and reuse with a computational model that simplifies correctness reasoning. It is concluded that the promising results should now be followed with the development of tools to make SLOOP accessible to the average software engineer.

2. THE SLOOP LANGUAGE

A SLOOP program is expressed in the SLOOP specification language. It relies on an existing OO language for its major syntactical features, the associated constructs then having the corresponding semantics of the OO language. This has several advantages:

- It aids understandability, thereby reducing the learning curve.
- It facilitates the transformation of a specification into an executable program.
- The functionality offered by the base OO language's class library is accessible within the specification language.

The original version of the SLOOP language resembled Smalltalk [7]. Early trials with the SLOOP method also mapped the application to Smalltalk code. Here, however, Java is used as the base OO language. Differences between the two versions are purely syntactical.

The SLOOP language is required to express semantics relating to the SLOOP computational model. As in UNITY, features are required to express both synchrony

and asynchrony in a program. Notations to express these features have also been incorporated into the SLOOP language.

In this section, the syntax and semantics of the SLOOP language relevant to this article are given. Full details are available in [8].

2.1 SLOOP programs and classes

The static structure of a SLOOP program consists of an *activation section* and a *collection of classes*, grouped into one or more packages. The former contains a list of sequential statements, as well as a list of parallel statements. The concept of a parallel statement is explained later in this section.

The sequential statements in the *activation section* are executed initially, once only, in order of appearance. They ensure that the required classes are instantiated, but may optionally perform other functions. The sequential statements in the *activation section* correspond roughly to I, the initialization part of (1) above.

Once the sequential statements have completed execution, the statements in the parallel statement list in the *activation section* are executed infinitely often and in any order. These are the statements that invoke the parallel methods of the classes of the program. Note that all classes containing parallel methods must be instantiated during activation¹.

A SLOOP class description has the standard elements found in an OO language: the class name; its superclass; and its class and instance variables. As a notational convenience, provision is also made for a class macros section as described in [8]. This is followed by a properties section, which is illustrated in Section 3. Finally, the methods (both class and instance) of the class are defined.

There are two types of methods: sequential and parallel. Sequential methods are exactly as their counterparts in conventional OO languages. Syntactically, they may only invoke other sequential methods and, semantically, all reasoning about them assumes that they always terminate.

A parallel method is made up of parallel statements. Each parallel statement executes atomically, infinitely often and in any order with respect to all other parallel statements in the program. It may invoke other parallel *or* sequential methods. Parallel statements correspond to the A_i 's in (1) above but may contain the OO actions that extend the UNITY computational model. The concept of a parallel method allows one to refer to a group of parallel statements, thereby providing structure to the program. In [8] it is shown that even if parallel methods are invoked in a hierarchical fashion, reasoning about the

¹ Exceptions to this rule are discussed in [8].

parallel statements is still based on the fact that each parallel statement is executed infinitely often and in any order. The syntax of SLOOP methods and their associated statements will be described in greater detail below.

Note that the properties section of a class description allows for the specification of class invariants. Such an invariant is also part of the preconditions of all the class's parallel methods. However, these invariants need only hold after the class has been instantiated and initialised. It is for this reason that, while parallel statements are allowed in the activation section, their execution may only commence once the sequential statements in this section have completed.

Not all classes have to be instantiated in the activation section: classes can be instantiated as needed if they do not contain parallel methods and if the resulting objects are not referenced in parallel statements.

2.2 SLOOP Methods

A SLOOP method description includes the method signature (the message pattern in Smalltalk), a method macros section and a properties section. A parallel method is differentiated by having `p_` as a name prefix. The body of both a sequential and a parallel method is a *statement-list*. The BNF for a parallel method's *statement-list* is presented in Figure 1 (for brevity, the *quantified-statement-list* and *quantified-component* constructs are omitted from the present discussion and are therefore not defined below).

```

statement-list → statement {[] statement}*
statement →
    simple-statement | quantified-statement-list
simple-statement →
    statement-component {|| statement-component}*
statement-component →
    enumerated-component | quantified-component
enumerated-component →
    conditional-component-part-list|component-part
conditional-component-part-list→
    if boolean-expr component-part-list
    {~ if boolean-expr component-part-list}*
component-part-list →
    component-part {\+ component-part}*
component-part →
    [return]variable = simple-expr /
    [return] method-call
simple-expr → method-call | primary

```

Figure 1: Statement lists in methods

The `[]` symbol is used to separate individual *statements* in a *statement-list*. Each *statement* consists of one or more *statement-components* separated by the `||` symbol. (Instead of using `||`, the tilde symbol (`~`) may optionally be used to

highlight the fact that the *statement-components* being listed are conditional.)

An *enumerated-component* is either a single *component-part* that is executed unconditionally; or it is a list of *component-parts*, separated by the `\+` symbol, that are conditionally *executed*, the *condition* being a *boolean-expr*. A *component-part* (which may either be a *method-call* or an assignment) executes atomically.

A *method-call* refers to a Java method invocation. (Its Smalltalk equivalent is a message expression [7].) It consists of the method's object, the method name and zero or more arguments.

As usual, an assignment has a variable on the left hand side, and a *simple-expr*, which is either a *method-call* or a *primary* on the right hand side. A *primary* is either a variable name, or a Java expression. The 'return' keyword preceding a *method-call* or assignment in a method indicates that the value of that expression is to be returned by the method. In the absence of this keyword, the default return value of a method is its implicit parameter – *i.e.* the instance (or class) to which the instance (or class) method belongs.

The following program excerpt demonstrates the above concepts. It contains two valid SLOOP statements, S1 and S2. The purpose of these statements is to receive objects from a producer, transform them and subsequently buffer them until they can be passed to a consumer on a First In First Out basis, while ensuring that a record is kept of the maximum length reached by `buff`, the FIFO queue containing the buffered objects. The `[]` symbol separates the two statements. The first statement has two *statement-components* separated by the `||` symbol. Both statements have multiple *component-parts*.

The two statements are executed infinitely often and in any arbitrary order. When a statement is executed, all its *component-parts* are executed simultaneously.

```

if (next.notNull() && buff.size() < max)
{this.use(next)      \+
 buff.addLast(next)  \+
 next = nil
}
||
if (next.notNull() &&
    buff.size() < max &&
    buff.size()+1 > maxToDate)
{maxToDate = buff.size()+1}      (S1)

[]

if (consumer.isReady()&& buff.size() > 0)
{consumer.pass(buff.first()) \+
 buff.removeFirst()}            (S2)

```

Section 3 will elaborate on these constructs by deriving the body of a SLOOP parallel method.

2.3 SLOOP statement evaluation order

Each statement executes infinitely often, atomically and in an arbitrary order. When a statement is executed, all *statement-components* (for the purposes of this discussion these will be limited to *enumerated-components*) may execute more or less simultaneously, but subject to rules below that slightly restrain the simultaneity. These are based on UNITY rules, but are appropriately adapted for SLOOP's extended computational model.

When a parallel statement is executed, synchronization rules control the simultaneous execution of *statement-components* as follows:

- 1) Evaluate simultaneously each *boolean-expr* in each conditional statement.
- 2) In all component-parts, evaluate simultaneously all method calls:
 - a) that are part of an argument of some other method call;
 - b) that compute (normally by reflection) the implicit parameter of some other method call;
 - c) that appear in an assignment.
- 3) In all *component-parts* do the following simultaneously:
 - a) execute all assignments;
 - b) evaluate the outermost method calls.

The order in which arguments of a method call are evaluated is assumed to be the same as in the base OO language being used – in this case, Java.

The following rules apply when SLOOP statements are evaluated:

Rule A: The evaluations steps 1) and 2) may not modify class or instance variables that occur in the statement concerned.

Rule B: Methods that are executed in step 3) should not modify any class or instance variables that are referenced by methods that are executed as part of 3) in other component-parts of the same statement.

The above evaluation steps are demonstrated in an example in Section 3.3. In the next section it is shown how the SLOOP notation provides a vehicle for analyzing and designing a system in such a way that correctness reasoning is simplified.

3. REASONING ABOUT CORRECTNESS

A specifier should neither under-specify a target system (by neglecting to articulate important properties) nor over-specify it (by introducing redundant properties) [9]. To assist in this objective, SLOOP provides a comprehensive and relevant checklist of generic correctness properties. Its use is mandatory. During the analysis phase the relevance of each property type within each category is systematically considered. Properties that are deemed to be relevant are then articulated in concrete terms that refer to the system's

artifacts. During the design phase these property statements are redefined in terms of the SLOOP logic.

The checklist is based on temporal logic correctness properties defined by Manna and Pnueli [10]. There are three categories of correctness properties, viz. safety², liveness³ and precedence⁴ (until) properties. Each category is subdivided into different property types (*e.g.* clean behaviour, global invariants and unless properties are property types in the safety property category). The complete checklist is given in [8].

For cross-referencing purposes, a unique reference string is assigned to each correctness property used during each development phase. The string has the format: PCt-nn, where:

P is A, D or I (standing for Analysis, Design or Implementation) to indicate the phase during which the property was formulated;

C is S, P or L (standing for Safety, Precedence or Liveness) to indicate the property category;

t is the checklist-provided number of the property type within the property category; and

nn is a specifier-assigned number to distinguish system-specific properties of the same type from one another.

To illustrate the SLOOP software development method, consider the following small dispatcher system example. It demonstrates the method's emphasis on correctness properties and the role played by these properties. The problem statement is as follows:

Develop a dispatcher system which acts as a generic user of objects, receiving them from a producer, using the received objects in some way and dispatching them to a consumer. The dispatcher has to queue the objects in a FIFO order until the consumer is ready to receive the next object. The dispatcher keeps a record of the maximum length ever reached by the queue. It also ensures that the size of this queue never exceeds a specified maximum value.

3.1 Analysis phase correctness properties

Figure 2 shows a class diagram derived from the problem statement (the arrows in the diagram indicate the associations between the classes). Further analysis suggests the following instance fields associated with the Producer and Dispatcher classes, respectively. (The

² Safety properties describe properties that need to be satisfied for all execution sequences of the program, *i.e.* they are properties that assert that something bad will never happen.

³ Liveness properties deal with events that must occur eventually, *i.e.* they are properties that assert that something good will eventually happen.

⁴ Precedence properties are described in terms of safety and liveness operators. An example of a precedence property is fair responsiveness.

Consumer and DispatcherQ classes will not be further elaborated in this example.)

The Producer instance fields include:

- A reference to an instance of Dispatcher, which will dispatch objects provided by the Producer.

The Dispatcher instance fields include:

- `buff`, a DispatcherQ instance,
- `consumer`, the Consumer instance to which a Dispatcher dispatches elements in `buff`,
- `maxToDate`, the maximum length of `buff` reached up to this point in the computation,
- `max`, the maximum length allowed for `buff`,
- `next`, an object received from a Producer, but not yet added to `buff`.

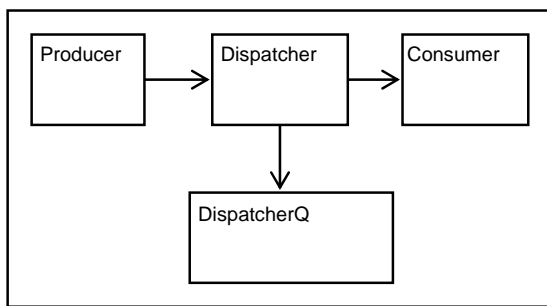


Figure 2: Class Diagram

The Producer and Consumer classes will be assumed to be part of the system environment. Their correctness properties will therefore not be discussed. Note that during the analysis phase no assumptions are made about the target architecture (e.g. whether it is a synchronous shared-memory or a distributed system architecture).

To simplify the discussion below, assume that scrutiny of the property checklist leads to the conclusion that the following broad property types are the only ones relevant to the current example.

- AS2 Clean behaviour: Safety properties that specify the absence of abnormal conditions.
- AS3 Global invariant: Safety properties that should always hold.
- AS4 Unless: Safety properties specifying that once p holds, it will continue to hold unless q holds.
- AP1 Safe liveness: Precedence properties specifying that once p holds, it will continue to hold unless q holds, and eventually q will indeed hold.
- AL1 Total correctness: Liveness properties of terminating code, indicating that if preconditions hold initially, then the postcondition will hold eventually.

After carefully considering the safety properties AS2, AS3 and AS4, the following appear to be a reasonable reflection of the system's safety behavioural requirements:

- AS2-01: The dispatcher never indicates that it is ready to accept `next` from the producer if it has not yet added the element passed to it on a previous occasion to `buff`.

- AS2-02: The length of `buff` is always less than or equal to `max`.
- AS3-01: The length of `buff` is always less than or equal to `maxToDate`.
- AS4-01: `maxToDate` is non-decreasing.
- AS4-02: Once the dispatcher has accepted a new object from the producer, it remains a new object unless it has been used and added to `buff`.
- AS4-03: Once an object is added to `buff`, it remains there unless it is passed to the consumer.
- AS4-04: If an object is added to `buff`, it is always added to the end of the queue.
- AS4-05: Objects are removed from `buff` in the order that they are added to the queue.

Likewise, after careful consideration of the precedence properties, the following suggest themselves as relevant to the current system:

- AP1-01: If the dispatcher has received a new object from the producer and the size of `buff` is less than `max`, then these conditions continue to hold until the new object has been both used (in some unspecified way) and added to `buff`.
- AP1-02: Once the consumer is ready to receive an object and `buff` is non-empty, then these conditions continue to hold until an element of `buff` is removed and passed to the consumer.

For the sake of brevity, the liveness (AL1) properties that apply to this example will only be discussed in Section 3.2.

These properties describe the dynamic behaviour of the system. In practice, some (such as AS2-02, AS3-01 and AS4-01) followed rather directly from the problem statement. Others expressed the nature of a FIFO queue (AS4-04 and AS4-05). The rest of them (AS2-01, AS4-02, AS4-03, AP1-01 and AP1-02) were evolved in an iterative process, as one reflected more deeply on the problem. Note that initially there may be a measure of uncertainty about classification. For example, should AS2-02 and AS4-01 not be seen as global invariants? In the former case, the matter is non-critical, while in the latter case, the "unless" nature of the property became apparent during an iteration through the next phase. Finally note that in the SLOOP context, deadlock freedom (a safety property) is not explicitly addressed during the analysis and design phases. It will be discussed in Section 4.

3.2 Design phase correctness properties

During the design phase this syntax is used to express in a more formal way class and method properties that were evolved during the analysis phase. The syntax largely corresponds to the well-known syntax used for terms in first order predicate logic, but it allows for a few extensions. In this section, the relevant extensions are mentioned before providing the design phase properties for the running example.

The first extension is the use of a set of fundamental logical relations in SLOOP terms. They are similar to the UNITY logical relations. The only ones relevant to the current example are the *unless*, *invariant* and *ensures* relations. A complete list is provided in [8].

To define these relations, let C denote a class in which PM and SM are arbitrary parallel and sequential methods respectively. " $\forall s : PM::p$ " asserts that predicate p holds for all statements s in PM . " $\forall PM::p$ " asserts that p holds for any PM . Similar interpretations apply to formulae containing existential quantification and sequential methods, SM . Finally, for predicates p and q , and for some statement s , the notation $\{p\} s \{q\}$ asserts that if p is true before an execution of s commences, then s will terminate, at which point q will be true. The *unless*, *invariant* and *ensures* relations are defined as follows:

p unless $q \equiv \forall SM :: \{p \wedge \neg q\} SM \{p \vee q\} \wedge$

$\forall PM :: \forall s:PM :: \{p \wedge \neg q\} s \{p \vee q\}$

invariant $p \equiv (\text{initial condition} \Rightarrow p) \wedge p$ unless false

p ensures $q \equiv (p$ unless $q) \wedge$

$\exists PM :: \exists s:PM :: \{p \wedge \neg q\} s \{q\}$

These three relations will be used to characterize class instance properties, whereas the *results-in* relation will be needed to characterize a sequential method, say SM of a class C . It is defined as follows:

p_{entry} results-in $q_{\text{exit}} \equiv (p_{\text{entry}} \Rightarrow \diamond \text{at } \text{loc}_{\text{exit}} \wedge q_{\text{exit}})$

where p_{entry} represents SM 's preconditions, \diamond is the eventually temporal logic operator, loc_{exit} represents an exit location of SM and q_{exit} represents SM 's postconditions.

Another extension to the syntax of SLOOP logical terms is to allow the use of accessor method calls as expressions within a logical term. (An accessor method does not have side-effects. It simply reports on the state of the implicit parameter object.) In addition, a particular accessor predicate method is presumed to exist for all SLOOP classes, viz. `postHolds`. A call of the form:

`iparm.postHolds(someMeth(args))`

returns a Boolean to indicate whether the call `iparm.someMeth(args)` renders the postcondition of `someMeth` true or false with respect to the arguments, `args`. The rationale for this construct is provided in Section 3.3.

Finally, note that a quantification of the form $\forall x$ where $p(x)$ refers to all x for which predicate $p(x)$ is true. In certain predicates below the following particular form is used:

$\forall \text{obj where next = obj} :: \text{next} = \text{obj unless } q$

This form should be construed to mean that the quantified variable `obj` refers to the "old" value of the program variable `next` – i.e. prior to the execution of any statement implied by the *unless* relation.

The example system's design phase correctness properties may now be articulated in terms of the notation and syntax outlined above. The reader may easily verify that each

analysis phase correctness property has been re-stated in formal terms. Notice that objects, variable names, accessor method names and return types suggest themselves while applying one's mind to the problem of expressing these properties in the formal SLOOP notation. These should be noted separately but also referenced explicitly in the correctness properties.

AS2-01: *invariant* `this.isReady() \Rightarrow next.isNil()`

AS2-02: *invariant* `buff.size() \leq max`

AS3-01: *invariant* `buff.size() \leq maxToDate`

AS4-01: $\forall k$ where $0 \leq k \wedge k \leq \text{max} ::$
`maxToDate = k unless maxToDate > k`

AS4-02: $\forall \text{obj where next = obj} :: \text{next} = \text{obj unless}$
`this.postHolds(use(obj)) \wedge buff.isIn(obj) \wedge`
`next.isNil()`

AS4-03: $\forall \text{obj where buff.isIn(obj)} ::$
`buff.isIn(obj) unless`

`consumer.postHolds(pass(obj))`

AS4-04: $\forall \text{objY where } \neg \text{buff.isIn(objY)} ::$

$\neg \text{buff.isIn(objY) unless buff.isLast() = objY}$

AS4-05: $\forall (\text{objX}, \text{objY})$ where

$\neg(\text{objX} = \text{objY}) \wedge \text{buff.isIn(objX)} \wedge$

$\text{buff.isIn(objY)} ::$

$\text{buff.indexOf(objX)} < \text{buff.indexOf(objY) unless}$

$\neg \text{buff.isIn(objX)} \wedge$

buff.isIn(objY)

AP1-01: $\forall \text{obj where next = obj} ::$

$\text{next} = \text{obj} \wedge \text{buff.size()} < \text{max ensures}$

`this.postHolds(use(obj)) \wedge`

`buff.isIn(obj) \wedge next.isNil()`

AP1-02: $\forall \text{obj where buff.isIn(obj)} ::$

`buff.isIn(obj) \wedge consumer.isReady() ensures`

$\neg \text{buff.isIn(obj)} \wedge$

`consumer.postHolds(pass(obj))`

Note: An alternative for AS2-01 that is more of a direct translation of its verbal statement, but which is the logical equivalent of the version above, is:

AS2-01: *invariant* $\neg \text{next.isNil()} \Rightarrow \neg \text{this.isReady()}$

3.3 Refinement to SLOOP statements

Once the design phase correctness properties have been specified, then SLOOP methods can be defined. The liveness and/or precedence properties provide the basis for deriving the parallel methods. These will now be derived, with accompanying motivating narrative.

The infinitely often execution of the statements of the parallel methods has to result in the desired progress. Properties *API-01* and *API-02* specify progress to be made by a Dispatcher class instance. Briefly, if the latter has accepted a new element from the Producer instance, then the new element should be used and added to `buff` once there is space in that queue. In addition, the elements in `buff` should be passed to the Consumer instance whenever the latter is ready to accept them. This is the crux of the functionality of the Dispatcher class.

Since all the parallel statements of the class execute infinitely often, in principle they could all be grouped into a single monolithic parallel method. However, in general it is judicious to group parallel statements into separate parallel methods by functionality. That way, when a certain functionality needs to be overridden during subclassing, parallel methods that have nothing to do with that particular functionality are unaffected. In the present example, the functionality of the Dispatcher class is very simple and a single parallel method called `p_dispatch` therefore suffices as a container for all parallel statements.

A useful heuristic for deriving parallel statements from properties of the form "*p ensures q*", is to rely on information in the conjuncts in *p* to derive the condition for the *if* clause. The conjuncts in *q* then determine *component-parts* of these statements. Note however that the conjuncts in the correctness property cannot be naïvely mapped to expressions in the SLOOP statement. For example, the use of `obj` in *API-01* to refer to the old value of `next` has already been noted. There is thus no reason to refer to `obj` in the SLOOP statement derived from *API-01*.

Property API-01: Based on these observations, the following statement in the `p_dispatch` method, if executed infinitely often, will satisfy correctness property *API-01*:

```
if (next.notNull() && buff.size() < max)
  {this.use(next)  \+
   buff.add(next)  \+
   next = nil
  }                                     (S1)
```

Property API-02: This property likewise suggests the following parallel statement:

```
if (consumer.isReady() &&
    buff.isIn(obj))
  {consumer.pass(obj) \+
   buff.remove(obj)
  }                                     (S2)
```

We shall refer to these two statements and their later refinements as *S1* and *S2* respectively. At this stage *S2* still refers to `obj` since *API-02* does not provide any additional information regarding the identity of `obj`. It will be necessary to consider the other correctness properties before *S2* can be refined further.

The derivation of *S1* and *S2* demonstrates the advantages of the SLOOP computational model: these statements were derived without reference to location counters. Since they are executed infinitely often, the correct results will be achieved, provided their respective preconditions become true at some point and remain true until the particular statement is selected for execution.

While the above versions of *S1* and *S2* take care of the liveness and precedence properties, the safety properties (*AS2*, *AS3* and *AS4*) have to be considered as well. They

provide the necessary information for the refinements of these statements.

Property AS4-04: This property prescribes how the new object should be added to `buff`, *i.e.* always at the end. Consequently the `buff.add(next)` method call in *S1* should be refined to `buff.addLast(next)`.

Property AS3-01: This property describes the invariant relationship between the size of `buff` and the value of `maxToDate`. Inspection of *S1* shows that its execution may violate this invariant. Specifically, when `buff.addLast(next)` executes, the size of `buff` is implicitly incremented by one. The value of `maxToDate` should thus be updated whenever both of the following are true: (1) the conditions apply under which `buff.addLast(next)` executes; and (2) the resulting size of `buff` is greater than the current value of `maxToDate`. These considerations suggest amending *S1* as follows:

```
if (next.notNull() && buff.size() < max)
  {this.use(next)  \+
   buff.addLast(next)  \+
   next = nil      \+
   if buff.size()+1 > maxToDate)
     {maxToDate = buff.size()+1}
  }
```

However, the SLOOP syntax does not allow the *component-parts* to be *if*-statements. Instead, another *statement-component* is added to *S*, yielding a revised *S1* in which two *statement-components* execute as one atomic action:

```
if (next.notNull() && buff.size() < max)
  {this.use(next)  \+
   buff.addLast(next)  \+
   next = nil
  }
||
if (next.notNull() &&
    buff.size() < max &&
    buff.size()+1 > maxToDate)
  {maxToDate = buff.size()+1}          (S1)
```

Recall that all *if* clauses of all components of a particular statement are evaluated before any of the component parts are executed. All evaluations of `buff.size()` in statement *S1* will therefore yield the same result. Thereafter, evaluations mentioned in step 2 of section 2.3 take place, obtaining values for the following (in any arbitrary order): `this`, `next`, `buff`, `nil`, `(buff.size() + 1)`. Finally, in step 3 of section 2.3 the following *component-parts* are executed (in any arbitrary order), thereby achieving the outcome desired of *S1*:

```
this.use(next)
buff.addLast(next)
next = nil
maxToDate = buff.size() + 1
```

Property AS4-05: This property requires that elements should only be removed from the head of `buff`. This indicates that in statement *S2*, all references to `obj` can be changed to `buff.first()`. Furthermore the call to

```

buff.remove(obj) can be changed to
buff.removeFirst(). The revised version of S2
therefore becomes:
if(consumer.isReady() && buff.size() > 0)
    {consumer.pass(buff.first()) \+
      buff.removeFirst()
    }

```

(S2)

Property AS2-01: The correctness properties may also assist in defining sequential methods. For example, property *AS2-01* can be used to derive the definition of the method `isReady()`. Being sequential, the method has to terminate, which means that a total correctness property may be defined for it. The property *AS2-01* suggests the following total correctness property:

DL1-01: true *results-in*

```
methodReturnValue = next.isNil()
```

where `methodReturnValue` is a *pseudo-variable* that references the method's return value. The Java code for the body of `isReady()` that complies with DL1-01 is simply: `return next.isNil()`

Properties *AS4-02* and *API-01* refer to a sequential method called `use()`. The problem formulation suggests its existence, but does not indicate anything about its behaviour. In the Dispatcher class this method may therefore merely return the implicit parameter. Once a more refined requirements statement is given, a subclass may override the method as required. Other sequential methods mentioned in Dispatcher's correctness properties, such as `isIn()`, `isLast()`, `size()` and `indexOf()`, will not be investigated further here. These are DispatcherQ instance methods – their implicit parameter being the DispatcherQ instance, `buff`. An existing library class (e.g. the Java array list class) could potentially be used to implement this class – a matter that should be investigated during the design phase. In such an event, the possibility that the reused class already has instance methods with the required functionality should be investigated. In the SLOOP method, these classes need not be re-implemented, but it may be necessary to develop the class properties (for later reuse) along the lines illustrated above. The reuse of the correctness properties is the topic of the next section.

Note that properties *AS2-02*, *AS4-01*, *AS4-02* and *AS4-03* have not been used to derive methods. Rather than indicating what statements should be included, these properties dictate what may not be included. For example, property *AS2-02* forbids any statement that causes the size of `buff` to exceed `max`. Similarly, property *AS4-01* ensures that no statement will ever set the value of `maxToDate` to a value less than its current value. Property *AS4-02* forbids `next` from being set to nil at any point other than in *S1*. Similarly, property *AS4-03* implies that the only time when an element may be removed from `buff` is when it is passed to the consumer, as in *S2*.

Viewed dynamically, the system in the above example contains a parallel method, `p_dispatch`, which is invoked infinitely often from within the *activation-section* of the SLOOP program. The two parallel statements comprising the `p_dispatch` method are therefore executed infinitely often. The system reacts to events via these two statements. A more complex system would contain more classes and therefore more parallel methods. The only implication of invoking more parallel methods would be that there would be more parallel statements executing infinitely often.

Since the program has to be correct regardless of the execution order of the parallel statements, there is no need to take location counters into consideration. The various correctness properties specified for the program are indeed satisfied by merely executing statements *S1* and *S2* infinitely often.

3.4 Reusing correctness properties

SLOOP relies on the “design by contract” notion articulated by Meyer [3]. When a method call appears in a correctness property it is assumed that it will provide the correct result. It is not necessary to reason about its correctness from first principles; that is done when that particular method is defined. When it is necessary to refer to the postconditions of another method (possibly of another class) then the `postHolds()` method may be used. Property *API-02* contains such an example.

SLOOP therefore assumes that correctness properties have been set for each class. When constructing a system, these should be examined when investigating classes to be reused. The correctness properties of a reusable class therefore constitute an important part of its interface. These reusable classes therefore have to collaborate within the constraints of the correctness properties specified for the required system as a whole, while the integrity of the correctness properties specified for each individual class have to be preserved during the collaboration with other classes.

When evaluating classes for potential reuse, it is likely that the correctness properties of the required classes and those present in the repository of reusable classes will not be specified in terms of the same concepts and variables. The SLOOP method assists the software designer in the task of finding a matching class in the following way: It recommends that this step is performed before the informal correctness properties of the required class have been refined into a formal specification. Since the SLOOP method mandates that the correctness properties of the repository class are specified both informally and formally, it aids understandability and the software designer is better able to find a possible match.

An added advantage of this approach is that it allows the software designer to reuse the mapping of informal to formal specification that has already been done in the reusable class. Furthermore, it is likely that the reusable

class will highlight deficiencies in the specification of the required class and these refinements can therefore also be reused.

Since a SLOOP program consists of a collection of classes and each class may contain parallel methods, the effect of the union of all these parallel statements needs to be considered in terms of the correctness properties of the program. Due to the data encapsulation feature that is inherent in an object-oriented method, the union of the parallel statements of the various classes is actually a restricted union (modification of variables is restricted to the statements belonging to the class containing the variables) and the correctness properties of the individual classes still hold in the union of the classes.

Classes can also be added through specialization. The effect of inheritance on correctness properties is described in the next section. The effect of program structuring on correctness properties is discussed at length in [8].

3.5 SLOOP and inheritance

There are three distinct ways in which SLOOP's correctness properties are affected by inheritance: correctness properties in a subclass may reuse, override or add to those of its superclass. When a correctness property is overridden, preconditions may not be strengthened and postconditions may not be weakened.

All correctness properties are identified by a number, followed by a class name in brackets. All correctness property numbers are therefore unique with respect to a specific class. When a correctness property of an ancestor applies to a descendant as well, the property is not listed again in the properties section of the descendant. When this property is referenced with respect to the descendant, the number and class name of the ancestor are used. If a new property is added for the descendant, a unique number for that descendant class is allocated. If a correctness property is redefined in a descendant, then the class name and number associated with the property in the ancestor are retained in the descendant.

SLOOP also allows for parallel methods in descendant classes to override parallel methods in ancestor classes. Additional parallel methods may also be defined in the descendant classes.

4. MAPPINGS TO TARGET ARCHITECTURES

One of the characteristics of the SLOOP method is that the target architecture is ignored during the analysis and design phases. Reasoning about correctness is in terms of the atomic parallel statements. During the implementation phase a mapping to the target architecture is performed, the main concern being to preserve the atomicity of the parallel statement. In doing so, the correctness properties derived in previous phases are carried over to the implementation.

In [5] various heuristics are defined for the mapping of UNITY programs to sequential, to asynchronous shared-memory, to synchronous shared-memory and to distributed systems. They are used as a basis for mapping SLOOP programs to various architectures.

Mapping to a sequential architecture is the simplest. All the statements are assigned to the same process on the same processor. Although the parallel statements are executed sequentially, their order of appearance is irrelevant. The only important issue is that they should be enclosed in an infinite loop in order to ensure that they are executed infinitely often. Each parallel statement should appear at least once within this loop. Since there is no concurrency, the atomicity of each statement is assured.

In a synchronous shared-memory architecture, only one parallel statement is executed at a time. Each of its *statement-components* is assigned to a different (logical) processor. Execution takes place in three phases or 'clock ticks' corresponding to the evaluation steps listed in Section 2.3. Thus, all *statement-components* undergo the processing mentioned in steps 1, 2 and 3 at clock ticks 1, 2 and 3 respectively. Although some processors might be idle during some clock cycles (all parallel statements do not necessarily have the same number of *statement-components*), this mapping ensures the atomicity of each parallel statement. The rules as specified in Section 2.3 must be preserved in order to ensure the integrity of the data. The *statement-components* assigned to each processor have to be enclosed in an infinite loop.

In an asynchronous shared-memory architecture, there is no common clock. As a result, all the *statement-components* of a given parallel statement have to be executed by the same processor. In this case the various parallel methods are assigned to different processors. At each invocation of a parallel method only one of its statements is executed. This arrangement ensures that although multiple statements may be grouped into a single method for structuring purposes, the atomic unit of execution is still the parallel statement. It also allows for the arbitrary interleaving of parallel statements that is required by the computational model. As a special case, multiple parallel statements may execute simultaneously, if they do not reference the same objects.

In a distributed system the various objects, along with their methods, have to be assigned to the processors available in the system. The parallel methods of the objects assigned to a particular processor have to be enclosed in an infinite loop on that processor. Again each invocation of a parallel method results in the execution of only one of its statements. As in the case of the asynchronous shared-memory architecture, multiple parallel statements may be executed simultaneously, provided they do not reference the same objects.

One way of achieving this latter requirement is by ensuring that the resources necessary for executing a

statement are reserved for that statement prior to its execution. An object may only be reserved by one client at a time. Various object reservation algorithms can be implemented to ensure that deadlock does not occur. Details can be found in [8].

Note that these deadlock prevention algorithms are concerned with the correctness of the mapping procedures - they are completely independent of the application itself. Once their correctness has been shown, they can be reused for all applications. Thus, the design of the SLOOP program remains the same, *i.e.* there is no need to be concerned with absence of deadlock properties in the SLOOP program itself.

Another important aspect of the mapping to the various architectures is the preservation of the semantics of the SLOOP parallel statement. For example, if a statement containing multiple components is executed on a single processor, the components need to be ordered so as to preserve the semantics of the statement. It might even be necessary to introduce additional components in order to achieve this. For example, in mapping the SLOOP statement $x = y \parallel y = x$ to a single processor, a statement component $t_{\text{emp}} = x$ has to be introduced before the execution of $x = y$ and the statement component $y = x$ has to be modified to $y = t_{\text{emp}}$ and executed after the statement $x = y$.

Other aspects of distributed object computing, such as locating the objects and converting the messages into a format that can be transmitted over communication channels, form part of middleware infrastructures such as CORBA [11].

5. RELATED WORK

The SLOOP method is a semi-formal method which has as its basis a formal notation for specifying the structure and behaviour of a software system. It is intended for software practitioners rather than theoreticians and is therefore not aimed at competing with formal methods such as the Vienna Development Method (VDM) [12], Z [13], B [14] and Communicating Sequential Processes (CSP) [15] (and their many variants). Research within the UNITY framework has also focused on more formal aspects [16]. The SLOOP method should thus be compared with methods that address roughly the same problem space as it does.

The most significant difference between SLOOP and most other semi-formal methods is its computational model. Since the SLOOP computational model greatly simplifies correctness reasoning, the SLOOP method has a distinct advantage over methods such as UML [17] and the

Business Object Notation (BON) [18]⁵ that use conventional computational models.

Action systems have a mathematical model that is equivalent to that of UNITY [19] and therefore also to that of SLOOP. Recent work on action systems includes some extensions to support modularisation [20] and the addition of object-oriented constructs [21]. The concepts described in [21] have been implemented in an experimental language called DisCo (*Distributed Cooperation*). Although DisCo is categorized as a formal specification language [22], it is stated in [23] that the notation is appropriate for the complete spectrum of users, from software practitioners to theoreticians. The SLOOP method differs in several ways from the work presented in [20] and [21], most notably the following:

- The extension to action systems described in [20] does not deal with object-orientation. It shows how Dijkstra's guarded command language [24] can be extended with the concept of procedures (including local, imported and exported variables), *i.e.* it adds support for modularisation. It also describes how the language Oberon can be modified to support action systems. Oberon is particularly suitable because it already supports concepts such as imported and exported variables, etc.
- Modular action systems [20] do not have object-orientation features such as polymorphism and inheritance.
- Multiple partitioning options are available for the actions and variables of modular action systems. This raises the possibility of problems such as deadlock. In the case of modular action systems the solutions to the problems associated with mappings are not treated as reusable artifacts. In contrast, this is an important aspect of the SLOOP method, as was discussed in section 4.
- In the object-oriented action system described in [21] the actions and classes are viewed as separate entities. Multiple classes may participate in a joint action. The actions replace the concept of methods. A class therefore has attributes, but no methods associated with it. An action does not form part of a class. The statements in an action may modify the variables of the objects participating in the action directly.
- SLOOP parallel methods form an integral part of a specific class. They may invoke methods of other class(es) and in that sense there is synchronisation with the other class(es), but a parallel method belongs to a single class and can only be refined by the subclasses of that class. SLOOP parallel methods are therefore in line with the concept of encapsulation and information hiding. The contents of the parallel method are not important to any class other than the containing class and its subclasses. Only its correctness properties need to be visible. A SLOOP parallel statement may invoke sequential methods, which ensures that the structuring capabilities of object-orientation are exploited fully.

⁵ BON is designed to work seamlessly with Eiffel [3], *i.e.* a BON specification is transformed into an Eiffel program during the implementation phase.

- Another difference between object-oriented action systems and SLOOP is the way in which inheritance is handled. In the case of object-oriented action systems the preconditions of an operation can be strengthened during specialisation. This is the opposite of the inheritance rule in SLOOP, which specifies that preconditions may not be strengthened (they may remain the same or be weakened). This is to ensure that subclasses will always accept requests from clients that are unaware of the fact that they are not dealing with the parent class (useful in polymorphism) [3]. The rationale for this inheritance rule in SLOOP is discussed in more detail in both [3] and [8].
- In the SLOOP method the emphasis is on specifying a set of correctness properties and using those as the basis for the derivation of the SLOOP program. Informal correctness reasoning consists of providing correctness arguments showing that the statements of the SLOOP program satisfy the specified properties. The SLOOP class and method specifications include correctness property specifications. In the case of DisCo (which is based on object-oriented action systems), the emphasis is on specifying the system in terms of a set of classes and actions. Class specifications may contain invariants. The actions are specified in terms of the modifications to the relevant variables. An animation tool is provided to validate the DisCo specification against the informal requirements of the system. Verification of invariant properties can be performed using a prototype tool which employs a Prototype Verification System (PVS) theorem prover.
- The formal basis for action systems is the Temporal Logic of Actions [21], whereas SLOOP is based on the UNITY programming logic [5].
- The issues involved regarding the mapping to various architectures are not considered in [21]. SLOOP explores this aspect to a great level of detail.
- Seamlessness is a very important aspect of the SLOOP method. During the analysis and design phases the focus is on the specification of correctness properties. The notation that is used for the formal specification of the correctness properties contains Java method invocations, making the derivation of SLOOP statements from the formal correctness properties relatively simple. If the target implementation language is Java, the transition from the design phase to the implementation phase is seamless. The DisCo method focuses on the analysis and design phases of the software development life cycle.
- The emphasis in [20] and [21] is on formally applying refinement calculus on action systems that are extended to include modularisation or object-oriented concepts respectively. The focus in SLOOP is to take advantage of the simplifying aspects of the computational model in order to enable informal reasoning about correctness of object-oriented systems. The SLOOP method therefore provides a list of correctness properties and shows why they are important for design when an informal approach is used.

6. APPLICABILITY AND CONCLUSIONS

In [8] a non-trivial running example was used to show how the SLOOP method scales up. The method has also

been applied successfully to several different types of design problems. This was achieved by incorporating many of the architectural and design patterns described in [25] and [26], respectively, into a system that was designed using the SLOOP method. Examples of the patterns that have been used include the Pipes and Filters architectural pattern, the Reflection architectural pattern, the Factory Method and Singleton creational design patterns, the Adapter and Flyweight structural design patterns and the State and Template Method behavioural design patterns.

The most remarkable aspect of these investigations was the fact that once the design phase had been completed no further modifications to the design or the logic were required. The executable programs produced the correct results when they were first run. It is strongly believed that this can be attributed to the "constructive approach" design philosophy. The emphasis on correctness properties during the analysis, design and implementation phases promotes a disciplined and careful approach towards the software development process.

Another important feature is the computational model. Not only does it simplify correctness reasoning, but it is also possible to map a SLOOP program first to a sequential executable program and then to a concurrent one without changing a single SLOOP statement. Both mappings produced the correct results when they were first run. The method thus combines the excellent structuring capabilities inherent in an object-oriented method with the benefits gained through a semi-formal approach towards software development.

Nevertheless, the question needs to be asked: Is the method accessible to the average software engineer? The answer to this question hinges critically on answers to the following sub-questions:

- Is the availability of a checklist of general properties a sufficient aid for identifying all relevant system-specific properties?
- How difficult is it to articulate the informally specified analysis phase correctness properties?
- Once these informal properties are available, how difficult is it to formulate the more formal design phase properties?

Deriving SLOOP code from the design phase properties is arguably somewhat less challenging than the latter two tasks, but would, of course, also need to be considered. Clearly, performing these tasks will require both training and management commitment. However, such training as well as the performance of the actual tasks could be greatly facilitated if an appropriate tool in support of the SLOOP method was developed. One option is to leverage Java Modeling Language (JML) tools [27] for this purpose. Since JML is based on a different computational model, modifications to the tools would be necessary.

In addition a SLOOP tool could also, for example, keep track of checklist properties that have been considered, provide links between the various correctness property formulations, compare design phase properties with parallel method statements and advise on how the latter might be formulated, assist in mapping SLOOP programs to target programs and to target hardware architectures, have elements of a literate programming environment built into it, etc. The next phase of this research will aim to develop and field-test such a tool.

7. REFERENCES

- [1] B. Selic: "An efficient object-oriented variation of the statecharts formalism for distributed real-time systems", *IFIP Conference on hardware description languages and their applications*, Ottawa, Canada, April 26 - 28, 1993.
- [2] J. Sifakis: "Integration, the price of success", *Formal methods 1999*, Toulouse, France, Vol. 1, LNCS 1708, pp. 52-55, September 20-24, 1999.
- [3] B. Meyer: *Object-oriented software construction*, Prentice-Hall, Inc, New Jersey, second edition, 1997.
- [4] N. Francez: *Program verification*, Addison-Wesley, 1992.
- [5] K.M. Chandy and J. Misra: *Parallel program design: a foundation*, Addison-Wesley, U.S.A., 1988.
- [6] R. Gerth and A. Pnueli: "Rooting UNITY", *Communications of the ACM*, No. 1, pp 11-19, 1989.
- [7] A. Goldberg and D. Robson: *Smalltalk-80, the language*, Addison-Wesley, 1989.
- [8] M. M. Ross: *UNITY-inspired object-oriented concurrent system development*, Ph.D thesis, University of Pretoria, 2001.
- [9] J. Wing: "A specifier's introduction to formal methods", *Computer*, pp 8-24, September 1990.
- [10] Z. Manna and A. Pnueli: "Verification of concurrent programs: the temporal framework", *The correctness problem in computer science*, International lecture series in computer science, Academic Press, London, pp. 215-274, 1981.
- [11] R. Orfali, D. Harkey and J. Edwards: *Instant CORBA*, John Wiley & Sons, Inc., U.S.A, 1997.
- [12] C. B. Jones: *Systematic software development using VDM*, Prentice-Hall Inc., London, 1986.
- [13] J.M. Spivey: *The Z notation: A reference manual*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, second edition, 1992.
- [14] J.-R. Abrial: *The B-book - assigning programs to meanings*, Cambridge University Press, 1996.
- [15] C.A.R. Hoare: *Communicating sequential processes*, Prentice-Hall Inc., London, 1985.
- [16] M. Charpentier and K. M. Chandy: "Towards a compositional approach to the design and verification of distributed systems", *Formal methods 1999*, Toulouse, France, Vol. 1, LNCS 1708, pp. 570-589, September 20-24, 1999.
- [17] Rational Software Corporation: "UML Resource Center" <http://www-306.ibm.com/software/rational/uml/>.
- [18] R.F. Paige and J. S. Ostroff: "Developing BON as an industrial-strength formal method", *Formal methods 1999*, Toulouse, France, Vol. 1, LNCS 1708, pp. 834-853, September 20-24, 1999.
- [19] R.-J.R. Back and R. Kurki-Suonio: "Decentralization of process nets with centralized control", *Distributed Computing*, No. 3, pp. 73-87, 1989.
- [20] R.-J.R. Back, and K. Sere: "From action systems to modular systems", *Symposium of Formal Methods Europe (FME) 94*, lecture notes in computer science, No. 873, pp. 1-25, 1994.
- [21] R. Kurki-Suonio: "Fundamentals of object-oriented specification and modeling of collective behaviours", *Object-oriented behavioral specifications*, Kluwer Academic, London, pp. 101-120, 1996.
- [22] A. Ruiz-Delgado, D. Pitt and C. Smythe: "A review of object-oriented approaches in formal methods", *The Computer Journal*, Vol. 38, No. 10, pp. 777 - 784, 1995.
- [23] M. Katara: "A short tutorial on DisCo". On-line Web tutorial at <http://www.cs.tut.fi/~laitos/DisCo/tutorial/Tutorial.html>.
- [24] E.W. Dijkstra: *A discipline of programming*, Prentice-Hall Inc., Englewood Cliffs, N.J, 1976.
- [25] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal: *Pattern-oriented software architecture: a system of patterns*, John Wiley & Sons Ltd, England, 1996.
- [26] E. Gamma, R. Helm, R. Johnson and J. Vlissides: *Design patterns, elements of reusable object-oriented software*, Addison-Wesley, 1995.
- [27] G.T. Leavens and Y. Cheon: "Design by contract", *On-line paper* at <http://www.jmlspecs.org/>, 8 February 2004.