FORMAL DEVELOPMENT OF OBJECT-BASED SYSTEMS IN A TEMPORAL LOGIC SETTING

E. Canver and F. W. von Henke

Abt. Künstliche Intelligenz
Universität Ulm
89069 Ulm, Germany
email: {canver,vhenke}@ki.informatik.uni-ulm.de

Abstract: This paper presents TLO, an approach to the formal development of object-based systems in a temporal logic framework. The behavior of an object-based system is viewed as derivable from the behaviors of its constituent component objects. Temporal logic is a formalism well suited for specifying behavior of concurrent systems; it also provides conceptually simple notions of composition and refinement: Composition of objects is expressed as conjunction of the associated component specifications. The refinement relation between a low-level and a high-level specification requires that the former specification implies the latter.

Specifically in an object-based approach, systems and their components need to be viewed as open systems: Each object guarantees some service (behavior), provided its environment conforms to certain assumptions. Hence, such components are most appropriately specified in an assumption/guarantee style.

TLO adopts TLA as the underlying logical formalism. It encompasses a specification language for object-based designs and a corresponding method for specification and verification. The concepts are illustrated by an example involving both functional and fault-tolerance requirements.

INTRODUCTION

Object-based systems are in general viewed as entities interacting with their environment. In this view, an object is an open system that exhibits some behavior that may depend on the behavior of its environment; an object may be "active", i.e. have some internal activity. Several objects may be composed to form a concurrent system.

P. Ciancarini et al. (eds.), *Formal Methods for Open Object-Based Distributed Systems*

The temporal logic of actions (TLA) [11, 12] is a well-established formalism for specifying the behavior of concurrent state transition systems; related methodologies for specifying, composing, and refining open systems have been proposed. Open systems are typically specified in an *assumption/guarantee* style; composition is represented by simple logical conjunction of specifications and refinement by logical implication. This suggests that TLA may be quite sufficient for specifying the behavior of concurrent systems. However, just as first-order predicate logic is theoretically sufficient, but in practice not an adequate specification language for sequential systems, we regard TLA as too much of a "raw" logic for being truly practical, specifically since the logic itself does not provide a concept of true modules or objects.

In this paper, we propose an object-based specification language TLO (Temporal Logic for Objects) that builds on TLA and the associated methodology as its semantic basis. TLO combines constructs for organizing specifications into interacting objects with a manner of using TLA formulae for specifications that directly reflects and supports the assumption/guarantee style. The semantics of a TLO specification is fully expressed by the TLA formulae into which it expands. Within the TLO framework, parameterized objects and compositions of objects into systems can be formally specified; the framework also supports refinement of objects. This allows to generate proof obligations for assessing properties and the correctness of refinements which then can be analyzed using the rules of TLA.

Extending formal languages with object-based and object-oriented paradigms is the goal of many approaches that regard the behavior of objects as a main topic of interest. For example, Object-Z [8] extends Z [17] with more advanced object-oriented structuring mechanism than currently present in our approach; it provides its own variant of temporal logic for specifying invariants and liveness properties. Calculi for refining Z specifications have been proposed; however, as of yet the notion of refinement is not formally defined for Object-Z. The formal object-oriented specification language TROLL [10] has initially been defined for designing information systems. TROLL has later been extended [7] by defining a temporal logic for TROLL and developing a theory for refinement based on this temporal logic. A proof system is not available for this approach. Maude [13] is a calculus for expressing the execution of concurrent object-oriented systems. In contrast to such approaches, TLO aims at specifying properties and refinements of systems in order to be able to formally verify them. Therefore we have chosen TLA as formal basis which provides a well developed theory for refinements and proofs, and we have extended it with object-based specification language constructs.

Temporal logic is also used in other approaches for specifying behavioral properties. TRIO$^+$ [14] is a formalism based on a temporal logic; objects are composed by "connections" which express equivalences on the values of items in imported modules; the data types available are restricted to enumeration and built-in scalar types. The semantics of a class together with its connections and parameter instantiations is given by flattening the textual structure into an equivalent specification without connections and parameter instantiations. Our work is more general than this approach since we retain the structure of the specifications in the TLA representation. We also allow arbitrary user-defined data types.

The work on DisCo [9] is similar to ours in that TLA is used as the target logic for reasoning about specifications. The DisCo approach is based on joint actions similar to the action systems of Back [5, 4]: several objects may participate in one action which allows them to interact with each other; there are no actions internal to the objects. This is in contrast to our approach, where objects may have internal activities. Another important difference to our approach is that DisCo specifications describe closed systems with the nondeterminism of action parameters used for describing unconstrained environments, whereas in our model of open systems the assumptions about the environment are stated explicitly.

A novelty of our approach is the use of "shared objects" as parameters of other objects for specifying the composition of interacting components of object-based systems. This allows us to clearly define, in terms of methods of the parameter object, in which way components may access the state of the shared object. Another advantage of this approach is the possibility to refine the interaction structure between components by refining the parameter object.

The remainder of the paper first gives a short overview of TLA and the associated methodology. Subsequent sections present the specification constructs and refinement methodology adopted in TLO. They are illustrated by the top-down development of a small case study for moving the contents of a file, involving two steps of refinement, with the second introducing failure assumptions and a fault tolerance mechanism. Because of space limitations, the presentation is much abbreviated; a full account will be given in a forthcoming report.

OVERVIEW OF TLA

The temporal logic of actions [11, 12] is a formalism suitable for describing state transition systems, their composition, and their refinement. Transition systems are usually described by TLA formulae in canonic normal form, $I \wedge \Box[N]_f \wedge L$, with formulae I, N, and L over state variables. State variables may change their values over time. I is a predicate describing the initial state. N is an action formula describing the state transitions by a relation over successive states. A primed occurrence of a state variable (like e.g. v') in formula N refers to the value of the variable in the second state whereas a non-primed occurrence refers to its value in the first state. The subscript f attached to the specification of state transitions allows the system to perform stuttering steps where the value of the expression f does not change. $\Box[N]_f$ expresses that each step of the system is a state transition corresponding to N or leaving expression f unchanged. L describes the liveness assumptions for the system often in the form of fairness assumptions on actions expressing that if an action is enabled long or often enough it will eventually be executed. TLA allows to write predicates of the form $Enabled(action\text{-}formula)$ expressing whether the action described by $action\text{-}formula$ can be executed in the current state. The specification of a system can be generalized by abstracting from concrete state variables occurring in the formulae. This is done by existential quantification over state variables [12].

Composition is expressed as conjunction and refinement as implication. For two TLA formulae, $\Phi \triangleq \exists x : \Pi_1$ and $\Psi \triangleq \exists y : \Pi_2$, describing systems, Φ is a refinement of Ψ if the implication $\Phi \Rightarrow \Psi$ holds. One way of proving this implication

is to construct a suitable refinement formula $\exists\, y : R$ such that

$$\begin{aligned} \Pi_1 \wedge R &\Rightarrow \Pi_2 \\ \Pi_1 &\Rightarrow \exists\, y : R \end{aligned}$$

In simple cases R is a refinement map of the form $\Box(y = map(x))$. When an atomic state transition is refined into several state transitions, then R can be used to express when and how state changes occurring in the refinement Φ become visible in the abstract system Ψ, for example by mapping several states of the refinement to the same abstract state; for further details see [12].

Large systems should be built from smaller cooperating components. Each such component can be viewed as an open system that guarantees some service, provided that its context or environment, i.e. the collection of the other components, satisfies certain assumptions. Specification, composition, and refinement of open systems can be encoded in TLA [2, 1].

An open system is specified in the form $OS \triangleq E \stackrel{+}{\triangleright} M$ with E describing the assumptions about the environment and M describing the properties guaranteed by the system; E and M are assumed to be given in TLA's canonic normal form potentially with existential quantification over state variables. $E \stackrel{+}{\triangleright} M$ means that the system satisfies M at least one step longer than the environment satisfies E. In other words, the system may fail to satisfy M only *after* the environment has failed to satisfy E. For a discussion of this form of specification see [2, 1]. Note, that a closed system M is just a special case of open systems by defining the environment specification E to be *true*. A system composed from open system components is encoded as conjunction of component specifications ($CS \triangleq \wedge_{i=1}^{n}(E_i \stackrel{+}{\triangleright} M_i)$) and refinement is again encoded by implication (e.g. $CS \Rightarrow OS$ if CS is supposed to be a refinement of OS).

If the visible variables of the open system specifications can be partitioned into input variables e and output variables m such that changes to input and output variables do not occur simultaneously then $CS \Rightarrow OS$ can be verified by proving

- $E \wedge (\wedge_{j=1}^{n} \mathcal{C}(M_j)) \Rightarrow E_i$ for all $i = 1, \ldots, n$
- $E \wedge (\wedge_{j=1}^{n} \mathcal{C}(M_j)) \Rightarrow \mathcal{C}(M)$
- $E \wedge (\wedge_{j=1}^{n} M_j) \Rightarrow M$

where $\mathcal{C}(M)$ denotes a formula defining just the safety aspects that are described by formula M; $\mathcal{C}(M) \triangleq I \wedge \Box[N]_f$ if M is given in TLA's canonic normal form.

SPECIFICATION OF OBJECT BASED SYSTEMS

An object is usually regarded as an entity that encapsulates some state and that provides an interface consisting of methods to access and modify its state. It may have internal activities, which means that the state of the object may change even if no method is called. The internal activities and the effects of method execution define the behaviour of an object. Objects exhibiting the same behaviour are considered to belong to the same class; thus by describing a class, a group of objects with the same behaviour is specified.

Object-based systems are built by composing objects that cooperate with each other. Each component object can be viewed as an open system, with the other objects constituting its environment: an object guarantees some service provided that certain assumptions about its environment hold.

In the following sections, an approach to specifying classes of objects and refining those object-based specifications is presented; it corresponds to the methodology for TLA introduced in the preceding section. Specification language constructs are illustrated for plain (non-parameterized), parameterized, and aggregated specifications. Each specification describes a class of objects. The meaning of such specifications is given in terms of TLA formulae.

Plain Specifications

```
Sys : CLASS
  BEGIN
    IMPORTING    data types
    attr         : ATTRIBUTE type
    meth(x:type): METHOD type == effects(RESULT)
    act-name     : ACTION == effects
    INITIAL      predicate
    STEPS        act
    FAIRNESS     conjunction of WF(act_1), SF(act_2), ...
  END Sys
```

Figure 1: Main elements of a specification

A specification characterizes a class of objects which exhibit the same set of behaviours. Figure 1 shows the main elements of a specification. Data types and operations (functions and relations) on them are specified in separate modules (cf. modules of the PVS system [16, 15]) and imported into TLO specifications. The attributes correspond to the state variables in TLA; in contrast to TLA, they are typed.

The methods of an object provide means for an environment to access and modify the attributes. A method can be parameterized and may return a value. The state transition caused by executing a method is specified as a relation over two successive states; thus, the effect of a method is considered to be an atomic state transition. If the method returns a value, the formula describing its effects may contain the reserved word RESULT, as denoted by *effects*(RESULT) in figure 1; RESULT can be used like a variable, e.g. in an equation like RESULT=*expr*.

In order to be executed, a method needs to be called by the environment. The interaction between the environment and the object involves input and output variables. The parameter x of method *meth* is considered to be an input variable and the result of *meth* is stored in a new (implicitly defined) output variable ($result_{meth}$).

The approach presented here allows objects to have internal activities that can be observed but not directly controlled by the environment. Such internal activities are caused by internal state transitions represented in the specification as actions. It

is possible to define a name for the formula describing an internal action with the syntactic form `act-name: ACTION == effects`.

The initial, steps, and fairness clauses define the behavior of objects belonging to the specified class. The initial state of an object is defined by a predicate over its attributes. The steps section describes which actions (including actions associated with methods) may occur in the behavior of the object. They are combined as simultaneous or interleaving state transitions, i.e. occurring within the corresponding formula *act* in conjunctions or disjunctions respectively. State transitions can be specified to comply with a strong or weak fairness assumption. Strong fairness asserts that a state transition will eventually occur in a behavior if it is enabled just often enough. Weak fairness asserts that a state transition will eventually occur in a behavior if it is enabled long enough.

The synchronization between an object and its environment is handled by two additional boolean state variables for each method *meth*: an input variable $start_{meth}$ and an output variable $stop_{meth}$. $start_{meth}$ denotes whether *meth* has been invoked by the environment and $stop_{meth}$ denotes whether the action corresponding to the effects of *meth* has occurred (after it has been invoked). The values of $start_{meth}$ and $stop_{meth}$ can be combined to encode four states of a method:

$$
\begin{array}{lcl}
idle_{meth} & \triangleq & \neg start_{meth} \wedge \neg stop_{meth} \\
activated_{meth} & \triangleq & start_{meth} \wedge \neg stop_{meth} \\
returning_{meth} & \triangleq & start_{meth} \wedge stop_{meth} \\
finishing_{meth} & \triangleq & \neg start_{meth} \wedge stop_{meth}
\end{array}
$$

Initially a method is in an idle state. Executing method *meth* is encoded in the transition system by the following steps:

1. The environment changes the parameter to contain some valid value and changes the state of the method to $activated_{meth}$. Invoking a method *meth* with parameter *par* is expressed by the action $Invoke(meth(par))$ that is defined by the formula $(idle_{meth} \wedge x' = par \wedge start'_{meth} = true)$.

2. The method is executed, setting $stop_{meth}$ to true. This is expressed by the formula $(activated_{meth} \wedge stop'_{meth} = true \wedge effects(result'_{meth}))$.

3. The environment waits for the result value to become available, consumes the result and signals to the object that this has been done by taking the method into the state $finishing_{meth}$. The corresponding action $Wait(meth)$ is defined by the formula $(returning_{meth} \wedge start'_{meth} = false \wedge result'_{meth} = result_{meth})$.

4. The object sets the method state back to idle. This is expressed by the formula $(finishing_{meth} \wedge stop'_{meth} = false \wedge attr' = attr)$.

The environment may call a method. This involves steps 1 and 3 of executing a method. A special syntactic form is provided for expressing the actions associated with calling a method: `CALL(meth(par))`. This mainly encompasses the actions $Invoke(meth(par))$ and $Wait(meth)$ to occur one after the other. If this form is part of a larger action formula, denoted $Act(\texttt{CALL(meth(par))})$, then its meaning in terms

of TLA is

$$Act(\texttt{CALL(meth(par))}) \triangleq \begin{array}{l} Enabled(Act(Invoke(meth(par)))) \wedge Invoke(meth) \\ \vee\ Act(Wait(meth)) \end{array}$$

The internal actions associated with a method correspond to steps 2 and 4. For a method defined by `meth(x:type) : METHOD type == effects(RESULT)` the corresponding TLA formula is

$$action_{meth} \triangleq \begin{array}{l} (activated_{meth} \wedge stop'_{meth} = true \wedge effects(result'_{meth}) \\ \vee\ (finishing_{meth} \wedge stop'_{meth} = false \wedge attr' = attr) \end{array}$$

If *meth* occurs in the steps or the fairness section, then it is replaced by $action_{meth}$. The plain specification *Sys* given above corresponds to the following TLA specification:

$$\begin{array}{lcl} Initial_{Sys} & \triangleq & predicate \wedge \neg stop_{meth} \\ Steps_{Sys} & \triangleq & \Box[act]_{attr} \\ Fairness_{Sys} & \triangleq & WF_{\langle attr, stop_{meth} \rangle}(act_1) \wedge \ldots \\ Sys & \triangleq & Initial_{Sys} \wedge Steps_{Sys} \wedge Fairness_{Sys} \\ Spec(Sys) & \triangleq & \neg start_{meth} \stackrel{+}{\triangleright} \exists\, attr : Sys \end{array}$$

Aggregating Components and Parameterized Specifications

```
ASys1 : CLASS                    AC : CLASS
  BEGIN                            BEGIN
    AGGREGATE                        meth(x:type): METHOD type == ...
      A, B : AC                      ...
    END AGGREGATE                  END AC
  END ASys1
```

Figure 2: Aggregating Components

A large object-based specification can be built by composing smaller object-based specifications. The mechanism used here for composing objects is aggregation (cf. figure 2). Specification `ASys1` describes a class of objects which contain as components objects of a class corresponding to specification `AC`. An expression `A:AC` in the aggregation section can be understood as defining an instance of class `AC` with name `A`. Formally, this is modeled by prefixing all visible names from class `AC` with the instance name `A`. If $Spec(AC)$ denotes the meaning of class `AC`, then the meaning of `A:AC`, denoted $Spec(A)$, is obtained from $Spec(AC)$ by prefixing all names of input and output variables and the names of the methods with the instance name `A`[1]. The meaning of specification `ASys1` is the conjunction of its component specifications ($Spec(ASys1) \triangleq Spec(A) \wedge Spec(B)$)

[1]Prefixing all visible names in a formula ϕ with an instance name A is denoted by $A.\phi$

```
PSys [ m FROM A: AClass ] : CLASS
  BEGIN
    Body          % analogous to plain specification
  END PSys
```

Figure 3: A parameterized object-based specification

The aggregation of plain specifications allows the specification of systems composed from several components which, however, don't interact with each other. The main use of aggregation comes with the parameterization of object-based specifications since this allows to model components that interact with each other.

An object-based specification may be parameterized. Parameters may be types, to express a common form of genericity; this kind of parameterization is similar to the parameterization of abstract data types and will not be discussed further because of space limitations.

A specification may also take one or more object specifications as parameters. More precisely, a parameter consists of a list of method names together with the name and class of an object providing the methods (cf. figure 3). With respect to the parameter object, the specified body is restricted to using only the listed methods. The parameter specification describes the assumptions expected to hold in the environment. For the purpose of explaining the semantics of a parameterized specification, the semantics of the body is that of the corresponding unparameterized specification. A parameter class is regarded as reduced to a view corresponding to the methods explicitly listed; this means, in particular, that unlisted methods are regarded as replaced by internal actions (with parameters and results existentially quantified).

Let $Spec(A) \triangleq \exists\, a : A.AClass$ denote the meaning of `A:AClass` as described above. The meaning of m `FROM A:AClass` is defined by abstracting from each method `h` of `AClass` that does not occur in the list m. For this $action_h$ in formula $Spec(A)$ is replaced by $\exists\, A.par, A.result_{A.h}, start_h, stop_h : action_h$, where $A.par$ denotes the formal parameters of method h. Abstracting a formula F from methods not occurring in a list m is denoted in the following by $Restrict(m, F)$.

The parameterized specification `PSys` describes an open system in an assumption/guarantee style, which corresponds semantically to the TLA formula

$$
\begin{array}{l}
Spec(PSys) \triangleq \\
\quad \exists\, a : \mathcal{C}(Restrict(m, A.AClass)) \ \stackrel{+}{\triangleright} \\
\quad \exists\, attr(Body_{PSys}) : \quad (\wedge_{meth \in m} \neg start_{meth}) \wedge \mathcal{C}(Body_{PSys}) \\
\qquad\qquad\qquad\qquad \wedge\ (Restrict(m, Fairness_{AClass}) \Rightarrow Fairness_{Body_{PSys}})
\end{array}
$$

The additional constraint $(\wedge_{meth \in m} \neg start_{meth})$ describes that in the initial state all methods occurring in the list m are not active.

Parameterized object-based specifications can be used to model a system composed of interacting components. In figure 4 an object-based specification is constructed by linking objects from classes `AC` and `BC` via a single method (more complex examples will be presented later).

```
ASys2 : CLASS                    AC : CLASS
  BEGIN                            BEGIN
    AGGREGATE                        atr: ATTRIBUTE type
      A:  AC                         meth(x:type): METHOD type ==...
      B:  BC[meth FROM A]            ...
    END AGGREGATE                  END AC
  END ASys2
                                 BC [ meth FROM P: AC ] : CLASS
                                   BEGIN
                                     ...
                                   END BC
```

Figure 4: Aggregating Instances of Parameterized Specifications

The meaning $Spec(BC)$ of specification BC is obtained as described above and the meaning of B: BC[meth FROM A] is obtained from $Spec(BC)$ by prefixing $Spec(BC)$ with the instance name B and then substituting the names of the input and output variables associated with method A.meth for the input and output variables of the parameter method P.meth:

$$Spec(B) \triangleq (B.Spec(BC))[B.P.x \leftarrow A.x, B.P.result_{P.meth} \leftarrow A.result_{A.meth}, B.P.start_{P.meth} \leftarrow A.start_{A.meth}, B.P.stop_{P.meth} \leftarrow A.stop_{A.meth}]$$

The meaning of ASys2 is defined by conjunction of the specifications of its aggregated components:

$$Spec(ASys2) \triangleq Spec(A) \wedge Spec(B)$$

A SMALL CASE STUDY: MOVING THE CONTENTS OF A FILE

The approach presented in this paper is illustrated by incrementally developing an example system called *Move*. The functional requirement for *Move* is that the contents from a source file are moved to a destination file. In the following this simple specification is successively refined.

In the first refinement, the contents of the source file are moved to the destination file one character at the time by first moving the character to some auxiliary buffer and then moving it from the buffer to the destination file. The system is split into several component objects for maintaining the different state components: a producer for emptying the source file, a consumer for filling the destination file and a buffer object for maintaining the buffer between producer and consumer.

Each such component can be viewed as an open system that can be refined separately. While the producer and the consumer are active components, the buffer appears to be passive. However, when refining the buffer, it may become an active component. Specifically when non-functional requirements have to be considered: e.g. the buffer may have to be implemented as a component for sending data over a non-reliable medium. Here we consider the possibility of lossy wires, and the buffer is further refined by introducing the alternating bit protocol.

```
Move : CLASS
  BEGIN
    IMPORTING   Files   % type and operations for files
    src         : ATTRIBUTE file
    dst         : ATTRIBUTE file
    move        : ACTION ==     src  ≠ empty
                            AND dst' = src
                            AND src' = empty
    INITIAL     dst = empty
    STEPS       move
    FAIRNESS    WF(move)
  END Move
```

Figure 5: A closed system specification for *Move*

$$
\begin{array}{lcl}
move & \triangleq & src \neq empty \wedge dst' = src \wedge src' = empty \\
Initial_{Move} & \triangleq & dst = empty \\
Steps_{Move} & \triangleq & \Box[move]_{\langle src,dst \rangle} \\
Fairness_{Move} & \triangleq & WF_{\langle src,dst \rangle}(move) \\
Move & \triangleq & Initial_{Move} \wedge Steps_{Move} \wedge Fairness_{Move} \\
Spec(Move) & \triangleq & \exists\, src, dst : Move
\end{array}
$$

Figure 6: TLA representation for specification `Move`

Specifying Move as a Closed System

Figure 5 presents the top-level specification of system *Move* describing an action for moving the complete contents of a source file to a destination file by means of one atomic state transition. System *Move* is specified as a closed system which does not interact with any environment. File-type and file-operations are imported with a theory named `Files`. In the remaining paper, such imports will be omitted. A TLA representation of this specification is shown in figure 6.

Refining Object Structure, Actions, and State

In a first step specification `Move` is refined into a specification describing the composition of several cooperating objects: a producer, a consumer, and a buffer. The buffer is assumed to be capable of receiving, holding, and delivering a single character. Thus also the granularity of the actions is changed: the producer reads the contents of the source file one character after the other and passes them on to the buffer from which the consumer can obtain the characters and build up the destination file. This involves intermediate steps and an "additional" state component for the buffer.

Each component constitutes an open system. The consumer and the producer communicate with each other via a buffer object, which provides two methods: a `put` method for storing some value in the buffer and a `get` method for obtaining the

```
Buf : CLASS
  BEGIN
    buf          : ATTRIBUTE char
    put(x:char)  : METHOD ==     buf  = empty
                              AND buf' = x
    get          : METHOD char
                       ==     buf ≠ empty
                              AND buf' = empty
                              AND RESULT = buf
    INITIAL      buf = empty
    STEPS        put(x) OR get
    FAIRNESS     WF(put(x)) AND WF(get)
  END Buf
```

Figure 7: Specification of Buffer

$$
\begin{array}{lcl}
put & \triangleq & action_{put} \\
get & \triangleq & action_{get} \\
Initial_{Buf} & \triangleq & buf = empty \wedge stop_{put} = false \wedge stop_{get} = false \\
Steps_{Buf} & \triangleq & \Box[put \vee get]_{\langle buf, stop_{put}, result_{get}, stop_{get} \rangle} \\
Fairness_{Buf} & \triangleq & WF_{\langle buf, \ldots \rangle}(put) \wedge WF_{\langle buf, \ldots \rangle}(get) \\
Buf & \triangleq & Initial_{Buf} \wedge Steps_{Buf} \wedge Fairness_{Buf} \\
Spec(Buf) & \triangleq & \neg start_{put} \wedge \neg start_{get} \stackrel{+}{\triangleright} \exists\, buf : Buf
\end{array}
$$

Figure 8: Semantics of Specification `Buf` in TLA

```
Prod [ put FROM B: Buf ] : CLASS
  BEGIN
    src_1      : ATTRIBUTE file
    call       : ACTION ==     src_1 ≠ empty
                           AND CALL(B.put(head(src_1)))
                           AND src_1' = tail(src_1)
    INITIAL    TRUE        % implicitly: B.start_put = false
    STEPS      call
    FAIRNESS   WF(call)
  END Prod
```

Figure 9: Parameterized object-based specification: the producer

current value stored in the buffer. The specification of the buffer is given in figure 7. Its semantics in terms of TLA is shown in figure 8. Note that the synchronization mechanism would require careful encoding if TLA were used directly; in TLO these details are part of the semantics.

The producer is specified in Module `Prod` shown in figure 9; its semantics in terms of TLA (fig. 10) is based on the semantics of instance `B` of specification `Buf` (fig. 8),

$$
\begin{array}{lcl}
Par\_Prod & \triangleq & Restrict(B.put, B.Buf) \\
C(Par\_Prod) & \triangleq & Restrict(B.put, B.Initial_{Buf}) \wedge Restrict(B.put, B.Steps_{Buf}) \\
call & \triangleq & src_1 \neq empty \wedge CALL(B.put(head(src_1))) \wedge src_1' = tail(src_1) \\
Initial_{Body\_Prod} & \triangleq & B.start_{put} = false \\
Steps_{Body\_Prod} & \triangleq & \Box[call]_{\langle src_1, x, start_{put} \rangle} \\
Fairness_{Body\_Prod} & \triangleq & WF_{\langle src_1, x, \ldots \rangle}(call) \wedge WF_{\langle src_1, x, \ldots \rangle}(wait) \\
Body\_Prod & \triangleq & Initial_{Body\_Prod} \wedge Steps_{Body\_Prod} \wedge Fairness_{Body\_Prod} \\
C(Body\_Prod) & \triangleq & Initial_{Body\_Prod} \wedge Steps_{Body\_Prod} \\
Spec(Prod) & \triangleq & \exists\, buf : C(Par\_Prod) \stackrel{+}{\triangleright} \\
 & & (\exists\, src_1 : C(Body\_Prod) \wedge \\
 & & (Restrict(B.put, B.Fairness_{Buf}) \Rightarrow Fairness_{Body\_Prod}))
\end{array}
$$

Figure 10: Semantics of Specification `Prod` in TLA

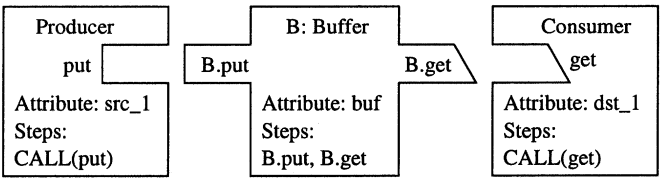

Figure 11: Refined Object Structure for *Move*

with the action *get* replaced by $\exists\, start_{B.get}, stop_{B.get}, B.result_{B.get} : action_{B.get}$ for abstracting from method `B.get`.

The specification of the consumer, `Cons`, is analogous to that of the producer. The parameter of `Prod` describes the environment as seen by the producer: the environment provides a buffer that can be filled calling the `B.put` method; if the buffer is filled, it will eventually again be emptied by the environment. As long as the source file is not empty the producer iterates calling the `B.put` method with the first character in the file and truncating the file by the first character.

There is no explicit initialization in the body of the producer specification. However, the implicitly defined variable $B.start_{B.put}$ being an output variable of the producer's body has to be initialized here. This initialization is implicit.

The components are connected to each other by means of their interfaces which must "fit together" for being composed. This is depicted in figure 11. The corresponding specification is presented in figure 12. The links between the components are established by instantiating the formal parameters of the producer and the consumer with the corresponding methods from the buffer component. They fit together if the assumptions stated for the parameters are satisfied by the instantiation.

The meaning of $Spec(ProdCons)$ is given by the meaning of its components:

$$Spec(ProdCons) \triangleq Spec(B) \wedge Spec(P) \wedge Spec(C)$$

```
ProdCons : CLASS
  BEGIN
    AGGREGATE
      B    : Buf
      P    : Prod[put FROM B]
      C    : Cons[get FROM B]
  END ProdCons
```

Figure 12: Producer/consumer system composed from several components

$Spec(B)$ is obtained from $Spec(Buf)$ by prefixing it with the instance name B ($Spec(B) \triangleq B.Spec(Buf)$). $Spec(P)$ is obtained from $Spec(Prod)$ by prefixing it with the instance name P and renaming the input and output variables with the substitution

$$\sigma \triangleq [\ P.B.x \leftarrow B.x,\ P.B.start_{P.B.put} \leftarrow B.start_{B.x},\ P.B.stop_{P.B.put} \leftarrow B.stop_{B.put}\ ]$$

The semantics of instances B and P are defined by

$$\begin{array}{lcl} Assuming_B & \triangleq & \neg B.start_{B.put} \wedge \neg B.start_{B.get} \\ Body_B & \triangleq & B.Buf \\ Spec(B) & \triangleq & Assuming_B \stackrel{+}{\triangleright} \exists\, buf : Body_B \end{array}$$

$$\begin{array}{lcl} Assuming_P & \triangleq & (P.\mathcal{C}(Par\_Prod))\sigma \\ Body_P & \triangleq & (P.\mathcal{C}(Body\_Prod))\sigma \wedge \\ & & (P.Restrict(B.put, B.Fairness_{Buf}) \Rightarrow Fairness_{Body\_Prod})\sigma \\ Spec(P) & \triangleq & \exists\, buf : Assuming_P \stackrel{+}{\triangleright} \exists\, src_1 : Body_P \end{array}$$

$Assuming_C$, $Body_C$, and $Spec(C)$ can be defined analogously.

In order to show that ProdCons refines Move, it is necessary to prove the validity of the implication

$$Spec(ProdCons) \Rightarrow Spec(Move)$$

All components can be regarded as open systems; thus the proof obligation can be reformulated as

$$(E_B \stackrel{+}{\triangleright} M_B) \wedge (E_P \stackrel{+}{\triangleright} M_P) \wedge (E_C \stackrel{+}{\triangleright} M_C) \Rightarrow (E_{Move} \stackrel{+}{\triangleright} M_{Move})$$

with

$$\begin{array}{lcllcl} E_B & \triangleq & Assuming_B & E_C & \triangleq & \exists\, buf : Assuming_C \\ M_B & \triangleq & \exists\, buf : Body_B & M_C & \triangleq & \exists\, dst_1 : Body_C \\ E_P & \triangleq & \exists\, buf : Assuming_P & E_{Move} & \triangleq & true \\ M_P & \triangleq & \exists\, src_1 : Body_P & M_{Move} & \triangleq & Spec(Move) \end{array}$$

By applying the rules for open systems in TLA the refinement can be shown to be correct by proving

- $\mathcal{C}(Body_B) \wedge \mathcal{C}(Body_P) \wedge \mathcal{C}(Body_C) \Rightarrow E_P$
- $\mathcal{C}(Body_B) \wedge \mathcal{C}(Body_P) \wedge \mathcal{C}(Body_C) \Rightarrow E_C$
- $\mathcal{C}(Body_B) \wedge \mathcal{C}(Body_P) \wedge \mathcal{C}(Body_C) \Rightarrow \neg start_{B.put} \wedge \neg start_{B.get}$
- $\mathcal{C}(Body_B) \wedge \mathcal{C}(Body_P) \wedge \mathcal{C}(Body_C) \Rightarrow \mathcal{C}(Spec(Move))$
- $M_B \wedge M_P \wedge M_C \Rightarrow Spec(Move)$

The first two formulae express the requirement that the parameter specifications of the producer and the consumer are correctly instantiated. The third formula expresses that the buffer has to be used in a suitable context which initializes the variables for synchronization appropriately. The fourth formula requires that the safety properties of *Move* are correctly refined and the fifth formula requires in addition that the liveness properties are correctly refined.

The last two proof obligations require, in general, a suitable refinement formula, $\exists\, x : R$, which maps the state variables of *ProdCons* to the state variables of *Move*. The "smaller" steps of *ProdCons* are not observable in *Move* because it specifies that the content of the source file is moved to the destination file in one atomic step; changes occurring in the refinement may be observable at the abstract level of *Move* only when the complete file has been moved. Also the additional state component of the buffer does not appear explicitly in *Move*. A refinement formula should express this by mapping the actual contents of the source and destination files (src_1, dst_1) from the refinement *ProdCons* to the abstract *Move* specification level only after the source file has completely been emptied and it should leave the abstract values of the files unchanged for intermediate steps. This is described by the following refinement formula:

$$\exists\, src, dst : \Box \left[\begin{array}{l} \textbf{if } buf + src_1 = empty \\ \quad \textbf{then } src = empty \wedge dst = dst_1 \\ \quad \textbf{else } src = dst_1 + buf + src_1 \wedge dst = empty \end{array} \right]$$

In fact a refinement map is also necessary for the first two proof obligations for asserting the correct instantiation of parameters; however, this is trivial because attribute *buf* of the buffer can be mapped directly to the attribute *buf* of the parameter specification.

Refining the Buffer

The buffer object appears to be a system without internal activity. It rather resembles a passive queue with the capacity of holding one character. If, however, the transmission of a character from a producer to a consumer has to be reliable despite the presence of a faulty transmission medium, then a simple and passive queue is usually not good enough.

In the following it is assumed that the functionality of the buffer object has to be provided on top of unreliable transmission media, which may lose information (lossy wires). However, it is also assumed that information is not forged (or that this can be detected) and that any information will be passed on if it is retransmitted just often enough. A solution suitable for such situations is the alternating bit protocol (ABP).

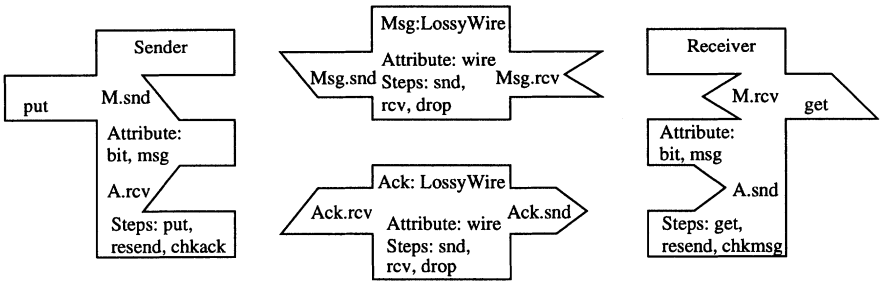

Figure 13: Components of the ABP

```
LossyWire [ t : TYPE ] : CLASS
  BEGIN
    wire          : ATTRIBUTE t
    snd(x:t)      : METHOD    == wire' = x
    rcv           : METHOD t == RESULT = wire
    drop          : ACTION    == wire' = empty
    INITIAL       wire = empty
    STEPS         snd OR rcv OR drop
    FAIRNESS      WF(snd) AND SF(wire ≠ empty AND rcv)
  END LossyWire
```

Figure 14: Specification of lossy wires

The components of the ABP refinement of the buffer are pictured in figure 13. The sender expects two wires to be accessible by means of methods `M.snd` and `A.rcv` and provides the method `put` which can be accessed from the environment; similarly for the receiver. Specifications `Sender` and `LossyWire` are presented and discussed below. The receiver is specified analogous to the sender.

Specification `LossyWire` (figure 14) describes properties of the environment in which the sender and receiver are expected to operate. This environment may largely consist of hardware (wire) rather than a pure software module. The specification is generic with respect to the type of messages to be processed. Writing some message to the wire is modeled by the method `snd`, and reading a message from the wire is modeled by method `rcv`. The fault hypotheses is also formalized in this specification: the wire may drop (or lose) some message. However, the strong fairness assumption asserts that if the wire is often enough not empty then eventually some (non-empty) message from the wire will be returned by method `rcv`.

The sender (figure 15) is modeled with a parameterized specification with the parameter describing two communication lines: a message and an acknowledgment wire. It provides a method `put`, which can be called to submit to the sender some value to be transmitted over the message wire. Such a message is stored temporarily in attribute `msg`. The sender tries repeatedly to transmit the message by calling method `M.snd` and checks whether the transmission was successful by calling `A.rcv`

```
Sender [ snd FROM M: LossyWire[[char,bit]],
         rcv FROM A: LossyWire[bit]          ] : CLASS
  BEGIN
    b        : ATTRIBUTE bit
    msg      : ATTRIBUTE char
    put(x)   : METHOD == msg  = empty AND msg' = x
    resend   : ACTION == msg ≠ empty AND CALL(M.snd(msg,b)
    chkack   : ACTION ==     msg ≠ empty AND CALL(A.rcv)
                         AND IF result_A.rcv' = b
                               THEN b' = 1 - b AND msg' = empty
                               ELSE b' = b AND msg' = msg
                             FI
    INITIAL   b = 0 AND msg = empty
    STEPS     put OR resend OR chkack
    FAIRNESS  WF(put) AND WF(resend) AND WF(chkack)
  END Sender
```

Figure 15: Specification of the sender

```
ABP : CLASS
  BEGIN
    AGGREGATE
      Msg : LossyWire[[char,bit]]
      Ack : LossyWire[bit]
      Snd : Sender[snd FROM Msg, rcv FROM Ack]
      Rcv : Receiver[rcv FROM Msg, snd FROM Ack]
    END AGGREGATE
  END ABP
```

Figure 16: Specification of the ABP

and comparing the received acknowledgment with the current bit. The parameter specification contains the failure assumption that the wires may lose some values, but may not alter the values in transmission. It is further assumed that the acknowledgment wire will eventually contain some (non-empty) information that will also be available for reading. If an acknowledgment that is compatible with the message written to the message wire is received, the current bit is alternated (`b'=1-b`) and the temporarily stored message `msg` is cleared. This in turn enables method `put(x)` for being called again. Due to its size, the formal semantics of this specification is not presented here; it is obtained analogously to the semantics of specification `Prod` (cf. figure 10).

Specification `ABP` shown in figure 16 defines the composition of sender, receiver, and wire components. The sender and the receiver provide the methods put and get, respectively. They are connected to each other by means of their parameters being instantiated with `Msg` and `Ack`. The sender can make use of method `Msg.snd` for writing messages to the message wire and it can read with method `Ack.rcv` from the acknowledgment wire. The receiver is instantiated with the complementary methods for reading from the message wire and for writing acknowledgments.

Proving formula $Spec(ABP) \Rightarrow Spec(Buf)$ shows that ABP correctly refines specification Buf. This can be handled analogously to the approach described above.

CONCLUSION AND FURTHER WORK

In this paper, we have presented elements of TLO, a specification language based on temporal logic for object-based designs, and a refinement methodology for such designs. The proposed approach has been illustrated by a small case study that exhibits essential features, such as composition of object specifications by aggregation and refinement of objects.

A key feature of TLO is that open systems are specified as parameterized objects. The parameter expresses assumptions about the environment; the body of the specification states the properties guaranteed by the system provided that the assumptions hold. Large systems can be specified as the composition of smaller open systems by aggregating and instantiating the parameters of the corresponding specifications appropriately. This approach also provides a high degree of flexibility: the producer and consumer systems presented above could directly be instantiated with the Snd.put and Rcv.get methods from specification ABP.

The design of TLO as presented here is still experimental; it also needs to be extended by features that make more extensive use of the class structure, such as inheritance and subtypes; this is left for future work. At this point, no implementation of TLO exists. However, an embedding of TLA in PVS has already been worked out. We expect that this will be expanded to provide full mechanized proof support for TLA formulae; furthermore, it will be complemented by mechanized generation of verification conditions from specifications for composition, instantiation and refinement of objects.

Acknowledgments

We would like to thank the referees for their valuable comments. This work has been funded in part by Esprit LTR Project 20072 "Design for Validation" (DeVa).

References

[1] M. Abadi and L. Lamport. Open Systems in TLA. In *Proc. of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 81 –90, 1994.

[2] M. Abadi and L. Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–538, May 1995.

[3] R. Alur and T. A. Henzinger, editors. *Computer Aided Verification*, volume 1102 of *LNCS*, New Brunswick, NJ, July 1996. Springer.

[4] R. Back. Refinement Calculus, Part II: Parallel and Reactive Programs. In de Bakker et al. [6], pages 67–93.

[5] R. Back and J. von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In de Bakker et al. [6], pages 42–66.

[6] J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems*, volume 430 of *LNCS*. Springer, May 1989.

[7] G. Denker. *Verfeinerung in objektorientierten Spezifikationen: Von Aktionen zu Transaktionen.* Number 6 in DISDBIS. infix-Verlag, Sankt Augustin, 1996.

[8] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z Specification Language. Technical Report 91-1, University of Queensland, Department of Computer Science, Software Verification Centre, May 1991. Version 1.

[9] H.-M. Järvinen and R. Kurki-Suonio. DisCo specification language: marriage of actions and objects. In *Proc. 11th International Conference on Distributed Computing Systems*, pages 142–151. IEEE Computer Society Press, 1991.

[10] R. Jungclaus, T. Hartmann, G. Saake, and C. Sernadas. Introduction to TROLL – A Language for Object-Oriented Specification of Information Systems. In G. Saake and A. Sernadas, editors, *Information Systems – Correctness and Reusability*, Informatik Bericht 91-03, pages 97–128. TU Braunschweig, 1991.

[11] L. Lamport. Introduction to TLA, July 1994. Available from http://www.research-.digital.com/SRC/personal/lamport/tla/tla.html.

[12] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[13] J. Meseguer. *A Logical Theory of Concurrent Objects and Its Realization in the Maude Language*, chapter 12, pages 314–390. The MIT Press, Cambridge, Massachusetts, 1993.

[14] A. Morzenti and P. San Pietro. Object-Oriented Logical Specification of Time-Critical Systems. *ACM Transactions on Software Engineering and Methodology*, 3(1):56–98, Jan. 1994.

[15] S. Owre, S. Rajan, J. Rushby, and N. Shankar. PVS: Combining Specification, Proof Checking, and Model Checking. In Alur and Henzinger [3], pages 411–414.

[16] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.

[17] J. M. Spivey. *The Z Notation.* Prentice-Hall International series in computer science. Prentice Hall, 1992.