A COMPOSITIONAL FORMALISM FOR OBJECT VIEWPOINTS

Einar Broch Johnsen and Olaf Owe
Department of Informatics, University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
{einarj,olaf}@ifi.uio.no

Abstract Aspect-oriented approaches have recently been proposed to address the problem of specifying dynamic object-based systems, by depicting the various roles of the objects separately. In this paper, we consider an approach based on the observable behavior of objects and propose a specification formalism for reusable object interfaces with input/output-driven assumption-guarantee predicates. The formalism supports compositional reasoning and exchange of object identities between objects in an environment where the number of objects is unbounded.

Keywords: object-orientation, interfaces, formal specification, refinement, viewpoints

Introduction

To facilitate the modeling process of complex open object-based distributed systems, we are encouraged to separate different concerns into various viewpoints, or aspects, by the RM-ODP [13]. This paper introduces an object-oriented specification formalism centered around this idea. The formalism considers objects running in parallel and communicating asynchronously, and it directly supports main object-oriented concepts such as object identity and multiple inheritance. Interfaces and classes are organized in distinct inheritance hierarchies to separate reuse and design of behavior from that of code. Such a distinction is also found in languages like Java, and is enforced by the COM architecture [11]. In this paper, the usual syntactic information of interface declarations is extended with behavioral constraints. Thus, interfaces are carriers of semantic requirements to be respected by the classes that implement them.

In open systems, components can be supplied by different manufacturers at different times. Therefore, we do not assume knowledge of implementation details concerning objects in the environment. Instead, an object's behavior is locally determined by its interaction with the environment [2]. For this reason, we prefer an approach to system development based on the *observable behavior*

In: B. Jacobs and A. Rensink (Eds.): Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002), p. 45–60, Kluwer Academic Publishers, March 2002.

of objects, rather than on internal states and state transitions as encountered in work based on formalisms such as Object-Z [23, 8] and timed automata [7].

Message-based process algebras such as CSP [12], the π-calculus [17], and FOCUS [9] do not have object identity, which makes viewpoint modeling of distributed systems difficult. Formal object-based approaches to message-based concurrency modeling, such as Actors [2] and Maude [10], are not oriented towards stepwise development and viewpoints, lacking notions of composition and refinement of specifications. Industrial development languages such as UML [19] and OCL [25] support viewpoint specification, but lack formal notions of composition and refinement and are less suitable for rigorous reasoning.

In the proposed formalism, objects are specified by interaction sequences of remote method calls both to and from the object, so we model *active* as well as passive objects. By composing viewpoints to different objects, we specify services cross-cutting the system architecture, resembling aspects in aspect-oriented programming [16]. Our work is related to other approaches to viewpoints in message-passing systems, e.g. [4, 18, 24], but our emphasis is on formal development of viewpoint specifications in the setting of open distributed systems. A refinement relation is proposed for viewpoints, which allows openness by service upgrades and restructuring.

In order to focus on ease-of-use, a schematic presentation of specifications is proposed by means of interface declarations with both syntactic and semantic information. We think of interfaces as *generic object viewpoints*. An object can offer several interfaces and an interface can be offered by several objects. Objects are always considered through the interfaces they offer. Hence, for specification purposes, we deal with interfaces that correspond to object viewpoints, rather than complete specifications. New objects may be created at any time, and object identities can be transmitted between objects. Old objects may offer new interfaces and an object that offers an interface I also offers all interfaces refined by I. In interface declarations, behavioral requirements are written in a rely-guarantee style [15], in a novel input/output-driven manner. The separation of interfaces and classes give us static correctness proofs. The specification formalism has been encoded into the PVS proof system [21], to provide a machine-assisted proof environment for the formalism [14].

Behavioral interfaces lets us capture certain forms of openness by textual analysis. Reasoning is both compositional and incremental, so software units can be written, formally analyzed, and modified independently, at different times and locations, while we have control of maintenance of earlier proven results. In this paper, only refinement and composition of viewpoint specifications are considered, ignoring implementation issues. In accordance with the idea of observability, we focus on safety specifications, in the sense of Alpern and Schneider [3], by means of prefix-closed sets of finite communication traces. These are described by trace patterns in a graphical specification style.

1. A Formalism for Partial Object Specifications

Objects are not static parts of a system, they evolve through interaction with the environment. One may think of the current "state" of an object as the result of its past interactions with the environment by way of messages or method calls. Objects are modeled by finite communication traces, known from e.g. CSP [12], but our communication events include information about the *identities* of the sender and receiver of calls.

A communication trace is a sequence over an alphabet of events reflecting remote method calls between objects. Let Mtd and Obj be fixed sets of methods and object identities, respectively. Say that a method m is offered by an object o_1 and that this method is called by o_2. This call is represented by a triple $\langle o_2, o_1, m\rangle$ where $o_1, o_2 \in Obj$ and $m \in Mtd$. For simplicity, we ignore additional event information until Section 2.1.

The triple is said to *involve* the objects o_1 and o_2. Calls by an object to methods in itself are internal and are not reflected in the alphabet nor in the observable traces of the object.

A specification of an object is a *partial* description of that object. Thus, we allow several specifications of the same object. To each specification Γ of an object o, we associate an alphabet α that consists of those communication events of o that are considered by Γ and a trace set over α. A specification of a component describes a system service by combining viewpoints of different objects. Specifications have unbounded *communication environments,* consisting of external objects involved in communication with the objects specified. Thus, new objects can always appear in the environment. Environments are modeled by infinite sets of object identifiers.

Definition 1 *A specification* Γ *is a quadruple* $\langle \mathcal{O}, \alpha, \mathcal{T}, \mathcal{E}\rangle$ *where (1)* $\mathcal{O}$ *and* $\mathcal{E}$ *are disjoint sets of object identities,* $\mathcal{O}, \mathcal{E} \subseteq Obj$*, (2)* α *is a set of events* $\langle o_1, o_2, m\rangle$ *such that* $m \in Mtd$ *and each event involves an object in* $\mathcal{O}$ *and another in* $\mathcal{E}$*, and (3)* $\mathcal{T}$ *is a prefix-closed subset of* α^**.*

S^* denotes the finite traces over a set S. We call $\mathcal{O}$ the *object set* of the specification Γ, α the *alphabet* of Γ, $\mathcal{T}$ the *trace set* of Γ, and $\mathcal{E}$ the *communication environment* of Γ. These are referred to as $\mathcal{O}(\Gamma)$, $\alpha(\Gamma)$, $\mathcal{T}(\Gamma)$, and $\mathcal{E}(\Gamma)$, respectively. Although the alphabets of specifications are statically defined, communication of object identities can still be modeled by a particular object's ability to communicate with another object at a specific point in history.

If the object set of a specification Γ is singleton, we call Γ an *interface* specification of o. In the literature, soundness for a singleton specification Γ of an object o is usually given by $\mathcal{T}^o \subseteq \mathcal{T}(\Gamma)$, where $\mathcal{T}^o$ is the trace set of o [5]. With partial specifications, we restrict the traces to the alphabet of the specification, so Γ is a *sound* specification of o if $\forall h \in \mathcal{T}^o : h/\alpha(\Gamma) \in \mathcal{T}(\Gamma)$. Soundness properties are discussed in Section 1.2.

Trace Notation. Let h be a trace and S a set of events. Then h/S denotes the trace obtained by deleting all events in the trace h that are not elements of S and $h \backslash S$ denotes the trace obtained by deleting the elements in S. By extension, h/o will represent the restriction of h to the set of events involving an object o. We will often use trace patterns in our specifications, mimicking a graphical specification style. Let h **prp** P denote that the trace h is a prefix of the pattern P, i.e. that there is an extension to h described by P. Patterns are regular expressions extended with a binding operator and pattern matching.

Example 1 Consider interface specifications `Read`, `Write`, and `ReadWrite`, of an object o controlling read and write access to some shared data. Concurrent read access is allowed, whereas write access is restricted to one object of the environment ($\mathcal{E}$) at the time. For *write control*, there are methods for writing as well as for opening and closing write access, which we name W, OW and CW, respectively, where W has a parameter ranging over $Data$. An interface specification, `Write`, of an object o controlling write access has the alphabet

$$\alpha(\texttt{Write}) \triangleq \{\langle x, o, OW\rangle, \langle x, o, CW\rangle, \langle x, o, W(d)\rangle \mid x \in \mathcal{E} \wedge d \in Data\}.$$

A write-cycle can be specified by the pattern

$$\textit{Wcycle} \triangleq [\langle x, o, OW\rangle\ \langle x, o, W\rangle^*\ \langle x, o, CW\rangle \bullet x \in \mathcal{E}].$$

We then define the trace set of `Write` by

$$\mathcal{T}(\texttt{Write}) \triangleq \{h : \alpha(\texttt{Write})^* \mid h\ \textbf{prp}\ \textit{Wcycle}^*\}.$$

Sequential write access is enforced by the binding operator $\bullet$ of Wcycle, expressing that the calling object is fixed for each repetition of the pattern. Similarly, *read control* can be specified over the alphabet given by methods R, OR and CR; and a read-cycle may be specified by the pattern

$$Rcycle \triangleq [\langle x, o, OR\rangle\ \langle x, o, R\rangle^*\ \langle x, o, CR\rangle \bullet x \in \mathcal{E}].$$

The trace set of the specification `Read` (of object o) can be given by

$$\mathcal{T}(\texttt{Read}) \triangleq \{h : \alpha(\texttt{Read})^* \mid \forall x : \mathcal{E} \bullet h/x\ \textbf{prp}\ Rcycle^*\}$$

where concurrent and overlapping read cycles are allowed (due to the projection to every x). Alternatively, we could define the trace set by h **prp** $MultiRcycle^*$ where the pattern *MultiRcycle* is specified as the interleaving of Rcycles:

$$\forall x : \mathcal{E} \bullet MultiRcycle/x = Rcycle^*.$$

An object o *controlling both read and write access* may be specified by the alphabet $\alpha(\texttt{ReadWrite}) \triangleq \alpha(\texttt{Read}) \cup \alpha(\texttt{Write}))$ and the trace set

$$\mathcal{T}(\texttt{ReadWrite}) \triangleq \{h : \alpha(\texttt{ReadWrite})^* \mid h\ \textbf{prp}\ [\textit{Wcycle}|MultiRcycle]^*\}.$$

1.1 Refinement

Refinement notions capture the correct evolution of specifications. Usually, a formal refinement relation states that a specification Γ' refines another specification Γ if Γ' implies Γ, which becomes $\mathcal{T}(\Gamma') \subseteq \mathcal{T}(\Gamma)$ for trace-based formalisms. This concept of refinement is found in e.g. Action Systems [6], CSP [22], FOCUS [9], and TLA [1]. An essential principle here is that possible behavior is restricted by refinement. We do not find this principle satisfactory for partial specifications. Instead, we want to consider more as we approach implementation. Consequently, behavior should be expanded by refinement in this setting. For open and object-based systems, such refinement can be understood in terms of *substitutability*, when components are replaced or extended on the fly. Thus, component refinement corresponds to a controlled form of component upgrade, allowing new functionality to be added to system services. For this purpose, we use refinement with projection.

Definition 2 *A specification* Γ' *refines another specification* Γ*, denoted* $\Gamma' \sqsubseteq \Gamma$*, if (1)* $\mathcal{O}(\Gamma) \subseteq \mathcal{O}(\Gamma')$*, (2)* $\alpha(\Gamma) \subseteq \alpha(\Gamma')$*, (3)* $\mathcal{E}(\Gamma) \subseteq \mathcal{E}(\Gamma')$*, and (4)* $\forall h \in \mathcal{T}(\Gamma') : h/\alpha(\Gamma) \in \mathcal{T}(\Gamma)$.

This relation is a partial order. Without alphabet expansion or object addition, the relation coincides with the traditional subset-relation and solely restricts the allowed behavior of the old specification. For partial specifications however, our refinement relation has an additional asset: As the alphabet of a specification may be expanded through refinement, a specification may refine several other specifications. For component specifications (see Section 1.2), new object identifiers may also be introduced.

Example 2 Reconsider Example 1. We clearly have that `ReadWrite` refines both `Read` and `Write`. If we extend the alphabet of `Write` with a read operation (R), the trace set can be defined by the prefix closure of repeated W+cycles:

$$\textit{W+cycle} \triangleq [\langle x, o, OW\rangle \; [\langle x, o, W\rangle \mid \langle x, o, R\rangle]^* \; \langle x, o, CW\rangle \bullet x \in \mathcal{E}].$$

The resulting specification, `Write+`, refines `Write`. However, the similarly extended `ReadWrite` specification would neither refine `Read` (since R events now are allowed outside Rcycles) nor `Write+` (since R events are allowed outside W+cycles), but it would refine `Write`.

1.2 Specifying Components

Let a *component* encapsulate an arbitrary number of objects, so that details regarding the implementation of the component are abstracted away, hiding internal activity. The objects of a component are explicit; we represent a component by the set of objects it encapsulates.

Definition 3 *The internal events of a set S of objects are the set of all possible communication events between objects of the set:*

$$\mathcal{I}(S) \triangleq \{\langle o_1, o_2, m\rangle \mid o_1 \in S \wedge o_2 \in S \wedge m \in Mtd\}.$$

Given alphabets and trace sets for the objects of a component, the alphabet and trace set of the component are constructed directly.

Definition 4 *Let C be a component, encapsulating a set $S = \{o_1, \ldots, o_n\}$. Define the alphabet α^C of C by $\alpha^C \triangleq \bigcup_{o \in S} \alpha^o - \mathcal{I}(S)$. Let $h \in \left(\bigcup_{o \in S} \alpha^o\right)^*$. Then the trace set of C is defined as $\mathcal{T}^C \triangleq \{h/\alpha^C \mid \bigwedge_{o \in S} h/\alpha^o \in \mathcal{T}^o\}$.*

Component composition becomes the union operation on sets, and is obviously commutative and associative. Due to object uniqueness, components are compositional in the sense that we can derive $\alpha^{C_1 \cup C_2}$ and $\mathcal{T}^{C_1 \cup C_2}$ directly from the alphabets and trace sets of C_1 and C_2.

A *component specification* can have a non-singleton object set. With partial specifications, the same object identifiers may be part of several specifications and an event that is internal in one specification may be observable in another. Composition in this case is non-trivial and its properties must be examined.

Definition 5 (Composability) *Two component specifications Γ and Δ are* composable *if and only if $\alpha(\Gamma) \cap \mathcal{I}(\mathcal{O}(\Delta)) = \emptyset$ and $\mathcal{I}(\mathcal{O}(\Gamma)) \cap \alpha(\Delta) = \emptyset$.*

For two composable component specifications it is possible to determine whether their composition is "meaningful" by considering the observable communication traces of the two specifications. Often, we can verify the simpler conditions $\mathcal{O}(\Gamma) \cap \mathcal{E}(\Delta) = \emptyset$ and $\mathcal{O}(\Delta) \cap \mathcal{E}(\Gamma) = \emptyset$ instead, which entail composability.

A related notion is found in [8]: two specifications are *consistent* if they have a common refinement. The (least) common refinement is obtained by unification of the two specifications. However, as the trace sets of our formalism are prefix-closed, two specifications always have a common refinement.

Definition 6 *Let Γ and Δ be composable component specifications. Then $\Gamma||\Delta$ is a specification $\langle \mathcal{O}, \alpha, \mathcal{T}, \mathcal{E}\rangle$, where (1) $\mathcal{O} \triangleq \mathcal{O}(\Gamma) \cup \mathcal{O}(\Delta)$, (2) $\alpha \triangleq \alpha(\Gamma) \cup \alpha(\Delta) - \mathcal{I}(\mathcal{O})$, (3) $\mathcal{T} \triangleq \{h/\alpha | h \in (\alpha(\Gamma) \cup \alpha(\Delta))^* \wedge h/\alpha(\Gamma) \in \mathcal{T}(\Gamma) \wedge h/\alpha(\Delta) \in \mathcal{T}(\Delta)\}$, and (4) $\mathcal{E} \triangleq \mathcal{E}(\Gamma) \cup \mathcal{E}(\Delta)$.*

Composition is commutative and associative. Due to hiding of internal events, a proof for a naive formulation of soundness for the composition rule is not straightforward. However, the following lemma ensures soundness for the composition of specifications of a given component.

Lemma 1 *Composition preserves soundness, i.e. if Γ and Δ are sound specifications of a component C, then $\Gamma||\Delta$ is a sound specification of C.*

To obtain reasoning control over the refinement of component specifications, we need another restriction, motivated as follows: Say that Γ' is a refinement of Γ, and Γ' includes an event that is in the alphabet of a third specification Δ such that this event is visible in the composition $\Gamma||\Delta$. Then this event should also be visible in $\Gamma'||\Delta$. Such events may be introduced by the inclusion of new objects in the refinement. No restriction is needed when old objects are extended with new methods.

Definition 7 (Properness) *Let Γ, Γ' and Δ be specifications such that $\Gamma' \sqsubseteq \Gamma$. Let α' denote the set of events involving objects of Γ' that do not involve any objects of Γ, so*

$$\alpha' \triangleq \{\langle o_1, o_2, m\rangle \mid (o_1 \in \mathcal{O}(\Gamma') \vee o_2 \in \mathcal{O}(\Gamma')) \wedge o_1 \notin \mathcal{O}(\Gamma) \wedge o_2 \notin \mathcal{O}(\Gamma)\}.$$

Then Γ' is a proper refinement of Γ with respect to Δ if $\alpha' \cap \alpha(\Delta) = \emptyset$.

For most practical cases, we can verify that new object identifiers are fresh, i.e. $(\mathcal{O}(\Gamma') - \mathcal{O}(\Gamma)) \cap \mathcal{E}(\Delta) = \emptyset$, which entails properness. For specifications satisfying the composability and properness criteria, we infer a compositional refinement property.

Theorem 1 *Let Γ, Γ' and Δ be component specifications such that Γ' is a proper refinement of Γ with respect to Δ and assume that Γ' and Δ are composable. Then $\Gamma'||\Delta \sqsubseteq \Gamma||\Delta$.*

2. Generic Object Viewpoints

In this section, we present a high-level syntax for interface declarations for asynchronously communicating objects. Interfaces are separated from the objects that implement them to focus on behavioral reusability. The syntax here is a subset of a richer language which also includes contracts between objects and class definitions [20]. The language is strongly typed, and objects are typed by interfaces that correspond to different viewpoints. Objects typed by the same interfaces can be used in the same places. An object offers an interface to its environment when its class implements that interface. An object offering an interface F can be used in place of an object offering a *superinterface* of F. This substitutability requires a rigorous definition of interface inheritance.

2.1 Asynchronous Communication

In distributed systems, communication between objects is typically asynchronous. In the traces of the communication model, each remote method call is represented by two distinct events, which we refer to as the initiation and the completion of the call, respectively. Each event contains information about input or output values, a tag for initiation or completion, identities of the sending and receiving objects, and a method name. (It is necessary to restrict the traces of Section 1 to a subtype which ensures that initiations and completions are matched correctly [14].) An operation is declared (in an interface) as

$$\textbf{opr}\ m(\textbf{in}\ p_1\colon T_1,\ldots,p_i\colon T_i\ \textbf{out}\ p_{i+1}\colon T_{i+1},\ldots,p_j\colon T_j)$$

where the keywords **in** and **out** initiate (optional) input and output parameter lists (and **in** is default). Say that an object o_1 offers this method to its environment, and that the method is invoked by another object o_2. This remote call is first reflected in the traces by an initiation event, represented as $o_2 \to o_1.m(p_1,\ldots,p_i)$. If a response is returned to o_2, this is reflected by a completion event in the traces, denoted $o_2 \twoheadleftarrow o_1.m(p_1,\ldots,p_i;p_j,\ldots,p_n)$. This graphical notation is used for readability instead of the triples of Section 1. The semicolon separates input and output values. For methods without explicit output values, there is simply no semicolon nor values after it. We refer to o_2 as the caller and to o_1 as the receiver of both events $o_2 \to o_1.m(\ldots)$ and $o_2 \twoheadleftarrow o_1.m(\ldots)$, where $m \in Mtd$.

Global traces are communication histories for an entire system or sub-system, whereas local traces describe the histories restricted to a subset of the communication events for one object or component. Asynchronous communication means that the calling object need not wait for the completion of a call; in the (local) traces, other communication events may occur between the initiation and completion events of a particular call. Synchronous communication can be captured in this model by disallowing such intermediary events.

2.2 High-level Interfaces

An interface contains syntactic definitions of operations and behavioral constraints, appearing as an assumption-guarantee specification [15]. The assumption is a requirement on the behavior of the objects in the communication environment. As customary in the assumption-guarantee paradigm, the guaranteed invariant need only hold when the assumption is respected by the environment. However, in our setting, the paradigm is extended to deal with input and output aspects of communicating systems. The semantic requirements of an interface rely on the available communication history up to present time of an object offering the interface, i.e. they are predicates on the finite traces. An interface declaration has the following general form:

```
interface $F$ [⟨type parameters⟩] (⟨context parameters⟩)
  inherits $F_1, F_2, \ldots, F_m$
begin
  with $G$
    opr $m_1(\ldots)$
       ⋮
    opr $m_n(\ldots)$
    asm <formula on local trace restricted to one calling object>
    inv <formula on local trace>
  where <auxiliary function definitions>
  end
```

where $F, F_1, \ldots, F_m$, and G are interfaces. Type parameters (square brackets) are data types or interfaces. Context parameters (ordinary brackets) are values (typed by data types) and objects parameters (typed by interfaces) and describe the minimal environment that any object offering the interface needs at the point of creation. Type parameterized interfaces appear as interface templates. We now consider interfaces where both type and context parameters are fixed. These interfaces describe viewpoints to objects.

For active objects, we may want to restrict access to calling objects of a particular interface. This way, the current object can invoke methods of the caller and not only passively complete invocations of its own methods. Use of the **with** clause restricts the communication environment of an object (considered through the interface) to external objects offering a given interface. For passive objects, no such knowledge is required and the **with** clause is omitted. The interface declared in the **with** clause is called a *cointerface*. Mutual dependency is specified if two interfaces have each other as cointerface. Remark that the **inherits** clause as well as the **with** clause indicate an ordering of interface declarations.

Denote by $o\colon F$ an object o which offers an interface F (as above, but for given parameters). For $o\colon F$, we can derive an alphabet $\alpha(o\colon F)$ of communication events from the declaration of F. To each method, we associate initiation and completion events, varying over possible input and output values and possible calling objects. The alphabet of $o\colon F$ is the set of events of the form $o'{\rightarrow}o.m(...)$ and $o'{\leftarrow}o.m(...)$ where o' is an object in the enviroment and m is a method declared in F (or a superinterface of F), as well as $o{\rightarrow}o'.m(...)$ and $o{\leftarrow}o'.m(...)$ where m is a method declared in some interface F', which is a formal parameter to F, a cointerface of F, or a cointerface to a superinterface of F. An invocation of a method m in an object o' by another object o is type correct if o' has a (super) interface declaring m (with matching type parameters) and with a cointerface (if present) which is a (super) interface of o.

The alphabet of $o\colon F$ is defined as maximal, considering all potentially external objects. In reality, the communication environment of an object evolves over time, due to information about calling objects and object identifiers transmitted as method parameters. To capture this evolution is part of the task of specifying the acceptable communication traces.

The assumption is a predicate on traces and object identifiers, declared following the **asm** keyword. The assumption is a pointwise requirement on the environment, i.e. it is expected to hold for the local traces restricted to a particular object at a time. Hence, in interface declarations, the assumption predicate varies over traces as well as over calling and current objects. (An object identifier offering the interface is understood as the current object.) Since assumptions are the responsibility of the environment, these are only expected to hold at points in the traces that end with input to the object o considered.

Inputs to an object o are events $o' \rightarrow o.m(\ldots)$ and $o \leftarrow o'.m(\ldots)$, reflecting initiations of calls to methods of o and answers to calls by o to methods of other objects. Outputs are events $o' \leftarrow o.m(\ldots)$ and $o \rightarrow o'.m(\ldots)$, reflecting completions of calls to methods of o and initiations of calls by o to methods of other objects. Let $\mathbf{in}_o(h)$ denote the longest prefix of h ending with input to o. Similarly define $\mathbf{out}_o(h)$ for output. Given an assumption predicate A in an interface declaration of F, we define a predicate A^{in} by

$$A^{in}(h,o) = \forall x \in \mathcal{E} : A(\mathbf{in}_o(h/x), o, x) \quad \text{for } h \in \alpha(o\colon F)^*,$$

and a similar predicate $A^{out}(h,o)$, hiding inputs to o by the function $\mathbf{out}_o(h)$. In interface declarations, we expect $A^{in}(h,o)$ to hold for each object in the environment. The invariant I is a predicate on traces, declared following the **inv** keyword. We define

$$I^{out}(h,o) = I(\mathbf{out}_o(h/\alpha(o\colon F))) \quad \text{for } h \in \alpha(o\colon F)^*.$$

The invariant of a specification is guaranteed to hold for the object offering the interface, so it is a predicate on the entire (local) trace. Hence, by an invariant I declared in an interface, we expect $I^{out}(h,o)$ to hold. Omitted predicate declarations are interpreted as **true**. If auxiliary functions, patterns, or predicates are needed for specification purposes, these are defined by equations, following the **where** keyword.

Traces are used explicitly in interface declarations to determine the behavior at some point in time. We therefore regard the behavior of objects offering the interface as a set $\mathcal{T}$ of possible (finite) traces. For every trace h in such a set $\mathcal{T}$, all prefixes of h represent prior points in the life of the object, so $\mathcal{T}$ is prefix-closed. The trace set $\mathcal{T}(o\colon F)$ of $o\colon F$ is defined as the prefix-closure of the set of finite sequences over $\alpha(o\colon F)$, defined by $\{h : \alpha(o\colon F)^* \mid A^{in}(h,o) \Rightarrow (I^{out}(h,o) \wedge A^{out}(h,o))\}$. We do not want the output from the object to violate the assumption of future extensions to a trace. $A^{out}(h,o)$ is therefore implicitly included as part of the invariant.

When an interface F inherits another interface F_i and o offers F to the environment, $\alpha(o\colon F_i)$ is included in $\alpha(o\colon F)$. For the traces, inclusion is by projection, so $h/\alpha(o\colon F_i) \in \mathcal{T}(o\colon F_i)$ for all $h \in \mathcal{T}(o\colon F)$. The trace set $\mathcal{T}(o\colon F)$ of $o\colon F$, where F inherits interfaces $F_1, \ldots, F_m$ and where A and I are the assumption and invariant predicates of F, is then given by the prefix-closure of

$$\left\{ h : \alpha(o\colon F)^* \;\middle|\; \begin{array}{l} \bigwedge_{1 \leq i \leq m} h/\alpha(o\colon F_i) \in \mathcal{T}(o\colon F_i) \\ \wedge\, A^{in}(h,o) \Rightarrow (I^{out}(h,o) \wedge A^{out}(h,o)) \end{array} \right\}.$$

Composition rules for interface declarations are found in [20].

3. Case Study: Design of a Bank System

Consider a bank system with three kinds of objects: users, teller machines and a center. Each teller machine communicates with one user (at a time) and the center. The center may communicate with several teller machines.

3.1 Syntax of the Interfaces

Interfaces are specified for the different viewpoints to the teller machine (Figure 1) and we use multiple inheritance to obtain a (more complete) spe-

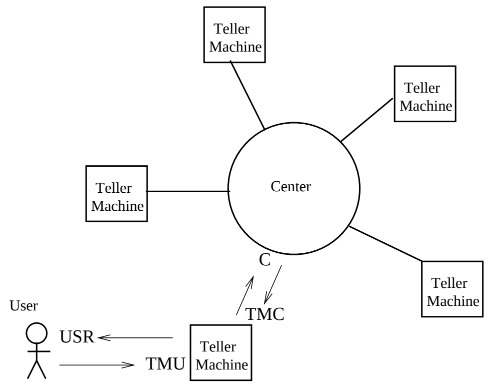

Figure 1. The bank system, illustrated by objects and interfaces (upper case).

cification of the teller machine from the partial descriptions. The interfaces are considered pairwise as they communicate with each other. We first consider the methods and later the behavior of the system. The interaction between the teller machine and the user consists of these operations: *insert*, a card is inserted into the teller machine, *giveCode*, the user attempts to validate the card, *query*, the user chooses activity, either“wd” (withdraw) or “end” (end session), *withdraw*, request withdrawal of some amount by user, *dispense*, the action of dispensing a sum to the user, and *return*, the card is returned to the user. To begin with, we specify an interface `USR` of users towards the teller machine and another interface `TMU` of teller machines towards users.

```
interface USR
begin
  with TMU
    opr giveCode(out $code$ : nat)
    opr query(out $choice$ : nat)
    opr withdraw(out $sum$ : nat)
    opr dispense(in $sum$ : nat)
    opr return()
end
```

```
interface TMU
begin
  with USR
    opr insert(in $card$ : nat)
end
```

The interfaces USR and TMU are mutually dependant. Due to the **with** clause, the teller machine can react to an event $u{\rightarrow}x.insert(card)$ by calling methods of u, available via the USR interface. Thus, $\alpha(x:\texttt{TMU})$ consists of the initiation $u \rightarrow x.insert(card)$, the completion $u \leftarrow x.insert(card)$, and the events of u: USR, for all u. The entire alphabet is available for requirement specification.

Interaction between the teller machine and the center consists of the operations *authorize*, which checks that a code and a card correspond, and *debit*, if the requested amount can be withdrawn from an account, do so and return true. Otherwise, return false. The methods are placed in the interface C of the center, leaving initiative with the teller machine.

```
interface C
begin
  with TMC
    opr  authorize($card$:nat, $code$:nat out $ok$:bool)
    opr  debit($sum$:nat, $card$:nat, $code$:nat out $ok$:bool)
end
```

A corresponding interface TMC, with formal parameter c:C and no methods, is declared for the teller machine. (Teller machines have static links to the center.)

3.2 An Abstract View of the System

Consider a requirement on the overall performance of the system, informally described by: "*Neither bank nor user loses money.*" The requirement, when formalized, says that transactions decrease the amount on a user's account with exactly the sum dispensed to the user, for every teller machine in the system. However, the requirement is only true in between transactions.

Let $\mathcal{R}(x,h)$ calculate the amount received by a card x after a history h of the current teller machine. It is sufficient to consider the completion events of insert and dispense. Ignoring other events, a sequence of dispense actions is preceded by an insert action that identifies the user. Similarly, the amount withdrawn from the account of the card x is calculated by a function $\mathcal{W}(x,h)$. All changes in the account balance can be identified through the completion events of the debit method. Cases are considered in the order listed (à la ML), where "others" matches any event, and parameters in event patterns can be left unspecified by a wildcard "_". The current object of an interface is denoted "this". The requirement must hold when the card has been returned to a user. For traces, ε is the empty trace, $\dashv$ denotes left append, and **ew** denotes "ends with".

```
interface TMA(c:C)
  inherits TMC(c), TMU
begin
  inv $\forall x \in \text{nat} : (h \textbf{ ew } \text{this}{\rightarrow}u.\text{return}()) \Rightarrow \mathcal{W}(x, h/c) = \mathcal{R}(x, h/\text{this}{:}\,\texttt{TMU})$
  where
    $\mathcal{R}$ : nat $\times \alpha(\text{this}{:}\,\texttt{TMU})^* \to$ nat
    $\mathcal{W}$ : nat $\times \alpha(c{:}\,\texttt{C})^* \to$ nat

    $\mathcal{R}(x, \varepsilon) = 0$
    $\mathcal{R}(x, u{\leftarrow}\text{this.insert}(x) \dashv \text{this}{\leftarrow}u.\text{dispense}(y) \dashv h) = \mathcal{R}(x, u{\leftarrow}\text{this.insert}(x) \dashv h) + y$
    $\mathcal{R}(x, u{\leftarrow}\text{this.insert}(\_) \dashv u{\leftarrow}\text{this.insert}(x) \dashv h) = \mathcal{R}(x, u{\leftarrow}\text{this.insert}(x) \dashv h)$
    $\mathcal{R}(x, u{\leftarrow}\text{this.insert}(\_) \dashv \text{others} \dashv h) = \mathcal{R}(x, u{\leftarrow}\text{this.insert}(x) \dashv h)$
    $\mathcal{R}(x, \text{others} \dashv h) = \mathcal{R}(x, h).$

    $\mathcal{W}(x, \varepsilon) = 0$
    $\mathcal{W}(x, \text{this}{\leftarrow}c.\text{debit}(y, x, \_; true) \dashv h) = \mathcal{W}(x, h) + y$
    $\mathcal{W}(x, \text{others} \dashv h) = \mathcal{W}(x, h).$
end
```

3.3 A Communication View of the Teller Machine

We now consider a communication view of the teller machine and specify the interleaving of events from `TMU` and `TMC` for two interaction scenarios:

1 The user gives an incorrect code for the card, so authorization fails. The card is returned to the user and the session terminates.

2 The user gives the correct code to the teller machine. The session continues by a query-driven menu where the user requests an amount, this request is accepted by the center and the amount is dispensed. At the end of the session, the card is returned.

These scenarios are formalized in the interface TM.

```
interface TM(c:C)
  inherits TMC(c:C), TMU
begin
  inv $h$ prp [start$(u, x, y)$ [goodSession$(u, x, y)$ | badCode$(x, y)$] quit(u) $\bullet$ $u$: USR, $x, y$: nat]$^*$
  where
    start$(u, x, y) = u{\leftrightarrow}\text{this.insert}(x)$ $\text{this}{\leftrightarrow}u.\text{giveCode}(y)$
    okCode$(x, y) = \text{this}{\leftrightarrow}c.\text{authorize}(x, y; true)$
    badCode $(x, y) = \text{this}{\leftrightarrow}c.\text{authorize}(x, y; false)$
    okWithdraw$(u, x, y) = \text{this}{\leftrightarrow}u.\text{query}(;$ “wd”$)$ badAmount$(u, x, y)^*$ Dispense$(u, x, y)$
    Dispense$(u, x, y) =$ okAmount$(u, x, y, sum)$ $\text{this}{\leftrightarrow}u.\text{dispense}(sum)$ $\bullet$ $sum$ : nat
    okAmount$(u, x, y, sum) = \text{this}{\leftrightarrow}u.\text{withdraw}(; sum)$ $\text{this}{\leftrightarrow}c.\text{debit}(sum, x, y; true)$
    badAmount$(u, x, y) = \text{this}{\leftrightarrow}u.\text{withdraw}(; sum)$ $\text{this}{\leftrightarrow}c.\text{debit}(sum, x, y; false)$
    goodSession$(u, x, y) =$ okCode$(x, y)$ okWithdraw$(u, x, y)^*$ $\text{this}{\leftrightarrow}u.\text{query}(;$ “end”$)$
    quit(u) $= \text{this}{\leftrightarrow}u.\text{return}()$
end
```

The symbol $\leftrightarrow$ in a pattern represents an initiation succeeded by its completion. Thus, $u{\leftrightarrow}\text{this.insert}(\_)$ reflects the initiation of a call to the method *insert* of “this by “u”, followed by its completion. Incidentally, `TM` refines `TMA`. The argument uses induction over the traces of a TM-object and is omitted here.

4. Conclusion

This paper presents a formalism for object viewpoints in open distributed systems. In contrast to state-based approaches, we do not assume knowledge of implementation details for objects in the environment. For open systems, this is particularly attractive as components may be provided and replaced by different manufacturers at different times. Instead of using transitions on internal states, object behavior is captured by observable interaction sequences. Reasoning in terms of input and output lets us specify active as well as reactive objects. By composing viewpoints to different objects, we specify viewpoints to system components, i.e. system services on distributed platforms, where several objects participate to provide the service. A refinement relation for viewpoint specifications is suggested to capture controlled forms of service upgrade and restructuring. Compositional reasoning by stepwise refinement is supported. Using explicit object identities, the formalism captures certain forms of openness, such as transmission of object identifiers and addition of new objects in component refinement.

In order to facilitate reuse of specifications, object interfaces are introduced that correspond to generic viewpoint specifications. The interfaces contain both syntactic declarations and behavioral constraints. For open distributed systems, it is common to employ an assumption guarantee style of object specification. With our observational approach, we introduce a particular flavor of input/output-driven assumption guarantee specifications, where the assumption considers histories that end with input while the guarantee considers histories that end with output. Trace patterns provide a graphical specification style.

Interfaces and classes are organized in distinct inheritance hierarchies and behavioral reasoning about objects of given interfaces is supplemented by static correctness proofs for the code. The language is supported by a proof environment in PVS to facilitate formal reasoning based on the semantic foundation of the formalism. In ongoing work, we investigate similar notions of refinement for fault tolerance, time-outs, and other robustness issues, methodology for capturing UML-type specifications into the formalism, and liveness specifications.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan. 1997.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[4] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In A. Yonezawa and S. Matsouka, editors, *Proceedings of Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, pages 187–209, Springer-Verlag, Sept. 2001.

[5] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Systems*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1991.

[6] R. J. R. Back and J. von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *Proceedings of CONCUR'94*, volume 836 of *Lecture Notes in Computer Science*, pages 367–384. Springer, Uppsala, Sweden, Aug. 1994.

[7] L. Blair and G. Blair. Composition in multi-paradigm specification techniques. In R. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proceedings of FMOODS'99*, pages 401–418. Kluwer Academic Publishers, Feb. 1999.

[8] E. Boiten, J. Derrick, H. Bowman, and M. Steen. Consistency and refinement for partial specification in Z. In M.-C. Gaudel and J. Woodcock, editors, *Proceedings of FME'96*, volume 1051 of *Lecture Notes in Computer Science*, pages 287–306. Springer, Mar. 1996.

[9] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Monographs in Computer Science. Springer, 2001.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001. To appear.

[11] The component object model specification. `http://www.microsoft.com/com`, 1995.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[13] ITU Recommendation X.901-904 ISO/IEC 10746. *Open Distributed Processing - Reference Model parts 1–4*. ISO/IEC, July 1995.

[14] E. B. Johnsen and O. Owe. A PVS proof environment for OUN. Research Report 295, Department of informatics, University of Oslo, 2001.

[15] C. B. Jones. *Development Methods for Computer Programmes Including a Notion of Interference*. PhD thesis, Oxford University, UK, June l981.

[16] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*. Springer, June 1997.

[17] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Prentice Hall, 1999.

[18] T. Nelson, D. Cowan, and P. Alencar. A model for dsecribing object-oriented systems from multiple perspectives. In *Proceedings of FASE 2000*, volume 1783 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 2000.

[19] Object Management Group. *UML Language Specification, version 1.4*, Sept. 2001.

[20] O. Owe and I. Ryl. A notation for combining formal reasoning, object orientation and openness. Research Report 278, Department of informatics, University of Oslo, 1999.

[21] 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.

[22] W. A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1997.

[23] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[24] P. Tarr, H. L. Ossher, W. H. Harrison, and S. M. Sutton, Jr. *N* degrees of separation: Multi-dimentional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. IEEE / ACM Press, May 1999.

[25] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Object Technology Series. Addison-Wesley, 1999.