

# An Exception Aware Behavioral Type System for Object-Oriented Programs

Filipe Militão and Luís Caires

CITI / Departamento de Informática, Universidade Nova de Lisboa, Portugal

We develop a type system for object oriented languages that combines standard type information with behavioral protocol specifications. The typing rules cover familiar constructs, as well as exceptions, which are a main novelty in this work: exceptions may cause abrupt control transfer in allowed behaviors, and have been particularly difficult to tackle with behavioral type systems. The type system guarantees protocol fidelity both at the method level and at the class level by checking consistency in the use of fields with the class' usage protocol. It also ensures that program execution always reaches a safe termination state, even in the presence of behavioral borrowing, that is, temporary aliasing of object references during methods calls.

## 1 Introduction

The ever increasing software complexity has always been the striving force behind the push for more advanced type systems in an effort to reduce the number of software bugs. Although virtually all modern day verification techniques guarantee the absence of important errors, such as calling non-existing methods or incorrect conversion of data structures, they still leave much room for improvements. Namely, most are incapable of checking the fulfillment of prevalent APIs assumptions such as specific restrictions in the allowed sequences of calls (that is, making sure a protocol is obeyed) or identifying possible sources of interferences in the use of objects (due to aliasing, for instance). Therefore, recent developments in typing methodologies try to address such issues (including other emerging use cases such as web services and concurrency) to protect programs from non-trivial and frequent mistakes.

We focus on the problem of checking the behavior restrictions in the use of methods, that is, how some objects require specific sequences of calls to be observed in order to function properly. File objects are an usual example: they require to be opened before used and closed at the end (to force flushing out any changes). This behavioral protocol is usually defined in attached documentation and/or checked only at run-time (with any error flagged as an exception). We propose a type system that uses a formal behavior description to statically ensure the absence of behavioral errors in sequential programs, so that such protocols are always obeyed from start to finish. While the idea of exploiting behavioral specifications to discipline the usage of objects in programs is not new, in this paper we approach language features much more challenging than in other approaches, such as borrowing behaviors, full class consistency check, and, prominently, exception handling. We have also implemented our type system in a working prototype interpreter and runtime system, the **yak** language [22].

Example 1 shows a *Travel* class that defines a **usage** protocol specifying its allowed behavior. Then, the **let** expression uses that class accordingly, where each method call

causes a transition in the allowed behavior. The comments show how the behavior of the new object evolves until it reaches a termination state, **stop**. This is but a tiny example: our full protocol description language is capable of describing exceptions, choices, and behavior selection based on the result of a call (as illustrated on Section 3.3).

*Example 1.* Travel example.

<pre>class TravelOrder {   usage flight.hotel.(buy+cancel)   void flight(){ ... }   void hotel(){ ... }   void buy(){ ... }   void cancel(){ ... } }</pre>	<pre>let t in   t = new TravelOrder();       //TravelOrder#flight.hotel.(buy+cancel)   t.flight(); //TravelOrder#hotel.(buy+cancel)   t.hotel(); //TravelOrder#buy+cancel   if ( ... ) t.buy() //TravelOrder#stop   else t.cancel() //TravelOrder#stop</pre>
--	--

In our language each class declares a behavioral protocol that restricts the use of some of its methods to contexts given by its **usage**. The syntax and semantics (Section 2) is similar to the one of mainstream OO languages. The behavioral information is only used by the type system (Section 3), that handles usual expressions such as branches, loops, exceptions and is also able to check the behavior in cases of recursion, behavioral borrowing and subtyping (Section 3.1). The consistency check procedure (Section 3.2) guarantees behaviors are obeyed, both at the method level and at the class level so that fields are used correctly during the lifetime of an object. Finally, in Section 3.3, we briefly address some flexibility issues of the typing rules and give additional examples. The original contributions of this paper include:

- A simplified type annotation that reduces the burden on the programmer (Section 3), namely, by using a consistency check phase (Section 3.2) that automatically handles the behavior of fields in each method. Additionally, our typing rules do not interfere with type abstraction or modularity and are flexible enough not to require a direct mapping of the protocol into code (Section 3.3). Although due to space limitations we cannot include a formal proof, our type system is provably sound [17].
- Addressing changes in behavior due to raised exceptions has not been much explored even if some proposals [14] explored ideas similar to ours, they have many limitations and are not used in an OO language. In this work, we address behavioral exceptions without compromising much expressiveness (Section 3).
- Although we enforce linear aliasing control [21], we do not require all method arguments to possess unique ownerships. Such features are useful to model usual call-by-reference and is conceptually consistent with borrowing behaviors of object references in limited scopes, such as during method calls (details in Section 3).

## 2 Syntax and Informal Semantics of the Core Language

We develop our type system for a core Java-like object oriented language, based on ClassicJava [11] (we leave out inheritance, for now). The syntax is shown in Figure 1. A program in this language is formed by several **class** definitions (*def*) each with a behavioral protocol defined after the **usage** keyword. The entry point is an expression (*e*), after the list of definitions. In a method call the callee object must be represented by

$e ::= v$ $\begin{array}{l}   e; e \\   x = e \\   \mathbf{if}(e) e \mathbf{else} e \\   \mathbf{while}(e) e \\   \mathbf{try} e \mathbf{catch}(N x) e \\   \mathbf{throw} e \\   \mathbf{let} x \mathbf{in} e \\   \mathbf{new} c() \\   \beta.m(e^*) \end{array}$	$c \in \text{class names}$ $m \in \text{method names}$ $x, y, f \in \text{variable names}$	$prog ::= def^* e \quad (\text{program})$ $def ::= \mathbf{class} c \{ \quad (\text{class decl})$ $\quad \quad \mathbf{usage} P \quad (\text{behavior})$ $\quad \quad \quad field^* \quad (\text{fields})$ $\quad \quad \quad meth^* \quad (\text{methods})$ $\quad \quad \quad \}$ $field ::= N x$ $meth ::= T m(arg^*) [\mathbf{throws} N^*]_{opt} \{e\}$ $arg ::= [\mathbf{owned}]_{opt} T x \quad (\text{argument})$ $v ::= \mathbf{null}$ $\quad   \mathbf{true}   \mathbf{false}$ $\quad   \beta$ $\beta ::= x   \mathbf{this}$ $T, U ::= \mathbf{void}   N \# P \quad (\text{type})$ $N ::= \mathbf{boolean}   c \quad (\text{name})$
---	--	--

**Fig. 1.** Syntax of the core language.

a variable or a **this** pointer ( $\beta$ ). An **owned** modifier informs the type system that an argument requires permissions to be returned or stored in a variable in the method's body (this will be describe in detail in the ownership control section of the type system). Due to lack of space, we cannot fully describe the operational semantics of our language, but this will not hinder much the understanding of the reader since most constructs are standard. Therefore, we move immediately to the most interesting issues in our work, namely the type system.

### 3 A Behavioral Type System for OO programs

In this section we describe the key ideas behind the design of our type system. We define a *behavioral type* to be a pair of standard (Java-like) type with a behavioral description for that type. Thus, our type notation is of the form:

$$Type \triangleq Name \# Behavior$$

The first part (*Name*) refers to a list of method declarations and to the **usage** protocol (i.e., the initial behavior), both defined in the class declaration for *Name*. The second part (*Behavior*) contains the changing protocol (due to a method call, for instance) that is used during the verification. The language for specifying such usage protocols is defined in (Figure 2). As shown in Figure 1 a type's behavioral protocol is declared after the keyword **usage** in the class definition.

The behavioral descriptions describe the stages of the protocol on which a call to a specific method is allowed. We use a regular expressions-like description language, that includes recursion ( $\&r$  to create the recursion point and then  $r$  to “jump” back to that position - assuming  $r$  to be unique), choice (+), allowing us to express alternatives in the protocol, and empty behavior (**stop**). We include a new behavioral construct within

$P, Q, V, O ::= m[\textit{exception}^*].R$	(method and continuation)
$r$	(recursion label)
$\&r(P)$	(recursion point)
$P + P$	(choice)
<b>stop</b>	(empty)
$R ::= P$	
$(\mathbf{true}.P) + (\mathbf{false}.P)$	(result based behavioral choice)
$\textit{exception} ::= N : P$	(behavioral exception, if $N$ raised then change to $P$ )

**Fig. 2.** Syntax for behavioral descriptions.

method calls (*exception*), to capture exceptional behavior. We also include a construct to express behavioral alternatives, based on the value returned by of a method call (only for booleans, although such a system could be easily extended to support other enumerable types). It is convenient to further explain these two novel constructs :

- “ $m[N : P|\dots].Q$ ”, the description for a method usage. This construction declares that a method ( $m$ ) can be called on that specific context and that it may throw any of the exceptions inside the square brackets. More precisely, after an exception of type  $N$  the allowed behavior for that type changes to  $P$ , this is what we call a *behavioral exception*. When no exception is raise the protocol continues with  $Q$ ;
- “ $(\mathbf{true}.P) + (\mathbf{false}.Q)$ ”, declares a change in the allowed behavior based on the result of a **boolean** call. Therefore, this construct is only allowed to appear immediately after a method usage. This description allows the type system to e.g., distinguish between the availability of the *next* method in an iterator, depending on the value returned by the *hasNext* method.

The choice construction offers an external decision (where the programmer is free to decide which behavioral path to take); and the exception/result-choice is related to an internal choice as they change the allowed behavior only based on the class’ internal code. We categorize the methods of a class in two groups: *behavioral methods* - methods whose name appears in the usage protocol and thus that have their use restricted; and *free* (non-behavioral) methods - methods that do not appear in the protocol and may be freely used in any context regardless of the state of the protocol, for example, methods such as *toString*). When a behavior reaches **stop** only free methods are available. Such categorization of methods remains fixed throughout the life of a type since it is related to the **usage** protocol, not to the dynamic state of a type.

Although our type system is fully static, it needs to track the dynamic state of the objects’ protocols. Therefore, the basic typing judgment uses effect-tracking to model such changes, as caused by the evaluation of expressions. It has the following form:

$$\Delta \vdash e : T \rightsquigarrow \Delta'$$

The intuitive reading of such judgment is: the initial environment ( $\Delta$ ) on which the expression ( $e$ ) is checked to be of type ( $T$ ) causes some side-effects on the initial environment, and leads to the final environment ( $\Delta'$ ). When expression  $e$  is of type **boolean**,  $\Delta'$  has the form  $(\Delta_T|\Delta_F)$ , reflecting the two possible final states (see [17]).

$$\begin{array}{c}
\text{boolean}\#\text{stop} \equiv \text{boolean} \\
\hline
\begin{array}{c}
\text{[R-NULL]} \\
\frac{}{\Delta \vdash \text{null} : N\#\text{stop} \rightsquigarrow \Delta} \\
\text{[R-TRUE]} \\
\frac{}{\Delta \vdash \text{true} : \text{boolean} \rightsquigarrow \Delta} \\
\text{[R-FALSE]} \\
\frac{}{\Delta \vdash \text{false} : \text{boolean} \rightsquigarrow \Delta} \\
\text{[R-PROG]} \\
\frac{\forall c.(c \in \text{def}^* \Rightarrow (\text{def}^* \vdash c : \text{OK} \rightsquigarrow \text{def}^*))}{\text{def}^* \vdash e : T \rightsquigarrow \text{def}^* \text{ stopped}(T)} \\
\frac{}{\emptyset \vdash \text{def}^* e : T \rightsquigarrow \emptyset} \\
\text{[R-NEW]} \\
\frac{c \in \Delta}{\Delta \vdash \text{new } c() : c\#\text{usage}(c)^\circ \rightsquigarrow \Delta} \\
\text{[R-SEQ]} \\
\frac{\Delta \vdash e_0 : T_0 \rightsquigarrow \Delta' \text{ stopped}(T_0) \quad \Delta' \vdash e_1 : T_1 \rightsquigarrow \Delta''}{\Delta \vdash e_0; e_1 : T_1 \rightsquigarrow \Delta''} \\
\text{[R-IF]} \\
\frac{\Delta \vdash e^{\text{cond}} : \text{boolean} \rightsquigarrow (\Delta_T | \Delta_F) \quad \Delta_T \vdash e^{\text{if}} : T \rightsquigarrow \Delta' \quad \Delta_F \vdash e^{\text{else}} : T \rightsquigarrow \Delta'}{\Delta \vdash \text{if}(e^{\text{cond}}) e^{\text{if}} \text{ else } e^{\text{else}} : T \rightsquigarrow \Delta'} \\
\text{[R-WHILE]} \\
\frac{\Delta \vdash e^{\text{cond}} : \text{boolean} \rightsquigarrow (\Delta_T | \Delta_F) \quad \Delta_T \vdash e : N\#P \rightsquigarrow \Delta \text{ stopped}(P)}{\Delta \vdash \text{while}(e^{\text{cond}}) e : N\#\text{stop} \rightsquigarrow \Delta_F} \\
\text{[R-THROW]} \\
\frac{\Delta \vdash e : N\#P \rightsquigarrow \Delta' \text{ stopped}(P)}{\Delta, (N \rightsquigarrow \Delta') \vdash \text{throw } e : T \rightsquigarrow \emptyset} \\
\text{[R-TRY]} \\
\frac{\Delta, (N \rightsquigarrow \Delta_{\text{catch}}) \vdash e^{\text{try}} : T \rightsquigarrow \Delta', (N \rightsquigarrow \Delta_{\text{catch}}) \quad \Delta_{\text{catch}}, (x : N\#\text{stop}), (N \rightsquigarrow \Delta_N) \vdash e^{\text{catch}} : T \rightsquigarrow \Delta', (x : N\#\text{stop}), (N \rightsquigarrow \Delta_N)}{\Delta, (N \rightsquigarrow \Delta_N) \vdash \text{try } e^{\text{try}} \text{ catch}(N x) e^{\text{catch}} : T \rightsquigarrow \Delta', (N \rightsquigarrow \Delta_N)}
\end{array}
\end{array}$$

**Fig. 3.** Basic typing rules.

A typing environment ( $\Delta$ ) is a set of declarations, that may contain:

1. ( $x : T$ ) - variable (labeled  $x$  and of type  $T$ );
2. ( $\text{def}$ ) - class definition;
3. ( $N \rightsquigarrow \Delta_N$ ) - exception handler for a type  $N$  where  $\Delta_N$  is the environment to be used on the nearest **catch** branch of a **try catch** in scope;

Since all the elements of an environment are uniquely declared, we use ( $\Delta = \Delta_0, \dots, \Delta_n$ ) to split them into other disjoint sub-environments. In Figure 3 we present the basic typing rules of our type system, we briefly discuss each of them.

**R-NULL** A **null** value can be used as some type with a **stop** behavior.

**R-PROG** To check a program we start by checking the consistency of each class ( $c$ ) before moving to the initial expression ( $e$ ). Since the resulting value cannot be used elsewhere it must be stopped, the behavior must be able to terminate at this point.

**R-NEW** The new object starts with the **usage** protocol and is uniquely owned (noted by the  $^\circ$ ) because it is a newly created value.

**R-SEQ** Since it ignores the result of its left side ( $T_0$ ), we require that type to be stopped so no behavior is lost. Any side-effects that it may produce are carried on to the right side by the environment  $\Delta'$  which produces the final result and the resulting environment of this expression.

**R-IF** Follows the usual **if else** flow with a double environment ( $\Delta_T | \Delta_F$ ) for the case when there is a result-based behavioral choice in  $e^{\text{cond}}$ .

$$\begin{array}{c}
\text{[R-LET]} \\
\frac{\Delta, (x : N\#\text{stop}^\circ) \vdash e : T \rightsquigarrow \Delta', (x : N\#P^\circ) \quad \text{stopped}(P)}{\Delta \vdash \text{let } x \text{ in } e : T \rightsquigarrow \Delta'} \\
\text{[R-TAKE]} \\
\frac{P \xrightarrow{P} \text{stop}}{\Delta, (\beta : N\#P^\circ) \vdash \beta : N\#P^\circ \rightsquigarrow \Delta, (\beta : N\#\text{stop}^\circ)}
\end{array}
\qquad
\begin{array}{c}
\text{[R-ASSIGN]} \\
\frac{\Delta \vdash e : N\#P^\circ \rightsquigarrow \Delta', (x : N\#Q^\circ) \quad \text{stopped}(Q)}{\Delta \vdash x = e : N\#\text{stop} \rightsquigarrow \Delta', (x : N\#P^\circ)} \\
\text{[R-BORROW]} \\
\frac{P \xrightarrow{V} Q}{\Delta, (\beta : N\#P) \vdash \beta : N\#V^\bullet \rightsquigarrow \Delta, (\beta : N\#Q)}
\end{array}$$

**Fig. 4.** Ownership control typing rules.

**R-WHILE** The  $\Delta$  environment models the loop invariant. The result type ( $N\#\text{stop}$ ) ignores the behavior  $P$  since we defined the **while** body as leaving a **null** result.

**R-THROW** The run-time **catch** mechanism does not handle behavior on the raised object, thus its behavior ( $P$ ) must be stopped. The exception handler defines that the catch environment must be the same as the one resulting from  $e$ . The empty environment ( $\emptyset$ ) signals that any following expression will be unreachable.

**R-TRY** The catch environment ( $\Delta_{catch}$ ) for the type  $N$  is used as a handler inside the **try** branch and as the initial environment in the **catch**. Both branches produce the same environment ( $\Delta'$ ) so that the behavior afterwards is independent of what happens at run-time, note the previous handler for type  $N$  is restored.

Any type system as ours, that tracks the flow of calls, has to deal with another important issue: aliasing control. Aliasing induces interference, which may easily break the prescribed usage protocols. We decided to use a kind of behavioral linearity for controlling access to each object's behavior. Thus, there is no true aliasing as the full behavior is only visible to one variable at a time. However, the non-behavioral or free view of any object has no such restrictions since the use of a stopped type never causes behavioral interferences. Nonetheless, using only this limited view on aliasing would be too restrictive and so we decided to include the option of "borrowing" these behaviors for arguments in calls. Thus, an object can be lent for the specific duration of a call and afterwards continue to be used as if it were always owned by the original variable.

To model this phenomenon, we introduce a simulation operation ( $P \xrightarrow{Q} V$ ). We assert  $P \xrightarrow{Q} V$  when an object subject to the usage protocol  $P$ , may still be used as defined by  $V$ , after a temporary usage as specified by  $Q$ . The simulation relation is formally defined by a set of rules, that we cannot include here for lack of space (see [17]). The simulation relation also deals with exceptions, since the argument's protocol  $Q$  must be fulfilled in all situations, even when an exception is raised. This solution leads to a whole new set of problems that we need to solve, namely: how can we save a value in a field or return it if it could be borrowed by someone else? For this reason, we use the concept of ownership to distinguish when an object is or not uniquely owned or if that uniqueness is required in a call. Thus, it is only possible to return values or save them in fields if they are **owned** (noted  $T^\circ$ ). After an ownership is taken away, the variable only retains a stopped view of that object. In the case of borrowed behaviors, the variable does not truly own the object, instead it has a non-owned type (noted  $T^\bullet$ ) that cannot be returned or assigned (thus, is incompatible with the owned type) but still can call the

$$\begin{array}{c}
\text{[R-CALL]} \\
m(T_i)[N_j] T^\circ \in \text{methods}(N) \quad i \in \{0, \dots, n\} \quad j \in \{0, \dots, k\} \\
\Delta = \bigcup_{i=0}^n \Delta_{e_i} \quad \Delta_{e_i} \vdash e_i : T_i \rightsquigarrow \Delta'_{e_i} \quad \bigcup_{i=0}^n \Delta'_{e_i} = \Delta', \Delta_{f\text{-before}}, (N_j \rightsquigarrow \Delta_{\text{catch}_j}) \\
\beta \xrightarrow{m} O|P \quad \Delta_{f\text{-before}} \xrightarrow{\beta, m} \Delta_{f\text{-after}} \quad \beta \xrightarrow{m, N_j} V_j \quad \Delta_{f\text{-before}} \xrightarrow{\beta, m, N_j} \Delta_{f\text{-catch}_j} \\
\Delta''_O = \Delta', \Delta_{f\text{-after}}, (\beta : N\#O) \quad \Delta_{\text{catch}_j} = \Delta', \Delta_{f\text{-catch}_j}, (\beta : N\#V_j) \\
\Delta''_P = \Delta', \Delta_{f\text{-after}}, (\beta : N\#P) \\
\hline
\Delta, (\beta : N\#Q) \vdash \beta.m(e_0, \dots, e_n) : T^\circ \rightsquigarrow \Delta''_O | \Delta''_P
\end{array}$$

**Fig. 5.** Method Call typing rule.

methods of its protocol. When no explicit ownership notation ( $T$ ) is given it is assumed that any ownership value will do. We now explain some key aspects of the ownership control typing rules.

**R-LET** Creates a variable ( $x$ , with an initial type compatible with **null**) to be used inside its body ( $e$ ). Therefore, before it falls out of scope we must make sure that any remaining behavior it may have ( $N\#P$ ) is stopped.

**R-ASSIGN** An assignment can only occur with variables that are owned. Thus, the old content ( $N\#Q$ ) will be lost and must be stoppable.

**R-TAKE** Taking a content (owned read) is removing all the behavior it may have. It will lose the possession of any behavior, keeping only a **stop** state.

**R-BORROW** A borrow read causes some of the behavior to be read in a non-owned way. That is, it cuts a prefix of the behavior and what remains can continue to be used afterwards. This kind of read can only occur when checking an argument of a method call and as such, together with the disjoint checking of arguments (that will be explained further down), it makes it possible to do behavioral borrowing by using those types for non-owned arguments.

*Example 2.* Some examples involving ownership control.

<pre> <b>let</b> v <b>in</b>   v = <b>new</b> C();   //v: C#a.b.c   //"void m(C#a.b x)"   //'borrows' C#a.b   m(v); //v: C#c   v.c() //v: C#stop </pre>	<pre> <b>let</b> v <b>in</b>   v = <b>new</b> F();   //v: F#a.(b+stop)   //"void m(owned F#a x)"   // =&gt; 'x' requires ownership   // F#a.(b+stop) &lt;: F#a   m(v); // #b+stop is hidden   //v: F#stop </pre>	<pre> <b>let</b> v0 <b>in</b>   <b>let</b> v1 <b>in</b>     v0 = <b>new</b> D();   //v0: D#a v1: D#stop   v1 = v0;   //v0: D#stop v1: D#a   v1.a()   //v0: D#stop v1: D#stop </pre>
---	--	---

**R-CALL** This is the most complex rule of our type system. First, each argument is checked in a disjoint sub-environment ( $\Delta_{e_i}$ ) that excludes any interferences between them. Then, it must move the behavior of the caller from state  $Q$  to the normal (non-exceptional) behavior after the  $m$  call (written  $O|P$ : an alternative behavior to account for result-based behavioral choices; equal alternatives are used if there is no such choice). This effect is captured by the assertion  $\beta \xrightarrow{m} O|P$ , which is defined from the simulation relation (see [17]). Self-inflicted calls do not cause changes in the behavior: from our point of view, the protocol is only meant to express restrictions to clients of

the object (i.e. from the outside) and not internally. Thus, an object may freely call any of its own methods without causing a change in its current behavior. However, the type system must anyway keep track of changes in the behavior of the object fields, as stated in the  $\Delta_{f\text{-after}}$  environment with the field changes caused the  $\beta.m$  call. The type rule also needs to take into account the possibility of the method call raising an exception. To take care of that, a verification similar to the one expressed in the R-THROW rule must be performed for each raisable exception ( $N_j$ ). However, we must account for behavioral exceptions by stepping the state to the exception behavior  $V_j$  while also considering possible changes in the fields ( $\Delta_{f\text{-catch}_j}$ ).

### 3.1 Subtyping

In order to improve flexibility, our type system also includes a rich subtyping relation. The most interesting feature of our subtyping by structure is the use of behavioral protocol compatibility. Thus, in our setting, subtyping must also take into account the interchangeability of usage protocols (so that protocols can be safely replaced without violating expectations) which is achieved directly with the simulation operation.

As described above, we split a class' methods into two groups: behavioral and free methods. A subtype cannot move a method between these two groups, thus a behavioral method in a subtype  $T$  must also be behavioral in the supertype  $U$  (and an identical situation with free methods). In general, a type  $T$  is a subtype of a type  $U$  ( $T <: U$ ) if:

- for each method in  $U$ , there must exist a method in  $T$  with the same name and with a compatible method signature (usual method subtype) while also belonging to the same behavioral/free group. As usual, the subtype is free to define additional methods in any group.
- the simulation of  $U$ 's current behavior protocol must be compatible with the protocol of  $T$ , that is, by simulating in  $T$  the protocol of  $U$  it must be able to reach a **stop** state so that the subtype includes at least all the behavior of the supertype.

*Example 3.*

$$\text{TravelOrder}\#_{\text{hotel}}.(buy + cancel) \quad \text{Order}\#(buy + cancel)$$

These two types are incompatible: the *TravelOrder* requires a method “*hotel*” that does not exist in *Order* and both protocols are incompatible. However, once *TravelOrder* calls “*hotel*” the following relation becomes valid (but not the reversed):

$$\text{TravelOrder}\#(buy + cancel) <: \text{Order}\#(buy + cancel)$$

Notice that the condition for the behavioral/free methods is based on the **usage** protocol of each class. However, the protocol compatibility (through the simulation operation) uses the current state of the behavior and therefore this relation now holds.

### 3.2 Consistency Check

Another important feature of our type system is that it verifies that client code respects usage protocols, but also that server code (e.g., classes) implement the usage protocols they declare. Although we will not go into formal details, this is achieved by a

consistency check of the use of fields throughout a class' behavior (testing if it is OK). Essentially, it carries the behavior of fields over all possible paths of the **usage** protocol. Thus, these variables start with a **stop** behavior and at each termination point of the protocol they must also be in a stoppable state. For free methods, their use of fields is restricted to a constant and stopped state so that they cannot interfere with behavioral methods. Finally, at each methods, we must also check that all arguments have been completely used. We illustrate this consistency checking with a simple example.

*Example 4.* An example of *consistency check* with recursion.

```

class C {
  usage a.(b+c) // behavior paths: a -> b
               //                               -> c
  N v;
  void a(){
    v = new N(); // a << [v: N#stop]
                // v : N#m1+m2
  } // a >> [v: N#m1+m2]
  void b(){
    if( ... ) // b << [v: N#m1+m2]
      ( v.m2(); // -> v: N#m1+m2 (no change)
        v = new N(); // v: N#stop
        this.b() ) // { v: N#m1+m2 } -(b)-> { v: N#stop }
    else v.m1() // v: N#stop
  } // b >> [v: N#m1+m2]
  void c(){
    this.b() // c << [v: N#m1+m2]
            // { v: N#m1+m2 } -(b)-> { v: N#stop }
  } // c >> [v: N#stop]
}

```

### 3.3 Discussion and Further Examples

So far, some of the presented typing rules may seem too restrictive as, for example, they require exact matches in the environments of different branches. For this reason, we will briefly discuss some improvements to the basic type rules, that offer additional flexibility. Essentially, we define more flexible ways of combining typing environments.

*Example 5.* The *environment subtyping* allows for an environment to be safely used as another if they have compatible content and any “extra” variables are stopped.

$$\{(x : N\#b + d), (w : N\#u[M : q]), (y : N\#\text{stop})\} <: \{(x : N\#d), (w : N\#u[M : q|N : w])\}$$

*Example 6.* The *environment intersection* merges two environments into one that contains the common behavior (and therefore, is valid in both of the initial environments) and any name that does not appear in both will be added to the final environment.

$$\{(x : N\#b + d), (y : N\#\text{stop})\} \sqcap \{(x : N\#q + d)\} = \{(x : N\#d), (y : N\#\text{stop})\}$$

These relations allow us to define how a double environment can be converted to a single one ( $(\Delta_T | \Delta_F) <: (\Delta_T \sqcap \Delta_F)$ ) and conversely that any environment is a double environment where both alternatives are itself ( $\Delta = (\Delta | \Delta)$ ). Therefore, the rules can adapt to cases where a double environment is not produced or when one is not required. By combining these operations with the typing rules we gain some additional flexibility. For instance, the R-IF may have different environments in its branches that are merged using environment intersection. Or the R-THROW rule does not need to produce exactly the same environment as the one in the exception handler, it just need to be an environment subtype of it. In conclusion, different environments can be merged using intersection and two environments are compatible if the subtype relation holds.

We leave the presentation of our formal proof of soundness of the type system, based on subject-reduction and type-safety proofs, to the companion technical report [17]. Before concluding the section, we present some examples of programs and their typings.

*Example 7.* Consistency check with behavioral exceptions.

```

class C {
  usage a.b[boolean: c].c // paths: a -> b -> c
  N v; // -> throw boolean -> c
  void a(){ // a << [v: N#stop]
    v = new N(); // v : N#(b.(b+c))+d.(d+c)
  } // a >> [v: N#(b.(b+c))+d.(d+c)]
  void b() throws boolean{ // b << [v: N#(b.(b+c))+d.(d+c)]
    if( ... ) //
      ( v.d(); // v: N#d+c
        throw true ) // throw boolean >> [v: N#d+c]
    else v.b() // v: N#b+c
  } // b >> [v: N#b+c] (by 'v: N#b+c' intrs '{empty}')
  void c(){ // c << [v: N#d+c] c << [v: N#b+c]
    v.c() // v : N#stop
  } // c >> [v: N#stop] c >> [v: N#stop]
}

```

*Example 8.* The **while** body makes a choice that will have behavioral repercussions in the following cycles. Thus, the initial environment cannot be used directly, it must first be subtyped into one that correctly models the loop invariant,  $(v : N\#\&r(b.r + stop))$ . Since we cannot statically know how many times it will loop (or if it will at all) using a call to “a” after that **while** is always considered to be illegal since the while’s body made a behavioral choice and thus changed the allowed behavior.

```

void method(N#a+&r(b.r+stop) v){ // -> v: N#a+&r(b.r+stop)
  while( ... ) // subtyping environment to
    v.b() // v: N#&r(b.r+stop)
} // <- v: N#&r(b.r+stop) (stoppable)

```

*Example 9.* The environment intersection can be used to merge two distinct branches (**if** and **else**) with different behaviors into an environment that contains the shared protocol.

```

void method(N#a+b+stop v){ // -> v: N#a+b+stop
  if( ... ) v.b() // v: N#stop
  else v.toString() // v: N#a+b+stop (toString is 'free')
} // <- intersection results in 'v: N#stop'

```

*Example 10.* Result based choice and exceptions.

```

void m( N#a.(true.b+false.c) v ){ // >> [ v : N#a.(true.b+false.c) ]
  try ( if( v.a() ) // v: N#b | v: N#c
        throw true // --> throw boolean + v: N#b
      ) // v: N#stop
  catch(boolean b) // v: N#stop ( 'v: N#stop' intrs '{empty}' )
    v.b() // v: N#stop
} // << [ v: N#stop ]

```

## 4 Related Work

The core idea behind our types is based on the spatial-behavioral type system of [6], where it is developed a type system for a resource aware model of behavior, using the  $\pi$ -calculus as the underlying model. This work is an attempt to adapt and expand some of his ideas to a mainstream Java-like language and was developed during Militão’s Masters thesis [16], that also lead to a publicly available prototype [22]. This line of research has its remote roots in Nierstrasz’s regular types for objects [18].

DeLine and Fähdrich [8,7,9] explore the idea of enforcing protocols in an object oriented language using pre/post conditions to check invariants, that can include a state-machine like protocol. This work also includes features similar to those of ESC/Java [10], by checking other kinds of properties. They includes rules for inheritance (which we do not handle) and subtyping based on a simplified version of behavioral subtyping as proposed by Liskov, et al [15]; even they still allows substitutability violations in some cases. Unlike in our work, they do not use linear typing to ensure sound state transitions, they do not tackle with exception handling, neither consider behavioral borrowing on references passed to method calls. Typestates [19] for objects have been further developed by Aldrich and Bierhoff [2,3,4]. where subtyping relation, by means of state refinement respecting substitutability is defined. Building on the notion of fractional permissions [5], they define *access permissions* [3] whose expressiveness goes beyond linear types. These allow for advanced aliasing control, for example, it is even capable of modeling some situations where it can statically verify the absence of concurrent modifications in the use of iterators [1]. Nonetheless, they do not consider behavioral borrowing as we do here.

A different approach [20] adapts session-types [12] to define dynamic interfaces, that controls access to object methods. This work differs from ours mainly in that they do not guarantee termination of behaviors, nor account for exceptions. The aliasing control is similar, as they also require linearity but without the option of borrowing.

In [13] Igarashi and Kobayashi create a type system for a call-by-value, simply-typed  $\lambda$ -calculus based language that guarantees usage correctness. In later work [14], they expand their proposal to include exceptions similar to our behavioral exceptions. However, they limit the raise construct to a single typeless exception at a time, thus it is not possible to jump to a specific catch branch based on the type of the thrown object, as we do, and is often needed in realistic programs.

## 5 Concluding Remarks

We have presented a behavioral type system for object oriented programs that statically verifies usage conformance of objects by enforcing behavioral fidelity and termination of protocols declared in class specifications, while extending other existing proposals with flexible aliasing control, and exception handling. We implemented a version of this type system into a prototype interpreter [22], and we have provided a formal proof of its correctness [17]. Our use of a consistency check phase reduces the burden on the programmer by automatically checking the correct use of object fields in accordance to the declared behavioral protocol, without the need for additional annotations. All constructions (exceptions, branches and loops) are checked in a flexible way, that does not require the usage protocol to be directly expressed in the client code. A subtyping relation guarantees that all behavioral expectations are met in accordance with the general substitutability principle. Although we use a linear ownership control with the notion of owned and non-owned types, we account for the possibility of borrowing types in well defined scopes and in a coherent way with normal call-by-reference. We hope in the future to be able to extend our verification techniques to programs with concurrency.

## References

1. Kevin Bierhoff. Iterator specification with tpestates. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 79–82, New York, NY, USA, 2006. ACM.
2. Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with tpestates. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 217–226. ACM, 2005.
3. Kevin Bierhoff and Jonathan Aldrich. Modular tpestate checking of aliased objects. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA*, pages 301–320. ACM, 2007.
4. Kevin Bierhoff and Jonathan Aldrich. Plural: checking protocol compliance under aliasing. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE Companion*, pages 971–972. ACM, 2008.
5. John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.
6. Luís Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theor. Comput. Sci.*, 402(2-3):120–141, 2008.
7. R. DeLine and M. Fähndrich. The fugue protocol checker: Is your software baroque, 2003.
8. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69, 2001.
9. Robert DeLine and Manuel Fähndrich. Tpestates for objects. In Martin Odersky, editor, *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.
10. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.
11. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL*, pages 171–183, 1998.
12. Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In S. Doaitse Swierstra, editor, *ESOP*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1999.
13. Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *POPL*, pages 331–342, 2002.
14. Futoshi Iwama, Atsushi Igarashi, and Naoki Kobayashi. Resource usage analysis for a functional language with exceptions. In John Hatcliff and Frank Tip, editors, *PEPM*, pages 38–47. ACM, 2006.
15. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
16. Filipe Militão. Design and implementation of a behaviorally typed programming system for web services. Master’s thesis, Universidade Nova de Lisboa, July 2008.
17. Filipe Militão and Luís Caires. An exception aware behavioral type system for object-oriented programs. Technical Report UNL-DI-3-2009, CITI / FCT-UNL, 2009.
18. Oscar Nierstrasz. Regular types for active objects. In *OOPSLA*, pages 1–15, 1993.
19. Robert E. Strom and Shaula Yemini. Tpestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.
20. Vasco T. Vasconcelos, Simon Gay, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Dynamic interfaces. In *International Workshop on Foundations of Object-Oriented Languages (FOOL'09)*, 2009.
21. Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North, 1990.
22. yak home page. <http://ctp.di.fct.unl.pt/yak/>.