A modular theory of object orientation in higher-order UTP

Frank Zeyda[1], Thiago Santos[2], Ana Cavalcanti[1], and Augusto Sampaio[3]
frank.zeyda@york.ac.uk thiago.lvl.santos@gmail.com

[1] University of York, Deramore Lane, York, YO10 5GH, UK
[2] Banco Central do Brasil, Rua da Aurora, 1259, Santo Amaro, Recife, PE, CEP 50040-090, Brazil
[3] Universidade Federal de Pernambuco, Centro de Informática, Caixa Postal 7851, Recife, PE, CEP 50732-970, Brazil

Abstract. Hoare and He's Unifying Theories of Programming (UTP) is a framework that facilitates the integration of relational theories. None of the UTP theories of object orientation, however, supports recursion, dynamic binding, and compositional method definitions all at the same time. In addition, most of them are defined for a fixed language and do not lend themselves easily for integration with other UTP theories. Here, we present a novel theory of object orientation in the UTP that supports all of the aforementioned features while permitting its integration with other UTP theories. Our new theory also provides novel insights into how higher-order programming can be used to reason about object-oriented programs in a compositional manner. We exemplify its use by creating an object-oriented variant of a refinement language for real-time systems.

Keywords: unification; semantics; models; integration; refinement

1 Introduction

The development of semantic theories is central to the creation of sound methods for program verification. While ongoing research has produced a mélange of specialised theories and calculi for a wide array of languages, a challenge one is currently faced with is unification: identification of commonalities in those languages and transfer of results between them. The Unifying Theories of Programming (UTP) [8] address this issue by providing a meta-theoretical framework that sustains a unified notion of computation as predicates over relevant observations. The UTP is not a programming language in itself; it rather defines a mathematical infrastructure in which arbitrary modelling and programming languages can be uniformly described and combined.

Semantic models for object-oriented languages have been an active area for research. A seminal work is Abadi and Cardelli's calculus of objects [1]. More recently, Hoare and He's Unifying Theories of Programming (UTP) [8] has been applied in this domain [7,11,13,16] too. The use of UTP is attractive as it fosters the integration of object-oriented theories with theories that address complementary paradigms. We have, for instance, UTP theories of process algebras [8,10], hardware description languages [3], and timed calculi [11,15]. The UTP has native support for refinement and thus by default supports refinement-based verification techniques based on algebraic laws and refinement strategies.

A primary motivation for the use of UTP is that the existing theories of object orientation are not adequate to model languages and technologies that also require models for orthogonal aspects such as reactive behaviours, real-time execution, or memory utilisation. Java, and UML and its variants, are examples of such languages; their complex models require unification of various features related to memory model, communication, synchronisation, time, and so on.

Whereas theory integration is a major concern, we also regard the following four features as essential: language independence, recursion, dynamic dispatch of calls, and compositional definitions. Language independence ensures that we can use arbitrary theories to define the model of method behaviour. Compositionality is crucial to formalise and reason about concepts in isolation, such as defining and overriding individual methods. It turns out that none of the existing UTP works [7,11,13,16] on object orientation can do justice to all four issues at once.

While higher-order programming (HOP) is used in some form in all of the existing UTP works, designing a fully compositional theory that includes mutual recursion, based on HOP in UTP, is particularly challenging. Mutually-recursive methods in this context can only be specified through concurrent assignments of all procedure variables for methods that take part in the recursion. Redefinition of individual methods participating in a (mutual) recursion is thus not possible, and this crucially destroys compositionality which requires, by definition, theory constructs for (re)defining individual methods. Nevertheless, HOP has proved itself very useful, even necessary, in theories of object orientation [14].

We note that handling recursion in non-UTP theories such as [2] can be more straightforward, but simplicity is usually gained by assuming a fixed syntax. Our agenda is different: we want to retain language independence and thus have to take an entirely semantic view of programs, as prescribed by the UTP.

The contribution of this paper is a novel UTP theory of object orientation that solves the four issues pointed out above and, at the same time, lends itself for integration with other theories of programming. For this, we extend and combine two existing works: our theory of object orientation in [13] and the theory of methods in [20]. The result is a comprehensive and modular theory of the object-oriented paradigm that is fully compositional in terms of declaring classes, attributes and methods, supports mutual recursion, dynamic binding, refinement, and makes no assumptions about the syntax and semantics of the base language in which we write methods. We also illustrate how our theory can be used to create new object-oriented languages, based on existing theories.

In Section 2, we review preliminary material. Section 3 details the problems in the existing UTP work(s) on object orientation. In Section 4, we extend the theory of methods in [20] to support parameters, and Section 5 presents our novel theory of object orientation. In Section 6, we exemplify its use by creating an object-oriented variant of a language for reactive real-time systems and, lastly, in Section 7, we conclude and discuss related and future work.

2 Preliminaries

In this section, we discuss the UTP and its higher-order extension. Programs and their specifications are characterised in the UTP by relations that determine the observable behaviours of a computation. Relations are encoded by alphabetised predicates: that is, predicates equipped with an alphabet of variables, obtained by the operator $\alpha(\_)$, that determines the observable quantities of interest.

As an example, we consider the predicate $D \mathrel{\widehat=} ok \wedge n > 0 \Rightarrow ok' \wedge n' = n - 1$ with alphabet $\{n, n', ok, ok'\}$. Whereas n of type $\mathbb{N}$ is a program variable, ok of type boolean is an auxiliary variable that captures termination. D encodes a computation that, if started (ok) in a state where $n > 0$, terminates (ok') while decrementing the value of n. Dashed variables are used to record immediate or final observations, and undashed variables, initial observations. Predicates that only refer to initial (undashed) variables are called conditions.

The construction used in the definition of D is called a design, here with precondition $n > 0$ and postcondition $n' = n - 1$. The UTP introduces a special notation $P \vdash Q =_{\mathrm{df}} ok \wedge P \Rightarrow ok' \wedge Q$ for designs with P and Q as pre and postcondition; thus D can be equally written as $n > 0 \vdash n' = n + 1$.

Signature Standard predicate calculus operators apply to alphabetised predicates too. Disjunction is used to model nondeterminism, and relational composition to model sequential execution. Further operators for designs are $\mathbf{II}$ (skip), which retains the values of all variables, and assignment $(\_ := \_)$. These operators implicitly define the alphabet of the result. For skip and assignment, it can also be explicitly given by a subscript, as in $\mathbf{II}_A$ and $x :=_A e$.

The UTP conditional $D_1 \lhd b \rhd D_2$, defined by $(b \wedge D_1) \vee (\neg\, b \wedge D_2)$, is written in infix form and corresponds to the more familiar **if** b **then** D_1 **else** D_2 construct. In a recursion $\mu X \bullet F(X)$, occurrences of X in F are recursive calls. The semantics of recursion is defined by weakest fixed points in the underlying refinement lattice. Refinement is universally defined by reverse implication: $P \sqsubseteq Q =_{\mathrm{df}} [P \Leftarrow Q]$, where $[\_]$ denotes universal closure. The top and bottom of the refinement lattice of a theory are denoted by $\top$ and $\bot$.

Local variables are the object of the $\mathbf{var}\, x : T$ and $\mathbf{end}\, x$ constructs. Whereas $\mathbf{var}\, x : T$ opens the scope of a new local variable x of type T, $\mathbf{end}\, x$ terminates it. Their definitions are $\exists\, x : T \bullet \mathbf{II}_A$ and $\exists\, x' : T \bullet \mathbf{II}_A$ for some alphabet A, where $\alpha(\mathbf{var}\, x) =_{\mathrm{df}} A \setminus \{x\}$ and $\alpha(\mathbf{end}\, x) =_{\mathrm{df}} A \setminus \{x'\}$. Both constructs are not binders, but sequentially composed with a predicate that may use x.

Healthiness conditions Typically, not all predicates over a given alphabet are considered valid models of computation. To delineate valid predicates, each UTP theory defines a set of healthiness conditions. These are idempotent and monotonic functions on predicates whose cumulative fixed points determine the predicates of the theory. For instance, $\mathbf{H1}(P) = ok \Rightarrow P$ in the theory of designs rules out predicates that constrain program variables before the program has started. While monotonicity and continuity of the healthiness conditions ensure that the predicates of a theory form a complete lattice, monotonicity of the operators guarantees well-definedness of recursions (weakest fixed points).

A signature and healthiness conditions together define a UTP theory: that is a set of predicates together with operators that define the semantics of language constructs. Unification and clarity is achieved by engineering the predicate model in such a way that common operators, such as nondeterminism, sequential composition, conditional statements, refinement, and so on, have similar definitions across theories. Theories are linked either by aggregating their healthiness conditions, or relating their predicate models using Galois connections [8].

The difficult task in constructing UTP theories is to elicit the denotational model and healthiness conditions. Once those are in place, we obtain many laws for free due to the uniformity of operators. Moreover, proofs that only depend on healthiness conditions naturally carry over to theory combinations. When conducting refinements, which encompasses both transforming specifications into software designs, and software designs further into executable code, all we need to care for are the algebraic laws. Simplicity is gained by discarding the semantic baggage at that point, being only a means to an end to prove the laws.

A theory of invariants Invariants are conditions that initially are assumed to hold, and are preserved by all terminating behaviours. The theory of invariants [4], in essence, ensures that violating an invariant is a situation from which we cannot recover, similar to nontermination. For this, the theory introduces a healthiness condition $\mathbf{SIH}(\psi) \mathrel{\widehat=} \mathbf{ISH}(\psi) \circ \mathbf{OSH}(\psi)$ for each state invariant ψ. The functions $\mathbf{ISH}(\psi)$ and $\mathbf{OSH}(\psi)$ are defined as follows.

$$\begin{aligned}\mathbf{ISH}(\psi)(D) &=_{\mathrm{df}} D \lor (ok \land \neg\, D[ok \setminus \mathit{false}] \land \psi \Rightarrow ok' \land D[ok \setminus \mathit{true}]) \ \text{ and}\\ \mathbf{OSH}(\psi)(D) &=_{\mathrm{df}} D \land (ok \land \neg\, D[ok \setminus \mathit{false}] \land \psi \Rightarrow \psi')\end{aligned}$$

where ψ is a condition. Intuitively, $\mathbf{ISH}(\psi)(D)$ strengthens the precondition of a design D for it to abort if we start in a state where the invariant ψ does not hold, and $\mathbf{OSH}(\psi)(D)$ strengthens its postcondition in order to ensure that the invariant is preserved. We note that the substitutions $\neg\, D[ok' \setminus \mathit{false}]$ and $D[ok' \setminus \mathit{true}]$ extract the original pre and postcondition of D [8].

It is possible to show that $\mathbf{SIH}(\psi)$-healthy designs can be written in the form $P \land \psi \vdash Q \land \psi'$. In [4], it is also shown that $\mathbf{SIH}(\psi)$ is idempotent and monotonic, and closed with respect to the relevant theory operators.

Higher Order HO UTP, in addition, includes procedure values and variables. For instance, $p := \{\!|\, \mathbf{val}\, x, y : \mathbb{N};\ \mathbf{res}\, z : \mathbb{N} \bullet y \neq 0 \vdash z := x \operatorname{div} y \,|\!\}$ assigns to p a procedure that takes two value parameters, x and y, of type $\mathbb{N}$, and one result parameter, z, also of type $\mathbb{N}$. A procedure p is called via $p(a_1, a_2, \dots)$ where the a_i are the arguments passed to the call. For instance, a call $p(6, 3, a)$ yields the design predicate $3 \neq 0 \vdash a := 6 \operatorname{div} 3$ (equivalent to $a := 2$).

In the definition of a procedure value, **val** is used to introduce a value parameter, **res** to introduce a result parameter, and **valres** for a value-result parameter. HO UTP gives a model to parametrised procedures through functional abstraction. For instance, p above is encoded by a function that takes two values of type $\mathbb{N}$ and one variable (name) of type $\mathbb{N}$. The alphabet of the predicate resulting from the call in the above example depends on the argument provided for z. For instance, $\alpha\, p(6,3,a) = \{a, a', ok, ok'\}$ whereas $\alpha\, p(6,3,b) = \{b, b', ok, ok'\}$.

In HO UTP, we also have a collection of laws to reason about higher-order predicates, that is, predicates whose alphabets contain procedure variables. A general law for a procedure call is the following. It is the manifestation of the copy-rule, enabling us to replace a procedure variable by its body in a call.

Law 1. $p := \{\!|Q|\!\}\ ;\ p(a) \equiv p := \{\!|Q|\!\}\ ;\ Q(a)$

An important restriction of HO UTP is that recursion is prohibited. For instance, we cannot define $p := \{\!|\, \mathbf{res}\, n : \mathbb{N} \bullet (n := n - 1\ ;\ p(n)) \lhd n > 0 \rhd \mathbf{I\!I} |\!\}$, with the intention $p(x) \equiv x := 0$, since the procedure variable p that is assigned occurs within the program value. This is to ensure that procedure types have finite constructions, and in [20] we have presented a formal proof that this is sufficient to ensure soundness of the HO UTP model in the context of using arbitrary UTP theories for the bodies of procedure values.

Procedure variables can be used in theories of object orientation to record methods [13]. This is a common approach that has the advantage of allowing us to capture declarative concepts at a high level of abstraction. We next illustrate, however, some essential challenges in adopting this approach.

3 The problem: syntax and compositionality

The challenges we address are presented here in the context of the UTP theory of object orientation in [13], but they equally arise in any other treatment that uses higher-order programming to encode method behaviours [7,11,16]. In [13], we first extend the theory of designs by introducing additional observational variables to capture class definitions and the subclass relation. The theory signature provides operations to declare classes, their attributes, and methods. For instance, $\mathbf{meth}\ C\ m \mathrel{\widehat=} (pds \bullet body)$ is used to define a new method m with body $body$ and parameters pds in a class C, provided C has already been declared.

Procedure variables are used to record the behaviours of methods. That is, for each new method, a procedure variable is introduced to record the program that corresponds to the body of the method. Crucially, the same variable is also used to record overridings of that method in subclasses. Multiple overridings result in a cascade of tests that determines, at call time, which method body has to be executed, testing against more concrete types first. In a class hierarchy where $C_1 \prec C_2 \prec C_3$ ($\prec$ means 'is extended by'), we may have the program

$$\{\!|\mathbf{valres\ self};\ pds \bullet (b_3 \lhd \mathbf{self}\ \mathit{is}\ C_3 \rhd (b_2 \lhd \mathbf{self}\ \mathit{is}\ C_2 \rhd (b_1 \lhd \mathbf{self}\ \mathit{is}\ C_1 \rhd \bot)))|\!\}$$

as part of the definition of a procedure variable that records a method that is first introduced in C_1 and later overridden in C_2 and C_3. The parameter **self** provides a reference to the object on which the method is called, and $obj\ \mathit{is}\ C$ is a test that determines whether an object obj is of a given class type C.

The above solves the problem of dynamic binding in a simple and elegant manner, but the approach also has apparent ramifications. First, since the cascade of tests has to be syntactically modified with each definition of an overriding method, we essentially require the value of m to be encoded as syntax rather than directly as a predicate. While [13] does not explore this in detail, there is either way no sound justification that permits us to encode the procedure m as a predicate; for instance, the seminal account [8] on the UTP requires it to be a program (syntax) whose meaning is determined by a subtheory of designs.

A second problem is that, due to the restrictions on the types of higher-order variables, we cannot define recursive methods in this way as this would result in procedure variables that refer to themselves in their alphabets. Consider, for instance, the following declaration of mutually-recursive methods m_1 and m_2.

$$\begin{array}{l} \mathbf{meth}\, C\, m_1 \mathrel{\widehat=} (\mathbf{res}\, n : \mathbb{N} \bullet (n := n - 1\ ;\ \ m_2(n)) \lhd n > 1 \rhd \mathbf{II})\ ; \\ \mathbf{meth}\, C\, m_2 \mathrel{\widehat=} (\mathbf{res}\, n : \mathbb{N} \bullet (n := n - 1\ ;\ \ m_1(n)) \lhd n > 1 \rhd \mathbf{II}) \end{array} \tag{1}$$

whose behaviour is to set the value of n to zero by a call to either $m_1(n)$ or $m_2(n)$. Such definitions are prohibited by the theory above because m_1 includes a variable m_2 in its alphabet which, in turn, includes the variable m_1. More precisely, the circular inclusion of variables in the procedure body alphabets creates a circular dependency in the types of m_1 and m_2, which is prohibited in HO UTP as we noted. A possible solution is to use a recursive predicate, but this forces us to define the methods in a single assignment $m_1, m_2 := \mu X, Y \bullet \langle F(X, Y), G(X, Y)\rangle$ for some F and G, and thus destroys compositionality of method definitions. For instance, it is subsequently not possible to redefine or override one of the above methods individually — any update to m_1 or m_2 has to be done 'in bulk' with the recursion calculated anew. The loss of compositionality thus prevents us from modular reasoning at the level of individual method overridings.

Despite the above problems, the use of procedure variables *per se* is a powerful tool to pave the way for modular instantiation of one UTP theory with another. The theory of methods in the next section tackles the identified problems.

4 A theory of parametrised methods

Our first theory is not a comprehensive theory of object orientation, but rather addresses the particular problem of using higher-order variables to record method behaviour. First, it establishes, via a constructive proof that a program model exists, a sound basis for using predicates of arbitrary UTP theories to specify procedure values. It is therefore possible to identify procedure values directly with the predicates of any designated UTP theory for the method bodies. We thereby eradicate any dependency on a fixed syntax and remain in the realm of semantic models, adhering to the philosophy and approach of the UTP.

To address the problem of compositionality, the theory of methods uses the notion of ranks. Intuitively, the rank determines the maximal nesting level of program abstractions in a predicate. For instance, the predicates of rank 0 are just the standard predicates; predicates of rank 1 include procedure variables whose values are standard predicates; predicates of rank 2 moreover admit values being rank 1 predicates, and so on. Thus, $x := 1$ is a rank 0 predicate, $m_1 := \{\!|x := 1|\!\}$ is a rank 1 predicate, and $m_2 := \{\!|x := 1\ ;\ \mathbf{call}\, m_1|\!\}$ is a rank 2 predicate. Formally, the rank of a variable depends on its type: basic types like $\mathbb{N}$, $\mathbb{B}$, $\mathbb{P}(\mathbb{N})$, and so on, have a rank 0, and for procedure types the rank is one more than the maximum rank of the variables in the procedure's alphabet. Predicate ranks are determined by the maximum rank of its alphabet variables.

For theories of object-oriented programming, as we explain next, we only need and admit rank 1 and rank 2 procedure variables. To emphasise the ranks of variables, we use a single overbar for rank 1 variables and a double overbar for rank 2 variables. Each method of an object-oriented program is now encoded by two variables rather than one, with the same name but at different ranks. Where methods are defined, we use rank 2 variables; where methods are called, we use rank 1 variables, regardless of using recursion. The use of different variables for defining and calling methods implies that call dependencies do not implicitly constrain the ranks of method variables anymore. Furthermore, it paves the way for a compositional treatment of recursive methods. Below, we recapture the example (1) at the end of Section 3 in the context of the theory of methods.

$$\begin{array}{l}\overline{\overline{m}}_1 := (\mathbf{res}\, n : \mathbb{N} \bullet (n := n - 1\ ;\ \overline{m}_2(n)) \lhd n > 1 \rhd \mathbf{II})\ ; \\ \overline{\overline{m}}_2 := (\mathbf{res}\, n : \mathbb{N} \bullet (n := n - 1\ ;\ \overline{m}_1(n)) \lhd n > 1 \rhd \mathbf{II})\end{array} \tag{2}$$

Unlike (1), the above is a valid higher-order predicate since there are no recursions in the types of $\overline{\overline{m}}_1$ and $\overline{\overline{m}}_2$ due to the procedure variables being at different ranks in the recursive calls (they are indeed different variables).

A single healthiness condition in the theory of methods establishes a connection between rank 1 and rank 2 procedure variables. As they are different variables, there exists *a priori* no formal relationship between them. The healthiness condition **MH** of the theory enforces a formal link: they have to be equivalent if we quantify over standard (non-procedure) variables.

$$\mathbf{MH}(P)\ =\ P \land (\forall\, \overline{m}\ \overline{\overline{m}} \mid \{\overline{m}, \overline{\overline{m}}\} \subseteq \alpha P \bullet [\mathbf{call}\,\overline{m} \Leftrightarrow \mathbf{call}\,\overline{\overline{m}}]_0)$$

The $[\_]_0$ is a restricted universal closure that only quantifies over standard (non higher-order) variables. The **call** construct abbreviates a method call without parameters: that is, **call** m is just the same as $m()$. We next generalise the theory of methods to cater for the use of parameters as employed in the example above.

Parametrised procedure types The type of a parameterless procedure in higher-order UTP is, in essence, equated with the alphabet of the body predicate. For parametrised procedures, this is insufficient because we also need to consider the number and types of parameters in order to distinguish procedures with different parametrisations by their types. A complication arises due to result parameters: here, the alphabet of the predicate obtained via a procedure call moreover depends on the variable(s) being passed as arguments to the call; hence, we cannot assume a fixed predetermined alphabet in that case.

To overcome these issues, we recast the notion of procedure type in [8] as specified in Fig. 1. We use two type constructors there: *BaseType* to construct the type of a standard value, and *ProcType* to construct a procedure type.

Procedure types are encoded by a pair consisting of a sequence of parameter types and an alphabet. Here, however, the alphabet only includes global variables

$$
\begin{array}{|l|}
\hline
<\!\mathit{type}\!> ::= <\!\mathit{base\ type}\!> \mid <\!\mathit{procedure\ type}\!> \\
<\!\mathit{procedure\ type}\!> ::= \mathit{ProcType}(\mathrm{seq}\,(<\!\mathit{parameter\ type}\!>), <\!\mathit{alphabet}\!>) \\
<\!\mathit{parameter\ type}\!> ::= \mathit{ValArg}(<\!\mathit{type}\!>) \mid \mathit{ResArg}(<\!\mathit{type}\!>) \\
<\!\mathit{alphabet}\!> ::= \mathbb{F}\,(<\!\mathit{variable}\!> : <\!\mathit{type}\!>) \\
<\!\mathit{base\ type}\!> ::= \mathit{BaseType}(\mathit{int}) \mid \mathit{BaseType}(\mathit{bool}) \mid \ldots \\
\hline
\end{array}
$$

Fig. 1. Recast notion of procedure types that supports parameters.

of the procedure predicate. Alphabets are encoded by finite sets ($\mathbb{F}$) of pairs $v : T$ that define a name v and a type T. Parameters can be either value parameters (*ValArg*) or result parameters (*ResArg*). Both constructors take the type of the parameter. By way of an example, $\{\!|\, \mathbf{val}\, x : \mathbb{N};\ \mathbf{res}\, y : \mathbb{N} \bullet y := x + z \,|\!\}$ is of type $\mathit{ProcType}(\langle \mathit{ValArg}(\mathit{NatType}), \mathit{ResArg}(\mathit{NatType})\rangle, \{z : \mathit{NatType}\})$ where *NatType* abbreviates $\mathit{BaseType}(\mathit{nat})$.

Procedure ranks revisited To justify the sound use of parametrised procedures, we require a notion of rank for the new model of procedure types outlined above. Following the same approach as in [20], we can then perform an inductive construction of a program model, which is sufficient to establish the consistency of the morphisms $\{\!|\ldots|\!\}$ and $p(\mathit{args})$ for the construction and destruction of parametrised procedure values. The rank is defined inductively as follows.

$$
\begin{aligned}
&\mathit{rank}(\mathit{BaseType}(t)) = 0 \quad \text{and} \\
&\mathit{rank}(\mathit{ProcType}(\langle v_1 : t_1, v_2 : t_2, \ldots\rangle, \{w_1 : \tilde{t}_1, w_2 : \tilde{t}_2, \ldots\})) = \\
&\qquad \mathit{max}\,\{\mathit{rank}(t_1), \mathit{rank}(t_2), \ldots, \mathit{rank}(\tilde{t}_1), \mathit{rank}(\tilde{t}_2), \ldots\} + 1
\end{aligned}
$$

As before, the rank of basic types is zero. For procedure types, it is one more than the maximum of the ranks of the types of global variables used in the procedure predicate and the types of parameters. We note that our notion of type and rank entail procedures being passed as arguments, although the theory of methods does not require this. The soundness of permitting it is an added contribution of our work; it may be useful in other uses of higher-order UTP.

To establish consistency of the procedure model, we inductively construct a program model for predicates up to a given rank n, denoted by $\mathit{Pred}(n)$. $\mathit{Pred}(0)$ yields the standard predicates, which trivially have a model. Rank 1 predicates are obtained by extending rank 0 predicates with additional predicates whose alphabets include procedure variables whose bodies and arguments can range over rank 0 predicates and values. In each step, the set of constructible predicates monotonically increases, that is $\mathit{Pred}(0) \subset \mathit{Pred}(1) \subset \ldots$.

A complete model *Pred* of procedure values of any rank is obtained by taking the limit of this chain: $\mathit{Pred} =_{\mathrm{df}} \bigcup\{n : \mathbb{N} \bullet \mathit{Pred}(n)\}$. Parametrised procedures are then introduced as a new type that is isomorphic to *Pred*. A mechanisation in Isabelle/HOL is available [19] that soundly introduces (parameterless) procedures up to rank 2. We recall that, for the theory of methods, rank 2 predicates are sufficient. The generality of the result may be useful elsewhere, though.

The soundness of the higher-order program model is indeed the primary concern in generalising the theory of methods. Procedure values and calls are, as in [8], modelled by functions and their application. Finally, we need to recast the healthiness condition **MH** to cater for parametrised method variables.

$$\mathbf{MH}(P) \;=\; P \land (\forall\, \overline{m}\ \overline{\overline{m}} \mid \{\overline{m}, \overline{\overline{m}}\} \subseteq \alpha P \bullet [\forall\, args \bullet \overline{m}(args) \Leftrightarrow \overline{\overline{m}}(args)]_0)$$

The quantification $\forall\, args \bullet \ldots$ ranges over well-formed argument lists only, namely those whose arguments are of the correct length and type.

Having generalised the theory of methods to deal with parameters, we next combine it with the theory of object orientation in [13] to overcome the issue in the latter (Section 3) with dependency on syntax and compositionality.

5 A modular theory of object orientation

Our integrated theory is an extension of the theory of designs, and, therefore, includes the auxiliary boolean variables ok and ok' to record termination. Besides, it also includes additional auxiliary variables to capture specific aspects of object-oriented programs. These are listed below.

- cls of type $\mathbb{P}(CName)$ to record the names of classes used in the program;
- $atts$ of type $CName \nrightarrow (AName \nrightarrow Type)$ to record the class attributes;
- sc of type $CName \nrightarrow CName$ to record the subclass hierarchy;
- an open set $\{\overline{\overline{m}}_1, \overline{\overline{m}}_2, \ldots\}$ of procedure variables for method definitions; and
- an open set $\{\overline{m}_1, \overline{m}_2, \ldots\}$ of procedure variables for method calls.

Above, $CName$ is the set of all class names, $AName$ the set of all attribute names, and $Type$ is defined as $CName \cup prim$ where the elements in $prim$ represent primitive types, like integers or booleans. The functions $atts$ and sc are partial ($\nrightarrow$) since they only consider classes that are currently declared, namely those in cls. The function sc maps each class to its immediate superclass; the subclass relation is obtained via its reflexive and transitive closure: $C_{sub} \preceq C_{super} =_{\mathrm{df}} (C_{sub}, C_{super}) \in sc^*$. There also exists a special class **Object** $\in CName$ that does not have a superclass.

Healthiness conditions The theory has seven healthiness conditions. They are characterised by invariants that constrain the permissible values of cls, $atts$ and sc, as well as the procedure variables for methods. Table 1 summarises the first six constraints, which are related to cls, $atts$ and sc. Whereas the table specifies the invariants themselves, the corresponding healthiness conditions are obtained by application of $\mathbf{SIH}(\ldots)$, as explained in Section 2.

The invariant **OO1** requires **Object** always to be a valid class of the program. **OO2** and **OO3** determine the shape of the subclass relation: it has to be a tree with **Object** at its root. Attributes have to de defined for all classes (**OO4**), they have to be unique (**OO5**), and their types, if they are not primitive, must refer to classes that have already been declared (**OO6**).

	Invariant ψ for $\mathbf{SIH}(\psi)$	Description
OO1	$\mathbf{Object} \in \mathit{cls}$	**Object** is always a class of the program.
OO2	$\operatorname{dom} \mathit{sc} = \mathit{cls} \setminus \mathbf{Object}$	Every class except **Object** has a superclass.
OO3	$\forall\, C : \operatorname{dom} \mathit{sc} \bullet (C, \mathit{Object}) \in \mathit{sc}^+$	**Object** is at the top of the class hierarchy.
OO4	$\operatorname{dom} \mathit{atts} = \mathit{cls}$	Attributes are defined for all classes.
OO5	$\forall\, C_1, C_2 : \operatorname{dom} \mathit{atts} \mid C_1 \neq C_2 \bullet \operatorname{dom}(\mathit{atts}(C_1)) \cap \operatorname{dom}(\mathit{atts}(C_2)) = \varnothing$	Attribute names are unique across classes.
OO6	$\operatorname{ran}(\bigcup \operatorname{ran} \mathit{atts}) \subseteq \mathit{prim} \cup \mathit{cls}$	Attributes have primitive or class types.

Table 1. Healthiness conditions for the theory of object orientation.

A further healthiness condition (**OO7**) not in Table 1 corresponds to **MH** in the theory of parametrised methods. We recast it in terms of an invariant too.

$$\mathbf{OO7}(P) = \mathbf{SIH}(\forall\, \overline{m}\ \overline{\overline{m}} \mid \{\overline{m}, \overline{\overline{m}}\} \subseteq \alpha P \bullet [\forall\, \mathit{args} \bullet \overline{m}(\mathit{args}) \Leftrightarrow \overline{\overline{m}}(\mathit{args})]_0)(P)$$

Theory predicates hence have to maintain the fundamental correspondence between rank 1 and rank 2 method variables. Finally, we let **OO** denote the composition of all healthiness conditions: $\mathbf{OO} =_{\mathrm{df}} \mathbf{OO1} \circ \mathbf{OO2} \circ \ldots \circ \mathbf{OO7}$.

Operations We provide operations to declare classes, attributes and methods in a compositional manner. We use $\mathbf{class}\ C\ \mathbf{extends}\ B$ to declare a new class C that extends a class B, $\mathbf{att}\ C\ x : T$ to declare a new attribute x of type T in a class C, and $\mathbf{meth}\ C\ m \mathrel{\widehat{=}} (\mathit{pds} \bullet \mathit{body})$ to define or override a method m in a class C. To declare more than one class, attribute or method, we sequence multiple applications of the aforementioned constructs. We focus here on the definition of methods and refer to [12] for a complete account of our theory.

To define and override methods, we recast the respective constructs in [13] in the context of the theory of methods. Below, pds are the arguments of the method and body is the program for the method body.

$$\begin{array}{l}
\mathbf{meth}\ C\ m \mathrel{\widehat{=}} (\mathit{pds} \bullet \mathit{body}) =_{\mathrm{df}} \\
\mathbf{let}\ \mathit{mp} \mathrel{\widehat{=}} \{\!\!\{\, \mathbf{valres}\ \mathbf{self};\ \mathit{pds} \bullet (\mathit{body} \lhd \mathbf{self}\ \mathit{is}\ C \rhd \bot_{oo})\,\}\!\!\} \bullet \\
\quad \mathbf{OO}\left(\begin{array}{l} \mathbf{var}\ \overline{\overline{m}}\,; \\ \left(\begin{array}{l} C \in \mathit{cls} \land \\ \forall\, t \in \mathit{types}(\mathit{pds}) \bullet t \in \mathit{prim} \cup \mathit{cls} \end{array}\right) \vdash \left(\mathit{mp} \sqsubseteq \overline{\overline{m}}' \land w = w'\right) \end{array}\right) \\
\mathbf{where}\ w = \mathit{in}\,\alpha(\mathbf{meth}\ C\ m = (\mathit{pds} \bullet \mathit{body})) \\
\mathbf{provided}\ \{\overline{m}, \overline{m}', \overline{\overline{m}}\} \cap \alpha(\mathbf{meth}\ C\ m = (\mathit{pds} \bullet \mathit{body})) = \{\overline{m}, \overline{m}'\}
\end{array}$$

A new procedure variable $\overline{\overline{m}}$ is introduced to record the method. That variable must not already be in the input alphabet of the predicate, although we assume that the corresponding rank 1 variable $\overline{m}$ is included in it. The operation changes the value of the rank 2 variable only, which holds the method definition. The link to the respective rank 1 variable is established via application of $\mathbf{OO}(\_)$.

The operation is specified by a design whose precondition requires that the class C in which the method is defined has been declared, and that the types of method parameters are either primitive or declared classes. The postcondition states that the new value of $\overline{\overline{m}}$ refines[4] mp while leaving other variables unchanged. The procedure mp first includes an additional implicit parameter **self** for self reference. It then wraps the method body into a conditional that tests if the target object (**self**) is of the correct type. If so, the *body* program is executed. Otherwise, we execute $\bot_{oo}$, which corresponds to program failure in the theory of object orientation and arises if an undefined method is called.

We observe that, above, *body* can in fact be any predicate. Our earlier discussion of soundness of the theory of methods in Section 4 relaxes the caveat in the earlier work that it has to be syntax. Secondly, this definition is compositional since the higher-order type of $\overline{\overline{m}}$ can be *a priori* determined: while it needs to include all rank 1 variables for methods, the types of those variables are fixed and not affected by the definition of further methods. Finally, recursion at the level of method definitions is possible since $\overline{m}$ may itself be included in the alphabet of $\overline{\overline{m}}$ without giving rise to issues related to recursions in procedure types — we recall that $\overline{m}$ and $\overline{\overline{m}}$ are different variables.

For overriding a method in a subclass, $\mathbf{meth}\ C\ m \mathrel{\widehat=} (pds \bullet body)$ has a different definition. As hinted in Section 3, we do not introduce a new variable in that case, but instead alter the procedure that $\overline{\overline{m}}$ records. That is, for every overriding of m in a subclass D, we inject an additional test $body \lhd \mathbf{self}\ is\ D \rhd \ldots$ at a suitable position into the conditional in mp. This is a syntactic transformation that requires part of the procedure value to be encoded in syntax. The mix of syntax and semantics turns out not to be an issue though, and neither does it compromise soundness and language independence. To formalise the combination of the two, we adopt the approach in [20] by first defining a generic datatype *METHOD* for the syntactic fragment into which the method bodies are embedded.

$$\begin{aligned}&\mathit{METHOD}[\mathit{PRED}] ::= \\ &\mathit{CondSytx}\ \langle\!\langle \mathit{METHOD} \times \mathit{CVAL} \times \mathit{METHOD} \rangle\!\rangle \mid \mathit{BotSytx} \mid \mathit{Body}\ \langle\!\langle \mathit{PRED} \rangle\!\rangle\end{aligned}$$

The type parameter *PRED* is instantiated with the predicate model of the embedded theory for method behaviour; in this way, we retain language independence. To give a semantics to the syntactic fragment, we inductively define a denotational function $[\![\_]\!]$ that maps elements of $\mathit{METHOD}[\mathit{PRED}]$ (syntax) to predicates in *PRED*. For instance, $\mathit{CondSytx}(M_1, c, M_2)$ elements are translated into conditionals $[\![M_1]\!] \lhd c\ is\ \mathbf{self} \rhd [\![M_2]\!]$, and *BotSytx* into $\bot_{oo}$. For $\mathit{Body}\ P$, we just have P. We hence require the operators $P \lhd b \rhd Q$ and $\bot_{oo}$ to be defined in the respective theory for method behaviour. Thanks to their uniform characterisations in UTP, we can always introduce them if they are not already available. The inductive definition of $[\![\_]\!]$ can easily be shown to terminate. Soundness of the altered procedure model is established using a similar proof as in [20].

We omit further aspects of our integrated theory for reasons of space. The report [12] provides a comprehensive account and, in particular, additionally

[4] Using refinement here instead of equality ensures monotonicity of the construct.

addresses issues of definedness, the encoding and creation of objects, and support for references in our theory. We conclude by observing that we now can encode (2), as it was our initial motivation and goal.

$$\begin{array}{l}
\mathbf{meth}\, C\; m_1 \;\widehat{=}\; ((\mathbf{res}\, n : \mathbb{N} \bullet n := n - 1\,;\; \overline{m}_2(n)) \lhd n > 1 \rhd \mathbf{II})\,; \\
\mathbf{meth}\, C\; m_2 \;\widehat{=}\; ((\mathbf{res}\, n : \mathbb{N} \bullet n := n - 1\,;\; \overline{m}_1(n)) \lhd n > 1 \rhd \mathbf{II}) \\
\mathbf{where}\ \alpha(\mathbf{meth}\, C\; m_1 \;\widehat{=}\; \ldots\,) =_{\mathrm{df}} \{\overline{m}_1, \overline{m}_1', \overline{m}_2, \overline{m}_2', \overline{\overline{m}}_1, \overline{\overline{m}}_1', \overline{\overline{m}}_2, \overline{\overline{m}}_2'\} \text{ and} \\
\alpha(\mathbf{meth}\, C\; m_2 \;\widehat{=}\; \ldots\,) =_{\mathrm{df}} \{\overline{m}_1, \overline{m}_1', \overline{m}_2, \overline{m}_2', \overline{\overline{m}}_1, \overline{\overline{m}}_1', \overline{\overline{m}}_2, \overline{\overline{m}}_2'\}
\end{array}$$

Whereas the first method declaration constrains $\overline{\overline{m}}_1$, the second one constrains $\overline{\overline{m}}_2$. The procedures refer to each other via $\overline{m}_1$ and $\overline{m}_2$, and at the point where $\overline{m}_2$ is first called, $\overline{\overline{m}}_2$ does not have to be declared yet. Whereas above, the methods were simple imperative programs, in the next section we investigate the case where methods have more elaborate semantic models.

6 Example: an object-oriented real-time language

The ability to instantiate the method model is a feature of our theories that segregates it from other theories of object orientation. This requires the inclusion of additional healthiness conditions that constrain procedure variables to record predicates of particular theories, rather than admit any kind of predicate. If the theory to be used to describe method behaviour has a set $\mathcal{H}$ of healthiness conditions, we proceed as follows. For each function $\mathbf{H} \in \mathcal{H}$, we define a pair $\hat{\mathbf{H}}_1$ and $\hat{\mathbf{H}}_2$ that embed $\mathbf{H}$ into the theory of object orientation.

$$\begin{array}{l}
\hat{\mathbf{H}}_1(P) \;=_{\mathrm{df}}\; \mathbf{SIH}\left(\forall\, \overline{m} \in \alpha P \bullet \psi_{\mathbf{H}}(\overline{m})\right)(P) \\
\hat{\mathbf{H}}_2(P) \;=_{\mathrm{df}}\; \mathbf{SIH}\left((\forall\, \overline{m} \in \alpha P \bullet \psi_{\mathbf{H}}(\overline{m})) \Rightarrow (\forall\, \overline{\overline{m}} \in \alpha P \bullet \psi_{\mathbf{H}}(\overline{\overline{m}}))\right)(P) \\
\mathbf{where}\ \psi_{\mathbf{H}}(m) \;\widehat{=}\; (\forall\, \mathit{args} \bullet \mathbf{H}(m(\mathit{args})) = m(\mathit{args}))
\end{array}$$

The embedded healthiness conditions are again invariants, here constraining procedure variables. In particular, $\hat{\mathbf{H}}_1$ forces all programs recorded by procedure variables at rank 1 in the alphabet of P to be fixed points of $\mathbf{H}$. $\hat{\mathbf{H}}_2$ does the same for procedure variables at rank 2, albeit assuming that the property already holds for rank 1 variables, since the procedures recorded in rank 2 predicates typically use rank 1 variables, namely when they call other methods.

The function $\psi_{\mathbf{H}}(m)$ abbreviates the property that a method m, if called on a valid argument list *args*, yields a predicate that is a fixed point of $\mathbf{H}$. By introducing $\hat{\mathbf{H}}_1$ and $\hat{\mathbf{H}}_2$ for all healthiness conditions $\mathbf{H}$ of $\mathcal{H}$, we obtain that the programs recorded by procedure variables are fixed points of all healthiness conditions in $\mathcal{H}$ and so valid predicates of the embedded theory.

To illustrate an embedding of a theory, we consider *Circus* Time [18], a theory of reactive processes that supports communication events, state operations, and real-time. The auxiliary variables of the theory include tr and tr' to record time traces of interactions. They are of type $\mathrm{seq}^+(\mathrm{seq}\, \mathit{Event} \times \mathbb{P}\, \mathit{Event})$ so that each trace element consists of a pair whose first component is a sequence of events in

Healthiness condition
$\mathbf{R1}(A) =_{\mathrm{df}} A \wedge tr \leq tr'$
$\mathbf{R2}(A) =_{\mathrm{df}} A[\langle(\langle\rangle, last(tr).2)\rangle, tr' - tr) \,/\, tr, tr']$ where $tr \leq tr'$
$\mathbf{R3}(A) =_{\mathrm{df}} \mathbf{II} \lhd wait \rhd A$ where $\mathbf{II} =_{\mathrm{df}} (\neg\, ok \wedge tr \leq tr') \vee (ok' \wedge \mathbf{II}_{\{wait, tr, state\}})$

Table 2. Healthiness conditions of *Circus* Time.

a time slot, and whose second component is the set of events refused at the end of the slot. The variables ok and ok' of boolean type record the observation that the predecessor or current process has not diverged. Termination is captured here by the boolean variables $wait$ and $wait'$. Specifically, $wait$ records that the predecessor has terminated, and $wait'$ records termination of the current process.

Healthiness conditions are listed in Table 2. The first healthiness condition $\mathbf{R1}(A)$ establishes that a process action A cannot alter the previous history of interactions. The second one $\mathbf{R2}(A)$ enforces insensitivity of A to interactions that took place before it started. And the third one $\mathbf{R3}(A)$ masks out any behaviours of A until its predecessor action has terminated ($wait$ is true). The operators '$\leq$' and '$-$' are special prefix and sequence subtraction operators on timed traces, whose definition can be found in [18].

The three healthiness conditions in Table 2 give rise to six healthiness conditions in the integrated theory. Hence, in addition to **OO1** to **OO7**, we have, for instance, the following pair of healthiness conditions for **R1**:

$$\hat{\mathbf{R1}}_1(A) =_{\mathrm{df}} \mathbf{SIH}\left(\forall\, \overline{m} \in \alpha P \bullet \psi_{\mathbf{R1}}(\overline{m})\right)(A)$$
$$\hat{\mathbf{R1}}_2(A) =_{\mathrm{df}} \mathbf{SIH}\left((\forall\, \overline{m} \in \alpha P \bullet \psi_{\mathbf{R1}}(\overline{m})) \Rightarrow (\forall\, \overline{\overline{m}} \in \alpha P \bullet \psi_{\mathbf{R1}}(\overline{\overline{m}}))\right)(A)$$

The lifted version of the remaining healthiness conditions are analogous.

Inside our new theory, we can encode, for instance, actions such as

$$\begin{array}{l}\mathbf{var}\; o : C;\; r : T \bullet c := \mathbf{new}\; C()\,; \\ \quad (in\,?\,x \longrightarrow o.calc(x, r)) \blacktriangleleft 5\,;\; \mathbf{wait}\; 0 \mathinner{..} 10\,;\; out\,!\,r \longrightarrow \mathbf{Skip}\end{array}$$

Above, in and out are communication channels, and o is a local object, initialised with a **new** instance of a class type C. We first wait for a communication on a channel in that inputs a value x. The synchronisation deadline $(\ldots \blacktriangleleft 5)$ specifies that a communication on in with the environment must take place within 5 time units. Subsequently, the method $calc(\ldots)$ is called on o, and a nondeterministic **wait** models a time budget of 10 time units for the call.

While x is a value parameter of $calc$, we assume the result of the call is deposited in a result parameter r; we, lastly, output r through a communication on the channel out. Whereas the interaction and time operators above are provided by the embedded theory (*Circus* Time), the method call $o.m(x, r)$ is translated into a procedure call $\overline{\overline{m}}(o, x, r)$ in the host theory, so that the target object becomes an additional argument of the procedure call.

The above mix of reactive, timed and object-oriented operators is to a certain extent already possible in TCOZ [11], however, our combined theory here inherits the generality in supporting *calc* to be defined recursively, redefined and overridden. To reason about programs such as the above, we use the laws of the embedded theory (*Circus* Time), alongside new special laws in the theory of methods, for instance, to reason about recursive methods.

The example shows that it is in essence very easy to integrate an existing UTP theory with our theory of object orientation. Certain operators, however, might have to be defined in the embedded theory, namely to construct the cascades of tests to resolve dynamic binding in method definitions and overloading.

The language we defined in this section is interesting in its own right, as it is a step towards resolving the dichotomy between active process behaviour and passive class objects which is present in many of the current works that combine object orientation with reactive theories. While those works typically provide good coverage of object-oriented concepts for class objects, they have little to no support to deal with the same features in terms of processes. The theory we have defined promises to enable progress in this area.

7 Conclusion

We have presented a novel theory of object orientation that segregates itself from other works by facilitating the integration with theories that address complementary aspects. In particular, we are free to define and instantiate the semantic model for method behaviour. This was achieved by extending and combining two existing unifying theories: one that addresses object orientation and another one that uses a novel approach to encode methods as higher-order programs.

Our theory is compositional in the presence of recursive method definitions, and enables us to reason about declarative concepts at a fine level of granularity. For instance, we can formulate a law that sequenced definitions of mutually-recursive methods commute, or that individual recursive methods can be overridden by a refinement of the method in a subclass.

We note that our theory here has also been integrated with the UTP theory of pointers in [6] to support object references and data sharing. A detailed discussion of this integration can be found in [12]; here, we decided to omit those details as it is not a central part of the particular problem we solve. It appears, moreover, that we can perform this integration by instantiating our theory of object orientation, namely with a theory of method behaviour that already includes a treatment of pointers; this makes pointers an orthogonal aspect.

The practical relevance of our theory is illustrated by two notable application examples. Firstly, Safety-Critical Java (SCJ) [17] is a recent technology that has been proposed to enable the verification and certification of Java programs; it requires a highly-integrated theory that includes object orientation, a specialised execution and memory model [4], and time. Secondly, SysML [5] is an extension of UML 2.0 that adds support for system-level specification; its semantics likewise involves the combination of a theory of object orientation with other theories [9], here VDM and CSP. We are currently looking at both these languages in order to define semantic models.

An open problem is refinement strategies that take advantage of the combinations of laws that arise from integrating our theory with others. While the UTP model we present can already be used to prove general properties of object-oriented designs such as the soundness of refactorings, a repository of novel laws for the verification of concrete applications is expected to emerge, too.

Related work Most of the existing UTP-related works on object orientation give a semantics for a fixed language. Smith's work [16] defines a semantics for Abadi and Cardelli's theory of objects [1]; He et al. [7] a model for rCOS, a language for refinement of object systems; and Qin et al. [11] a semantics for TCOZ, an integration of Object Z and Timed CSP. Our earlier work in [13] does not introduce a fixed language, but, as explained in Section 3, it lacks a justification that its combination with arbitrary theories for method models does not raise unsoundness issues in its use of higher-order UTP.

We next examine in more detail to what extent the existing UTP works address the issue of dynamic dispatch, recursion and compositionality.

Dynamic dispatch In Smith's work [16], dynamic dispatch emerges naturally as it is a theory of an object-based language (that is, [1]), rather than a theory of object orientation. Whereas rCOS [11] gives a comprehensive semantic account of the issue, TCOZ [7] leaves an explanatory gap here by only defining the denotation of fresh and overridden methods, but not, in detail, how method calls are resolved based on dynamic type information.

Recursion Only Smith's work [16] and our earlier theory [13] fully support recursion. In rCOS [7], recursion fails due to the denotational function that maps rCOS programs to their UTP models not terminating for recursive methods, and TCOZ [11] excludes recursive methods altogether from its class operations, since recursion is not part of the language of Object Z on which TCOZ is based.

Compositionality In [20], we first pointed out fundamental issues that prevent *any* theory that uses higher-order UTP to encode methods in a naïve way from being fully compositional. These issues indeed apply to [7,11,16]; they are not elicited in those works though as HOP is only used in an informal manner. As explained in Section 3, our earlier work [13] suffers from these problems too.

Future work Future work consists of two strands: first we require a comprehensive set of laws to reason about object-oriented constructs in our theory and the paradigm in general. Some laws have already been defined and proved in [12], but, in particular, we require additional laws to reason about method definition and overriding in the presence of recursion, exploiting **OO7** in Section 5.

Secondly, the integration of languages has to be examined in more detail, especially in terms of proof strategies. Finally, we have also started to mechanise our theory in a theorem prover: Isabelle/HOL. So far, our mechanisation provides a provably sound model for higher-order predicates up to rank 2, and a generic encoding of parametrised procedures. We are currently completing this work.

Acknowledgements We would like to thank the anonymous reviewers for their useful suggestions. This work was funded by the EPSRC grant EP/H017461/1.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, New York, Berlin, Heidelberg, January 1996.
2. M. Abadi and R. Leino. A Logic of Object-Oriented Programs. In *Proceedings of TAPSOFT '97*, volume 1214 of *LNCS*, pages 682–696. Springer, April 1997.
3. A. Butterfield, A. Sherif, and J. Woodcock. Slotted-*Circus*. In *Integrated Formal Methods (IFM 2007)*, volume 4591 of *LNCS*, pages 75–97. Springer, July 2007.
4. A. Cavalcanti, A. Wellings, and J. Woodcock. The Safety-Critical Java memory model formalised. *Formal Aspects of Computing*, 25(1):37–57, January 2013.
5. Object Management Group. OMG Systems Modeling Language (OMG SysML^{TM}). Technical Report Version 1.3, OMG, June 2012.
6. W. Harwood, A. Cavalcanti, and J. Woodcock. A Theory of Pointers for the UTP. In *Proc. of ICTAC 2008*, volume 5160 of *LNCS*, pages 141–155. Springer, 2008.
7. Jifeng He, Xiaoshan Li, and Zhiming Liu. rCOS: A refinement calculus for object systems. *Theoretical Computer Science*, 365(1-2):109–142, November 2006.
8. C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall, Upper Saddle River, NJ, USA, 1998.
9. A. Miyazawa, L. Lima, and A. Cavalcanti. SysML Blocks in CML. Technical Report COMPASS White Paper WP02, Seventh Framework Programme: Comprehensive Modelling for Advanced Systems of Systems (Grant 287829), April 2013.
10. M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for *Circus*. *Formal Aspects of Computing*, 21(1-2):3–32, February 2009.
11. S. Qin, J. S. Dong, and C. Wei-Ngan. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 321–340. Springer, September 2003.
12. T. Santos. A Unifying Theory of Object-Orientation. Technical Report (Qualifying Dissertation), Federal University of Pernambuco, Centre of Informatics, Brazil, 2007. http://www.cin.ufpe.br/~acas/pub/TheoryObjectOrientation.pdf.
13. T. Santos, A. Cavalcanti, and A. Sampaio. Object-Orientation in the UTP. In *Proceedings of UTP 2006*, volume 4010 of *LNCS*, pages 18–37. Springer, 2006.
14. S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular Verification of Higher-Order Methods with Mandatory Calls Specified by Model Programs. *ACM SIGPLAN Notices*, 42(10):351–368, October 2007.
15. A. Sherif, A. Cavalcanti, H. Jifeng, and A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *FAC-J*, 22:153–191, July 2009.
16. M. A. Smith and J. Gibbons. Unifying Theories of Objects. In *Integrated Formal Methods (IFM 2007)*, volume 4591 of *LNCS*, pages 599–618. Springer, July 2007.
17. The Open Group. Safety Critical Java Technology Specification. Technical Report JSR-302, Java Community Process, January 2011.
18. J. Woodcock. CML definition 4. Technical Report COMPASS Deliverable 23.5, FP7 Grant 287829, 2013. Available at http://www.compass-research.eu.
19. F. Zeyda. Mechanising Higher-Order UTP in Isabelle/HOL. Technical report, University of York, York, YO10 4DL, UK, November 2013. Available at http://www.cs.york.ac.uk/circus/publications/techreports/index.html.
20. F. Zeyda and A. Cavalcanti. Higher-Order UTP for a Theory of Methods. In *Proceedings of UTP 2012*, volume 7681 of *LNCS*, pages 204–223, August 2012.