Typechecking UML Static Models

Tony Clark

Department of Computing, University of Bradford, West Yorkshire, UK, BD7 1DP
a.n.clark@scm.brad.ac.uk

Abstract. UML static models are expressed using a mixture of class diagrams and OCL expressions. In a well formed static model, the OCL expressions and class diagrams are type consistent. Checking for type consistency of static models involves both inclusion and parametric polymorphism. This paper defines a semantics of type consistency in terms of a type theory for UML static models. The type theory is shown to be correct with respect to a value semantics for OCL. The existence of a consistency checking algorithm for UML static models is established.

1 Introduction

The Unified Modeling Language (UML) is a general-purpose visual modelling language that is designed to specify, visualise and construct the artifacts of an object-oriented system [14]. UML supports the OO development process by providing constructs for describing a proposed software system as a collection of data and behavioural models.

UML defines system states using *static models* that freely describe configurations of objects in terms of classes and associations. Each class declares state variables and methods. Associations between classes declare the potential for message communication between instances.

Expressions in an *object constraint language* (OCL) [15] restrict the freely constructed system states. OCL is used to express class invariants and method specifications (see [8] and [9] for more information on the use of conditions in OO designs). A class invariant is a condition that must hold for any instance of the class. Methods are specified using OCL pre- and post-conditions. In all cases OCL expressions are conditions on the values of object state variables, method calls and associations.

It is possible for the static models and the OCL expressions to be type inconsistent. Type inconsistencies result in references to a non-existent object fields and to the application of operators such as + to operands of incorrect types.

The aim of this paper is to define a set of rules for checking the type consistency of UML static models. To achieve this we define a value semantics for OCL expressions with respect to a UML class diagram. A type theory for OCL expressions is defined and shown to be sound and complete with respect to the value semantics. The type theory is used to define type consistency for UML static models.

2 UML Static Models

A UML static model denotes a collection of legal system states and method calls. Each method call is a pair of system states (a pre-state and a post-state), a method, a list of arguments and a return value. The model is expressed using UML class diagrams and OCL expressions. Both class diagrams and OCL expressions denote collections of system states and method calls; the legal system states are the intersection of the two collections. This section defines the components of a static model.

2.1 Requirements and Class Diagram

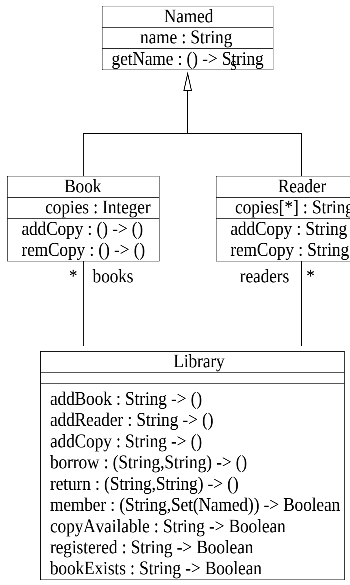

Fig. 1. A Static Model for a Library System

Software to control a library is required. The library contains a number of copies of books and has a number of registered readers. Two different books cannot have the same name. Two different readers cannot have the same name. Readers borrow and return copies from the library. The system must record the number of copies currently shelved and on loan. The user may query whether a copy is available and whether a reader is registered.

Figure 1 shows a UML class diagram for this system. Since both readers and books have names, the class `Named` is defined as a shared super-class of `Reader` and Book. An instance of Book contains an integer copies being the number of copies currently shelved. An instance of Reader contains a set of strings copies being the names of books currently borrowed (assuming a limit of one copy per book).

The declaration of copies[*]:String in Reader corresponds to the OCL type Set(String). The operator Set is a *type constructor* whose argument defines the type of set elements. Library is the source of two associations, readers and books; the cardinality constraint * indicates that a single library is associated with many instances of Reader and Book respectively.

A meaning for a class diagram such as that shown in figure 1 is a freely constructed collection of object states. Class diagrams alone are not sufficiently expressive; they cannot express exactly the required set of object states. For example, the class diagram shown in figure 1 permits more than one reader with the same name.

OCL is used to restrict the possible states in terms of *invariants* and *method specifications*. An invariant is a condition that must hold at all times for each instance of a class, referred to as self. For example:

```
Library:
  self.readers->forAll(r1 |
    self.readers->forAll(r2 |
      not(r1 = r2) implies
        not(r1.getName() = r2.getName())))
```

which requires the names of readers registered in a library to be different (assuming strings are equal when their corresponding characters are equal).

A method specification is a pair of conditions, called *pre* and *post*, such that if the pre-condition holds before the method is invoked then the post-condition holds after the method has completed. Specifications are given in the form *pre* implies *post*, for example:

```
Book::addCopy() true implies copies' = copies + 1;
Book::remCopy() true implies copies' = copies - 1
```

which requires that the addCopy and remCopy methods of Book respectively increase and decrease the value of copies when they are invoked.

The formal parameters of a method may be referenced in both the pre- and post-conditions. The change of state in an object is represented using the Z technique of adding ' to state variables in the post-condition. If a method has a return value then it is referenced in a post-condition as reply.

2.2 Collection Types

A number of programming languages have advocated the use of parametric classes including Eiffel [12] and recently Java extensions [1] and [16]. OCL provides a number of builtin collection types including Collection and Set. Each collection class may be supplied with a single parameter which is the type of

```
Library::addBook(book)
  not(self.bookExists(book)) implies
    self'.books->forAll(b | self.books.includes(b) or book = b.name) and
    self.books->forAll(b | self'.books.includes(b))
Library::member(name,named)
  named->exists(n | n.getName() = name) implies reply
Library::copyAvailable(name)
  self.books->exists(b | b.name = name and b.copies > 0)  implies reply
Library::bookExists(name)
  self.member(name,self.books) implies reply
Library::registered(name)
  self.member(name,self.readers) implies reply
```

Fig. 2. Specification of class `Library`

the elements in the collection. The collection types form a hierarchy and provide collection access and modification methods. The collection types are not the same as classes defined by the user. This is reflected in the special purpose OCL syntax which is used to access the collection methods.

We have taken the view that it is useful to treat collection types as *parameterised classes* that are defined at the same level as user classes in a design. Parameterised types have proved to be very useful in other areas of program development [11] [10]. The specification of the class `Reader` provides an example of the use of collection methods `including` and `excluding`:

```
Reader::addCopy(name)
  true implies self'.copies = self.copies.including(name)
Reader::remCopy(name)
  true implies self'.copies = self.copies.excluding(name)
```

Figure 2 specifies methods of class `Library`. The associations `books` and `readers` are of type `Set(Book)` and `Set(Reader)` respectively. In general, associations have multiplicity constraints. Associations with multiplicity other than one-to-one correspond to one or more set-valued attributes. Invariants are used to express multiplicity constraints other than `1` and `*`. For example if there is an upper limit of `10` library books per reader then the following invariant can be used: `Reader: self.copies.size <= 10`

The method `member` is used by both `bookExists` and `registered` and provides an example of type conformance (or *inclusion polymorphism* see [3]). The type of `self.books` is `Set(Book)` and the type of `self.readers` is `Set(Reader)`. The domain type of the method `member` is `Collection(Named)`. OCL permits a method to be supplied with a value whose type is different to the method domain providing the argument type conforms to the domain type. In this case since `Set` conforms to `Collection` and both `Reader` and `Book` conform to `Named` the use of `member` is type correct in `bookExists` and `registered`.

$$
\begin{array}{l}
[\![e_1 \rightarrow \texttt{forAll}(v_1 \mid e_2)]\!] = \\
\quad [\![e_1]\!] \rightarrow \texttt{iterate}(v_1 v_2 = \texttt{true} \mid \texttt{if } [\![e_2]\!] \texttt{ then } v_2 \texttt{ else false}) \\
[\![e_1 \rightarrow \texttt{exists}(v_1 \mid e_2)]\!] = \\
\quad [\![e_1]\!] \rightarrow \texttt{iterate}(v_1 v_2 = \texttt{false} \mid \texttt{if } [\![e_2]\!] \texttt{ then true else } v_2) \\
[\![e_1 \rightarrow \texttt{select}(v_1 \mid e_2)]\!] = \\
\quad [\![e_1]\!] \rightarrow \texttt{iterate}(v_1 v_2 = [\![e_1]\!] \mid \texttt{if } [\![e_2]\!] \texttt{ then } v_2 \texttt{ else } v_2\texttt{.excluding}(v_1)) \\
[\![e_1 \rightarrow \texttt{reject}(v_1 \mid e_2)]\!] = \\
\quad [\![e_1]\!] \rightarrow \texttt{iterate}(v_1 v_2 = [\![e_1]\!] \mid \texttt{if } [\![e_2]\!] \texttt{ then } v_2\texttt{.excluding}(v_1) \texttt{ else } v_2) \\
[\![e_1 \rightarrow \texttt{collect}(v_1 \mid e_2)]\!] = \\
\quad [\![e_1]\!] \rightarrow \texttt{iterate}(v_1 v_2 = \texttt{Bag}\{\} \mid v_2\texttt{.including}([\![e_2]\!]))
\end{array}
$$

Fig. 3. Translation of Iteration Expressions

2.3 Iteration Expressions

OCL provides *iteration expressions* to manipulate collections. An iteration expression selects elements from a collection one at a time and performs an operation. The result of the iteration is defined in terms of the results of all the component operations. For example, let `c` be a collection, `v` be a variable and e be a condition then `c->forAll(v|e)` is `true` when `e` is `true` for all the elements of `c`.

OCL provides the following types of iteration expression: `forAll` for universal quantification; `exists` for existential quantification; `select` and `reject` for filtering and `collect` for transformation. All types of iteration expression can be expressed using a basic form: `c->iterate(v1 v2 = i | e)` where `c` is a collection, `v1` is a variable bound to each element in the collection in turn, `i` is a value, `e` is any OCL expression and `v2` is a variable set by the iteration. Before evaluation `v2` is set to `i`. After each iteration, `v2` is set to the value of `e`. The result of the iteration expression is the final value of `v2`.

Figure 3 defines a semantics preserving translation $[\![.]\!]$ from *general* OCL iteration expressions to *basic* OCL iteration expressions. In each case the new variable v_2 is not free in e_2. The total translation is the unique extension to a homomorphism over OCL syntax. Consider the specification of `member` in figure 2. After translation, the result is:

```
Library::member(name,named)
  named->iterate(n v = false |
    if n.getName() = name
    then true
    else v) implies reply
```

3 Syntax and Semantics of Static Models

A class diagram denotes a collection of system states and an OCL expression is a condition on states. The condition can be viewed as an expression denoting a value with respect to a system state that contains values for all the free variables in the expression. This section defines a value semantics for OCL expressions.

A *system state* is a collection of object states. An *object state* (α, τ, ϕ) consists of a unique object identifier α, a type τ and a partial function ϕ from variables to values and functions over values. The function ϕ is referred to as a *value context* and can be extended with an association between variable v and value x to produce $\phi[v \mapsto x]$.

A value, $x, y, \ldots$, may be atomic (integer, character, boolean, *etc.*), a collection of values of the same type or an object state. A function maps indexed collections of values to a single value. An object state is *well-formed* when all occurrences of the same object identifier label equivalent object states and when occurrences of types conform to the type signatures given in the UML class diagram. Sets, bags and sequences are represented as objects with appropriate states. For notational convenience a set object will be written $\{x_1, \ldots, x_n\}$ and a bag object as $<x_1, \ldots, x_n>$ (sequences are handled the same way and will not be included).

An OCL expression denotes a boolean value (or undefined). Component expressions may denote values. We conflate these two categories for the sake of simplicity. An OCL expression e is defined:

$$
\begin{array}{rl}
e ::= & v \mid k \mid v(e,\ldots,e) \mid e.v \mid e.v(e,\ldots,e) \mid \texttt{Set}\{e,\ldots,e\} \mid \\
& e \rightarrow \texttt{iterate}(v\ v = e \mid e) \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \\
& e \rightarrow \texttt{forAll}(v \mid e) \mid e \rightarrow \texttt{exists}(v \mid e) \mid e \rightarrow \texttt{select}(v \mid e) \mid \\
& e \rightarrow \texttt{reject}(v \mid e) \mid e \rightarrow \texttt{collect}(v \mid e)
\end{array}
$$

where v is a variable and k is an atomic constant. Prefix and infix operators are not differentiated from unqualified method calls.

Attributes and methods of collections are accessed the same way as attributes and methods of objects, for example `c.size` and `c1.includesAll(c2)`. Methods of collections that bind identifiers *always* include exactly one identifier: multiple identifiers are provided by *currying* the expression, for example: `e1->forAll(i1,i2|e2)` is curried as `e1->forAll(i1|e1->forAll(i2|e2))`.

Let e be an OCL expression, ϕ be a value context and x be a value then x is a value of e with respect to ϕ when $\phi \vdash e \Rightarrow x$ as defined in figure 4. The semantics of class invariants is defined with respect to a value context containing a binding for the variable `self` its state variables and its associations. For a method specification the value context contains additional bindings for `self'` *etc.*

Theorem 1. *Let ϕ be a value context, e be an OCL expression then there exists at most one value x such that $\phi \vdash e \Rightarrow x$.*

The proof is by induction on the structure of e with respect to the rules in figure 4. In each case we assume by induction at most one value for each of the subexpressions of e; when defined, the uniqueness of x follows from its definition in each rule. $\Box$

$$\phi \vdash v \Rightarrow \phi(v) \tag{1}$$

$$\phi \vdash k \Rightarrow k \tag{2}$$

$$\frac{\phi \vdash e_i \Rightarrow x_i \;\; i = 1, \ldots, n}{\phi \vdash v(e_1, \ldots, e_n) \Rightarrow \phi(v)(\tilde{x})} \tag{3}$$

$$\frac{\begin{array}{c}\phi \vdash e \Rightarrow (\alpha, \tau, \phi') \\ v \in \mathit{dom}(\phi')\end{array}}{\phi \vdash e.v \Rightarrow \phi'(v)} \tag{4}$$

$$\frac{\begin{array}{c}\phi \vdash e_1 \Rightarrow \texttt{true} \\ \phi \vdash e_2 \Rightarrow x\end{array}}{\phi \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow x} \tag{5}$$

$$\frac{\begin{array}{c}\phi \vdash e_1 \Rightarrow \texttt{false} \\ \phi \vdash e_3 \Rightarrow x\end{array}}{\phi \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow x} \tag{6}$$

$$\frac{\begin{array}{c}\phi \vdash e \Rightarrow (\alpha, \tau, \phi') \\ \phi \vdash e_i \Rightarrow x_i \;\; i = 1, \ldots, n \\ v \in \mathit{dom}(\phi')\end{array}}{\phi \vdash e.v(e_1, \ldots, e_n) \Rightarrow \phi'(v)(\tilde{x})} \tag{7}$$

$$\frac{\phi \vdash e_i \Rightarrow x_i \;\; i = 1, \ldots, n}{\phi \vdash \mathit{Set}\{\tilde{e}\} \Rightarrow \{\tilde{x}\}} \tag{8}$$

$$\frac{\begin{array}{c}\phi \vdash e_1 \rightarrow S \\ \phi[v_1 \mapsto x_i] \vdash e_2 \Rightarrow y_i \text{ for each } x_i \in S\end{array}}{\phi \vdash e_1 \rightarrow \texttt{forAll}(v_1 \mid e_2) \Rightarrow \bigwedge_{i=1,n} y_i} \tag{9}$$

$$\frac{\begin{array}{c}\phi \vdash e_1 \rightarrow S \\ \phi[v_1 \mapsto x_i] \vdash e_2 \Rightarrow y_i \text{ for each } x_i \in S\end{array}}{\phi \vdash e_1 \rightarrow \texttt{exists}(v_1 \mid e_2) \Rightarrow \bigvee_{i=1,n} y_i} \tag{10}$$

$$\frac{\begin{array}{c}\phi \vdash e_1 \Rightarrow S \\ \phi[v_1 \rightarrow x] \vdash e_2 \Rightarrow \texttt{true} \text{ for each } x \in S_1 \\ \phi[v_1 \rightarrow x] \vdash e_2 \Rightarrow \texttt{false} \text{ for each } x \in S_2 \\ S_1 \cup S_2 = S\end{array}}{\phi \vdash e_1 \rightarrow \texttt{select}(v_1 \mid e_2) \Rightarrow S_1} \tag{11}$$

$$\frac{\begin{array}{c}\phi \vdash e_1 \Rightarrow S \\ \phi[v_1 \rightarrow x] \vdash e_2 \Rightarrow \texttt{true} \text{ for each } x \in S_1 \\ \phi[v_1 \rightarrow x] \vdash e_2 \Rightarrow \texttt{false} \text{ for each } x \in S_2 \\ S_1 \cup S_2 = S\end{array}}{\phi \vdash e_1 \rightarrow \texttt{reject}(v_1 \mid e_2) \Rightarrow S_2} \tag{12}$$

$$\frac{\begin{array}{c}\phi \vdash e_1 \Rightarrow S \\ \phi[v_1 \mapsto x_i] \vdash e_2 \Rightarrow y_i \text{ for each } x_i \in S\end{array}}{\phi \vdash e_1 \rightarrow \texttt{collect}(v_1 \mid e_2) \Rightarrow \langle y_1, \ldots, y_n \rangle} \tag{13}$$

$$\frac{\begin{array}{c}\phi \vdash e_1 \Rightarrow \{x_1, \ldots, x_n\} \\ \phi \vdash e_2 \Rightarrow x \\ \phi[v_1 \mapsto x_i, v_2 \mapsto y_i] \vdash e_3 \Rightarrow y_{i+1} \;\; i = 1, \ldots, n,\; y_1 = x\end{array}}{\phi \vdash e_1 \rightarrow \texttt{iterate}(v_1\ v_2 = e_2 \mid e_3) \Rightarrow y_{n+1}} \tag{14}$$

Fig. 4. OCL Semantics

Theorem 2. *Figure 3 defines a full and faithful transformation for iteration expressions. For any OCL expression e and any value context ϕ, if $\phi \vdash e \Rightarrow x$ then $\phi \vdash [\![e]\!] \Rightarrow x$ and if $\phi \vdash [\![e]\!] \Rightarrow x$ then $\phi \vdash e \Rightarrow x$.*

The proof is by induction on the structure of the expression e. For example, consider a `forAll` expression $e = e_1 \rightarrow \texttt{forAll}(v_1 \mid e_2)$. In order to show that the transformation is *full* there are two cases to consider. Let $\phi \vdash e_1 \Rightarrow S$ for some set S and assume by induction that $\phi \vdash [\![e_1]\!] \Rightarrow S$. When $\phi \vdash e \Rightarrow \texttt{true}$ then by definition $\phi[v_1 \mapsto x] \vdash e_2 \Rightarrow \texttt{true}$ for all $x \in S$. Assuming by induction $\phi[v_1 \mapsto x] \vdash [\![e_2]\!] \Rightarrow \texttt{true}$ then by definition $\phi[v_1 \mapsto x, v_2 \mapsto \texttt{true}] \vdash \texttt{if}\ [\![e_2]\!]\ \texttt{then}\ v_2\ \texttt{else}\ \texttt{false} \Rightarrow \texttt{true}$. Therefore, by definition $\phi \vdash [\![e_1]\!] \rightarrow \texttt{iterate}(v_1\ v_2 = \texttt{true} \mid e_2) \Rightarrow \texttt{true}$ as required. When $\phi \vdash e \Rightarrow \texttt{false}$ then by definition $\phi[v_1 \mapsto x] \vdash e_2 \Rightarrow \texttt{false}$ for some $x \in S$. Assume by induction that $\phi[v_1 \mapsto x] \vdash [\![e_2]\!] \Rightarrow \texttt{false}$ and therefore $\phi[v_1 \mapsto x, v_2 \mapsto y] \vdash [\![e_2]\!] \Rightarrow \texttt{false}$ for any y. By definition, $\phi[v_1 \mapsto x, v_2 \mapsto y] \vdash \texttt{if}\ [\![e_2]\!]\ \texttt{then}\ v_2\ \texttt{else}\ \texttt{false} \Rightarrow \texttt{false}$. Therefore, by definition $\phi \vdash [\![e_1]\!] \rightarrow \texttt{iterate}(v_1\ v_2 = \texttt{true} \mid e_2) \Rightarrow \texttt{false}$ as required. All other cases in the proof have the same structure. □

4 Type Checking

OCL expressions and class diagrams must be shown to be type consistent. To do this we define a relationship between class diagrams interpreted as types and OCL expressions [2]. An expression is well-typed with respect to a class diagram when it is possible to determine for any system state consistent with the class diagram whether or not the OCL expression holds.

This section defines how class diagrams are interpreted as *types*, and uses a static model type theory to define a well-typed relation between OCL expressions and types.

4.1 Static Models

An OCL type τ is defined $\tau ::= b \mid c(\tau, \ldots, \tau) \mid (\tau, \ldots, \tau) \rightarrow \tau$ where b is an atomic type such as `Boolean` or `Integer`, c is a class constructor and $\rightarrow$ constructs function types. A function type may only occur as part of a class signature.

Each well formed OCL value x has a unique type τ written $x : \tau$. If x is a constant k then $k : \mathit{typeOf}(k)$. If x is a set $\{\tilde{x}\}$ where each of the elements has the same type τ then $\{\tilde{x}\} : \mathit{Set}(\tau)$. If x is a function f such that every elements of the domain of f is of type $\tilde{\tau}$ and the range type if τ then $f : \tilde{\tau} \rightarrow \tau$. If x is an object (α, τ, ϕ) such that $\mathit{dom}(\mathit{sig}(\tau)) = \mathit{dom}(\phi)$ and for all $v \in \mathit{dom}(\phi)$, $\phi(v) : \mathit{sig}(\tau)(v)$ then $x : \tau$. If ϕ is a value context and $\mathcal{A}$ is a signature then $\phi : \mathcal{A}$ when $\mathit{dom}(\phi) = \mathit{dom}(\mathcal{A})$ and for all $v \in \mathit{dom}(\phi)$, $\phi(v) : \mathcal{A}(v)$.

A type scheme ρ is defined $\rho ::= t \mid b \mid c(\rho, \ldots, \rho) \mid (\rho, \ldots, \rho) \rightarrow \rho$ where t is a type variable. Type schemes denote a set of types constructed by substituting types for variables in the type scheme. Type scheme ρ_1 is an instance of type scheme ρ_2 when ρ_1 is the result of substituting type schemes for variables in ρ_2.

A class context $\mathcal{C}$ is a set of classes. In the following definitions the class context is assumed to be known. Each class is named $c(\tilde{t})$ where t are type parameters. Let $c(\tilde{\rho})$ represent the class constructed by substituting each type scheme ρ_i for the corresponding t_i in the definition of the class. Each class has a single super-class $\mathit{super}(c(\tilde{\rho})) = c'(\tilde{\rho}')$ such that the type variables in $\tilde{\rho}'$ are at most those contained in $\tilde{\rho}$. Each class has a signature $\mathit{sig}(c(\tilde{\rho}))$ that is a partial function from variables to type schemes. Signatures associating variables with types are referred to as *type contexts*. The operator $\_'$ is applied to a signature and appends the character ' to each name in the signature. Each class has a set of method names $\mathit{meth}(c(\tilde{\rho}))$. Each method m of a class has a set of specifications $\mathit{specs}(c(\tilde{\rho}))(m)$. Each specification is $(\tilde{v}, e_1, e_2)$ where $\tilde{v}$ are formal parameters, e_1 is the pre-condition and e_2 is the post-condition.

An invariant is $(c(), e)$ where $c()$ is a non-parametric class and e is the condition. A static model is a pair $(\mathcal{C}, \mathcal{I})$ where $\mathcal{I}$ is a set of invariants.

4.2 Type Conformance

Type checking ensures that no illegal field reference can be made and that all values supplied to basic operations are contained within the operation's domain. Since inheritance cannot remove fields, it is sufficient to require that the actual type of an OCL expression is a sub-type, or conforms to, the expected type.

$\mathcal{C} \vdash b_1 \leq b_2$	when basic types b_1 and b_2 conform.
$\mathcal{C} \vdash c(\tilde{t}) \leq c(\tilde{t})$	when $\mathcal{C}$ contains a class $c(\tilde{t})$.
$\mathcal{C} \vdash c_1(\tilde{t}) \leq c_2(\tilde{\rho})$	when $\mathcal{C}$ contains a class $c_1(\tilde{t})$ and $\mathit{super}(c_1(\tilde{t})) = c_2(\tilde{\rho})$.
$\mathcal{C} \vdash c_1(\tilde{\rho}_1) \leq c_2(\tilde{\rho}_2)$	when $\mathcal{C} \vdash c_1(\tilde{\rho}'_1) \leq c_2(\tilde{\rho}'_2)$ and $(\tilde{\rho}_1, \tilde{\rho}_2)$ an instance of $(\tilde{\rho}'_1, \tilde{\rho}'_2)$.
$\mathcal{C} \vdash c_1(\tilde{\rho}_1) \leq c_2(\tilde{\rho}_3)$	when $\mathcal{C} \vdash c_1(\tilde{\rho}_1) \leq c_2(\tilde{\rho}_2)$ and $\mathcal{C} \vdash c_2(\tilde{\rho}_2) \leq c_3(\tilde{\rho}_3)$.
$\mathcal{C} \vdash \tilde{\tau} \to \tau \leq \tilde{\tau}' \to \tau'$	when for each i, $\mathcal{C} \vdash \tau'_i \leq \tau_i$ and $\mathcal{C} \vdash \tau \leq \tau'$.
$\mathcal{C} \vdash \mathcal{A} \leq \mathcal{A}'$	when $\mathcal{A}$ and $\mathcal{A}'$ are signatures such that $\mathit{dom}(\mathcal{A}') \subseteq \mathit{dom}(\mathcal{A})$ and for each $v \in \mathit{dom}(\mathcal{A}')$, $\mathcal{C} \vdash \mathcal{A}(v) \leq \mathcal{A}'(v)$

Fig. 5. Definition of Type Conformance

UML uses *name equivalence* for types as opposed to *structural equivalence*. A class $c_1(\tilde{\rho}_1)$ conforms to another class $c_2(\tilde{\rho}_2)$ with respect to a given set of classes $\mathcal{C}$ when c_1 inherits from c_2 in $\mathcal{C}$ and the actual parameters conform and are consistent with $\mathcal{C}$. Type scheme ρ_1 conforms to ρ_2 when $\mathcal{C} \vdash \rho_1 \leq \rho_2$ which is the smallest relation defined by the rules given in figure 5.

Consider the example static model defined in section 2. The type conformance rules give rise to the following: `Reader()` $\leq$ `Named()`; `Set(Reader())` $\leq$ `Set(Named())`; `Set(Reader())` $\leq$ `Collection(Named())`.

The following theorem establishes the uniqueness of an upper bound on a class:

Theorem 3. *Let $c_1(\tilde{\rho}_1)$ be a class scheme and let c_2 be a class constructor. For all type schemes $\tilde{\rho}_2$ such that $\mathcal{C} \vdash c_1(\tilde{\rho}_1) \leq c_2(\tilde{\rho}_2)$ there exists a unique type scheme $\tilde{\rho}_3$ such that $\mathcal{C} \vdash c_2(\tilde{\rho}_3) \leq c_2(\tilde{\rho}_2)$ and $\mathcal{C} \vdash c_1(\tilde{\rho}_1) \leq c_2(\tilde{\rho}_3)$*

The proof follows since $\leq$ is a partial order. Suppose that $\tilde{\rho}$ and $\tilde{\rho}'$ are two non-equal candidates then $\mathcal{C} \vdash c_2(\tilde{\rho}) \leq c_2(\tilde{\rho}')$ and $\mathcal{C} \vdash c_2(\tilde{\rho}') \leq c_2(\tilde{\rho})$. By the definition of type conformance this can only occur if $\tilde{\rho}$ is an instance of $\tilde{\rho}'$ and vice versa, leading to a contradiction $\tilde{\rho} = \tilde{\rho}'$ as required. $\square$

Theorem 4. *There is an algorithm el which, given a type τ returns the most specific element type $el(\tau) = \tau_e$ such that $\mathcal{C} \vdash \tau \leq \texttt{Collection}(\tau_e)$.*

Let τ be a type such that $\mathcal{C} \vdash \tau \leq \texttt{Collection}(\tau_e)$ for some type τ_e. By definition there must be some unique class $c(\tilde{\tau}) = \tau$ such that after a finite collection of unfoldings $super(\ldots super(c(\tilde{\tau}))) = \texttt{Collection}(\tau_e)$. $\square$

Theorem 5. *Given a class context $\mathcal{C}$ and a pair of types τ_1 and τ_2 there exists an algorithm conf such that $conf(\tau_1, \tau_2)$ returns true when τ_1 conforms to τ_2 and false otherwise.*

Consider the form of the types τ_1 and τ_2. If either is an atomic type then *conf* can immediately report success or failure. If both types are instances of the same class then we assume by induction the existence of an algorithm for checking the component types and therefore *conf* reports success when all corresponding component types succeed. Otherwise, the types must be instances of different classes, in which case *conf* reports success when $conf(super(\tau_1), \tau_2)$ succeeds. Since each class has only finitely many super-classes, *conf* will terminate. $\square$

4.3 Type Theory

Let a type context $\mathcal{A}$ be a partial function from variables to types. Given a class context $\mathcal{C}$, an OCL expression e, a type context $\mathcal{A}$ whose domain is the set of free variables of e and a type τ then the relation $\mathcal{C}, \mathcal{A} \vdash e : \tau$ states that τ is a type for the values denoted by e in the given context.

The typing relation is defined as a theory in figure 6. The rules are described in outline as follows. The types of free variables are given by the current context (rule 15). The types of constants are pre-defined (rule 16). Predefined operators and methods of the current object must be supplied with arguments whose types conform to the domain of the operator (rule 17). The type of an attribute is determined with respect to the signature of an object type (rule 18). If a method call is qualified with an object (rule 19) then the type of the method is determined with respect to the signature of the object type; the actual argument types must conform to the formal argument types. The type of the test expression in a conditional (rule 21) must be `Boolean`; notice that we require that both consequent and alternative expressions have the same type. The component expressions of a set (rule 22) must all be the same type.

An iteration expression (rule 20) expects the type of e_1 to conform to a collection type with τ_e as the type of the elements in the collection. As it stands

$$\mathcal{C}, \mathcal{A} \vdash v : \mathcal{A}(v) \quad (15)$$

$$\mathcal{C}, \mathcal{A} \vdash k : \mathit{typeOf}(k) \quad (16)$$

$$\frac{\mathcal{C}, \mathcal{A} \vdash e_i : \tau_i' \quad \mathcal{A}(v) = \tilde{\tau} \to \tau \quad \mathcal{C} \vdash \tau_i' \leq \tau_i}{\mathcal{C}, \mathcal{A} \vdash v(\tilde{e}) : \tau} \quad (17)$$

$$\frac{\mathcal{C}, \mathcal{A} \vdash e : \tau \quad \mathit{sig}(\tau)(v) = \tau'}{\mathcal{C}, \mathcal{A} \vdash e.v : \tau'} \quad (18)$$

$$\frac{\mathcal{C}, \mathcal{A} \vdash e_i : \tau_i' \quad \mathcal{C}, \mathcal{A} \vdash e : \tau \quad \mathit{sig}(\tau)(v) = \tilde{\tau} \to \tau' \quad \mathcal{C} \vdash \tau_i' \leq \tau_i}{\mathcal{C}, \mathcal{A} \vdash e.v(\tilde{e}) : \tau'} \quad (19)$$

$$\frac{\mathcal{C}, \mathcal{A} \vdash e_1 : \tau_1 \quad \mathcal{C}, \mathcal{A} \vdash e_2 : \tau_2 \quad \mathcal{C} \vdash \tau_1 \leq \mathit{Collection}(\tau_e) \quad \mathcal{C}, \mathcal{A}[v_1 \mapsto \tau_e, v_2 \mapsto \tau_2] \vdash e_3 : \tau_2}{\mathcal{C}, \mathcal{A} \vdash e \to \mathtt{iterate}(v_1 v_2 = e_2 \mid e_3) : \tau_2} \quad (20)$$

$$\frac{\mathcal{C}, \mathcal{A} \vdash e_1 : \mathit{Boolean} \quad \mathcal{C}, \mathcal{A} \vdash e_2 : \tau \quad \mathcal{C}, \mathcal{A} \vdash e_3 : \tau}{\mathcal{C}, \mathcal{A} \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau} \quad (21)$$

$$\frac{\mathcal{C}, \mathcal{A} \vdash e_i : \tau}{\mathcal{C}, \mathcal{A} \vdash \mathit{Set}\{\tilde{e}\} : \mathit{Set}(\tau)} \quad (22)$$

Fig. 6. Type Theory for OCL

rule 20 is ambiguous since there may be many different types τ_e that satisfy the requirement; however, by theorem 3 we know that if τ_1 is a sub-class of *Collection* then there is a unique lower bound on the element type τ_e.

Theorem 6. *Let $\mathcal{C}$ be a class context, $\mathcal{A}$ be a type context and e be an OCL expression. There exists at most one type τ such that $\mathcal{C}, \mathcal{A} \vdash e : \tau$ providing that e contains no expressions denoting unrestricted empty collections.*

The proof is by induction on the structure of the OCL expression e with respect to the rules defined in figure 6. In each case the uniqueness of the type follows by assuming the uniqueness of the types for all sub-expressions. Theorem 3 ensures that there is a unique type for iteration expressions. The restriction on empty collections is required because an expression `Set{}` has an infinite number of types by rule 22. Note that the translation defined in figure 3 introduces an empty bag, but rule 20 can deduce the type of bag elements locally since they are required to be the same type as that of e_3. $\square$

4.4 Correctness of the Type Theory

The OCL type theory must be shown to be correct. This is done by establishing that it is sound and complete. Soundness is established by showing that when the type theory attributes a type τ to an expression, the semantics associates the expression with a value whose type conforms to τ. Completeness is established by showing that when the semantics associates a value x with an expression e then the type of x is that defined by the type theory.

Theorem 7. *The type theory for UML static models is sound. For any class context $\mathcal{C}$, type context $\mathcal{A}$, OCL expression e and type τ such that $\mathcal{C}, \mathcal{A} \vdash e : \tau$ then for any $\phi : \mathcal{A}'$ such that $\mathcal{C} \vdash \mathcal{A}' \leq \mathcal{A}$ and $\phi \vdash e \Rightarrow x$ such that $x : \tau'$ then $\mathcal{C} \vdash \tau' \leq \tau$.*

The proof is by induction and proceeds by case analysis on the structure of e. When $e = v$ the theorem holds by definition of $\leq$. When $e = k$ both types are $typeOf(k)$. When $e = v(\tilde{e})$ assume by induction that the theorem holds for the argument expressions. By contravariance, the function $\phi(v)$ is applicable to the argument values when application is defined by the type theory and the type of the result must conform to the type $ran(\mathcal{A}(v))$. When $e = e'.v$ assume by induction that the theorem holds for e' and therefore holds for e by the definition of $\leq$. When $e = e'.v(\tilde{e})$ assume by induction that the theorem holds for the argument expressions and the object expression, when a suitable method exists then by definition of $\leq$ there must be a suitable function and by contravariance the range type of the function conforms to the range type declared for the method. When $e = e_1 \rightarrow \texttt{iterate}(v_1 v_2 = e_2 \mid e_3)$ then assume by induction that the theorem holds for e_1 and e_2. Since $\mathcal{C}, \mathcal{A}[v_1 \mapsto \tau_e, \tau_2] \vdash e_3 : \tau_2$ then assume by induction that $\phi[v_1 \mapsto x_i, v_2 \mapsto y_i] \vdash e_3 \Rightarrow y_{i+1}$ such that $y_i : \tau_i'$ and $\mathcal{C} \vdash \tau_i' \leq \tau_2$. When $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ then assume by induction that the condition produces `true` or `false` and that the theorem holds for both the consequent and alternative. When $e = \texttt{Set}\{\tilde{e}\}$ assume by induction that the theorem holds for each sub-expression and since e must be well formed (all values must be the same type) the theorem holds by definition of $\leq$. $\square$

Theorem 8. *The type theory is complete. For any value context $\phi : \mathcal{A}$, OCL expression e and value x such that $\phi \vdash e \Rightarrow x$ where $x : \tau$ with respect to a class context $\mathcal{C}$ then $\mathcal{C}, \mathcal{A} \vdash e : \tau$.*

The proof is by induction and proceeds by case analysis on the structure of e. When $e = v$ then the theorem holds by definition of $\phi : \mathcal{A}$. When $e = k$ then $k : typeOf(k)$ for all contexts. When $e = v(\tilde{e})$ then assume by induction that the theorem holds for the argument expressions and therefore it holds by the definition of $\phi : \mathcal{A}$. When $e = e'.v$ then assume the theorem holds for the object expression e' and therefore it holds for e by the definition of a well formed object (α, τ, ϕ'). When $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ then the theorem must hold for the condition expression e_1. If e is well formed then both the consequent and alternative expressions produce values of the same type and therefore the theorem holds for e by an inductive assumption on both e_2 and e_3. When $e = e'.v(\tilde{e})$ assume that the theorem holds for the object expression and the argument expressions. If e' denotes a well formed object then the types of v agree and therefore the theorem holds for the result of the function application. When $e = e_1 \rightarrow \texttt{iterate}(v_1 v_2 = e_2 \mid e_3)$ then assuming that the theorem holds for e_1 and e_2 it also holds providing that e_3 produces values of the same type τ_2. When $e = Set\{\tilde{e}\}$ then providing the component values are all of the same type then the theorem holds. $\square$

4.5 Type Consistency of Static Models

A static model $(\mathcal{C}, \mathcal{I})$ is *type consistent* when all of the invariants $(c(), e) \in \mathcal{I}$ are well typed with respect to $\mathcal{C}$ and when all ground classes in the context are well typed. Let $\mathcal{A}_0$ be a type context containing types for the builtin variables. An invariant is well typed when:

$$\mathcal{C}, \mathcal{A}_0(\mathit{sig}(c()))[\texttt{self} \mapsto c()] \vdash e : \texttt{Boolean}$$

Let $c()$ be a class defined in $\mathcal{C}$, $c()$ is well typed if for every method $m \in \mathit{meths}(c())$ with type $\mathit{sig}(c())(m) = \tilde{\tau} \to \tau$, for every specification $(\tilde{v}, e_1, e_2) \in \mathit{specs}(c())(m)$ the following holds: let $\mathcal{A} = \mathcal{A}_0(\mathit{sig}(c()))[\texttt{self} \mapsto c(), v_i \mapsto \tau_i]$ in

$$\mathcal{C}, \mathcal{A} \vdash e_1 : \texttt{Boolean}$$

$$\mathcal{C}, \mathcal{A}(\mathit{sig'}(c()))[\texttt{self'} \mapsto c(), \texttt{reply} \mapsto \tau] \vdash e_2 : \texttt{Boolean}$$

5 Implementation

Given a static model $\mathcal{M}$ section 4.5 defines the conditions under which $\mathcal{M}$ is type consistent. In order to provide a tool to perform consistency checking for UML static models we must prove that there exists a suitable algorithm.

Theorem 9. *Given a static model $\mathcal{M}$ there is an algorithm con such that con$(\mathcal{M})$ returns true when $\mathcal{M}$ is type consistent and false otherwise.*

Let $\mathcal{M} = (\mathcal{C}, \mathcal{I})$. The algorithm *con* independently steps through the classes $\mathcal{C}$ and the invariants $\mathcal{I}$. In both cases the proof relies on the existence of an algorithm *type* such that given a type context $\mathcal{A}$ and an OCL expression e, if there exists a type τ such that $\mathcal{C}, \mathcal{A} \vdash e : \tau$ then $\mathit{type}(\mathcal{C}, \mathcal{A}, e) = \tau$ otherwise *type* reports failure. By theorem 6 the type τ is unique if it exists and by theorem 2 we need only find an algorithm for $[\![e]\!]$. The existence proof is by induction on the structure of e using the rules defined in figure 6. In each case the type of the expression, if it exists, is a function of the types of its sub-expressions. The algorithms *conf* and *el* are used to determine $\mathcal{C} \vdash \tau_1 \leq \tau_2$ and the type of elements τ_e in an iteration expression respectively. $\square$

A consistency checker for UML static models is currently being implemented in Java. The input to the checker is a textual representation of static models although the tool does not rely on this representation and in principle could accept input from a graphical tool given a suitable intermediate representation. The textual language is parsed using a program generated automatically by JavaCC from a static model grammar. The parser produces a static model represented as a tree of Java objects. The tool then performs consistency checking and reports success or failure.

6 Conclusion

The aim of this work is to define rules for checking the type consistency of UML static models. This has been achieved by defining an abstract representation for UML static models, defining a type theory for OCL expressions, showing the type theory is correct with respect to a value semantics for OCL, defining the conditions under which a static model is type consistent and finally establishing the existence of a consistency checking algorithm.

This work contributes to the ongoing efforts to formalise the UML and thereby establish the UML as a sound basis for the development of software systems and more generally addressing methods and techniques which contribute to rigorous object-oriented development including [5], [4], [6] and [7]. In addition, this work will be of use to tool developers wishing to provide computer support for UML.

The following points address the limitations of consistency checking described in this paper and possible areas for future work:

- *Intersection types* have not been used. These would allow heterogeneous sets and expressions of different types in the consequent and alternative parts of conditional expressions. Their addition would appear to be straightforward given a definition for least upper bounds with respect to $\leq$.
- Expressions denoting empty collections are not allowed. Their admission complicates type checking since $\mathcal{C}, \mathcal{A} \vdash \mathit{Set}\{\} : \mathit{Set}(\tau)$ for any type τ. One approach to this problem is to use conditional type schemes [13] such that $\mathcal{C}, \mathcal{A} \vdash e : \rho$ such that all possible typings τ are instances of ρ.
- UML models contain *dynamic* information expressed via sequence, collaboration, statechart and activity diagrams which must be shown to be consistent with static information. In each case the diagram can be given a semantics which is linked to the value semantics for OCL defined in this paper.
- The OCL may be used as part of a UML based software development method. A semantics for OCL is a prerequisite for the development of such methods which are likely to rely on activities including checking type consistency, checking the logical consistency of OCL (*i.e.* establishing the existence of at least one UML model satisfying the conjunction of OCL constraints) and checking OCL refinements. The OCL semantics defined in this paper is a first step towards providing a basis for consistency checking and refinement.

References

1. Brache G., Odersky M., Stoutamire D. & Wadler P. (1998) Making the future safe for the past: Adding Genericity to the Java Programming Language. *in proceedings of the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, (OOPSLA 98).
2. Cardelli L. (1984) Basic Polymorphic Type Checking. *Science of Computer Programming*, **8**(2), 147 – 72.
3. Cardelli L. & Wegner P. (1985) On understanding types, data abstraction and polymorphism. *ACM Computing Surveys.* **17**(4).

4. Clark, A. N. & Evans, A. S.: Foundations of the Unified Modeling Language. Presented at the Second Northern Formal Methods Workshop, June 14, 1997. Published in the proceedings of the 2nd BCS-FACS Nothern Methods Workshop, Ilkley, 1997. BCS FACS Electronic Workshops in Computing.
5. Clark, A. N. & Evans, A. S.: Semantic Foundations of the Unified Modelling Language. In the proceedings of the First Workshop on Rigorous Object-Oriented Methods: ROOM 1, Imperial College of Science Technology and Medicine, London, June, 1997.
6. Clark, A. N.: A Semantics for Object-Oriented Systems. Proceedings of the Third Northern Formal Methods Workshop. September 1998. BCS FACS Electronic Workshops in Computing, 1999.
7. Clark, A. N.: A Semantic Framework for Object-Oriented Development. Technical Report, Department of Computing, University of Bradford, 1999.
8. Cook, D. & Daniels J.: *Designing Object Systems: Object-Oriented Modelling with Syntropy* , Prentice Hall, 1994.
9. D'Souza, D. & Wills, A.: Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 1999.
10. Goguen, J.: Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis (Eds.), Software Reusability, Volume I: Concepts and Models, pp. 159 – 225. Addison Wesley, 1989.
11. Meyer, B.: Genericity Vs. Inheritance. In Meyrowitz N. (Ed.) ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM, 1986, pp. 391 – 405.
12. Meyer, B: *Eiffel the Language*. Prentice-Hall Object-Oriented Series, 1992.
13. Ohori, A., & Buneman, P.: Static type Inference for Parametric Classes. Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems Languages and Applications, pp 445 – 456.
14. Booch G., Jacobson I. & Rumbaugh J.: The Unified Modeling Language User Guide. Addison Wesley Longman, 1998.
15. UML Consortium. Object Constraint Language Specification Version 1.1. Available from `http://www.rational.com` 1997.
16. Odersky M. & Wadler P. (1997) Pizza into Java: Translating theory into practice. *Symposium on Principles of Programming Languages*, pp 146 – 159.