Higher-Order TRIO

Carlo A. Furia, Dino Mandrioli, Angelo Morzenti,
Matteo Pradella[*], Matteo Rossi, Pierluigi San Pietro
Dipartimento di Elettronica ed Informazione
Politecnico di Milano,
[*]CNR IEIIT-MI
{furia, mandrioli, morzenti, pradella, rossi, sanpietro}@elet.polimi.it

0 Index

1 Introduction: Why Higher-Order TRIO

TRIO [C3M399] is a first-order metric temporal logic with object-oriented constructs that is suitable to describe time-critical systems. TRIO is typed, but the number of types available in TRIO is limited to a small set. To model general complex systems (that is systems ranging from invoice management applications to power plants), however, we feel that great flexibility in defining what are the objects composing the system is needed.
Higher-Order TRIO (HOT for short) is a thorough revisitation of the TRIO language, one in which the user can freely define new types, which can be complex (or, for that matter, simple and basic) at will.
In fact, in origin, HOT derives from a very specific need, which is to effectively model system (software) architectures, that require suitable mechanisms and constructs to easily represent information such as "object A sends object B another object C", etc.
Representing this kind of information in usual TRIO would not be impossible, but certainly complicated and cumbersome (in two words, practically unfeasible).
For this, we need a new definition of types in TRIO. More precisely, we will start from the idea that *Class = Type*.
As a consequence, TRIO becomes a higher-order temporal logic, as it is now possible to quantify over variables that are typed by a *Class* (which can include functions, predicates, etc.). We will investigate the consequences of the equality *Class = Type* in the next sections.

This document is structured as follows. Section 2 presents some basic terminology and definitions. Section 3 introduces the syntax of HOT, while Section 4 gives a lambda calculus-based semantics for the language. Section 5 adds some syntactic sugar to basic HOT to simplify writing useful and common properties. Section 6 presents an alternative semantics for HOT based on set theory (Set theory-based HOT, SHOT for short). Section 7 hints at genericity in HOT. Finally, Section 8 draws some conclusions and outlines future work related to HOT.

Before concluding this introduction, notice that, with respect to TRIO, the modal part (that is, Time) of the HOT logic remains unchanged. The representation of temporal evolution of systems in HOT is still built around the *Dist* operator (and all the usual derived temporal operators). Then, this document will not delve into the details of HOT temporal operators, whose definition and semantics is the same as in old TRIO.

2 Some terminology and basic definitions

In the definition of HOT, let us start with a list of basic concepts and their synonyms (summarized in Table 1).

Concept	Synonyms
Item	Function. Predicate. Relation.
Formula	
Class	Type (and subtype). Domain. Set. (Module.)
Object	Instance. Value (of a type). Model. History.

Table 1 - List of HOT concepts and synonyms.

Items are the founding elements of the HOT logic. In HOT (and TRIO) terms, we call *item* what, in usual logic lore, is called *function* or *predicate* (notice that an item is either a function or a predicate, but in traditional logic a predicate is not a function, even though it is easy to express the former using the latter, and vice-versa). HOT items can have arguments (and return values), which are *typed* elements. The arguments (and returned values) of HOT items can be of any HOT type (see below). For example, we might define a HOT item `it` to be a predicate with two arguments of type `t1` and `t2`:

```
items:
TD it(t1, t2) : boolean;
```

Notice that, just like in TRIO, HOT items can be defined to be time-dependent (`TD`) or time-independent (`TI`).

Items are the building blocks for HOT ***formulae***. HOT (well-formed) formulae are, as usual, a combination of functions, predicates (that is, items), logical connectors (`&`, `|`, `->`, `<->`, `not`, etc.), temporal operators (*Dist*, *Futr*, *Past*, etc.) and quantifiers (`all`, `exists`). For example:

```
p1(f1, f2(c1)) -> Futr(all(o)(p2(o)), t);
```

Notice that every HOT variable (for example `c1`, `o` and `t` in the formula above) ranges over the values of some type, which is defined through a HOT class.

So, as stated before, a HOT ***class*** defines a *type*. A class identifies a *domain*, or, equivalently, a *set* of elements.

Given a HOT domain (i.e. class), an element of the domain is an ***object***. The term *object* is synonym for *instance* (of a class) and *value* (of a type). A HOT *object* corresponds also to a *model* for the corresponding class (or, in TRIO terms, to a *history*). We will analyze in Section 4, which defines HOT semantics, the meaning of making *object* synonym with *model* (and *history*).

3 Syntax

Basic elements

The basic elements of the HOT language are:

- constants (of a type)
- variables (of a type)
- items
- connectors and quantifiers (`not`, `&`, `all`, etc.)
- temporal operators (`Dist` and all the usual derived operators `Futr`, `Past`, etc.)

Constants and variables must have a *type*, which corresponds to the name of a HOT *class*. The type `boolean` corresponds to the truth values `true` and `false`.

Items can only be declared *inside* HOT classes.

Terms

- a constant of type `t` is a term of type `t`
- a variable of type `t` is a term of type `t`
- if `i(T1, ... Tn):Tret` is an item declared in class `C`, where `T1, ... Tn` and `Tret` are all type names, and `v1, ... vn` are terms of type `T1, ... Tn` and `c` is a term of type `C`, then `c.i(v1, ... vn)` is a term of type `Tret`.
- nothing else is a term.

Predicates are terms of type `boolean`.

Formulae

- a predicate is a formula
- if `f1` and `f2` are formulae, `not f1` and `f1 & f2` are formulae (all other usual logic connectors can be defined from `not` and `&`)
- if `v` is a variable of some type `t` (that is, ranging over the values of a class `t`) and `f` is a formula, all `v(f)` is a formula (`ex`, as usual, is defined as `not all not`)
- if `f` is a formula and `t` is a term of type `temporal domain`, `Dist(f, t)` is a formula
- nothing else is a formula.

HOT formulae can be declared only *inside* HOT classes.

Examples

Here are some examples of definitions of HOT classes.
The first example defines the natural numbers using Peano's axioms. Class `Natural` has three items, `isZero`, `succ`, and `+`. `isZero` identifies the natural number 0, that is, there is only one natural number such that `isZero` is true. For short, we refer to this particular natural number with constant 0 in the axioms of class `Natural`. `succ` is a function that returns the successor of a natural number, while + represents the usual addition. `NatPredicate` is a class (i.e. a type) representing the set of possible predicates on natural numbers (that is, the set of functions from naturals to booleans), and is defined below.

```
class Natural
visible: isZero, succ, +;
items:
  TI isZero : boolean;
  TI succ : Natural;
  TI +(Natural): Natural;

axioms:
  vars:
  n, n1, n2 : Natural;
  z : Natural;
  p : NatPredicate;   /* defined below */

  formulae:
  Peano1:  ex z(z.isZero);  /* call it 0 */

  Peano1_bis:  n1.isZero & n2.isZero -> n1 = n2;

  Peano2 : all n1(ex n2(n2 = n1.succ));
```

```
    Peano3: n1.succ = n2.succ -> n1 = n2;

    Peano4: not ex i(ex z(z.iszero & (z = i.succ)));

    Peano5: all p(p.val(0) ->
                   (all i(p.val(i) -> p.val(i.succ))
                    ->
                    all i(p.val(i)) ));

    ax+_1: all n1, n2(n1.+(n2.succ) = (n1.+(n2)).succ);
    ax+_2: all n1 (n1.+(0) = n1);
  end
```

Notice that, to represent `a + b` given the definition above one should write `a.+(b)`, since + is an item of class Natural. However, HOT allows also the classic (and more natural) syntax `a + b`. Section 5 deals with this (and other) syntactic features of the language.

Class `NatPredicate` defines (a class corresponding to) the set of all predicates on natural numbers. In classic logic, if *p* is a predicate over natural numbers and *n* is a natural number, we would write *p(n)* to signify the value of predicate *p* in *n*. Again, the syntax for pure HOT is a little more cumbersome, as we define an item, `val`, which takes a natural number as argument, and corresponds to the value of the predicate in that natural number. Then, instead of *p(n)*, in HOT we write `p.val(n)`. Again, shortcuts for notations such as `p.val(n)` might be devised in the future, if necessary.

```
  class NatPredicate
  visible: val;
  items:
    TI val(Natural) : boolean;
  end
```

Class `Invoice` defines invoices. It has only one axiom, which states that the serial number of an invoice separates it from other invoices (that is, no two invoices have the same serial number).

```
  class Invoice
  visible: sr_num, amount, date;
  items:
    TI sr_num : Natural;
    TI amount : Real;
    TI date : Date;

  axioms:
    vars:
    i1, i2: Invoice;

    formulae
    sr_num_unique: all i1, i2(i1.sr_num = i2.sr_num -> i1 = i2);
  end
```

Class Sensor defines the behavior of any possible sensor of a system. Every sensor has a serial number, which separates it from every other sensors. The quantity measured by the sensor is represented by item `measure`, item `limit` is a threshold that said quantity should not exceed, while predicate `over_limit` is true when the measured quantity is above its limit. Finally, predicate `working` is true when the sensor is correctly functioning (false otherwise).
The two axioms of the class state the uniqueness of serial numbers for sensors, and the behavior of predicate `over_limit`.

```
class Sensor
visible: measure, limit, over_limit, working;
temporal domain: Real;
items:
  TI sr_num: Natural;
  TD total measure: Real;
  TI limit: Real;
  TD over_limit : boolean;
  TD working : boolean;

axioms:
  vars:
  s1, s2: Sensor;

  formulae:
  def_over_limit: over_limit <-> measure > limit;
  sr_num_unique: all s1, s2(s1.sr_num = s2.sr_num -> s1 = s2);
end
```

Finally, class `Controller` models controllers provided with two sensors. The hypothesis is that the two sensors monitor the same quantity, and item `avg` keeps track of the average of the two measurements (as defined by axiom `def_avg`). The sensors of the controller can be replaced if they stop working. Axiom `sensor1_change` states exactly this property for sensor `s1`, that is, that if sensor `s1` stops working, it will be replaced with a *different* sensor within `SUBS_TIME` time units.
Axiom `sensors_never_the_same` states that it is not possible to wire the controller in a way such that it reads data from just one sensor.

```
class Controller
temporal domain: Real
items:
  TD s1: Sensor;
  TD s2: Sensor;
  TD avg: Real;

axioms:
  vars:
  sid1, sid2: Natural;

  formulae:
  sensors_never_the_same: Alw(s1 != s2);
  def_avg: avg = (s1.measure + s2.measure)/2;

  sensor1_change:
  not s1.working & s1.sr_num = sid1
  ->
  WithinF(s1.working & s1.sr_num != sid1, SUBS_TIME);
/* SUBS_TIME is a pre-defined positive real constant */
end
```

4 Semantics (of Basic HOT)

The semantics of HOT is given in terms of typed lambda calculus, as used in the type theory presented in [And86]. Then, HOT classes are to be interpreted as *sets* of *values*, and everything else is a *function* (even connectors and quantifiers, as shown in [And86]). Most notably, items of classes are functions.

Suppose, for example, that we declare in HOT the following class C (let us focus on time-independent items for the time being, we will tackle the issue of time later in this document):

```
class C
...
TI f(β) : γ;
...
end.
```

The first line declares C to be a set of values. Given an item `i` of class `C`, let us call $\mathtt{sig_C(i)}$ its signature in class `C`. For example, the signature of item `f` is $\mathtt{sig_C(f)} = \beta \rightarrow \gamma$, while the signature of item `+` in class `Natural` is `Natural -> Natural`.

Definition 1 (time-independent items). A time-independent item `i` of a class `C` is interpreted as a function `i` that has domain `C` and range the signature of `i`. That is,

```
i : C -> sigC(i)
```

which means that symbol `i` has type $\mathtt{C \rightarrow sig_C(i)}$.

For example, item `f` of class C is interpreted as a function f with domain C and range $\beta \rightarrow \gamma$:

```
f : C -> (β -> γ)
```

(or, borrowing a notation from the PVS logic [OSS99] `f` is a function of type `[C -> [β -> γ]]`; alternatively, using the notation of [And86], $f_{\gamma\beta C}$).
As additional examples, item `+` in class `Natural` is interpreted as

```
+ : Natural -> (Natural -> Natural)
```

item `val` of class NatPredicate corresponds to the following function

```
val : NatPredicate -> (Natural -> bool)
```

and item `sr_num` of class `Sensor` is interpreted as

```
sr_num : Sensor -> Natural
```

Then, to each HOT class `C` that has items $i_1, \ldots, i_n$ correspond `n` functions i_j ($j = 1, \ldots n$) that have type $\mathtt{C \rightarrow sig_C(i_j)}$. That is, every value `c` of type `C` is associated with `n` values $i_j(c)$, each one with type $\mathtt{sig_C(i_j)}$.
The `n` values associated with value `c` uniquely identify `c`. That is, a value `c` of class `C` is identified by the set of its images through functions $i_1, \ldots i_n$.

Definition 2. Given a HOT class C with items $i_1, \ldots, i_n$ every value `c` of type `C` is uniquely identified by the `n` images associated with `c` through functions $i_1, \ldots i_n$. More precisely, using the interpretation of HOT as typed lambda calculus and classic logic notation:

$$\begin{aligned}\forall c1,c2{:}C \quad (&i_1(c1) = i_1(c2) \wedge \\ &i_2(c1) = i_2(c2) \wedge \\ &\ldots \wedge \\ &i_n(c1) = i_n(c2)\end{aligned}$$

```
                  →
                  c1 = c2)
```

For example, definition 2 for class `Natural` presented above translates to

```
∀n1,n2:Natural (isZero(n1) = isZero(n2) ∧
                 succ(n1) = succ(n2) ∧
                 +(n1) = +(n2)
                 →
                 n1 = n2)
```

while for class `NatPredicate` definition 2 corresponds to

```
∀np1,np2:NatPredicate (val(np1) = val(np2)
                        →
                        np1 = np2)
```

and for class `Invoice` to

```
∀i1,i2:Invoice (sr_num(i1) = sr_num(i2) ∧
                 amount(i1) = amount(i2) ∧
                 date(i1) = date(i2)
                 →
                 i1 = i2)
```

Notice that axiom `Peano3` of class `Natural` and `sr_num_unique` of class `Invoice` are stronger than definition 2 for those two classes, which is entirely admissible.

Then, a HOT class is defined by its axioms in the sense that its axioms constrain the values of its items, and the values of its items define entirely a HOT class.

Time

A temporal domain τ in HOT can be any type with a total order and metric (that is, a notion of *distance* between instants is defined). HOT items can be time-dependent or time-independent. Time-dependent items can vary over time, meaning that they evaluate to different values depending on the instant in which they are evaluated.

Definition 3 (time-dependent items). A time-dependent item `td_i` of a class `C` with temporal domain τ is interpreted as a function `td_i` that has domain `C` and range the functions from τ to the signature of `td_i`. That is,

$$td\_i : C \to (\tau \to sig_C(i))$$

which means that symbol `td_i` has type $C \to (\tau \to sig_C(i))$.

For example, item `measure` of class `Sensor` (which has temporal domain `Real`) is interpreted as

```
measure : Sensor -> (Real -> Real)
```

while item `s1` of class `Controller` corresponds to

```
s1 : Controller -> (Real -> Sensor)
```

As usual with TRIO, however, in formulae time is implicit, so it is never explicitly represented or cited; for example, if `s` is a variable of type `Sensor`, `s.measure` is interpreted as `measure(s)(t)`, where `t` (`t` $\in$ τ) is the current instant.
The usual *Dist* temporal operator of TRIO is used to manipulate the implicit temporal variable of type τ.

Objects as histories

Given the definitions presented above (and in particular definitions 2 and 3), the values of a class `C` correspond to the *histories* of `C`.
Take, for example, class `Controller`. Definition 2 for class `Controller` translates to

```
∀c1,c2:Controller  (s1(c1) = s1(c2) ∧
                    s2(c1) = s2(c2) ∧
                    avg(c1) = avg(c2)
                    →
                    c1 = c2)
```

which means that two different controllers have different behaviors over time (so that every controller is identified by a precise *history*).

Now, take class `Sensor` as further example. Definition 2 for class `Sensor` translates to

```
∀s1,s2:Sensor (sr_num(s1) = sr_num(s2) ∧
               measure(s1) = measure(s2) ∧
               over_limit(s1) = over_limit(s2) ∧
               working(s1) = working(s2)
               →
               s1 = s2)
```

However, axiom `sr_num_unique` of class `Sensor` states that the serial number is enough to identify sensors: if two sensors have different serial numbers, then they are different values of class `Sensor`. This means that different sensors can have the same interpretation on `measure`, `over_limit` and `working` items, but they have different serial numbers (that is, they have different histories, as the serial number is part of them, too).

Implicit quantification over the values of a class (explained through an example)

Let us consider axiom `def_over_limit` of class `Sensor`. Let us repeat the axiom here for the sake of clarity, making explicit the implict temporal closure:

```
def_over_limit: Alw(over_limit <-> measure > limit).
```

As mentioned above, `over_limit`, `measure` and `limit` are in fact to be interpreted in the following way:

```
over_limit : Sensor -> (Real -> bool)
measure    : Sensor -> (Real -> Real)
limit      : Sensor -> Real
```

As axiom `def_over_limit` is, by definition, valid for all sensors, it is in fact to be interpreted as the higher-order formula:

```
∀s:Sensor (Alw(over_limit(s) <-> measure(s) > limit(s))).
```

That is, given an axiom of a class `C` in which unqualified references to class items appear (i.e. references without explicit mention of the containing class, such as all references in axiom `def_over_limit`), the unqualified references are implicitly universally quantified over all values of `C`.

Semantic remark (just to be precise). HOT domains can be infinite (we need to be able to represent numbers of various kinds, from naturals to reals), so HOT type theory includes an axiom of infinity[1].

5 Syntactic sugar

5.1 `non-contextual` *and* `infix` *keywords*

Consider class `Natural` defined in Section 3. In that class, operator + is defined as follows:

```
TI +(Natural) : Natural
```

which implies that in classic, pure object-oriented notation, to represent `a + b` one should write `a.+(b)`, since + is an item of class Natural (which takes a single Natural as argument, and returns the sum of its argument and the Natural value to which it is applied). However, sometimes it is useful (and more intuitive for the specifier) to be able to use the classic mathematical notation.
To this end, HOT allows the following declaration:

```
non-contextual TI + (Natural, Natural) : Natural
```

in which case axioms `ax+_1` and `ax+_2` of class `Natural` are rewritten as follows:

```
ax+_1: all n1, n2(+(n1, n2.succ) = (+(n1, n2)).succ);
ax+_2: all n1 (+(n1, 0) = n1);
```

The precise meaning of the `non-contextual` keyword is detailed below, in this same section.
To simplify syntax further and make it more similar to usual notations, HOT introduces another keyword, `infix`, which allows one to use an infix notation for (non-contextual) binary items. By using the `infix` keyword, the declaration of item + above becomes:

```
infix non-contextual TI + (Natural, Natural) : Natural
```

in which case axioms `ax+_1` and `ax+_2` are rewritten as follows:

```
ax+_1: all n1, n2(n1 + n2.succ = (n1 + n2).succ);
ax+_2: all n1 (n1 + 0 = n1);
```

[1] This causes the HOT logic to be *incomplete* with respect to standard models (that is, models in which every function type is required to represent the set of all possible functions from the domain set, to the range set).

Keyword `infix` can be used *only* in conjunction with keyword non-contextual, and can be applied only to (non-contextual) items with *exactly* two arguments. Other that that, `infix` has no additional semantics, and is pure syntactic sugar.

Semantics of the `non-contextual` keyword

From the discussion of Section 4 it is clear that the interpretation of an item of a class can differ between two values of the same class; for example, item `measure` of class `Sensor` can have different temporal profiles for different values of class `Sensor` (say , for $\hat{s}_1 \in$ `Sensor` `measure` at time instant 1 might have image 5.5, while for $\hat{s}_2 \in$ `Sensor` `measure` at the same time instant 1 might have image 8.3).
That is, the interpretation of an item `it` of class `C` depends on the *context* in which it is evaluated, where by the term *context* we mean the actual value of class `C` that is being considered (given `c1,c2` $\in$ `C`, with `c1`$\neq$`c2`, `c1.it` can have different interpretation from `c2.it`).
On the other hand, we might want to state that the value returned by a function is independent of the context in which it is applied, but depends only on its arguments. Operator (i.e. function) `+` (the version with two arguments, one for each operand) of class Natural is such an example: if we define `+` as follows:

```
TI + (Natural, Natural) : Natural
```

we expect that, if `n1,n2` $\in$ `Natural`, even if `n1`$\neq$`n2`, `n1.+(n3,n4)=n2.+(n3,n4)` and it corresponds to the usual num `n3+n4` for natural numbers). Unless we state otherwise with a suitable axiom, the declaration above does not rule out that for `n1,n2` $\in$ `Natural` and `n1`$\neq$`n2`, it might happen that `n1.+(n3,n4)`$\neq$`n2.+(n3,n4)` (where `n3,n3` $\in$ `Natural`).
So, we might like to have a syntactic mechanism to state that the interpretation of an item is independent of the context in which is it evaluated, and is entirely determined by its arguments.
Keyword `non-contextual` achieves exactly this: it says that the interpretation of the item is independent of its context.

Definition 4 (semantics of keyword `non-contextual`). For every non-contextual item `nci` (either time-independent or time-dependent) of a class `C` (with signature $\mathtt{sig}_{\mathtt{C}}\mathtt{(nci)}$) the following holds:

```
∀c1,c2:C (nci(c1) = nci(c2))
```

So, given the following declaration in class `Natural`,

```
non-contextual TI + (Natural, Natural) : Natural
```

the following formula holds:

```
∀n1,n2:Natural (+(n1) = +(n2))
```

which, in HOT terms, corresponds to saying that

```
all n1,n2(all n3,n4( n1.+(n3,n4) = n2.+(n3,n4) ))
```

Then, for `non-contextual` items, the specific value in which they are evaluated (`n1`, and `n2`, in the formula above) is irrelevant, and can be skipped altogether. We can then write `+(n3, n4)` (or, better, `n3+n4` if we declare `+` to be also `infix`) directly, instead of referring to a specific value as in `n1.+(n3, n4)`, without risk of ambiguities.

5.2 Modules

TRIO has a notion of *module*: a TRIO module is an instance of a TRIO class contained in another TRIO class. Furthermore, in TRIO the notion of module is *primitive*.
HOT does not have a *primitive* notion of module. Rather, it has *linguistic constructs* that allow one to obtain the same semantics of TRIO modules in HOT from basic HOT concepts.
HOT offers the keyword `module` as a shortcut to automatically introduce the HOT axioms and definitions corresponding to the semantics of TRIO modules.
HOT allows the following syntax:

```
module <module_name> : [array <array_range> of] <module_type>
```

where `module`, `array` and `of` are terminals, `<module_name>`, `<module_type>` and `<array_range>` nonterminals which expand, respectively, to two identifiers (the name and type of the module) and one domain (the range of the array).
The syntax above is translated in either of the two following HOT declarations, depending on the fact that the module is an array or not:

```
TI <module_name> : <module_type;
```

```
TI <module_name>(<array_range>) : <module_type>;
```

Every module defines a *different* value of type `<module_type>`, as defined by the following definition.

Definition 5 (semantics of keyword `module`). Given a class `C` with modules m_1, ... m_n and ma_1, ... ma_m, all of the same type `t` (where m_1, ... m_n are single modules, while ma_1, ... ma_m are arrays of modules with ranges r_1, ... r_m), the following formula holds:

$$
\begin{aligned}
&m_1 \neq m_2 \wedge m_1 \neq m_3 \wedge \ldots \wedge m_1 \neq m_n \wedge (\forall i \in r_1)(m_1 \neq ma_1(i)) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\forall i \in r_2)(m_1 \neq ma_2(i)) \wedge \ldots \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\forall i \in r_m)(m_1 \neq ma_m(i)) \wedge \\
&m_2 \neq m_3 \wedge \ldots \wedge m_2 \neq m_n \wedge (\forall i \in r_1)(m_2 \neq ma_1(i)) \wedge \ldots \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\forall i \in r_m)(m_2 \neq ma_m(i)) \wedge \\
&\ldots \wedge \\
&(\forall i_1, i_2 \in r_1)(i_1 \neq i_2 \rightarrow ma_1(i_1) \neq ma_1(i_2)) \wedge \\
&(\forall i_1 \in r_1)(\forall i_2 \in r_2)(ma_1(i_1) \neq ma_2(i_2)) \wedge \ldots \wedge \\
&(\forall i_1 \in r_1)(\forall i_m \in r_m)(ma_1(i_1) \neq ma_m(i_m)) \wedge \\
&\ldots \wedge \\
&(\forall i_1, i_2 \in r_m)(i_1 \neq i_2 \rightarrow ma_m(i_1) \neq ma_m(i_2))
\end{aligned}
$$

Then, if class `UnmodifiableController` contains the following declarations:

```
class UnmodifiableController
...
module uc_s1 : Sensor;
module uc_s2 : Sensor;
```

```
...
end
```

the following definitions and formula hold for the class:

```
TI uc_s1 : Sensor;
TI uc_s2 : Sensor;

uc_s1 != uc_2;
```

5.3 Quantification over the modules of a class

To make writing HOT specifications lighter, a shortcut that allows users to easily express quantification over all the *modules* (in the sense defined in Section 5.2) of a class that have a certain type is defined.
So, if `v` is a variable of type `t` and `f` is a HOT formula, `allmod v(f)` is also a HOT formula, whose semantics is given by the following definition.

Definition 6 (semantics of quantifier `all modules`). Given a class `C` with modules $m_1, \dots m_n$ and $ma_1, \dots ma_m$, all of the same type `t` (where $m_1, \dots m_n$ are single modules, while $ma_1, \dots ma_m$ are arrays of modules with ranges $r_1, \dots r_m$), and given a variable `v` of type `t` and a formula `f(v)` (where `v` is free in `f`), formula `allmod v(f(v))` corresponds to:

$$f(m_1) \wedge f(m_2) \wedge \dots \wedge f(m_n) \wedge$$
$$(\forall i \in r_1)(f(ma_1(i)) \wedge \dots \wedge (\forall i \in r_m)(ma_m(i))$$

Notice that this is *not* a quantification over *all* instances of type `t`, but only on those instances that correspond to modules of class `C`.
For example, if in class `UnmodifiableController` outlined in Section 5.2 one writes `allmod s(f(s))`, where `s` is a variable of type `Sensor` and `f` a formula in which `s` is free, this would simply correspond to writing
`f(uc_s1) ∧ f(uc_s2)`, which is *not* the same as `all s(f(s))`.

`exmod v(f(v))` is naturally defined as `not allmod v(not f(v))`.

5.4 Key of a HOT class

Every value of a HOT class corresponds to an interpretation (an assignment to the items of the class), that is, in usual TRIO terms, a *history*: different instances of the same TRIO class differ for at least the value of one of their items in one time instant (see definition 2 of Section 4). This means, conversely, that two instances of a TRIO class are in fact the same if they have the same interpretation for all their items.
In some cases, however, the condition that identifies the instances of a class could be stronger (for example, for the sensors modeled in Section 3, it is enough that two instances have the same serial number for them to be the same).
HOT offers the keyword **`key`** to explicitly state a condition that is enough to uniquely identify the values of a class.
Keyword **`key`** is used in the **`formulae`** subsection in the **`axioms`** declaration, and has the following syntax:

```
key: <condition>;
```

Only one key declaration is allowed in each class.
In the formula corresponding to a `key` declaration a class `C`, there must be exactly two free variables of type `C`. For example, the property defined by axiom `Peano3` of class `Natural` of Section 3 might be used as key for the class, but in this case it should be declared as follows:

```
class Natural
...
axioms:
  vars:
    n1, n2: Natural
...
  key: n1.succ = n2.succ;
...
end
```

The precise semantics of a **key** declaration is given by the following definition.

Definition 7 (semantics of keyword `key`). Given a HOT class C with condition `cond` declared as `key`, every value `c` of type `C` is uniquely identified by condition `cond`. More precisely, using again the interpretation of HOT as typed lambda calculus:

```
∀c1,c2:C  (cond(c1, c2)
           →
           c1 = c2)
```

From definition 7 it descends that in the above revised declaration of class `Natural`, the `key` declaration translates to the following formula:

```
∀n1,n2:Natural  (n1.succ = n2.succ → n1 = n2)
```

which corresponds to axiom `Peano3` of the original declaration.
Take now, as a second example, class `Sensor` also of Section 3. We could revise its definition and substitute axiom `sr_num_unique` with the following key declaration:

```
key: s1.sr_num = s2.sr_num;
```

which translates to the following formula:

```
∀s1,s2:Sensor  (s1.sr_num = s2.sr_num → s1 = s2)
```

which is precisely axiom `sr_num_unique`.

6 A different approach: set theory

So far we have based HOT on lambda calculus, but it is worth exploring a different approach to found HOT on: set theory (see for example [Men97]). To separate this second approach from the first one, we call it SHOT (Set theory-based Higher Order Trio).
SHOT is very similar to HOT, but employs a different semantic approach, which is reflected also in few selected syntactic aspects. In the rest of this section we present SHOT by highlighting its differences with respect to HOT (which will be graphically stressed).

6.1 Syntax

Basic elements

The basic elements of the SHOT language are:

- constants
- variables
- items
- operations
- (types)

Constants and variables must have a *type*, which corresponds to the name of a HOT *class*.
We will write `x : D` to stress that `x` has type `D`, and `x(D1) : D2` for functions with signature `D1 -> D2` (the latter like in good old first-order TRIO) .

Type `Bool` is defined as `{true, false}`.

Items and operations can only be declared *inside* SHOT classes.

Terms

- a constant is a term
- a variable is a term
- if `i : T1 × T2 × .. × Tn -> Tret` is an item declared in class `C`, and `v1, ... vn` are terms of type `T1, ... Tn` and `c` is a term of type `C`, then `c.i(v1, ... vn)` is a term of type `Tret`.
- an operation `f : A -> B`, applied to a term `a : A` is a term `f(a): B`
- nothing else is a term.

Logical connectives (`and`, `or`, `implies`, `not`) are constants having type `Bool × Bool -> Bool`, and `Bool -> Bool` respectively. Formulae and predicates are terms of type `D1 × D2 × .. Dk -> Bool`, where `Di` are types of free variables. Moreover SHOT has the usual quantifiers for every type: `all t : T`; `ex t : T`.

6.2 Semantics

SHOT is an extensional logic: two types or functions or anything else are equal iff they have the same type and are equal on all their arguments.

Classes define new types, and may be based on existing types.
In fact, we separate two types of classes: classes defining primitive types, and classes defining (derived) types based on other types. They differ on the number of items declared in them: classes defining primitive types do not have *any* items declared in them, while classes defining derived types contain *at least* one item declaration.

The declaration of a primitive type `PC` simply corresponds to stating that there is a set of name `PC`, and its axioms define it (see class Natural declared below in Section 6.3)..

Suppose instead we have a class C with N (N > 0) items $i_1...i_N$ (where the first P items are time-independent, and the other N-P ones are time-dependent) and M (M ≥ 0) operations $op_1...op_M$ (of which the first L ones are time-independent, the others time-dependent); suppose also that every item i has signature $sig_C(i)$, and every operation op has signature $sig_C(op)$ (see also Section 4 for the definition of sig_C). The declaration of class C would roughly be the following one:

```
class C
temporal domain: τ;

items:
  TI i_1   : sig_C(i_1);
  ...
  TI i_P   : sig_C(i_P);
  TD i_P+1 : sig_C(i_P+1);
  ...
  TD i_N   : sig_C(i_N);

operations:
  TI op_1   : sig_C(op_1);
  ...
  TI op_L   : sig_C(op_L);
  TD op_L+1 : sig_C(op_L+1);
  ...
  TD op_M   : sig_C(op_M);

axioms:
...
end
```

The first line states that C is a type.
The items section states that
$C \subseteq \wp(sig_C(i_1) \times ... sig_C(i_P) \times (\tau \rightarrow (sig_C(i_{P+1}) \times ... sig_C(i_N))))$.
The operations section defines name and signature of permitted operations on type C, while actual operations are defined by the axioms.
Axioms define if and how C is a proper subset of
$\wp(sig_C(i_1) \times ... sig_C(i_P) \times (\tau \rightarrow (sig_C(i_{P+1}) \times ... sig_C(i_N))))$, and its corresponding operations.

An object c of type C has type
$sig_C(i_1) \times ... sig_C(i_P) \times (\tau \rightarrow (sig_C(i_{P+1}) \times ... sig_C(i_N)))$.
An object is completely defined by its items, but *not* by its operations.

While item signatures are used to define C, item names are used as name of components. Dot notation is defined as projection. E.g. let t be the current time instant and I the interpretation function. Then $c.i_3$ is interpreted at the current time instant as $I(c)(t)|_{sig_C(i3)}$. ($I(c)$ is a tuple :
$sig_C(i_1) \times ... sig_C(i_P) \times (\tau \rightarrow (sig_C(i_{P+1}) \times ... sig_C(i_N)))$.
$I(c)(t)$ is the object value at time t, i.e. $I(c)(t) \in$
$sig_C(i_1) \times ... sig_C(i_P) \times (sig_C(i_{P+1}) \times ... sig_C(i_N)))$.

Notice that item signatures in C must not contain C itself nor of any derived type that depends on C (we define the relation "depend on" for types in the following way: if a type C1 has an item with either an argument or the range of type C2, then C1 depends on C2 and, recursively, on all types on which C2 depends). Notice also that, by the definition itself of primitive type as introduced above, a primitive type does not depend on any other types.

On the other hand, an operation declared in a SHOT class C must have *at least* one argument or the range which is either of type C itself, or of a type that depends on C.[2]

Finally, notice that, with the new semantics, definitions 1, 2 and 3 of Section 4 are not needed any more (in fact, definitions 1 and 3 are made unnecessary by the interpretation of classes as subsets of powersets, while definition 2 derives from the axioms of the underlying set theory).

6.3 Examples

Here we present the examples introduced in Section 3, but with the new separation in items and operations.

```
class Natural

/* no items: Natural is a primitive set
 * (i.e. it is not defined in terms of other sets)
 */


operations:
  TI succ(Natural) : Natural;
  /* i.e. N -> N */

  TI +(Natural, Natural): Natural;
  /* i.e. N x N -> N */

axioms:
  vars:
  n, n1, n2 : Natural;
  p : Natpredicate;   /* defined below */

formulae:
  Peano1: 0 in Natural;

  Peano2: all n1(ex n2(n2 = succ(n1)));

  Peano3: succ(n1) = succ(n2) -> n1 = n2;

  Peano4: not ex n (0 = succ(n));

  Peano5: all p (p(0) ->
                (all n (p(n) -> p(succ(n)))
                 ->
                 all n (p(n)) ));

  ax+_1: all n1, n2(+(n1,succ(n2)) = succ(+(n1,n2)));
  ax+_2: all n1 (+(n1,0) = n1);

end
```

As usual, + is more commonly used as infix, so we will write 3 + 4 instead of +(3,4)...

[2] Notice that, in a SHOT class, it would be possible to separate items from operations without explicitly using keywords `items` and `operations` (items never include types that depend on the enclosing class, while operations always do). However, the keywords help clarify the role of class elements, hence their introduction, despite the fact that they are redundant.

Class `NatPredicate` defines the set of all predicates on natural numbers.

```
class NatPredicate
items:

 TI v(Natural) : Bool;

end
```

We can define a useful shortcut for classes `C` with *exactly* one item `i`: given an instance `c` of type `C`, instead of writing `c.i`, we decide to simply write `c`, since `c.i` in SHOT is a projection operation, but if there is only one item, we project on the whole class. Hence in axiom `Peano5` in class `Natural` we write `p(n)` instead of `p.v(n)`.

A new version of the `Invoice` class, with a `copy` operation (the `copy` operation is partial, because it may not always be invoked: when it is invoked, it returns a copy of the `Invoice`):

```
class Invoice
visible: sr_num, amount, date;
items:
  TI sr_num : Natural;
  TI amount : Real;
  TI date : Date;

operations:

  TD partial copy(Invoice) : Invoice;

axioms:
  vars:
  i1, i2: Invoice;

  formulae:
  sr_num_unique: all i1, i2(i1.sr_num = i2.sr_num -> i1 = i2);

  copy_def: copy(i1) = i2
            ->
            i1.sr_num <> i2.sr_num &
            i1.amount = i2.amount &
            i1.date = i2.date;
end
```

An `Invoice_DB` object has an event such that, when it occurs, a copy of a certain invoice is made:

```
class Invoice_DB
visible: copy_invoice;
items:
  TD invoices(Natural) : Invoice;
  event copy_invoice(Natural);

axioms:
  vars:
  n1, n2 : Natural;
  i: Invoice;

  formulae:
  copy_invoice_def: copy_invoice(n1) &
                    Invoice.copy(invoices(n1)) = i
                    ->
                    ex n2(n2 <> n1 & NowOn(invoices(n2) = i);
end
```

A new version of the `Sensor` class, with a `destroy` command (when the sensor is destroyed, it stops working for the rest of the temporal domain):

```
class Sensor
visible: measure, limit, over_limit, working;
temporal domain: Real;
items:
  TI sr_num: Natural;
  TD total measure: Real;
  TI limit: Real;
  TD over_limit : boolean;
  TD working : boolean;

  event destroy;

axioms:
  vars:
  s1, s2: Sensor;

  formulae:
  def_over_limit: over_limit <-> measure > limit;
  sr_num_unique: all s1, s2(s1.sr_num = s2.sr_num -> s1 = s2);

  destroy_def: destroy -> AlwF(not working);
end
```

Notice that neither class Sensor, nor class Controller as declared in Section 3 have to be modified if one switches from the lambda calculus semantics to the set theory semantics.

7 Genericity

HOT/SHOT classes can be parametric with respect to values of classes and with respect to classes. The header of a generic HOT/SHOT class has the following syntax:

```
class <class_name> ( <par_decls> )
```

where nonterminal `<par_decl>` is defined as follows:

```
<par_decls> := <par_decl>; <par_decls> |
               <par_decl>
<par_decl> := const <par_name> : <par_type> |
              domain <par_name>
```

For example, the following declarations are admissible in HOT/SHOT:

```
class NaturalRange (const lower_bound : Natural,
                    const upper_bound : Natural)
...
end

class Stack (domain ObjType)
...
end

class MaxDepthBinaryTree (const max_depth : Natural,
                          domain ObjType)
...
end
```

It probably makes sense to allow users to define some *constraints* on the parameters passed to the class (for example, one might require for class `NaturalRange` that the upper bound must be greater than or equal the lower bound, or, for class

`MaxDepthBinaryTree`, one might require that a total order is defined for type `ObjType`), but we leave this topic to further revisions of HOT/SHOT.

8 Conclusions and future work

This document introduced a revised version of the TRIO specification language, one in which the concepts of type and class coincide.
We feel that the resulting language is extremely compact and clean, meaning that it is based on a small set of core concepts, and every other concept is derived from it.

A few issues have not been tackled in this document (or have been analyzed only superficially), but will be dealt with in the future.
Most notably, a more thorough discussion of genericity is in order, as we gave only a brief idea of how the new type system makes the concept of generic classes simpler with respect to usual TRIO.

In addition, we feel that the compactness and simplicity of its core is an important asset of the HOT/SHOT language. In particular, we intend to exploit these important characteristics to rigorously introduce the notions of inheritance and subtyping in HOT/SHOT.
In fact, we feel that, while the concepts of inheritance and subtyping have been studied for many years now, and many languages (programming or not) include a notion of either (or both) of them, the way they have been dealt with is still largely unsatisfactory.
More precisely, we plan on introducing subtyping in HOT/SHOT as a constrained (semantic) variant of inheritance.

Finally, the concepts of inheritance and subtyping will evolve into a notion of refinement for HOT/SHOT classes, which will add a method on top of the rigorously-defined aforemenetioned ideas.

9 References

[And86] P. B. Andrews. *An Introduction to Mathematical logic and type theory: to truth through proof.* Academic Press. 1986.

[C3M399] Ciapessoni, E., Coen-Porisini, A., Crivelli, E., Mandrioli, D., Mirandola, P. and Morzenti, A. 1999. From Formal models to formal based methods: an industrial experience, *ACM Transactions on Software Engineering and Methodologies 8, 1*, 79-113.

[CPRM03] A. Coen-Porisini, M. Pradella, M. Rossi, D. Mandrioli. *A Formal Approach for Designing CORBA-based Applications*, ACM Transactions On Software Engineering and Methodology, vol. 12, n. 2, April 2003.

[Men97] E. Mendelson. *Introduction to Mathematical Logic*. Lewis Publishers, 4th edition, 1997

[OS99] S. Owre, N. Shankar. *The Formal Semantics of PVS*. Technical Report CSL-97-2R. SRI International. March 1999.

10 Appendix A: `Invoice` and `Sensor` revised

Section 6 introduced new operations (in set theory terms) to the `Invoice` and `Sensor` classes that were absent in their original definitions given in Section 3. This appendix presents how the original declarations of Section 3 could be modified to include the new items, if the semantics of HOT is given in terms of typed lambda calculus.

The new version of the `Invoice` class, with a `copy` operation (the `copy` operation is partial, because it may not always be invoked: when it is invoked, it returns a copy of the `Invoice`):

```
class Invoice
visible: sr_num, amount, date;
items:
  TI sr_num : Natural;
  TI amount : Real;
  TI date : Date;

  TD partial copy : Invoice;

axioms:
  vars:
  i1, i2: Invoice;

  formulae:
  sr_num_unique: all i1, i2(i1.sr_num = i2.sr_num -> i1 = i2);

  copy_def: i1.copy = i2
            ->
            i1.sr_num <> i2.sr_num &
            i1.amount = i2.amount &
            i1.date = i2.date;
end
```

An `Invoice_DB` object has an event such that, when it occurs, a copy of a certain invoice is made:

```
class Invoice_DB
visible: copy_invoice;
items:
  TD invoices(Natural) : Invoice;
  event copy_invoice(Natural);

axioms:
  vars:
  n1, n2 : Natural;
  i: Invoice;

  formulae:
  copy_invoice_def: copy_invoice(n1) &
                    invoices(n1).copy = i
                    ->
                    ex n2(n2 <> n1 & NowOn(invoices(n2) = i);
end
```

A new version of the `Sensor` class, with a `destroy` command (when the sensor is destroyed, it stops working for the rest of the temporal domain):

```
class Sensor
visible: measure, limit, over_limit, working;
temporal domain: Real;
items:
  TI sr_num: Natural;
  TD total measure: Real;
  TI limit: Real;
  TD over_limit : boolean;
  TD working : boolean;

  event destroy;

axioms:
  vars:
  s1, s2: Sensor;

  formulae:
  def_over_limit: over_limit <-> measure > limit;
  sr_num_unique: all s1, s2(s1.sr_num = s2.sr_num -> s1 = s2);

  destroy_def: destroy -> AlwF(not working);
end
```