# A Graph-Based Implementation for Mechanized Refinement Calculus of OO Programs

Zhiming Liu[1], Charles Morisset[2], and Shuling Wang[1,3]

[1] UNU-IIST, P.O. Box 3058, Macau S.A.R., China
[2] Royal Holloway, University of London
Information Security Group
Egham, Surrey TW20 0EX, U.K.
[3] State Key Lab. of Computer Science
Institute of Software, Chinese Academy of Sciences

**Abstract.** This paper extends the mechanization of the refinement calculus done by von Wright in HOL, representing the state of a program as a graph instead of a tuple, in order to deal with object-orientation. The state graph structure is implemented in Isabelle, together with definitions and lemmas, to help the manipulation of states. We then show how proof obligations are automatically generated from the rCOS tool and can be loaded in Isabelle to be proved. We illustrate our approach by generating the proof obligations for a simple example, including object access and method invocation.

**Keywords:** Isabelle, Proof obligations, rCOS, Theorem proving.

## 1 Introduction

Software verification is about demonstrating that an *implementation* (executable code) of the software meets its *specification* (formal description of the behavior) and several techniques are available in order to achieve this goal. Testing and model checking usually aim to verify if a property holds on a subset of instances of a program, or on a model of the program, respectively, while theorem proving aims to build the proof of correctness, that is, the semantics of the implementation logically implies the specification. For all these techniques, there are several challenges to address:

*i*) An increasing number of software is written using the OO approach, and therefore the execution states of a program are complex, due to the complex relations among objects, aliasing, dynamic binding, and polymorphism. This makes it hard to understand and reason about the behavior of the program.

*ii*) When a tool is provided to help the development of software, it should offer an environment where the user can specify, analyze, implement and verify a program. Therefore, the different verification techniques need to be integrated within the tool.

*iii*) In general, it is not possible to automatically verify if an implementation satisfies a specification, therefore the tool is required to guide the user through the different steps of the verification process.

J. Davies, L. Silva, and A. Simão (Eds.): SBMF 2010, LNCS 6527, pp. 258–273, 2011.
© Springer-Verlag Berlin Heidelberg 2011

The refinement for Component and Object Systems (rCOS) [10,16] method provides an interesting framework to address these challenges. Firstly, rCOS has a formal semantics based on an extension of the Unifying Theories of Programming (UTP) [11] to include the concepts of components and objects. The graph-based operational semantics [12] has recently been defined for OO programs. Secondly, the rCOS tool (available at `http://rcos.iist.unu.edu`) provides a UML-like multi-view and multi-notational modeling and design platform. In particular, two verification processes are already implemented: the automated generation of test cases to check the robustness of a component [15], and the automated generation of CSP processes to verify the compatibility between the sequence diagram and the state diagram of a contract [7]. Lastly, rCOS extends the refinement calculus [1,17], which is a program construction method, where a non-deterministic specification is incrementally refined to deterministic code, using pre-defined rules. This approach creates several refinement steps, which fill the gap between the specification and the implementation, therefore reduces the proof complexity, by replacing a single complex proof by many simpler ones.

**Related Work.** The mechanization of the refinement calculus was firstly done in [24], which has been extended to include pointers [2] and also object-oriented programs [4,20]. In particular, a refinement calculus has been defined for Eiffel contracts [19], and encoded in PVS [18]. Although this approach addresses a similar issue than the one exposed here, the authors encode the calculus using a shallow embedding, that is, a class in Eiffel is encoded as a type in PVS, a routine in Eiffel is encoded as a function in PVS, etc. Proofs of refinement are then done over PVS *programs* rather than PVS *terms*, and so require the understanding of the underlying semantics of PVS. We use here a deep embedding, following [24], and the proofs of refinement are done, roughly speaking, over the abstract syntax tree of the original program, and so only require to know how to write a proof in Isabelle/Isar. The Program Refinement Tool [3] provides a deep embedding of a refinement calculus, and even if it does not support OO programs natively, it could be extended with an existing formalization which does [22]. However, rCOS also provides a semantics for components, and even if we do not address in this paper the issue of verification of component protocols, this work is part of a larger framework where other verification techniques exist [21]. In other words, the work presented here is not a standalone tool, but adds up to a collection of tools that helps a developer to specify, implement and verify an application.

On the other hand, different memory models for object-oriented programs have been encoded in theorem provers [9,23,13]. However, the memory in these approaches is either modelled as a function from addresses or pointers to values or using records to represent objects. Although such a modelling is very expressive, and has been shown to be adapted to automated demonstration, we propose here a representation of the memory by a directed and labeled graph, that is intuitive than a representation by a function or set of records. The graph structure helps in the formulation of properties and carrying out interactive proofs.

**Contribution.** The main contribution of this paper is twofold. The first one is the implementation in Isabelle of rCOS using a graph to represent the state of a program. It is an extension of the mechanization of the refinement calculus done by von Wright in HOL [24]. The other one is the automated generation of proof obligations for refinement steps in rCOS. Concretely, we have implemented in the rCOS tool a plug-in which is able to take the bodies of two methods defined with the rCOS language, translate each body into a predicate transformer in Isabelle and generate automatically an Isabelle lemma stating that the first predicate transformer refines the second one (the proof still has to be done by the user). This process is linked with the definition of refinement steps within the tool. The most technical part of this work is the translation from rCOS designs to Isabelle statements.

**Organization.** Section 2 introduces the rCOS language. Section 3 recalls the previous mechanization of the refinement calculus. Section 4 presents the graph-based representation of the memory and its implementation in Isabelle/HOL. Section 5 extends the mechanization of refinement calculus to object-orientation. Section 6 presents an example to illustrate our approach. Finally, Section 7 concludes and presents the future work.

## 2 rCOS

The rCOS method consists of two parts: a component/object-oriented language, with a formal semantics, and a modeling tool, enforcing a use-case based methodology for software development, providing tool support and static analysis. We give here a brief description of the language, and we refer to [5,6] for further details.

### 2.1 Language

The rCOS language is an extension of UTP [11], to include object-oriented and component features. The essential theme of UTP that helps rCOS is that the semantics of a program P (or a statement) in any programming language can be defined as a predicate, called a *design*. The most general form of a *design* is a pair of pre- and post-conditions [1,17], denoted as $p(x) \vdash R(x,x')$, of the *observable* x of the program. Its meaning is defined by the predicate $p(x) \wedge ok \Rightarrow R(x,x') \wedge ok'$, which asserts that if the program executes from a state where the *initial value* x satisfies p(x), the program will terminate in a state where the final value x' satisfies the relation R(x,x') with the initial value x. Observables include program variables and parameters of methods or procedures. The Boolean variables ok and ok' represent observations of termination of the execution preceding the execution of P (*i.e.* ok is true) and the termination of the execution of P (*i.e.* ok' is true), respectively. Non-deterministic choice is defined as $d_1 \sqcap d_2$, where $d_1$ and $d_2$ are designs.

The language also includes traditional imperative statements, and a design can be: SKIP and CHAOS, an assignment p := e, where p is a navigation path,

and e is an expression; a conditional statement $d_1 \lhd$ b $\rhd d_2$, where $d_1$ and $d_2$ are designs and b is a boolean expression; a sequence $d_1; d_2$, where $d_1$ and $d_2$ are designs; a loop do $b$ $d$, where $d$ is a design and $b$ is a boolean expression; a local variable declaration and un-declaration var T x = e ; end x, where T is a type.

Objects are created through the command $C$.new (p), where $C$ is a class type and p is a navigation path. It creates a new object of type $C$ whose attributes have the initial values as declared in $C$, and attaches the new object to p. A method invocation has the form e.m(ve, re), where m is a method and e, ve, re are expressions. Intuitively, it first records the value of the actual value parameter ve in the formal value parameter of m, and then executes the body of m. At the end it returns the value of the formal return parameter to the actual return parameter re.

The rCOS language includes the notion of components, which provide or require contracts. A contract includes an interface (a set of field and method declarations), the specification of each method and a protocol stating the allowed sequences of method calls (for instance, for a buffer, the method put must be called before the method get). A component provides a contract through a class, which is the usual notion of class, where each method has to be defined using a design. Note that the design of a method does not have to be executable in general, only if the user wants to generate Java code, since executable rCOS designs are quite similar to Java programs. For instance, all the following examples are correct rCOS programs.

```
class A {
  int x;
  public m(int v) { x := v }
}

class B1 {
  A a;
  public foo() {
   [ true |− a.x' = 2 ∨ a.x' = 3 ] }
}
```

```
class B2 {
  A a;
  public foo() {
   [ true |− a.x' = 1 ] ;
   a.x := a.x + 1 }
}
class B3 {
  A a;
  public foo() {
    a.m(1) ; a.x := a.x + 1 }
}
```

The method $B_1$::foo is abstract and non-deterministic: it just specifies, under the true precondition, that the value of the field x of the field a should be either equal to 2 or to 3. The method $B_2$::foo mixes abstract pre/post-conditions with a concrete assignment while $B_3$::foo is completely concrete and could be directly translated to Java. In this example, we can see that $B_1$::foo is refined by $B_2$::foo, which is refined by $B_3$::foo. We detail in the following section the mechanization of the notion of refinement.

## 3 Mechanized Refinement

The refinement calculus [1,17] is a program construction method, where a non-deterministic specification is incrementally refined to deterministic code, using

pre-defined rules. This calculus has been fully implemented with a theorem prover, HOL, in [24,8] and then extended, in particular in [14], which introduces, among others, procedures and recursive functions. The implementation is actually the definition of a predicate transformer semantics, i.e. the weakest precondition. For any design $d$ and any predicate $q$ over states, the function $d\ q$ is defined as the weakest precondition that should be true on states before executing $d$ such that $q$ holds after executing $d$. Therefore, a design is usually considered as a predicate transformer, since it takes a predicate ($q$) as input and returns another predicate (the weakest precondition of $q$). We recall here the definitions of assignment and refinement from [24]. We use `State` to represent the type of program states. We introduce first the types of predicates over states and the type of predicate transformers.

**types** State pred = State ⇒ bool
State predT = State pred ⇒State pred

The `assign` predicate transformer takes a function `e`, which takes a state and returns the state where the corresponding assignment is done. The weakest precondition of a predicate `q` is calculated by checking `q` on a state where the assignment has been done.

**definition** assign :: (State ⇒ State) ⇒ (State predT) **where**
assign e q ≡ λu. q (e u)

A design `c1` is refined by a design `c2` if, and only if, the weakest precondition of `c1` implies the one of `c2` for any state.

**definition** implies :: (State pred) ⇒ (State pred) ⇒ bool **where**
implies p q ≡ ∀ u. (p u) ⇒ (q u)

**definition** refines :: (State predT) ⇒(State predT) ⇒bool (**infixl** ref 40) **where**
c1 ref c2 ≡ ∀ q. (implies (c1 q) (c2 q))

In addition to the definition of the semantics, helpful theorems are introduced in [24]. For instance the one stating that the loop `do g c` refines the loop `do g d` if the design `c` refines the design `d`.

**theorem** do_ref : d ref c ⇒ (do g d) ref (do g c)

Although the previous definitions do not directly depend on the structure of the state, the latter is defined as a tuple [24], where each element of the tuple is the value of a variable of the program. For instance, if a program has two variables $x$ and $y$, set respectively to 1 and 3, the state of such a program is the pair $(1, 3)$. The names of the variables are therefore lost in the translation, and any operation concerning $x$ has to be translated as an operation concerning the first element of the pair. As a result, dealing with local variables and method calls implies to extend and narrow the state, respectively. Moreover, this approach does not directly handle references and therefore such a representation for states cannot be applied for OO programs. The usual way to tackle this issue is to represent OO states as records or as a function from pointers to values [2,19,4,20]. A new approach uses graphs instead [12], and we present it in the next section.

## 4 Graph Representation

In [12], the state of a program is represented as a directed labeled graph. We recall here the definition of such a graph and give its implementation in Isabelle/HOL, together with basic operations to manipulate state graphs.

### 4.1 State Graph

A state graph describes the values of variables, together with a family of objects and their relations. Due to the existence of nested local variables and method invocations, we also need to describe scopes in state graphs. A scope is represented as a node in a state graph, called *scope node*. Two scope nodes are adjacent iff the scopes they represent are directly nested. They are connected by an edge labeled by \$, the one corresponding to the inner scope as the source and the other one as the target of the edge respectively. In particular, the top scope node with no incoming \$ edge represents the current scope, and is thus the current root of the state graph. For instance, in Fig. 1(a), $r$ is the root of the graph.

The outgoing edges of a scope node, except for the \$ edge, represent the variables defined in the corresponding scope. A non-scope node in a state graph represents an object or a primitive datum, called *object node* and *value node*, respectively. An object node is labeled by the runtime type of the object, while a value node is labeled by the primitive value. An outgoing edge from an object node is labeled by a field name of the source object and refers to the target object representing the value of this field. There is no outgoing edge from a value node.

Let $\mathcal{A}$ be the set of names of variables (including the special variable *this* which refers to the current object) and fields, and $\mathcal{A}^+ = \mathcal{A} \cup \{\$\}$. Let $\mathcal{C}$ be the set of classes and $\mathcal{W}$ the set of constant values. The formal definition of a *state graph* is then given as follows.

**Definition 1 (State Graph).** *A state graph is a rooted, directed and labeled graph $G = \langle V, E, T, F, r\rangle$, where*

- $V = R \cup N \cup L$ *is the set of nodes, where $R$ is the set of scope nodes, $N$ is the set of object nodes and $L$ is the set of value nodes,*
- $E \subseteq V \times \mathcal{A}^+ \times V$ *is the set of edges,*
- $T : N \to \mathcal{C}$ *is a mapping from object nodes to types,*
- $F : L \to \mathcal{W}$ *is a mapping from value nodes to values,*
- $r \in R$ *is the root of the graph and it has no incoming edges,*
- *starting from $r$, the \$-edges, if there are any, form a path such that except for $r$ each node on the path has only one incoming edge.*

All the nodes on the \$-path are scope nodes, the top of which is the root of the state graph. When a new scope is entered, a new node together with a \$-edge from it to the current root are pushed onto the \$-path; and when a scope is exited, the top node of the \$-path is popped out, together with all edges outgoing from it. As an illustration, Figure 1 (a) shows the state graph after the command a.b.x :=1; var **int** c=2; is executed. Moreover, Figure 1 (b) shows a state graph with recursive objects, where the types and values of nodes are ignored.

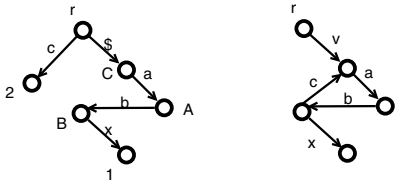


**Fig. 1.** (a): a state graph; (b): a state graph with recursive objects

### 4.2 Graph Implementation

A state graph requires the sets of scope nodes, object nodes and value nodes to be disjoint. We therefore define the datatype `vertex` as the union of four disjoint types: `onode` for object nodes, `snode` for scope nodes, `vnode` for value nodes, and $\perp$ for the undefined vertex, the latter being introduced mainly for the definition of graph operations.

**datatype** vertex = N onode | R snode | L vnode | $\perp$

A state graph is defined as the cartesian product of four elements:

**types**
edgefun = vertex ⇒ label ⇒ vertex
onodefun = onode ⇒ctype
vnodefun = vnode ⇒val
graph = edgefun ∗ onodefun ∗ vnodefun ∗ snode list

The first element of a graph is the function `edgefun`, which given a vertex $a$ and a label $x$, returns the vertex $b$ if $(a, x, b)$ is in the set of edges, $\perp$ otherwise (note that $\perp$ is different from the node corresponding to *null*). Such a definition automatically ensures the determinism for edges. The second (resp. the third) element of a graph corresponds to the function $T$ (resp. the function $F$). The last element is a list of scope nodes, where the head of the list stands for the current root, the second element stands for the previous one, and so on. This representation of scope nodes allows us not to implement $-edges.

In order to ensure that there is no edge starting with the undefined vertex $\perp$, we introduce the following property.

**definition** isGoodFunction:: graph ⇒bool **where**
isGoodFunction g ≡ ∀ x. (getEdgeFun g) $\perp$ x = $\perp$

where `getEdgeFun g` is used to get the first component of `g`.

Moreover, we need some properties concerning the list of scope nodes. Indeed the `snode` list is well-formed (and in this case the graph satisfies `isCorrectSnode`, which we do not detail here due to lack of space) if, and only if,: (1) the scope nodes cannot be undefined; (2) all the scope nodes are unique; (3) from each scope node, there must exist at least one outgoing edge; (4) a scope node can never be the target of an edge. Thus, a graph `g` is *well-formed*, denoted by `wfGraph g` if, and only if, it satisfies `isGoodFunction` and `isCorrectSnode`.

### 4.3 Graph Operations

This section gives the implementation of some basic graph operations. Due to space limitation we only present a subset of definitions below, more can be found at `http://isg.rhul.ac.uk/morisset/sbmf/graph_utl.thy`.

First, the function `swingEdge` swings an edge with the given source and label to point to a new vertex by updating the `edgefun` of the graph. It will return the original graph when the edge to be swung does not exist, or when the new target is undefined.

**definition** swingEdge:: vertex ⇒ label ⇒ vertex ⇒ graph ⇒ graph **where**
swingEdge n x m g ≡ (**if** ((getEdgeFun g) n x) = ⊥ **then** g
**else if** m = ⊥ **then** g
**else let** new_EF = ((λ v. λl. (**if** (v = n & l = x) **then** m
**else** ((getEdgeFun g) v l))) **in** (new_EF, snd g))

We then introduce the type `path` as a list of labels, hence representing navigation paths. For implementation optimisation reasons, the path is actually stored as the reverse list of labels: for instance, the rCOS expression a.b.x is defined as the list `[''x'', ''b'', ''a'']`. Given a path `p` and a graph `g`, the function `getOwner` returns the vertex that the tail of the path `p` refers to in the graph. Formally, it returns ⊥ if the path is empty, the corresponding scope node of the variable if the path is limited to one element (using the function `getSnodeOfVar`) and otherwise, recursively gets the owner of the tail of the path, from which it gets the next vertex associated with the head of the tail.

**consts** getOwner :: path ⇒graph ⇒ vertex
**primrec** getOwner [] g = ⊥
getOwner (x#t) g = (**if** t = [] **then** (R (getSnodeOfVar x g))
**else** ((getEdgeFun g) (getOwner t g) (hd t)))

Starting from the source `getOwner p g` and the label `hd p`, we can reach the target vertex that the path `p` refers to in `g`, which is exactly the definition of the function `getVertexPath`. This operation corresponds to the general evaluation of path expressions in a program state.

**definition** getVertexPath :: path ⇒ graph ⇒ vertex **where**
getVertexPath p g ≡ (**if** p = [] **then** ⊥
**else** ((getEdgeFun g) (getOwner p g) (hd p)))

Finally, the function `swingPath` swings the last edge of a path in the graph to point to a new vertex. In other words, it sets a new value to a path in a state graph, and therefore, can be used for implementing assignments in rCOS. As defined below, it uses `getOwner` and `hd p` to find the source and the label of the last edge respectively, and then swings the edge by using `swingEdge` directly. When the path is empty, `swingPath` returns the original graph.

**definition** swingPath :: path ⇒ vertex ⇒ graph ⇒ graph **where**
swingPath p n g ≡ (**if** p = [] **then** g **else** swingEdge (getOwner p g) (hd p) n g)

This function has been proved to preserve the well-formedness, *i.e.* for any graph `g`, any path `p` and any non-scope node vertex `n`, if `g` is well-formed then `(swingPath p n g)` is also well-formed.

Furthermore, we prove that a path after being swung to a vertex will actually point to the vertex. Before stating the fact, we define a path `p` to be well-formed w.r.t. `g`, denoted by `wfPath p g`, if, and only if, the vertex of `p` exists in `g`, and the owner of `p` appears exactly once as a source in the list of edges along the path, the latest of which is exactly the property `isGoodPath p g`. For instances, in the state graph in Figure 1 (b), the paths v.a, v.a.b.c and v.a.b.x satisfy `isGoodPath`, while v.a.b.c.a and v.a.b.c.a.b.x do not. We discuss in the conclusion about the limitations caused by this constraint.

Finally, the theorem `swingPathChangeVertex` is proved under three assumptions: `g` is well-formed; `p` is well-formed w.r.t. `g`; and `n` is not the undefined vertex. In particular, with the assumption `isGoodPath`, we can prove a key fact that the owner of the path `p` is not changed after it is swung.

**theorem** swingPathChangeVertex :
wfGraph g ⇒wfPath p g ⇒n ≠ ⊥ ⇒getVertexPath p (swingPath p n g) = n

We will introduce functions for implementing local variable (un-)declaration. Adding edges representing variables in a graph is done by the function `addVarList`, which given a function `f` of type `labelVerF` (mapping variables to their initial values) and a graph `g`, adds edges labeled with variables `lp` in the domain of `f` to the current root of `g` and lets them point to the associated vertices `f lp`.

**definition** addVarList:: labelVerF ⇒ graph ⇒ graph **where**
addVarList f g ≡ **case** (getSnodeList g) **of** [] ⇒ g
| a#p ⇒((λ vp. λ lp. (**if** (vp = (R a) & (∃ v. v ≠ ⊥ & f lp = v)) **then** (f lp)
**else** ((getEdgeFun g) vp lp))), snd g)

Moreover, we define a function `createSnode` which pushes a new scope node into the scope node list of a graph. The function `Vars` then combines the operations of creating a scope node and adding edges, and therefore implements local variable declaration.

**definition** Vars:: labelExpF ⇒graph ⇒ graph **where**
Vars f g ≡ addVarList (expToNode f (createSnode g)) (createSnode g)

where `labelExpF` maps variables to their initial expressions, and `expToNode` translates a function of type `labelExpF` to the corresponding one of type `labelVerF`, by changing in the range the expressions to their values in the graph. Finally, we introduce a function `removeSnode` which removes the top root from the scope node list, and in consequence all edges outgoing from the scope node. It implements local variable un-declaration.

**definition** removeSnode :: graph ⇒graph **where**
removeSnode g ≡ **case** (getSnodeList g) **of** [] ⇒ g
| t#q ⇒(λ v. λ l. (**if** (v = R t) **then** ⊥ **else** (getEdgeFun g) v l),
getOnodeFun g, getVnodeFun g, q)"

where the functions `getOnodeFun` and `getVnodeFun` are used to get the second and third components of a graph respectively.

We are now in position to introduce the function `addObject` which creates a new vertex (object) in a graph. Given a path `l`, a class type `s` and a graph `g`, this function first gets a fresh object node of type `s` not in `g`, done by `getNodeFromType`; then swings the path `l` to refer to the new vertex by using `swingPath`; and finally, attaches the attributes of class `s` to the new node and initialises them, which is implemented by the function `addAttrs`. In the definition, the function `getAttrsOfCtype` is used to extract the attribute information from class `s`.

**definition** addObject:: path ⇒ctype ⇒ graph ⇒ graph **where**
addObject l s g ≡ **let** v = (N (getNodeFromType s g)) **in** addAttrs v
(expToNode (getAttrsOfCtype s) g) (swingPath l v g)

The functions `removeSnode`, `Vars` and `addObject` are proved to preserve the graph well-formedness, and furthermore, the last two are proved to ensure that variables or attributes are initialised with the correct values.

## 5 Refinement of rCOS Designs

The graph-based representation of the memory presented in the previous section allows us to extend the mechanization of the refinement calculus presented in Section 3 to deal with object-orientation. Since we only consider well-formed graphs and paths, we integrate these conditions into the weakest precondition of each command. The complete definition of the refinement calculus for all constructs can be found at `http://isg.rhul.ac.uk/morisset/sbmf/rcos.thy`.

### 5.1 Primitive Designs

**Pre/post-condition.** The definition of the non-deterministic assignment is changed to include the well-formedness checks.

**definition**
nondass :: (graph ⇒ graph pred) ⇒ path list ⇒ (graph pred) ⇒ (graph pred) **where**
nondass P l q ≡ (λv. (wfGraph v) & (wfPathl l v) & (∀ v1. P v v1 ⇒ q v1))

where `wfPathl l v` is true if, and only if, every path in `l` satisfies `wfPath`. This list of paths corresponds to all the paths appearing in the post-condition. A pre/post-condition is then an assertion followed by a non-deterministic assignment.

**definition** pp :: (graph pred) ⇒ (graph ⇒ graph pred) ⇒ path list ⇒
(graph predT) **where**
pp p r l ≡ assert p ; nondass r l

where `assert` is the standard definition for the assertion. For instance, the pre/post-condition [ **true** |− a.b.x'=2] is translated into the following statement

pp (true) (λ g. λ g1.((getIntOfPath *a.b.x* g1) = 2)) [*a.b.x*]

where, for the sake of readability, we write a path as in code, e.g. $a.b.x$ stands for the path `[''x'',''b'',''a'']`.

**Assignment.** The definition of the assignment is changed as follows.

```
definition assign :: path ⇒ exp ⇒ (graph pred) ⇒ (graph pred) where
  assign p e q ≡ λu. wfGraph u & wfPath p u & wfExp e u &
                     q (swingPath p (getNodeExp e u) u)
```

where the path `p` is assigned to the expression `e`, which is required to be well-formed. The function `getNodeExp` returns the value of an expression, which is defined using `getVertexPath` when the expression to be evaluated is a path, otherwise itself when it is a constant value.

**Local Declaration and Un-declaration.** The commands `begin` and `end` declare/initialize new local variables and terminate them, respectively.

```
definition begin :: labelExpF ⇒(graph pred) ⇒ (graph pred) where
begin f q ≡ λu. wfGraph u & wflabelExpF f u & q (Vars f u)

definition end :: (graph pred) ⇒ (graph pred) where
end q ≡ λu. wfGraph u & q (removeSnode u)
```

where `f` is a well-formed function of type `labelExpF`, which means that for each local variable, it is initialised by a well-formed expression in `f`.

The command `locdec` defines the block for local declaration and un-declaration, where `f` is the same as above and `c` is the body of the block.

```
definition locdec :: labelExpF ⇒(graph predT) ⇒(graph predT) where
locdec f c ≡ begin f; c; end
```

**Method Invocation.** The command `method` implements a method invocation with the help of the command `locdec`.

```
definition method :: (label * exp) list ⇒ (graph predT) ⇒(graph predT) where
method l c ≡ locdec (getLabelExpF l) c
```

where `l` is of type `(label * exp) list`, each pair consisting of a formal value parameter and the corresponding actual value parameter of the method, and `c` is the method body followed by the assignment from the formal return parameter to the actual return parameter. In the `method` command, the function `getLabelExpF` translates a list of pairs of type `label * exp` to the corresponding mapping of type `labelExpF` (i.e. `label ⇒exp`). For instance, the method call a.m(1) in the example of Section 2 is translated as:

method [(*this*, Path *this.a*), ($v$, Val (Zint 1))] ;assign *this.x* Path $v$

When the method is called, the variable *this* is initialised by *this.a* (the caller), and $v$ by 1. Note that with this approach, recursive method calls are not directly handled, and require the definition of a fix-point, which we do not consider here.

**Object Creation.** The command `object` implements object creation $s$.new ($p$):

```
definition object :: path ⇒ ctype ⇒ (graph pred) ⇒ (graph pred) where
object p s q ≡ λu. wfGraph u & wfPath p u & wflabelExpF (getAttrsOfCtype s) u
                    & q (addObject p s u)
```

### 5.2 Composite Designs

With the predicate transformer semantics, the definitions of the composite designs, like the sequential composition, the loop or the conditional statement, do not depend on the representation of the memory state. Hence, we can directly re-use the definitions and theorems from [24]. For instance, the sequential composition `c; d` is refined by `e; f` if `c` is refined by `e` and `d` is refined by `f` and `c` is monotonic, and in fact, we have proved that all basic commands (i.e. `nondass`, `pp`, `assign`, `begin` and `end`) are monotonic, and the compound constructs `locdec, method, cond, do, seq` preserve monotonicity with respect to their subcomponents. Moreover, the other constructs such as the conditional `cond` and the loop `do` preserve refinement with respect to their subcomponents. By applying these theorems, we can refine a program by repeatedly refining its subcomponents, and then prove that the new generated program is a refinement of the old one.

## 6 Application

We describe in this section how to use the mechanization of the refinement calculus within the rCOS tool.

### 6.1 Tool Refinement

Refining a model is, by definition, a dynamic process: a new model is generated from a previous one, by applying some refinement rules. The main challenge is then to be able to consider both models at the same time, in order to generate the corresponding proof obligations. When the refinement concerns only method bodies, the rCOS tool provides a simple way to define a refinement operation. Firstly, a class is created, and stereotyped with a specific kind of refinement, for instance refining automatically every [**true** |− x'=e] by x := e. The most general refinement is the manual refinement, where the user provides an operation, its old design (mainly for sanity checks), and the refining design. In a second step, the user can, at any time, apply such a refinement by right-clicking on the corresponding class and selecting the "refine" operation, and the tool will then transform the model accordingly.

### 6.2 Provided Lemmas

In addition to the theorems introduced in [24], we provide lemmas corresponding to refinement steps. For instance, the lemma stating that for any path $p$ and any integer expression $n$, [**true** |− p'=n] is refined by p := n is defined as:

```
lemma ref_pp_assign :
  pp (true)  (λ g. λ g1.((getNatOfPath g1 p) = n)) [p] ref
  (assign p (Val (Zint n)))
```

The proof of this lemma, together with the proofs of other useful lemmas, can be found at `http://isg.rhul.ac.uk/morisset/sbmf/rcos_lib.thy`.

Another example is the Expert Pattern, which is an essential rule for OO functionality decomposition by delegating responsibilities through method calls to the objects, called the experts, that have the information to carry out the responsibilities. For instance, defining a setter for a field is a special case of the Expert Pattern, and therefore a refinement. In this case, we have proved, with the theorem `EPIsRefOne`, that the method bar() { p.x := n } is refined by the method bar() { p.m() } where m () {**this**.x := n} is a method of p, for any non-empty path `p` and constant n; a similar theorem `EPIsRefTwo` is provided, when the setter takes the value as an argument, that is, the method bar() { p.x := n } is refined by the method bar() { p.m(n) } where m (T v) {**this**.x := v} is a method of p, for any primitive type `T` and parameter `v`.

### 6.3 Example

Let us consider the examples given in Section 2. The user first defines the classes $A$ and $B_1$ and then introduces a manual refinement, concerning the operation foo, where the old design is $d_{old}$ =[**true**|−a.x'=2 ∨ a.x'=3] and the new design is $d_{new}$ ={a.m(1) ; a.x := a.x + 1}. When applying the refinement, the class $B_3$ is obtained. However, proving directly that $d_{old}$ is refined by $d_{new}$ might be complex, as several steps of refinement are involved. The user can either decompose the refinement proof in Isabelle or directly in the rCOS tool. Indeed, the latter allows one to easily compose refinement steps. In this example, the user can introduce the manual refinement $r_1$ of [**true**|−a.x'=2 ∨ a.x'=3] to [**true**|−a.x'=2]; the automatic refinement $r_2$ of pre/post-conditions to assignments; the the manual refinement $r_3$ of a.x := 2 to a.x:=1; a.x := a.x+1; the expert-pattern refinement $r_4$ on a.x := 1.

The proof obligations for each step can be generated automatically. For instance, the proof obligation corresponding to $r_3$ is the following statement:

```
assign this.a.x (Val (Zint 2)) ref
(assign this.a.x (Val (Zint 1)) ;
  (assign this.a.x (Plus (Path this.a.x) (Val (Zint 1)))))
```

The refinement $r_1$ strengthens the post-condition of the design, so the proof is quite straight-forward. The proofs of $r_2, r_3$ and $r_4$ directly follow from the lemmas described in the previous subsection. Finally, using the fact that the refinement relation is transitive, we can prove that $d_{old}$ =[**true**|−a.x'=2 ∨ a.x'=3] is refined by $d_{new}$ ={a.m(1) ; a.x := a.x + 1}. The complete proofs can be found at `http://isg.rhul.ac.uk/morisset/sbmf/example.thy`. Except the proofs of $r_1$ and $r_3$, all the other proofs could be derived automatically, since automatic

transformations are used, meaning we can directly use the lemmas associated with these transformations. Such an automatic association is, as we say in the conclusion, a future work.

## 7 Conclusion

The approach presented in this paper allows a user of the rCOS tool to automatically generate proof obligations of design refinement. The generated statements are defined using the predicate transformer semantics mechanized in [24] extended here to support object-oriented programs, thanks to a graph-based representation of the memory. A library of lemmas and theorems is available to the user in order to help her to prove the generated statements, concerning for instance the graph operations or the refinement calculus. There are two major strengths to this approach. The first one is the seamless integration within the rCOS tool, hence making the process transparent for the user, who can design a software using UML, with the proof obligations being automatically generated. Of course, these obligations still have to be discharged, but using a more generic back-end, like Why [9], would generate proof statements both for interactive theorem provers and automated demonstrators. However, since we are using higher-order terms, automated demonstration might not be very efficient, therefore we need to keep providing lemmas corresponding to refinement steps, especially the ones concerning OO concepts. This issue poses the one of the scalability, since large programs usually involve a large number of refinement rules and a statement containing a large number of rules can only be proven in a reasonable amount of time and space if each rule has been proven for the general case. The proof would then only be a succession of application of simple rules, which is the main strength of the refinement calculus. We therefore believe that the effort should be made to provide the developer with a large number of rules already proven rather than trying to automatically prove a complex transformation.

The other strength of this approach is the definition of the graph-based semantics. Although using a graph is not strictly more expressive than using records or a function from pointers to values, that is, it does not allow one to express more programs, the properties of a state can be expressed in a different, more abstract and intuitive way. In particular, the graph model helps the formulation and understanding of properties in the first place and on the other hand provides intuition for construction of their proofs to be checked by the theorem prover. In practice, the function `getEdgeFun` of a graph is conceptually close to a function from pointers to values: each object node is a pointer and each value node is a value. In other words, a graph can be seen as an extensional definition of a usual function from pointers to value. However we believe that a more abstract view of the memory helps reasoning about the state of the program. For instance, stating that a path `p` is alias-free in a state `g` is simply done by stating that there is no path `p2` different from `p` such that `getVertexPath p g = getVertexPath p2 g`.

Several limitations need however to be addressed, for instance the assumption `isGoodPath` in the theorem `swingPathChangeVertex` or in the weakest precondition

of the statements, which may be too strong. Intuitively, this theorem still holds without it but the proof is more complex, since in general we lose the fact that the owner of the path is the same before and after swinging, as we showed on the examples. A possible lead to address this issue is to consider that any path which does not satisfy `isGoodPath` can be "reduced" to a path which does. For instance, on Figure 1 (b), the path v.a.b.c.a points to the same object that v.a points to, but v.a satisfies `isGoodPath`.

Moreover, more designs need to be implemented in the translation process, for instance recursive method calls. However, we can extensively reuse [8,14], defined for imperative programs, by extending them to object-oriented programs. The method call does not currently support dynamic binding, but it could be done by looking up the actual type of the caller from the second element `onodefun` of the graph to fix the called method body.

In general, the next step is to integrate this mechanism within model transformations, which is an on-going work in the rCOS tool [21]. The principal challenge in this work is for the tool to handled¡ several models at the same time: before, during and after refinement. Such modifications are easy to handle when changing the design of a single method, but more complicated when for instance changing or deleting classes. The idea is then to establish a correspondence between the modifications done in the tool and the refinement rules involved. Moreover, a better interaction with Isabelle could help the user in the decomposition of the refinement steps. For instance, by using similar techniques than those used in automated demonstration, the tool could use the feedback from the theorem prover to ask the user to introduce more steps. We then could have a system where the user introduces a refinement rule, the tool tries to prove it automatically, if it cannot, it asks the user for an intermediary step, tries again, and so on until the whole rule is proved.

**Acknowledgment.** This work has been supported by the project GAVES of the Macao S&TD Fund, the 973 program 2009CB320702, STCSM 08510700300, and the projects NSFC-60721061, NSFC-60970031, NSFC-90718041 and NSFC-60736017. The authors would like to thank Volker Stolz for his useful remarks.

## References

1. Back, R.-J.: On the Correctness of Refinement Steps in Program Development. PhD thesis, Helsinki, Finland, Report A–1978–4 (1978)
2. Back, R.-J., Fan, X., Preoteasa, V.: Reasoning about pointers in refinement calculus. Technical Report 543, TUCS - Turku Centre for Computer Science, Turku, Finland (July 2003)
3. Carrington, D., Hayes, I., Nickson, R., Watson, G., Welsh, J.: A tool for developing correct programs by refinement. In: Proc. BCS 7th Refinement Workshop. Springer, Heidelberg (1996)
4. Cavalcanti, A., Naumann, D.A.: A weakest precondition semantics for refinement of object-oriented programs. IEEE Transactions on Software Engineering 26, 713–728 (2000)

5. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. Science of Computer Programming 74(4), 168–196 (2009); UNU-IIST TR 388
6. Chen, Z., Liu, Z., Stolz, V.: The rCOS tool. In: Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop, number CS-TR-1099 in Technical Report Series. Newcastle University (May 2008)
7. Chen, Z., Morisset, C., Stolz, V.: Specification and validation of behavioural protocols in the rCOS modeler. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 387–401. Springer, Heidelberg (2010)
8. Depasse, C.: Constructing Isabelle proofs in a proof refinement calculus. Research Report, UCL (2001)
9. Filliâtre, J.-C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (2003)
10. He, J., Liu, Z., Li, X.: rCOS: A refinement calculus of object systems. Theor. Comput. Sci. 365(1-2), 109–142 (2006); UNU-IIST TR 322
11. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs (1998)
12. Ke, W., Liu, Z., Wang, S., Zhao, L.: A graph-based operational semantics of oo programs. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 347–366. Springer, Heidelberg (2009)
13. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Trans. Program. Lang. Syst. 28(4), 619–695 (2006)
14. Laibinis, L.: Mechanised Formal Reasoning About Modular Programs. PhD thesis, Abo Akademi (2000)
15. Lei, B., Liu, Z., Morisset, C., Li, X.: State based robustness testing for components. In: FACS 2008. ENTCS, vol. 260, pp. 173–188. Elsevier, Amsterdam (2008)
16. Liu, Z., Morisset, C., Stolz, V.: rCOS: theory and tools for component-based model driven development. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 62–80. Springer, Heidelberg (2010)
17. Morgan, C.: Programming from specifications, 2nd edn. Prentice Hall International, Englewood Cliffs (1994)
18. Paige, R., Ostroff, J., Brooke, P.: Formalising eiffel references and expanded types in pvs. In: Proc. International Workshop on Aliasing, Confinement, and Ownership in Object-Oriented Programming (2003)
19. Paige, R.F., Ostroff, J.S.: ERC – An object-oriented refinement calculus for Eiffel. Form. Asp. Comput. 16(1), 51–79 (2004)
20. Sekerinski, E.: A type-theoretic basis for an object-oriented refinement calculus. In: Formal Methods and Object Technology. Springer, Heidelberg (1996)
21. Stolz, V.: An integrated multi-view model evolution framework. Innovations in Systems and Software Engineering (2009)
22. Utting, M.: An object-oriented refinement calculus with modular reasoning (1992)
23. van den Berg, J., Jacobs, B.: The loop compiler for java and jml. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001)
24. von Wright, J.: Program refinement by theorem prover. In: BCS FACS Sixth Refinement Workshop – Theory and Practise of Formal Software Development. Springer, Heidelberg (1994)