Isabelle Proof Obligations for Graph-based Refinement of rCOS Programs

Zhiming Liu, Charles Morisset and Shuling Wang

May 2010

UNU-IIST Report No. 430

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

Govindan Parayil, Director a.i.

P.O. Box 3058
Macao

Isabelle Proof Obligations for Graph-based Refinement of rCOS Programs

Zhiming Liu, Charles Morisset and Shuling Wang

Abstract

This paper presents an approach for the automated generation of proof obligations for OO program refinement defined in the rCOS language. We first extend 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 that 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.

Zhiming Liu is a senior research fellow at UNU-IIST. He is the leader of the rCOS ("Refinement of Component and Object Systems") research group, working on foundations, advances and applications in information engineering, especially on component-based model driven software design and object-oriented programming.

Charles Morisset was a former postdoctoral fellow of UNU-IIST. He is currently a postdoctoral research assistant at Royal Holloway, University of London. His research interests include the theory of access control and the security of information systems in general, and the area of formal methods, in particular the use of theorem proving in system specification.

Shuling Wang is a postdoctoral fellow of UNU-IIST. Her research interest is in formal methods of object-oriented programs, component systems, and concurrent systems, including syntax, semantics, and verification.

The work is supported in parts by the Projects ARV and GAVES funded by Macau Science and Technology Development Fund.

Contents

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 aim to find a counter-example, that is, a possible execution which would violate the specification, 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) Tool support is provided to help the development of software, offering an environment where the user can specify, analyze, implement and verify a program. Therefore, the validation process needs to be integrated within the tool.

iii) The gap between the specification and the implementation might prevent the user to use automated techniques, and she is then required to guide the validation process through the different steps.

The refinement for Component and Object Systems (rCOS) [9, 14] method provides an interesting framework to address these challenges. Firstly, rCOS has a formal semantics based on an extension of UTP [10] to include the concepts of components and objects. The graph-based operational semantics [11] 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 [13], and the automated generation of CSP processes to verify the compatibility between the sequence diagram and the state diagram of a contract [5]. Lastly, rCOS extends the refinement calculus [1, 15], 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 effort.

Related Work The mechanization of the refinement calculus was firstly done in [18], and we extend this work, with the constructs for object-oriented programs. Refinement calculus for object-oriented programs has also been studied in the literature [16, 2] but to the best of our knowledge, they lack a mechanized tool support. Moreover, different memory models for objects exist, in particular Why/Krakatoa [7], but we believe that the graph-based semantics we have implemented is more intuitive and therefore helps the interactive proving process. Finally, there is an on-going work in VDM/Overture [8] to provide a theorem-prover support, but not in the context of the refinement calculus.

Contribution The main contribution of this paper 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, which is done by extending the mechanization of the refinement calculus done by von Wright in HOL [18]. The state of a program is here represented as a graph.

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 [3, 4] for further details.

2.1 Language

The rCOS language is an extension of the Unifying Theories of Programming (UTP) [10], 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, 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 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. A design can also be defined as:

- a conditional statement $d_1 \lhd e \rhd d_2$, where d_1 and d_2 are designs and e is a boolean expression,
- a non-deterministic choice $d_1 \sqcap d_2$, where d_1 and d_2 are designs,
- a sequence $d_1; d_2$, where d_1 and d_2 are designs,

- a loop $\star(e, d_1)$, where d_1 is a design and e is a boolean expression or
- a primitive command, such as an assignment, a variable declaration, a method call, SKIP or CHAOS.

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, 15] 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 [18, 6] and then extended, in particular in [12], 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 the assignment and the refinement from [18]. We introduce first the types of predicates over states and the type of predicate transformers.

```
types  'a pred = "'a ⇒ bool"
       'a predT = "'a pred ⇒ 'a 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 :: "('a ⇒ 'a) ⇒ ('a pred) ⇒ ('a pred)" where
 "assign e q ≡ λu. q (e u)"
```

A design `c1` is refined by a design `c2` iff the weakest precondition of `c1` implies the one of `c2` for any state.

```
definition implies :: "('a pred) ⇒ ('a pred) ⇒ bool" where
 "implies p q ≡ ∀ u. (p u) ⇒ (q u)"

definition refines :: "('a predT) ⇒ ('a 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 [18]. For instance the one stating that the loop $\star(g, c)$ refines the loop $\star(g, d)$ 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, it is implicitly defined as a tuple, 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. Moreover, this approach does not directly handle references and therefore such a representation for states cannot be applied for oo programs, so we need to introduce a new way. The usual way is to represent oo states as records, but a new approach, more intuitive and with a better description of the model is to use graphs instead [11], as we see in the next section.

4 Graph Representation

In [11], in order to handle objects, 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. Moreover, due to the existence of nested local variables and method invocations, we need to describe scopes in state graphs. A scope is represented as a node in a state graph, called *scope node*. The adjacent scope nodes are connected by an edge labeled by \$, and in particular, the top scope node with no incoming \$ edge represents the current scope, and is thus the current *root* of the state 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 and fields, and $\mathcal{A}^+ = \mathcal{A} \cup \{this, \$\}$, where $this$ is the special variable denoting the current object. Let $\mathcal{C}$ be the set of classes and $\mathcal{V}$ 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 $G = \langle N, E, T, F, r\rangle$, where*

- *$N = R \cup O \cup L$ is the set of nodes, where R is the set of scope nodes, O is the set of object nodes and L is the set of value nodes,*
- *$E \subseteq N \times \mathcal{A}^+ \times N$ is the set of edges,*
- *$T : O \to \mathcal{C}$ is a mapping from object nodes to types,*
- *$F : L \to \mathcal{V}$ 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.

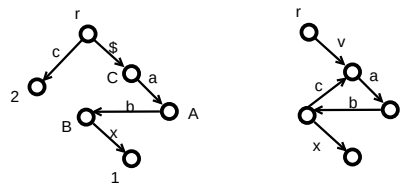

Figure 1: (a): a state graph; (b): a state graph with recursive objects

4.2 Graph Implementation

A state graph requires that the sets of scope nodes, object nodes and value nodes are disjoint. To implement this, we define the datatype `vertex` as the union of three disjoint types: `node` for object nodes, `root` for scope nodes, and `leaf` for value nodes.

datatype vertex = N node | R root | L leaf

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

types
edgefun = "vertex ⇒ label ⇒ vertex"
nodefun = "node ⇒ ctype"
leaffun = "leaf ⇒ val"
graph = "edgefun ∗ nodefun ∗ leaffun ∗ root 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 (where $\perp$ stands for the undefined vertex, different from the node corresponding to $null$). Note that with such a definition, the determinism for edges is automatically ensured for any state graph. 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 roots (scope nodes), where the head of the list stands for the current root, the second element stands for the previous root, and so on. This representation of roots 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) ⊥ x = ⊥

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

Moreover, we need some properties concerning the list of roots. Indeed the `root` list is well-formed (and in this case the graph satisfies `isCorrectRoot`, which we do not detail here due to lack of space) iff: (1) the root list is not empty; (2) the roots cannot be $\perp$; (3) all the roots are unique; (4) from each root, there must exist at least one outgoing edge; (5) a root can never be the target of an edge. Thus, a graph `g` is *well-formed*, denoted by `wf g` iff it satisfies `isGoodFunction` and `isCorrectRoot`.

definition wf::"graph ⇒ bool" **where**

"wf g ≡ isGoodFunction g & isCorrectRoot g"

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 in [11] and at `http://isg.rhul.ac.uk/morisset/ictac/grap`

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:: "graph ⇒ vertex ⇒ label ⇒ vertex ⇒ graph" **where**
"swingEdge g n x m ≡ (if ((getEdgeFun g) n x) = ⊥ then g
else if m = ⊥ then g
else **let** new_EF = ((λ v. λ l. (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 graph and a path, the function `getOwner` then returns ⊥ if the path is empty, the corresponding root of the variable if the path is limited to one element (using the function `getRootOfVar`) 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 :: "graph ⇒ path ⇒ vertex"
primrec "getOwner g [] = ⊥"
"getOwner g (x#t) = (if t = [] then (R (getRootOfVar g x))
else ((getEdgeFun g) (getOwner g t) (hd t)))"

Finally, the function `swingPath` swings the last edge of a path in the graph to point to a new vertex. 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 :: "graph ⇒ path ⇒ vertex ⇒ graph" **where**
"swingPath g p n ≡ (if p = [] then g else swingEdge g (getOwner g p) (hd p) n)"

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

Furthermore, we prove that a path after being swung to a vertex will actually point to the vertex. In order to prove this property, we first introduce the function `getVertexPath` which returns the vertex pointed to by the path in the graph.

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

Moreover, we define a path `p` to be well-formed w.r.t. `g`, denoted by `wfPath`, iff `p` is not empty, and `p` points to a vertex in `g`, and the owner of `p` appears exactly once as a source in the list of edges from the root to the vertex pointed to by the path, the latest of which is exactly the property `isGoodPath g p`. 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 well defined. 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 :
"wf g ⇒ wfPath g p ⇒ n ≠ ⊥ ⇒getVertexPath (swingPath g p n) p = n"

Adding edges representing variables in a graph is done by the function `addVarList`, which given a graph `g` and a function `f` of type `labelVerF` (mapping variables to vertices), adds edges labeled with variables `lp` in the domain of `f` to the root and makes them point to the associated vertices `f lp`.

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

Moreover, we define two functions respectively corresponding to entering or exiting a local scope: the function `createRoot` pushes a new root into the root list of a graph; the function `removeRoot` removes the top root from the root list, and in consequence all edges outgoing from the root. The function `Vars` combines the operations of creating a root and adding edges. Given a graph `g` and a function `f` of type `labelExpF` which maps from variables to expressions, it first creates a new root for `g`, and then adds edges labeled with variables in `f` to the new root to point to the vertices initialised in `f`. In the definition, the function `expToNode` translates a function of type `labelExpF` to the corresponding one of type `labelVerF`, by changing in the range the expressions to the vertices they represent in the graph.

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

Finally, the function `addObject` is defined for creating a new node (object). Given a graph `g`, a path `l`, and a class type `s`, it first gets a fresh node of type `s` that is not in `g`, done by `getNodeFromType`; then swings the path `l` to refer to the new node by using `swingPath`; and finally, attaches the attributes of class `s` to the new node and initialises them, the last of which is implemented by the function `addAttrs`. In the definition, the function `getAttrNodes` is used to extract the attribute information from class `s`.

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

Furthermore, we have proved properties for these functions, e.g. they preserve graph well-formedness

(except for `createRoot`), and `addVarList`, `Vars` and `addObject` ensure that variables or attributes are initialised with 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/ictac/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 ⇒ 'a pred) ⇒ path list ⇒ 'a pred ⇒ graph pred " **where**
"nondass P l q ≡ (λv. (wf v) & (wfPathl v l) & (∀ v1. P v v1 ⇒ q v1))"

where `wfPathl v l` is true iff 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 assignement.

definition pp :: "(graph pred) ⇒ (graph ⇒ 'a pred) ⇒ path list ⇒
'a pred ⇒ graph pred " **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 .((getNatOfPath g1 $a.b.x$) = 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 :: "(graph ⇒ graph) ⇒ path ⇒ (graph predT)" **where**
"assign e p q ≡ λu. wf u & wfPath u p & q (e u)"

where `p` is the path assigned; `e` is a graph to graph function. For instance, the rCOS design `a.b.x := 1` is translated into

assign (λg. swingPath g $a.b.x$ (getNodeN 1)) $a.b.x$;

where `getNodeN n` returns the vertex in `g` whose value is the natural `n`. Note that this function takes a `Nat` as input, which means we first need to know the type of an expression before translating it to Isabelle. Similar functions exist for the other types.

Local declaration and undeclaration

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. wf u & wflabelExpF u f & q (Vars u f)"

definition end :: "(graph pred) ⇒ (graph pred)" **where**
"end q ≡ λu. wf u & q (removeRoot u)"

where `f` is a function of type `labelExpF` that is required to be well-formed w.r.t. `u`, which means that for each associated expression in `f`, it is a path in `u`, or a constant.

The command `locdec` defines the block for local declaration and undeclaration, 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 q ≡ λu. ((begin f; c; end) q) u"

Method invocation

Intuitively, a method invocation of the form `e.m(ve, re)` first records the value of the actual value parameter `ve` in the formal value parameter of `m`, and then executes the body `c` of `m`. At the end it returns the value of the formal return parameter to the actual return parameter `re`. The command `method` implements this with the help of the command `locdec`.

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

where `l` is of type `(label * exp) list`, each pair of which is composed 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. For instance, the method call a.m(1) in the example of Section 2 is translated as:

method [($this$, Path $this.a$), (v, Val (Nat 1))]
  assign (λ g. swingPath g $this.x$ (getVertexPath g [v])) $this.x$"

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.

In the `method` command, the recursive function `getLabelExpF` translates such list to a function of type `labelExpF` which maps formal value parameters to corresponding actual value parameters of the method.

Object creation

The command `object` creates a new object of type `s`, and lets the path `p` refer to it.

definition object :: "path ⇒ ctype ⇒ (graph predT)" **where**
"object p s q ≡ λu. wf u & wfPath u p & q (addObject u p s)"

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 [18]. 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 action: 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 refined 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 [18], we present 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  (λ g . swingPath g p (getNodeN n)) p)"
```

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

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, as described in Table 1, the manual refinements r_1 and r_2, use the pre-defined refinements r_3 and r_4, and combine then into a single refinement operation.

The refinement r_1 strengthen the post-condition of the design, so the proof is quite straight-forward. The proof of r_3 is a bit more complex as it requires to manipulate the memory. The proofs of r_2 and r_4 directly follow from the lemmas described in the previous subsection. The complete proofs can be found in the appendix and at `http://isg.rhul.ac.uk/morisset/ictac/example.thy`. Finally, using the

r_1 : d_{old} = [**true**|−a.x'=2 ∨ a.x'=3]
d_{new} = [**true**|−a.x'=2]
↓
r_2 : Auto-refinement of pre/post-conditions to assignments
↓
r_3 : d_{old} = a.x := 2
d_{new} = a.x:=1; a.x := a.x+1
↓
r_4 : Expert-Pattern on a.x := 1

Table 1: Refinement steps

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}. Apart from the proofs of r_1 and r_3, all the other proofs could be derived automatically, since automatic transformations are used, and such an approach 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 [18] 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 [7], would generate proof statements both for interactive theorem provers and automated demonstrators. The other strength of this approach is the definition of the graph-based semantics, which eases the proving process, by giving an intuitive view of the memory, and therefore helping reasoning about the state of the program.

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 which satisfies `isGoodPath`. Moreover, more designs need to be implemented in the translation process, for instance recursive method calls. However, we can extensively reuse [6, 12], defined for imperative programs, by extending them to object-oriented programs. Finally, 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 `nodefun` 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 ongoing work in the rCOS tool [17]. The principal challenge in this work is for the tool to handle 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, 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.

References

[1] R.-J. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, Helsinki, Finland, 1978. Report A–1978–4.

[2] A. Cavalcanti and D. A. Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Transactions on Software Engineering*, 26:713–728, 2000.

[3] Z. Chen, Z. Liu, Anders P. R., V. Stolz, and N. Zhan. Refinement and verification in component-based model driven design. *Science of Computer Programming*, 74(4):168–196, February 2009. UNU-IIST TR 388.

[4] Z. Chen, Z. Liu, and V. Stolz. 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.

[5] Z. Chen, Charles M., and V. Stolz. Specification and validation of behavioural protocols in the rCOS modeler. In *3rd. FSEN 2009*, LNCS. Springer, 2009.

[6] C. Depasse. Constructing Isabelle proofs in a proof refinement calculus. *Research Report, UCL*, 2001.

[7] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, 2003.

[8] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[9] J. He, Z. Liu, and X. Li. rCOS: A refinement calculus of object systems. *Theor. Comput. Sci.*, 365(1-2):109–142, 2006. UNU-IIST TR 322.

[10] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[11] W. Ke, Z. Liu, S. Wang, and L. Zhao. A graph-based operational semantics of oo programs. In *Proceedings of ICFEM'09, LNCS volume 5885*, pages 347–366, 2009.

[12] L. Laibinis. *Mechanised Formal Reasoning About Modular Programs*. PhD thesis, Abo Akademi, 2000.

[13] B. Lei, Z. Liu, C. Morisset, and X. Li. State based robustness testing for components. In *FACS08*, ENTCS volume 260, pages 173–188. Elsevier, September 2008.

[14] Z. Liu, C. Morisset, and V. Stolz. rCOS: theory and tools for component-based model driven development. Technical Report 406, UNU-IIST, February 2009. Keynote, FSEN 2009, LNCS.

[15] C. Morgan. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., 1994.

[16] R. F. Paige and J. S. Ostroff. ERC – An object-oriented refinement calculus for eiffel. *Form. Asp. Comput.*, 16(1):51–79, 2004.

[17] V. Stolz. An integrated multi-view model evolution framework. *Innovations in Systems and Software Engineering*, 2009.

[18] J. von Wright. Program refinement by theorem prover. In *BCS FACS Sixth Refinement Workshop – Theory and Practise of Formal Software Development*. SpringerVerlag, 1994.

A Proof of refinement

The theory can be found at `http://isg.rhul.ac.uk/morisset/ictac/example.thy`. We do not put here the lemmas defined, the theory for the graph implementation can be found at `http://isg.rhul.ac.uk/morisset/i` the implementation of the rcos designs at `http://isg.rhul.ac.uk/morisset/ictac/rcos.thy`, and the useful lemmas (including `ref_pp_assign` and `EPIsRefTwo`) at `http://isg.rhul.ac.uk/morisset/ictac/rcos_lib.thy`.

Note that for r_1, r_2 and r_4, the statement might seem quite long, but the proofs are actually short, and very straightforward, following directly the refinement rules.

A.1 Refinement r_1

This manual refinement transforms [**true**|−a.x'=2 ∨ a.x'=3] into [**true**|−a.x'=2].

```
lemma r1_is_refinement :
  "pp (true) (λ g. λ g1.((getNatOfPath g1 [''x'',''a'','' this '']) = 2 |
                         (getNatOfPath g1 [''x'',''a'','' this '']) = 3))
       [[''x'',''a'','' this '']]
  ref pp (true) (λ g. λ g1.((getNatOfPath g1 [''x'',''a'','' this '']) = 2))
         [[''x'',''a'','' this '']]"
  (is "pp (true) (λg. ?Q1) ?path ref pp (true) (λg. ?Q2) ?path")
proof −
  have f1 : "implies ?Q2 ?Q1" by (simp add: implies_def)
  thus ?thesis by (simp add: pp_strengthen_post)
qed
```

A.2 Refinement r_2

This refinement automatically transforms [**true**|−a.x'=2] into a.x :=2.

```
lemma r2_is_refinement :
  "pp (true) (λ g. λ g1.((getNatOfPath g1 [''x'',''a'','' this '']) = 2))
       [[''x'',''a'','' this '']]
  ref
  assign (λ g . swingPath g [''x'',''a'','' this ''] (getNodeN 2))
         [''x'',''a'','' this '']"
by (simp add: ref_pp_assign)
```

A.3 Refinement r_3

This refinement transforms a.x :=2 into a.x :=1; a.x := a.x + 1. The proof might seem quite long, but we have preferred to make it "readable", by naming variables and decomposing each step, instead of

applying directly complicated tactics.

```
lemma r3_is_refinement :
  "(assign (λ g . swingPath g ['' x '' , '' a '' , '' this ''] (getNodeN (2:: nat ))) ['' x '' , '' a '' , '' this ''])  ref
  (assign (λ g . swingPath g ['' x '' , '' a '' , '' this ''] (getNodeN (Suc 0))) ['' x '' , '' a '' , '' this ''] ;
  (assign (λ g . swingPath g ['' x '' , '' a '' , '' this ''] (getNodeN (Suc (getNatOfPath g ['' x '' , '' a '' , '' this '']))))  ['' x '' , '' a '' , '' this
  (is "?A ref (?B ; ?C)")
proof −
  let ?path = "['' x '' , '' a '' , '' this '']" let ?Lone = "getNodeN 1" let ?Ltwo = "getNodeN 2"
  have f2 : "∀ q. (implies (?A q) (?B (?C q)))"
  proof −
    have f1: "!! g. wf g ⇒ wf (swingPath g ?path (getNodeN (Suc 0)))"
    proof −
      fix g assume h1 : "wf g"
      have f1 : "∀ x. getNodeN (Suc 0) ≠ R x" by (simp add:getNodeN_def)
      with h1 f1 show "wf (swingPath g ?path (getNodeN (Suc 0)))"
    by (simp add:h1 f1 wfswingPath [of "getNodeN (Suc 0)" g ?path])
    qed
    have f2 : "!! g. wf g ⇒ wfPath g ?path ⇒ wfPath (swingPath g ?path (getNodeN (Suc 0))) ?path"
    proof −
      fix g assume h1 : "wf g" assume h2 : "wfPath g ?path"
      with h1 h2 show "wfPath (swingPath g ?path (getNodeN (Suc 0))) ?path"
    by (simp add:swingPathNatPreserveswfPath [of g ?path "Suc 0"])
    qed
    have f3: "!! g q. wf g ⇒ wfPath g ?path ⇒ q (swingPath g ?path (getNodeN (2:: nat ))) ⇒
      q (swingPath (swingPath g ?path (getNodeN (Suc 0)))
      ?path (getNodeN (Suc (getNatOfPath (swingPath g ?path (getNodeN (Suc 0))) ?path ))))"
    proof −
      fix q g assume h1 : "wf g" assume h2 : "wfPath g ?path"
      assume h3 : "q (swingPath g ?path (getNodeN (2:: nat )))"
      let ?h = "(swingPath g ?path (getNodeN (Suc 0)))"
      show "q (swingPath ?h ?path (getNodeN (Suc (getNatOfPath ?h ?path ))))"
      proof −
    from h1 h2 have f1 : "(getNatOfPath ?h ?path) = Suc 0"
      by (simp add:sPCV_Nat [of g ?path "Suc 0"])
    have f2 : "getNodeN (Suc 0) ≠ undefVertex" by (simp add:getNodeN_def undefVertex_def)
    have f3 : "getNodeN 2 ≠ undefVertex" by (simp add:getNodeN_def undefVertex_def)
    from h1 h2 f2 f3 have f4 : "swingPath ?h ?path (getNodeN 2) = swingPath g ?path (getNodeN (2:: nat ))"
      by (simp add: swingPathTransitive [of g ?path "getNodeN (Suc 0)" "getNodeN 2"])
    with h3 have f5 : "q (swingPath ?h ?path (getNodeN (2:: nat )))" by simp
    have "Nat 2 = Nat (Suc (Suc 0))" by simp
    hence "getNodeN (2:: nat ) = getNodeN (Suc (Suc (0:: nat )))"
      by (simp add:getNodeN_def, arith )
    with f1 f5 have "q (swingPath ?h ?path (getNodeN (Suc (getNatOfPath ?h ?path ))))" by (simp)
    thus ?thesis by simp
      qed
    qed
    with f1 f2 f3 show ?thesis
      by (simp add: assign_def implies_def andd_def, auto, simp add: f1, simp add:f3)
  qed
  thus ?thesis by (simp add: refines_def seq_def)
qed
```

A.4 Refinement r_4

This refinement transforms a.x := 1 into a.m(1). We use directly `EPIsRefTwo`.

```
lemma r4_is_refinement :
" (assign (λ g . swingPath g [''x'',''a'','' this ''] (getNodeN (Suc 0)))
          [''x'',''a'','' this ''] ;
  (assign (λ g . swingPath g [''x'',''a'','' this ''] (getNodeN (Suc (getNatOfPath g [''x'',''a'','' this ''])))
          [''x'',''a'','' this ''])
ref
  (method [('' this '', Path [ ''a'','' this '']) , (''v'', Val (Nat (Suc 0)))]
          (assign (λ g. swingPath g [''x'','' this ''] (getVertexPath g [''v'']))  [''x'','' this ''])) ;
  (assign (λ g . swingPath g [''x'',''a'','' this ''] (getNodeN (Suc (getNatOfPath g [''x'',''a'','' this ''])))
          [''x'',''a'','' this ''])"
(is "(?A ; ?B) ref (?C ; ?B) ")
proof −
  have f1 : "?A ref ?C" by (simp add:EPIsRefTwo)
  thus ?thesis by (simp add: seq_ref_left )
qed
```

A.5 Refinement Chain

Finally, the proof that the whole chain is a refinement is very simple using the fact that the refinement relation is transitive. We have to decompose each step, otherwise the auto tactic of Isabelle generates too many possibilities.

```
lemma chain_is_refinement :
  "pp (true) (λ g. λ g1.((getNatOfPath g1 [''x'', ''a'','' this '']) = 2 |
                         (getNatOfPath g1 [''x'',''a'','' this '']) = 3))
      [[''x'',''a'','' this '']]
  ref
  (method [('' this '', Path [''a'','' this '']) , (''v'',Val (Nat (Suc 0)))]
          (assign (λ g. swingPath g [''x'','' this ''] (getVertexPath g [''v'']))  [''x'', '' this ''])) ;
  (assign (λ g . swingPath g [''x'',''a'','' this ''] (getNodeN (Suc (getNatOfPath g [''x'',''a'','' this ''])))
          [''x'' , ''a'', '' this ''])"
  (is "?A ref ?B")
proof −
  let ?path = "[''x'' , ''a'', '' this '']"
  let ?C = "pp (true) (λ g. λ g1.((getNatOfPath g1 ?path) = 2)) [?path]"
  let ?D = "assign (λ g . swingPath g ?path (getNodeN 2)) ?path"
  let ?E = "(assign (λ g . swingPath g ?path (getNodeN (Suc 0))) ?path ;
  (assign (λ g . swingPath g ?path (getNodeN (Suc (getNatOfPath g ?path )))) ?path))"
  have f1 : "?A ref ?C" by (simp add: r1_is_refinement )
  have f2 : "?C ref ?D" by (simp add: r2_is_refinement )
  have f3 : "?D ref ?E" by (simp add: r3_is_refinement )
  have f4 : "?E ref ?B" by (simp add: r4_is_refinement )
  from f1 f2 have "?A ref ?D" by (simp add: ref_transitive [of "?A" "?C" "?D"])
  with f3 have "?A ref ?E" by (simp add: ref_transitive [of "?A" "?D" "?E"])
  with f4 show ?thesis by (simp add: ref_transitive [of "?A" "?E" "?B"])
qed
```