



The United Nations  
University

**UNU-IIST**

International Institute for  
Software Technology

---

# **A Framework for Automated and Certified Refinement Steps**

---

**Andreas Griesmayer, Zhiming Liu, Charles Morisset, and Shuling Wang**

July 2011

## 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.

Peter Haddawy, Director



The United Nations  
University

**UNU-IIST**

**International Institute for  
Software Technology**

P.O. Box 3058  
Macao

---

# A Framework for Automated and Certified Refinement Steps

---

**Andreas Griesmayer, Zhiming Liu, Charles Morisset, and Shuling Wang**

## **Abstract**

The refinement calculus provides a methodology for transforming an abstract specification into a concrete implementation, by following a succession of refinement rules. These rules have been mechanized in theorem-provers, thus providing a formal and rigorous way to prove that a given program refines another one. In a previous work, we have extended this mechanization for object-oriented programs, where the memory is represented as a graph, and we have integrated our approach within the rCOS tool, a model-driven software development tool providing a refinement language. Hence, for any refinement step, the tool automatically generates the corresponding proof obligations and the user can manually discharge them, using a provided library of refinement lemmas. In this work, we propose an approach to automate the search of possible refinement rules from a program to another, using the rewriting tool Maude. Each refinement rule in Maude is associated with the corresponding lemma in Isabelle, thus allowing the tool to automatically generate the Isabelle proof when a refinement rule can be automatically found. The user can add a new refinement rule by providing the corresponding Maude rule and Isabelle lemma.

**Keywords:** Refinement, Software Engineering, Certification

**Andreas Griesmayer** is a research associate at Imperial College, London, UK. His research interests include software model checking and program repair, verification as well as automata and temporal logic. Email: andreas.griesmayer@imperial.ac.uk

**Zhiming Liu** is a Senior Research Fellow at UNU-IIST. His research interests include theory of computing systems, emphasizing sound methods and tools for specification, verification and refinement of fault-tolerant, realtime and concurrent systems, and formal techniques for object-oriented development. Email: Z.Liu@iist.unu.edu

**Charles Morisset** is a postdoc at Security Group, IIT - CNR, Italy. His research interests include the theory of access control, the security of information systems in general, and the area of formal methods, in particular the use of theorem proving in system specification. Email: charles.morisset@cnr.iit.it

**Shuling Wang** is currently a postdoctoral fellow of ISCAS, China. Her research interest is in formal methods of hybrid systems, including modeling and verification; object-oriented confinement and verification. Email: wangsl@ios.ac.cn

This work has been supported by the project GAVES and ARV funded by the Macao S&TD Fund, STCSM 08510700300, the projects NSFC-60970031, NSFC-91018012 and NSFC-60736017, the EU FP7-ICT project NESSoS (Network of Excellence on Engineering Secure Future Internet Software Services and Systems) under the grant agreement n. 256980, and the EU FP7-PEOPLE project DiVerMAS (Distributed System Verification with MAS-based Model Checking) under the grant agreement n. 252184.

## Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
<b>2</b>	<b>rCOS</b>	<b>8</b>
<b>3</b>	<b>Mechanized Refinement</b>	<b>9</b>
<b>4</b>	<b>Graph Representation</b>	<b>11</b>
4.1	State Graph . . . . .	11
4.2	Graph Implementation . . . . .	12
4.3	Graph Operations . . . . .	13
<b>5</b>	<b>Refinement of rCOS designs</b>	<b>14</b>
5.1	Primitive Designs . . . . .	14
5.2	Composite Designs . . . . .	16
<b>6</b>	<b>Generation of Proof Obligations</b>	<b>16</b>
6.1	Tool Refinement . . . . .	16
6.2	Provided Lemmas . . . . .	17
<b>7</b>	<b>Automatic Generation of Refinement Steps</b>	<b>17</b>
7.1	Maude . . . . .	18
7.2	Rewriting rules . . . . .	18
<b>8</b>	<b>Overall Mechanism</b>	<b>20</b>
<b>9</b>	<b>Related Work - Discussion</b>	<b>22</b>
9.1	Mechanization of refinement . . . . .	22
9.2	Certified model transformations . . . . .	23
9.3	Memory Model . . . . .	23
9.4	Automation of refinement . . . . .	23
9.5	Proof Generation . . . . .	24
<b>10</b>	<b>Conclusion</b>	<b>24</b>



## 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, roughly classified into two categories, are available in order to achieve this goal. The first category includes verification techniques taking both the specification and the implementation as inputs, and investigating whether the latter is an instantiation of the former. This investigation can be done for instance by running several instances of the implementation on specific inputs and monitoring whether the specification is respected during the execution (*e.g.* testing); by building an abstract model of the implementation and showing that the specification holds for any possible run (*e.g.* model-checking); by formally encoding both the specification and the implementation in a common language, using their respective semantics, and proving that the implementation logically implies the specification (*e.g.* theorem-proving).

The second category of verification techniques includes techniques where the implementation is generated from the specification, following a methodology guaranteeing by construction that the implementation meets the specification. For instance, Coq's extraction mechanism of a program from a formal specification [34] certifies that the program is correct with respect to this specification. Model-Driven Engineering [27] considers a program as a model, which can be derived from another one using model transformations [38]. Similarly, the refinement calculus [2, 39] provides a formal language in which both abstract specifications and concrete implementations can be expressed and mixed, and some formal refinement rules describing how to transform a program into another, more concrete one.

In general, these techniques offer equivalent results: if an implementation is proved to meet a specification, then there exists a refinement chain from the specification to the implementation, and conversely, if there exists a chain of refinement from a specification to an implementation, then it can be proven that this implementation satisfies this specification. Moreover, different techniques can be combined in order to exploit the strengths of each technique for a given context. For instance, a model-checker can be integrated into the Isabelle theorem-prover [17], and test-cases can be automatically generated from an Isabelle specification [6]. Similarly, model-checking can be used to verify refinement steps [21], and the refinement calculus has been encoded into a theorem-prover [45].

In a recent work [36], we have extended this encoding to object-oriented programs, by representing the memory of a state as a graph. Hence, a proof of refinement can be expressed as an Isabelle lemma, that needs to be proven by the user. Although we provide the user with a collection of lemmas to help her in this task, such a proof can still be challenging, in particular for users not familiar with Isabelle or theorem-proving in general. We address this challenge in this paper by providing a mechanism that automatically generates, when possible, the proof of the lemma in Isabelle.

This work takes place in the context of the refinement for Component and Object Systems (rCOS) [23, 11, 12, 35]. The rCOS language has a formal semantics based on an extension of the Unifying Theories of Programming (UTP) [24] to include the concepts of components and objects, and an operational semantics based on graphs [26]. This language is supported by the rCOS tool [13], which provides a UML-like multi-view and multi-notational modeling and design platform. The rCOS tool already provides the user with a collection of complementary verification techniques, such as the automated

generation of robustness test cases [31], and the automated generation of CSP processes to verify the compatibility between the sequence diagram and the state diagram of a contract [14].

**Contribution** The main contribution of this paper is the extension of previous work on encoding the refinement of two rCOS programs as Isabelle lemma [36], with a module that performs an automatic search for a sequence of pre-defined refinement steps showing that the refinement is correct. If such a sequence can be found, then the module directly generates the Isabelle proof of refinement for the initial lemma. This module is written using the Maude rewriting tool [15], such that refinement rules are defined as rewriting rules, and each rewriting rule is associated with the corresponding Isabelle lemma.

**Organization** Section 2 introduces the rCOS language. Section 3 recalls the previous mechanization of the refinement calculus. Section 4 gives the graph-based representation of the memory and its implementation in Isabelle/HOL, which is used in Section 5 to extend the mechanization of the refinement calculus to object-orientation. Section 6 presents an example to illustrate the application of refinement steps to rCOS programs. The framework for automated generation of a proof of refinement is introduced in Section 7 and demonstrated using an example in Section 8. Finally, we present related work in Section 9 and conclude and present future work in Section 10.

## 2 rCOS

The rCOS method consists of two parts: a component/object-oriented language with formal semantics, and a modeling tool, enforcing a use-case based methodology for software development, providing tool support and static analysis. We use only a subset of the language in the examples presented in this paper, however we give here a brief description of the whole language, and we refer to [12, 13] for further details.

The rCOS language is an extension of UTP [24], 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 [2, 39], 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 ; CHAOS ; an assignment  $p := e$ , where  $p$  is a navigation path, and  $e$  is an expression; a conditional statement  $d_1 \triangleleft b$

$\triangleright d_2$ , 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  $\text{do } b \text{ } d$ , where  $d$  is a design and  $b$  is a boolean expression; a local variable declaration and un-declaration  $\text{var } T \ x = e ; \text{end } x$ , where  $T$  is a type.

A new object of type  $C$  is created and attached to the path  $p$  through the command  $C.\text{new}(p)$ . A method invocation has the form  $e.m(i; o)$ , where  $m$  is a method,  $i$  stands for the input parameters and  $o$  for the output parameters. If there is no output parameter, we can write directly  $e.m(i)$ .

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  $\vdash$  a.x'=2  $\vee$  a.x'=3]}
}

class B2 {
  A a;
  public foo() {
    [true  $\vdash$  a.x'=1] ;
    a.x:=a.x+1}
}

class B3 {
  A a;
  public foo() {
    a.m(1) ;
    a.x := a.x+1}
}

```

The method  $B_1::\text{foo}$  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  $B_2::\text{foo}$  mixes abstract pre/post-conditions with a concrete assignment while  $B_3::\text{foo}$  is completely concrete and could be directly translated to Java. In this example, we can see that  $B_1::\text{foo}$  is refined by  $B_2::\text{foo}$ , which is refined by  $B_3::\text{foo}$ . We detail in the following section the mechanization of the notion of refinement.

### 3 Mechanized Refinement

The refinement calculus [2, 39] 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 encoded into the theorem prover HOL, an ancestor of Isabelle, in [45, 19] and then extended, in particular in [29], which introduces, among others, procedures and recursive functions. The encoding follows the weakest precondition approach: for any design  $d$  and any predicate  $q$  over states, the function  $\text{wp}(d, q)$  stands for 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 [45]. We use `State` to represent the type of a program state, which is defined as a tuple of value in [45], and represented as a graph in [36] and in this document. We introduce first the type of predicates over states and the type of predicate transformers.

**types** `State pred = State  $\Rightarrow$  bool`  
`State predT = State pred  $\Rightarrow$  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  $\Rightarrow$  State)  $\Rightarrow$  (State predT)`  
**where**  
`assign e q  $\equiv$   $\lambda$ u. 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)  $\Rightarrow$  (State pred)  $\Rightarrow$  bool`  
**where**  
`implies p q  $\equiv$   $\forall$  u. (p u)  $\Rightarrow$  (q u)`

**definition** `refines :: (State predT)  $\Rightarrow$  (State predT)  $\Rightarrow$  bool`  
**( infixl ref 40)**  
**where**  
`c1 ref c2  $\equiv$   $\forall$  q. (implies (c1 q) (c2 q))`

In addition to the definition of the semantics, helpful theorems are introduced in [45]. 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  $\Rightarrow$  (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 in [45], 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. Dealing with local variables and method calls thus 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 object-oriented programs. The usual way to tackle this issue is to represent a state as a record or as a function from pointers to values [3, 41, 9, 42]. A recent approach uses graphs instead [26], and we present it in the next section.

## 4 Graph Representation

In [26], the state of a program is represented as a directed labeled graph. We only give here a simple description of such a graph and its implementation in Isabelle/HOL, more details can be found in [26, 36].

### 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 if, and only if, 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,
- $T : N \rightarrow \mathcal{C}$  is a mapping from object nodes to types,
- $F : L \rightarrow \mathcal{W}$  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.

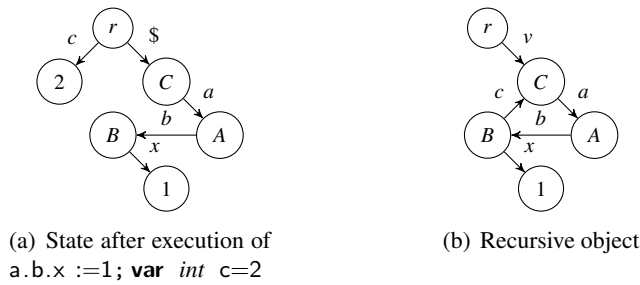


Figure 1: Examples of state graphs

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. Fig. 1(a) illustrates the state graph after execution of the command `a.b.x :=1; var int c=2`. Moreover, state graph with recursive objects is possible, as illustrated in Fig. 1(b).

## 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 type `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`  $\Rightarrow$  `label`  $\Rightarrow$  `vertex`

`onodefun` = `onode`  $\Rightarrow$  `ctype`

`vnodefun` = `vnode`  $\Rightarrow$  `val`

`G` = `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`:: `G`  $\Rightarrow$  `bool` **where**  
`isGoodFunction g`  $\equiv \forall x. (\text{getEdgeFun } g) \perp x = \perp$

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 briefly describes some basic graph operations that are needed for the encoding of the refinement calculus. Due to space limitation, we do not present here the implementation of these functions, more detailed explanations can be found in [36], and the complete list can be found at [http://www.doc.ic.ac.uk/~agriesma/mircos/graph\\_utl.thy](http://www.doc.ic.ac.uk/~agriesma/mircos/graph_utl.thy).

We first introduce the type `path` as a list of labels, which, for implementation optimization reasons, is reversed: the path `a.b.x` is represented by the list `["x", "b", "a"]`. The vertex corresponding to a path `p` in a graph `g` is given by `getVertexPath p g`.

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`. 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.

We also need to consider the well-formedness of paths: a path `p` satisfies `isGoodPath` if, and only if, the last-but-one vertex of `p` appears exactly once as a source in the list of edges along the path. For instance, in the state graph in Fig. 1(b), the paths `v.a`, `v.a.b.c` and `v.a.b.x` both 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. Furthermore a path `p` is said 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 `p` satisfies `isGoodPath`.

One of the most important theorems of our theory is `swingPathChangeVertex`: assuming that `g` is well-formed, that `p` is well-formed w.r.t. `g`, and that `n` is not the undefined vertex, then we can prove that a path after being swung to a vertex will actually point to the vertex.

The function `Vars` combines the operations of creating a scope node and adding edges, and therefore implements local variable declaration. Similarly, the function `removeSnode` 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. Finally, the function `addObject` creates a new vertex (object) in a graph. These functions are proved to preserve the graph well-formedness.

## 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://www.doc.ic.ac.uk/~agriesma/mircos/rcos.thy>.

### 5.1 Primitive Designs

#### Pre/post-condition

The definition of the non-deterministic assignment needs to include the well-formedness checks.

**definition**  $\text{nondass} :: (G \Rightarrow G \text{ pred}) \Rightarrow \text{path list} \Rightarrow (G \text{ pred}) \Rightarrow (G \text{ pred})$

**where**

$\text{nondass } P \ l \ q \equiv (\lambda v. (\text{wfGraph } v) \ \& \ (\text{wfPathl } l \ v) \ \& \ (\forall v1. P \ v \ v1 \Rightarrow q \ v1))$

where  $\text{wfPathl } l \ v$  is true if, and only if, every path in  $l$  satisfies  $\text{wfPath}$ . 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**  $\text{pp} :: (G \text{ pred}) \Rightarrow (G \Rightarrow G \text{ pred}) \Rightarrow \text{path list} \Rightarrow (G \text{ predT})$

**where**

$\text{pp } p \ r \ l \equiv \text{assert } p \ ; \ \text{nondass } r \ l$

where  $\text{assert}$  is the standard definition for the assertion. For instance, the pre/post-condition  $[ \text{true} \vdash a.b.x'=2 ]$  is translated into the following statement

$\text{pp} (\text{true}) (\lambda g. \lambda g1. ((\text{getNVal } a.b.x \ g1) = 2)) [a.b.x]$

where  $\text{getNVal}$  is the function returning the value (as a  $\text{nat}$ ) of the path  $p$  in  $g1$  and where, for the sake of readability, we abbreviate the path  $[ "x" , "b" , "a" ]$  as  $a.b.x$ .

#### Assignment

The definition of the assignment is changed as follows.

**definition**  $\text{assign} :: \text{path} \Rightarrow \text{exp} \Rightarrow (G \text{ pred}) \Rightarrow (G \text{ pred})$

**where**

$\text{assign } p \ e \ q \equiv \lambda u. \text{wfGraph } u \ \& \ \text{wfPath } p \ u \ \& \ \text{wfExp } e \ u \ \& \ q \ (\text{swingPath } p \ (\text{getNodeExp } e \ u) \ u)$

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 obtained 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` ::  $\text{labelExpF} \Rightarrow (G \text{ pred}) \Rightarrow (G \text{ pred})$   
**where**  
`begin`  $f \ q \equiv \lambda u. \text{wfGraph } u \ \& \ \text{wflabelExpF } f \ u \ \& \ q \ (\text{Vars } f \ u)$

**definition** `end` ::  $(G \text{ pred}) \Rightarrow (G \text{ pred})$  **where**  
`end`  $q \equiv \lambda u. \text{wfGraph } u \ \& \ q \ (\text{removeSnode } u)$

where  $f$  is a well-formed function of type `labelExpF`, which means that for each local variable, it is initialized 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` ::  $\text{labelExpF} \Rightarrow (G \text{ predT}) \Rightarrow (G \text{ predT})$   
**where**  
`locdec`  $f \ c \equiv \text{begin } f; \ c; \ \text{end}$

### Method invocation

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

**definition** `method` ::  $(\text{label} * \text{exp}) \text{ list} \Rightarrow (G \text{ predT})$   
 $\Rightarrow (G \text{ predT})$   
**where**  
`method`  $l \ c \equiv \text{locdec } (\text{getLabelExpF } l) \ c$

where  $l$  is of type  $(\text{label} * \text{exp}) \text{ list}$ , each pair consisting of a formal parameter and its actual value, 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  $\text{label} * \text{exp}$  to the corresponding mapping of type `labelExpF` (i.e.  $\text{label} \Rightarrow \text{exp}$ ). 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 substituted 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  $\Rightarrow$  ctype  $\Rightarrow$  ( G pred)  $\Rightarrow$  ( G pred)
where
object p s q  $\equiv$ 
   $\lambda$ u. wfGraph u & wfPath p u & q (addObject p s u)
  & wflabelExpF (getAttrsOfCtype 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 [45]. 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 Generation of Proof Obligations

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  $\vdash$  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 [45], we provide lemmas corresponding to refinement steps. For the sake of simplicity, we only focus here on lemmas concerning integers, however equivalent lemmas can be defined for other primitive types.

For instance, the lemma stating that for any path  $p$  and any integer  $n$ , the statement  $[true \vdash p'=n]$  is refined by the assignment  $p := n$  is defined<sup>1</sup> as:

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

Another example is the Expert Pattern, which is an essential rule for object-oriented 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. As a special instance of this pattern, we have defined the lemma `EPisRefTwo`, which states that the statement  $p.x := n$  is refined by the method  $p.m(n)$  where  $m(T\ v) \{this.x := v\}$  is a method of  $p$ , for any primitive type  $T$  and parameter  $v$ .

```
lemma EPisRefTwo :
  "p≠[] ⇒ b ≠ c ⇒
  (assign (a # p) (Val n) ref
  (method [(b, Path p), (c, Val n)]
  (assign (a # [b]) (Path [c ]))))"
```

This lemma only considers attribute accesses (if  $p$  is empty, then  $a \# p$  represents a local variable), and that  $b$  and  $c$  are the formal parameters of the method, and therefore must be different.

Finally, the lemma `assign_end` states that given a path  $p$  and two integers  $m$  and  $n$ , the statement  $p := m$  is refined by  $p := n; p := p + (m - n)$ .

```
lemma assign_end :
  "(assign p (Val (Zint m))) ref
  (assign p (Val (Zint n));
  (assign p (Plus (Path p) (Val (Zint (m - n))))))"
```

## 7 Automatic Generation of Refinement Steps

The techniques from the previous sections provide us with the tools to prove the refinement of a specification by a program, but require manual interaction to define the granularity and structure of the proof. This section presents an approach to automatize this step by formulating the proof obligations in rewriting logic and using the rewrite engine Maude to search for a sequence of refinement steps to create the

<sup>1</sup>Due to space limitation, we do not include in this document the proofs of the lemmas, which can be found at [http://www.doc.ic.ac.uk/~agriesma/mircos/rcos\\_lib.thy](http://www.doc.ic.ac.uk/~agriesma/mircos/rcos_lib.thy).

corresponding proofs of refinement in Isabelle. We first briefly introduce Maude [15], then present the rewriting rules corresponding to the refinement rules, and finally show how to extract an Isabelle proof. A detailed example is given in Section 8.

## 7.1 Maude

Maude is a rewrite tool that allows the specification of *equations* and *rewrite rules* which have a simple rewriting semantics in which instances of the left hand side are replaced by corresponding instances of the right hand side. The set of equations is designed to be confluent and terminating. This means that starting from a term, every possible sequence of applications of equations leads to a canonical form made from *ground terms*, which also give the means to define the type system of the implemented logic. Application of rewriting rules, on the other hand, needs neither be confluent nor terminating, and allows to express the evolution of the system. Eligible rules are selected by pattern matching: while *equations* have a deterministic result for any enabling term and are executed immediately by the Maude engine, the order of execution of *rewrite rules* may lead to different results. Thus, the application of rewrite rules spans a state space that can be explored by choosing among enabled rewrite rules. Intuitively, we thus define a state as a set of proof obligations that is known at any point in the execution and will use rewrite rules to generate new proof obligations for the proof, while the equations provide us with a canonical representation of equivalent terms, and discharge simple proof obligations.

## 7.2 Rewriting rules

In order to search for a sequence of steps to form a proof of refinement, we use Maude to systematically perform syntactical rewriting starting with an initial proof obligation that represents the refinement of a specification by an implementation. In the context of Maude, we write a proof obligation as  $(id, \{p_1\} \rightsquigarrow \{p_2\}, \text{from} : f \text{ status} : s)$ , where  $id$  is an identifier of the proof-obligation,  $\{p_1\} \rightsquigarrow \{p_2\}$  stands for the refinement step of  $p_1$  to  $p_2$ ,  $f$  refers to further proof-obligations that need to be fulfilled such that the rule can be discharged, and  $s$  is the status of the proof-obligation. The status can either indicate that it is still undetermined how to discharge the proof-obligation, (marked as `todo`), or it gives a rule with which it should be discharged in Isabelle.

We distinguish between two kinds of rewrite actions: 1) setting up new proof obligations according to the syntax of the present obligations and 2) discharging proof obligations according to the basic refinement steps proved by Isabelle as described in the previous sections. Note that some of those steps are implemented as equations for performance reasons. This can be done if the application of the rule is definitely required and the outcome is deterministic. E.g., the refinement between two identical terms is immediately discharged by the `ref-reflexive` rule. Other rules introduce new proof-obligations on a speculative basis. Such rules may or may not be required in the process of finding a proof and therefore, according to the definitions of equations and rewrite rules given above, need to be implemented as rewrite rules. For simplicity of presentation, we give here all of the steps in form of *rewrite rules*.

In the following, we present some selected rules for generation and discharging of proof obligations as

rewrite rules<sup>2</sup>. The syntax for expressions and statements in Maude is very similar to the syntax of the Isabelle lemmas; in fact, the communication between the tools can be done by a simple maude export expression and a script that replaces some reserved key symbols, in the remainder of the section we stick to a slightly simplified presentation of the rules; in particular, we will omit the environment taking track of details like number of open obligations and similar. We use  $S_1..S_4$  to denote statements,  $E_1, E_2$  for expressions and  $X$  for variables. A Maude rewriting rule has the following structure:

```
rl [name]  $po_1, \dots, po_k \Rightarrow po_{k+1}, \dots, po_n$ .
```

where *name* is the name of the rule,  $po_1$  is an undischarged proof-obligation, *i.e.* with a status equal to *todo*, and each  $po_i$  is a proof-obligation. For instance, the rule *ref-reflexive* can be defined as:

```
rl [ref-reflexive] :
  (  $id \{ X \} \sim \{ X \}$  stat : todo )  $\Rightarrow$ 
  (  $id \{ X \} \sim \{ X \}$  stat : ref-reflexive ) .
```

This rule can be read as follows: if we need to discharge the proof-obligation that the program  $X$  refines the program  $X$ , then we can directly do so by using the Isabelle lemma *ref-reflexive*. New proof obligations are introduced by pattern matching of present proof obligations. For instance, the refinement rule *ref-sequential-gen1* matches for sequences of statements and introduces a new proof obligation (note that  $id_2$  is a fresh identifier). This rule represents the fact that in order for the LHS to be refined by the RHS, the first statement on the LHS  $S_1$  needs to be refined by the first statement on the RHS  $S_3$ . (An analogous rule exists for the second statement.)

```
crl [ref-sequential-gen1] :
  (  $id_1 \{ S_1 ; S_2 \} \sim \{ S_3 ; S_4 \}$  status : todo )  $\Rightarrow$ 
  (  $id_1 \{ S_1 ; S_2 \} \sim \{ S_3 ; S_4 \}$  status : todo ) ,
  (  $id_2 \{ S_1 \} \sim \{ S_3 \}$  status : todo )
if new(  $S_1, S_3$  ) .
```

The rule is conditional and only creates a new proof obligation if  $\{S_1\} \sim \{S_3\}$  is not present yet. Similar rules exist for other constructs and can also match multiple present proof obligations to, e.g., add a missing part to apply the lemma of transitivity.

A second group of rewrite rules is used to close proof obligations when sufficient information is present. E.g., in our example of sequences of statements, we can discharge that  $S_1 ; S_2$  is refined by  $S_3 ; S_4$ , if, and only if, we have discharged proof-obligations  $id_2$ , stating that  $S_1$  is refined by  $S_3$ , and  $id_3$ , stating that  $S_2$  is refined by  $S_4$ . We discharge by recording to use the Isabelle lemma *ref-sequential*, with  $id_2$  and  $id_3$  as arguments:

```
rl [ref-sequential] :
  (  $id_1 \{ S_1 ; S_2 \} \sim \{ S_3 ; S_4 \}$  status : todo )
  (  $id_2 \{ S_1 \} \sim \{ S_3 \}$  status : stat1 ) ,
  (  $id_3 \{ S_2 \} \sim \{ S_4 \}$  status : stat2 ) ,
 $\Rightarrow$ 
  (  $id_1 \{ S_1 ; S_2 \} \sim \{ S_3 ; S_4 \}$  from:  $id_2 id_3$  status: ref-sequential)
if stat1 != todo and stat2 != todo
```

<sup>2</sup>The complete definition in maude rewriting logic can be found at <http://www.doc.ic.ac.uk/~agriesma/mircos/rcos.maude>.

The provided rules may not always be sufficient to fully discharge all proof obligations. E.g., the rule `ref-strengthen` refines a pre-post condition by replacing the post-condition by another one that logically implies it as follows:

```
rl [ref-strengthen] :
  ( id { [ |- E2 ] } ~>{ [ |- E1 ] } status : todo ) =>
  ( id { [ |- E2 ] } ~>{ [ |- E1 ] } status : ref-strengthen id2),
  ( id2 prove (E1 ⇒ E2) status : sorry ) .
```

We use the Isabelle keyword `sorry` to denote that the proof-obligation *cannot* be discharged in Maude, and therefore has to be done in Isabelle, where this keyword allows one not to provide the proof of a lemma. This approach makes it possible to consider the post-condition strengthening refinement rule regardless of the post-condition itself, by delegating the burden of the proof to Isabelle. However, some instances of this rule are quite simple, and can be done directly in Maude. For instance, the rule `ref-disj-left` chooses a member of the disjunction in a post-condition.

```
rl [ref-disj] :
  ( id { [ |- E1 ∨ E2 ] } ~>{ [ |- E1 ] } status : todo ) =>
  ( id { [ |- E1 ∨ E2 ] } ~>{ [ |- E1 ] } status : ref-disj-left) .
```

A major strength of this approach is its extensibility: new refinement rules can be easily added, simply by adding the corresponding rewriting rules in the Maude file, and the corresponding Isabelle lemmas in the Isabelle file, without any required modification to existing code. Hence, sets of rules can be dynamically loaded and unloaded, to adapt to different contexts.

## 8 Overall Mechanism

The Maude rules described in Section 7 aim at generating the proof of a refinement lemma as generated by the rCOS tool. Hence, the global architecture of our system, illustrated in Fig. 2, can be described as follows: given two rCOS programs  $p_1$  and  $p_2$ , the rCOS tool generates both the Isabelle statement **lemma**  $p_1 \text{ ref } p_2$ , as described in Section 6, and the Maude term  $(id \{p_1\} \sim\> \{p_2\} \text{ status})$ , as described in Section 7 (although, as said above, the Maude module can also take directly the Isabelle lemma as input, through a Python script). The Maude module tries to discharge the proof-obligation, by applying one or several refinement rules. If it succeeds, the corresponding proof for the Isabelle lemma is automatically generated. Recall the example given in Section 2, where that we want to prove that  $B1::\text{foo}()$  is refined by  $B3::\text{foo}()$ , *i.e.*, we need to show that  $[ \text{true} \vdash a.x' = 2 \vee a.x' = 3 ]$  is refined by  $a.m(1) ; a.x := a.x + 1$ .

The rCOS tool first generates the corresponding lemma in Isabelle, defined as:

```
lemma b1_foo_ref_b3_foo :
  "pp (true) (λ g. λ g1. ((getNVal this.a.x g1) = 2
    | (getNVal this.a.x g1) = 3)) [this.a.x]
  ref
  ((method [( ' this ' , Path this.a), ('v', Val (Zint 1))]
    (assign this.x (Path ['v']))) ;
  assign this.a.x (Plus (Path this.a.x) (Val (Zint 1))))"
```

where, for the sake of clarity, we abbreviate paths, *i.e.*  $this.a.x$  stands for  $[ 'x', 'a', 'this' ]$ . The rCOS tool also generates the corresponding proof-obligation in Maude:

```
{[ |- 2 = a.x ' ∨ 3 = a.x ' ]} ~>
[method([(' this ', ['a']), ['v'], 1)](this.x := ['v']) ;
  a.x := 1 + a.x} status: todo)
```

where, here again, we abbreviate paths. Maude is able to prove automatically the refinement, and outputs the following (for the clarity of the presentation, we have manually added the definition of the  $?X$  terms, the proof being equivalent without them, but much harder to read):

```
proof -
  let ?A = "(pp (λ g. True) (λ g. λ g1.
    ((getNVal this.a.x g1) = 2)) [this.a.x])"
  let ?B = "(assign this.a.x (Val (Zint 2)))"
  have f9: "?A ref ?B" by (simp add: ref_pp_assign )
  let ?C = "pp (λ g. True) (λ g. λ g1 .((getNVal this.a.x g1) = 2
    | (getNVal this.a.x g1) = 3)) [this.a.x ]"
  have f8: "?C ref ?A" by (simp add: ref_disj_left )
  from f8 f9 have f7: "?C ref ?B"
    by (simp add: ref_transitive [of ?C ?A ?B])
  let ?D = "(assign this.a.x (Val (Zint 1))) ;
    (assign this.a.x (Plus (Path this.a.x) (Val (Zint 1))))"
  have f6: "?B ref ?D"
    by (insert assign_end [of this.a.x 2 1], simp )
  from f7 f6 have f5: "?C ref ?D"
    by (simp add: ref_transitive [of ?C ?B ?D])
  let ?E = "assign this.a.x (Val (Zint 1))"
  have f4: "monotonic ?E" by (simp add: assign_monotonic)
  let ?F = "(method [(' this ', (Path this.a), ('v', (Val (Zint 1))))]
    (assign this.x (Path ['v'])))"
  let ?G = "assign this.a.x (Plus (Path this.a.x) (Val (Zint 1)))"
  have f3: "?G ref ?E" by (simp add: ref_reflexive )
  have f2: "?E ref ?F" by (simp add: EPIsRefTwo)
  from f2 f3 f4 have f1: "?D ref ?F ; ?G" by (simp add: seq_ref)
  from f5 f1 have f0: "?C ref ?F ; ?G"
    by (simp add: ref_transitive [of ?C ?D ?F ; ?G])
  from f0 show ?thesis by simp
qed
```

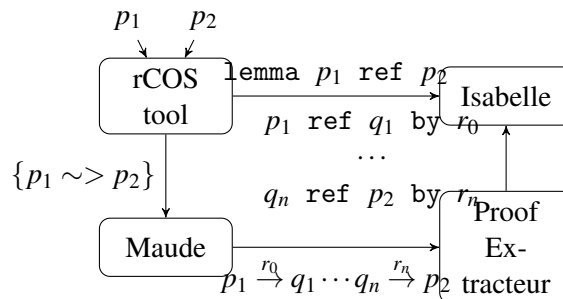


Figure 2: Workflow

This proof is correct, and can be instantly verified by Isabelle. Note that although it is quite long for a simple lemma, each step is atomic, since only one lemma is used for each step. In practice, it is possible to come up with a shorter, but equivalent proof of this lemma, by manually "inlining" the facts: every time that a fact  $f_i$  is used to prove a fact  $f_j$ , we try to prove directly  $f_j$  by adding the tactics used for  $f_i$ . If it succeeds, then  $f_i$  can be removed. However, in general, it is not trivial to find out which facts can be removed. The simplified version of the previous proof (and therefore corresponding to the one a human user could have directly generated) is:

```

proof –
  let ?path = this.a.x
  let ?A = "pp (λ g. True) (λ g. λ g1 .((getNVal this.a.x g1) = 2
    | (getNVal this.a.x g1) = 3)) [this.a.x]"
  let ?B = "(method [(' this ' , (Path this.a), ('v', (Val (Zint 1))))
    (assign this.x (Path ['v'])))] ;
    assign this.a.x (Plus (Path this.a.x) (Val (Zint 1))))"
  let ?C = "pp (true) (λ g. λ g1.
    ((getNVal ?path g1) = 2)) [?path]"
  let ?D = "assign ?path (Val (Zint 2))"
  let ?E = "(assign ?path (Val (Zint 1)) ;
    (assign ?path (Plus (Path ?path) (Val (Zint 1)))))"
  show ?thesis by
    (insert assign_end [of ?path 2 1],
    simp add:EPIsRefTwo seq_ref_left ref_disj_left
    ref_pp_assign ref_transitive [of ?A ?C ?D]
    ref_transitive [of ?A ?D ?E] ref_transitive [of ?A ?E ?B])
qed

```

## 9 Related Work - Discussion

### 9.1 Mechanization of refinement

The mechanization of the refinement calculus was firstly done in [45], which has been extended to include pointers [3] and also object-oriented programs [9, 42]. In particular, a refinement calculus has been defined for Eiffel contracts [41], and encoded in PVS [40]. 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 [45], 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 [8] 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 [44]. 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 [43]. 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.

## 9.2 Certified model transformations

The next step is therefore to express the refinement of models rather of simple statements, in particular for model transformations, which is an on-going work in the rCOS tool [43]. The principal challenge in this work is for the tool to handle several models at the same time: before, during and after refinement. For instance, in the example we have presented, we assume that the method  $m$  is already present in the class  $A$ . However, in practice, the software engineer might want to create the setter and change the code at the same time. In this case, creating the setter is a correct refinement, but in order to prove it, we need to also encode the structure of the whole model in Isabelle, in order to express that a whole model refines another one, and then define the model transformations in Isabelle. A related approach using the Coq theorem prover has been recently proposed [7], where the Class to Relational model transformation has been certified.

## 9.3 Memory Model

Different memory models for object-oriented programs have been encoded in theorem provers [20, 4, 28]. However, the memory in these approaches is either modeled as a function from addresses or pointers to values or using records to represent objects. Although such a modeling 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 might be more visual than a representation by a function or set of records. The graph structure helps in the formulation of properties and carrying out interactive proofs.

## 9.4 Automation of refinement

A range of tools are developed for supporting automatic refinement. Inspired by Dijkstra's weakest precondition calculus, they usually generate proof obligations corresponding to refinement requirement and then discharge them by automated reasoning via theorem proving. Most of them support both algorithm refinement (refining a program fragment or an entire method into an implementation, as we focus on in this paper), and data refinement (refining abstract data in specification to concrete data in implementation).

Robin [1] is an open toolset which integrates construction and verification of Event-B Models. The system behavior in Event-B is modeled by action systems, i.e. a collection of variables and guarded actions. Abstract specification of a model is constructed and then refined, following an incremental approach, however object-oriented programs are not directly supported.

ProofPower Z [25] is a tool interfaced with the YSE Zeta tool, which supports refinement from Z specification to Ada programs. The tool produces verification conditions for each refinement step for input into a theorem prover, and produces Ada code by applying the refinement steps. However, ProofPower and YSE are using different languages, making the communication sometimes difficult. Also, the unreadable output of proof in ProofPower provides few guideline for locating failures in source programs.

Based on ProofPower Z, the authors of [21, 46] develop the automatic refinement of Circus, which is a refinement language combining Z and CSP for describing state-rich reactive systems.

Perfect Developer [16] is a software tool for developing formal specifications and refining them into executable code. Compared to existing refinement tools before, it handles object-oriented features such as inheritance, recursive call, polymorphism, dynamic binding, in conformance with behavioral sub-typing principle. However, it does not support stepwise refinement, and requires a continuous strengthening of the code annotations, making the refinement less scalable.

Leino and Yessenov [33] develop a refinement system for object-oriented programs and where the verification engine is based on the SMT solver Z3. It supports automated stepwise refinement and all the intermediate steps are saved using syntax of code skeletons during the whole refinement process. This makes the location of failures in the source specification and code realizable. Moreover, it handles aliasing between data representations based on the permission mechanism in Chalice [32].

An important strength of our work compared with these different approaches, in addition to the integration within the rCOS tool, a complete platform for software engineering, is the generation of a *proof witness*, *i.e.* we are not only answering if the refinement is correct or not, but also providing the reason why. Hence, we can easily reuse previously proved lemmas, making our approach more scalable.

## 9.5 Proof Generation

Our approach for proof generation was loosely inspired by the work realized with the automated demonstrator Zenon [5]. Indeed, Zenon can prove first order logical formulae using the tableau method, and generate the proof in Coq. It was originally developed for the Focalize [22] environment, which provides an expressive programming language where properties of a program can be proved in Coq, potentially using Zenon to generate some of the Coq proofs. An interesting feature is the definition of a simple and intuitive proof language following Lamport's guidelines [30], which allows the user to break down a complex proof into small proofs, until a point where Zenon can prove automatically the statement. Such an integration is quite user-friendly, and therefore we aim at achieving a similar result.

In this context, the recent integration of Zenon with Isabelle and TLA+ [10] can probably be useful. We can also try to integrate our Maude module as an automatic theorem prover using the Sledgehammer in Isabelle [37].

## 10 Conclusion

This paper presents, for any two programs defined within the rCOS tool, the generation of an Isabelle lemma stating that one program refines the other, as originally introduced in [36], and extends this approach by describing a Maude module that searches for a sequence of refinement steps, each step being implemented as a rewriting rule. If such a sequence exists, then the module generates the Isabelle

proof of the previous lemma, otherwise the lemma still needs to be manually proven.

The strengths of this approach are fourfold. Firstly, the generation process is integrated within the rCOS tool, and when the refinement can be automatically proven by Maude, then the process is transparent for the user. Secondly, the user has still the possibility to manually prove some lemmas, when Maude cannot automatically prove the refinement. Thirdly, new rules can be added to the process, simply by adding the new lemma in the Isabelle library and the new rewriting rule in the Maude file, without any need to modify existing code. Finally, it generates the witness of the proof, instead of only returning yes or no. It follows that any proven refinement can be stored, and re-used later, to prove more complex refinements.

This work mostly focuses on defining the framework where the different entities, *i.e.* the rCOS tool, Isabelle and Maude, can communicate together, in order to have a complete chain from the user of the rCOS tool, who is potentially an expert in software engineering rather than an expert in theorem-proving, to the proof of refinement in Isabelle. Hence, we have mostly considered simple examples, although rich enough to validate our approach, but clearly lacking the complexity of real-world programs.

We believe that we have paved the way towards the certification of more complex programs, as we can leverage the incremental aspect of the refinement calculus. Indeed, a large, complex refinement chain, as it could be expected from a complex program, can always be decomposed as a sequence of simpler chains of refinement steps. Although some steps will probably always require a human interaction, such as the definition of a loop-invariant, some of tedious and repetitive steps can be automatically discharged.

However, a continuous effort of improving the library is required to eventually provide a system usable by non-expert users. For instance the theorem `swingPathChangeVertex` might still hold without the assumption `isGoodPath` 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 [19, 29], 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.

Finally, the Maude module can be optimized, in order not to generate proof-obligations that are clearly impossible, or in general to generate less proof-obligations. The use of rewrite strategies seems a suitable way to tackle this problem.

## References

- [1] Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in event-b. *Int. J. Softw. Tools Technol. Transf.* **12**, 447–466 (2010)
- [2] Back, R.J.: On the correctness of refinement steps in program development. Ph.D. thesis, University of Helsinki, Finland (1978). Report A-1978-4
- [3] Back, R.J., Fan, X., Preoteasa, V.: Reasoning about pointers in refinement calculus. Tech. Rep. 543, TUCS - Turku Centre for Computer Science, Turku, Finland (2003)
- [4] van den Berg, J., Jacobs, B.: The loop compiler for java and jml. In: TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 299–312. Springer-Verlag, London, UK (2001)
- [5] Bonichon, R., Delahaye, D., Doligez, D.: Zenon : An extensible automated theorem prover producing checkable proofs. In: N. Dershowitz, A. Voronkov (eds.) LPAR, *Lecture Notes in Computer Science*, vol. 4790, pp. 151–165. Springer (2007)
- [6] Brucker, A.D., Wolff, B.: HOL-TestGen: An interactive test-case generation framework. In: M. Chechik, M. Wirsing (eds.) Fundamental Approaches to Software Engineering (FASE09), no. 5503 in *Lecture Notes in Computer Science*, pp. 417–420. Springer-Verlag, Heidelberg (2009)
- [7] Calegari, D., Luna, C., Szasz, N., Tasistro, A.: A type-theoretic framework for certified model transformations. In: Davies et al. [18], pp. 112–127
- [8] 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 (1996)
- [9] Cavalcanti, A., Naumann, D.A.: A weakest precondition semantics for refinement of object-oriented programs. *IEEE Transactions on Software Engineering* **26**, 713–728 (2000)
- [10] Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the tla+ proof system. In: J. Giesl, R. Hähnle (eds.) IJCAR, *Lecture Notes in Computer Science*, vol. 6173, pp. 142–148. Springer (2010)
- [11] Chen, Z., Liu, Z., Ravn, A., Stolz, V., Yang, L.: A refinement driven component-based design. In: Proc. 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS07), pp. 277–289. IEEE Computer Society, Aucland, New Zealand (2007)
- [12] 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.
- [13] Chen, Z., Liu, Z., Stolz, V.: The rCOS tool. In: Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop, no. CS-TR-1099 in Technical Report Series. Newcastle University (2008). Available at <http://rcos.iist.unu.edu>

- [14] Chen, Z., Morisset, C., Stolz, V.: Specification and validation of behavioural protocols in the rcos modeler. In: F. Arbab, M. Sirjani (eds.) FSEN, *Lecture Notes in Computer Science*, vol. 5961, pp. 387–401. Springer (2009)
- [15] Clavel, M., Durn, F., Eker, S., Lincoln, P., Mart-Oliet, N., Meseguer, J., Quesada, J.: The maude system. In: P. Narendran, M. Rusinowitch (eds.) Rewriting Techniques and Applications, *Lecture Notes in Computer Science*, vol. 1631, pp. 671–671. Springer Berlin / Heidelberg (1999). DOI 10.1007/3-540-48685-2\_18
- [16] Crocker, D.: Perfect developer: A tool for object-oriented formal specification and refinement. In: Tools Exhibition Notes at Formal Methods Europe (2003)
- [17] Daum, M., Maus, S., Schirmer, N., Seghir, M.: Integration of a software model checker into isabelle. In: G. Sutcliffe, A. Voronkov (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, *Lecture Notes in Computer Science*, vol. 3835, pp. 381–395. Springer Berlin / Heidelberg (2005). 10.1007/11591191\_27
- [18] Davies, J., Silva, L., da Silva Simão, A. (eds.): Formal Methods: Foundations and Applications - 13th Brazilian Symposium on Formal Methods, SBMF 2010, Natal, Brazil, November 8-11, 2010, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 6527. Springer (2011)
- [19] Depasse, C.: Constructing Isabelle proofs in a proof refinement calculus. Research Report, UCL (2001)
- [20] Filliâtre, J.C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (2003)
- [21] Freitas, L., Cavalcanti, A., Woodcock, J.: Taking our own medicine: Applying the refinement calculus to state-rich refinement model checking. In: Z. Liu, J. He (eds.) Formal Methods and Software Engineering, *Lecture Notes in Computer Science*, vol. 4260, pp. 697–716. Springer Berlin / Heidelberg (2006)
- [22] Hardin, T., Pessaux, F., Weis, P., Doligez, D.: Reference Manual of Focalize (2009). [Http://focalize.inria.fr/](http://focalize.inria.fr/)
- [23] He, J., Liu, Z., Li, X.: rCOS: A refinement calculus of object systems. *Theor. Comput. Sci.* **365**(1-2), 109–142 (2006). DOI 10.1016/j.tcs.2006.07.034
- [24] Hoare, C., He, J.: *Unifying Theories of Programming*. Prentice-Hall (1998)
- [25] Imperial, P.S., Steggle, P., Software, I.: *Z tools survey* (1994)
- [26] Ke, W., Liu, Z., Wang, S., Zhao, L.: A graph-based operational semantics of oo programs. In: Proceedings of ICFEM'09, LNCS volume 5885, pp. 347–366 (2009)
- [27] Kent, S.: Model driven engineering. In: Proceedings of the Third International Conference on Integrated Formal Methods, IFM '02, pp. 286–298. Springer-Verlag, London, UK, UK (2002)
- [28] 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)

- [29] Laibinis, L.: Mechanised formal reasoning about modular programs. Ph.D. thesis, Abo Akademi (2000)
- [30] Lamport, L.: How to write a proof. *The American Mathematical Monthly* **102**(7), 600–608 (1995)
- [31] Lei, B., Li, X., Liu, Z., Morisset, C., Stolz, V.: Robustness testing for software components. *Sci. Comput. Program.* **75**(10), 879–897 (2010)
- [32] Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with chalice. In: A. Aldini, G. Barthe, R. Gorrieri (eds.) FOSAD, *Lecture Notes in Computer Science*, vol. 5705, pp. 195–222. Springer (2009)
- [33] Leino, K.R.M., Yessenov, K.: Automated stepwise refinement of heap-manipulating code (2010)
- [34] Letouzey, P.: A new extraction for coq. In: H. Geuvers, F. Wiedijk (eds.) TYPES, *Lecture Notes in Computer Science*, vol. 2646, pp. 200–219. Springer (2002)
- [35] Liu, Z., Morisset, C., Stolz, V.: rCOS: theory and tools for component-based model driven development. Tech. Rep. 406, UNU-IIST (2009). Keynote, FSEN 2009, LNCS
- [36] Liu, Z., Morisset, C., Wang, S.: A graph-based implementation for mechanized refinement calculus of oo programs. In: Davies et al. [18], pp. 258–273
- [37] Meng, J., Quigley, C., Paulson, L.C.: Automation for interactive proof: First prototype. *Inf. Comput.* **204**(10), 1575–1596 (2006)
- [38] Mens, T., Gorp, P.V.: A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science* **152**, 125 – 142 (2006). Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)
- [39] Morgan, C.: Programming from specifications (2nd ed.). Prentice Hall International (UK) Ltd. (1994)
- [40] 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)
- [41] Paige, R.F., Ostroff, J.S.: ERC – An object-oriented refinement calculus for Eiffel. *Form. Asp. Comput.* **16**(1), 51–79 (2004)
- [42] Sekerinski, E.: A type-theoretic basis for an object-oriented refinement calculus. In: Proc. of Formal methods and object technology. Springer (1996)
- [43] Stolz, V.: An integrated multi-view model evolution framework. *Innovations in Systems and Software Engineering* **6**, 13–20 (2010)
- [44] Utting, M., Robinson, K.: Modular reasoning in an object-oriented refinement calculus. In: R. Bird, C. Morgan, J. Woodcock (eds.) Mathematics of Program Construction, *Lecture Notes in Computer Science*, vol. 669, pp. 344–367. Springer Berlin / Heidelberg (1993)

- 
- [45] von Wright, J.: Program refinement by theorem prover. In: BCS FACS Sixth Refinement Workshop – Theory and Practise of Formal Software Development. SpringerVerlag (1994)
- [46] Zeyda, F., Cavalcanti, A.: Automating refinement of circus programs. In: Lecture Notes in Computer Science, Formal Methods: Foundations and Applications, vol. 6527, pp. 274–290 (2011)