Automated Model-based Verification of Object-Oriented Code

Jonathan Ostroff, Chen-wei Wang, Eric Kerfoot and Faraz Ahmadi Torshizi

Technical Report CS-2006-05

May 24, 2006

Department of Computer Science and Engineering

4700 Keele Street North York, Ontario M3J 1P3 Canada

Automated Model-based Verification of Object-Oriented Code

Jonathan Ostroff, Chen-Wei (Jackie) Wang, Eric Kerfoot and Faraz Ahmadi Torshizi *
Department of Computer Science and Engineering, York University,
4700 Keele St., Toronto, ON M3J 1P3, Canada.

Abstract

ESpec is a suite of tools that facilitates the testing and verification of Eiffel programs in an integrated environment. The suit includes unit testing tools and Fit tables (for customer requirements) that report contract failures. This paper describes ES-Verify (part of ESpec) for automatically verifying a significant subset of Eiffel constructs written with a value semantics. The tool includes a mathematical model library (sequences, sets, bags and maps) for writing high level specifications, and a translator that converts the Eiffel code into the language used by the Perfect Developer (PD) theorem prover. Preliminary experience indicates that the vast majority of verification conditions are quickly and automatically discharged, including loop variants and invariants. ES-Verify is the first automated Eiffel verification tool (to our knowledge) and allows the developer to use the clean syntax and object-oriented structures of Eiffel, together with its mature industrial strength design by contract mechanism.

1 Introduction

A software product is reliable if it is correct (performs its tasks according to specification) and robust (reacts appropriately to abnormal conditions). How should specifications be provided and how do we check that software behaves according to its specification? Design by Contract (DbC) is a promising method for answering these questions. A class can be specified via expressive preconditions, postconditions and class invariants [12].

A variety of OO languages have followed this contracting approach to software quality such as Eiffel [12], Spec# [2], ESC/Java [9], JML [11] and UML/OCL [3]. A "lightweight" formal approach to checking the correctness of code works by runtime assertion checking, i.e. as the code is executed the contracts are checked and an exception is raised if there is a contract violation. However, we would also like to reason formally about programs and to mechanize the process of verifying the correctness of the code. Automated verification of object-oriented code has been pursued in Spec#, ESC/Java and JML.

In this paper we describe automated verification for a significant subset of Eiffel for which we have developed the following components:

- An Eiffel Model Library (ML) for specifying the abstract state without exposing implementation details. This library is similar to model-based specifications as in B [1] and Z [13], except that it is object-oriented. ML contains classes such as ML_SEQ, ML_SET, ML_BAG and ML_MAP. These classes are both mathematical (i.e. immutable) and effective (i.e. executable). They are mathematical so that software properties can be specified abstractly and effective so that when the code (specified via ML) is executed, contract violations will be reported (if any). This mathematical library is thus useful for lightweight verification even in the absence of a theorem prover.

*Email: {jonathan, faraz}@cs.yorku.ca. Eric.Kerfoot@comlab.ox.ac.uk. Supported by a grant from NSERC.

- A base library (ES_BASE) of data structures (with classes such as ESV_ARRAY, ESV_LIST, ESV_SET and ESV_TABLE) for the efficient implementation of software products. These classes have a value semantics, but for efficiency are mutable. The classes are descendants of the standard Eiffel base library classes. The prefix ESV stands for Eiffel Spec Value (semantics). While class features are contracted via ML (which while executable are inefficient due to their mathematical immutability), the bodies of the features are implemented via the base classes (which are mutable and hence efficient, but not as suitable for specifications).

- A translator that will convert Eiffel code implemented via ES_BASE and specified via ML into specifications written in the Perfect Language [7]. The advantage of this translator is that there is a highly-productive theorem prover (Perfect Developer) for converting the specification (written in the Perfect Language) into complete verification conditions and automatically discharging their proofs.

The above components (which we call ES-Verify) for automated verification of Eiffel code is under development as part of the ESpec (Eiffel Specification) toolset which is a unified environment allowing software developers to combine Fit tables (for customer requirements and acceptance tests) with contract and unit testing tools. This means that a single integrated tool can be used to specify, develop, test and verify the requirements and design of a software product. Formal verification is a substantial addition to the capabilities of the ESpec toolset, allowing for the combination of lightweight validation as well as automated deductive verification.

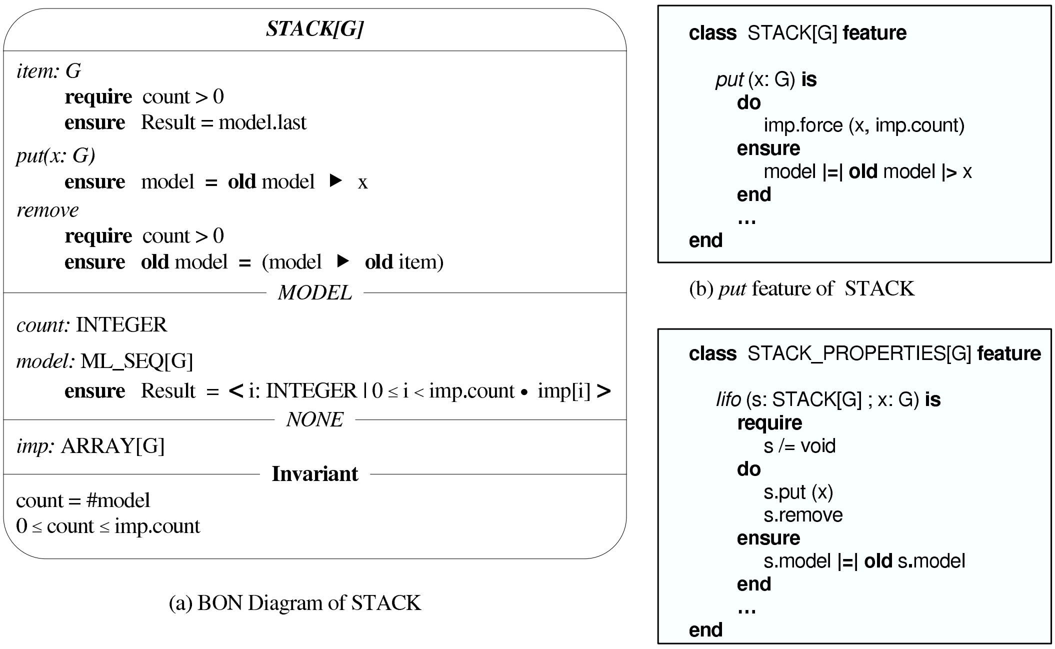
As stated, ES-Verify uses the Perfect langauge and a theorem prover. Although we are impressed with the expressiveness and power of the Perfect tools (see sequel) we have not used the Perfect specification language and theorem prover in the intended fashion. The intended use of Perfect is that developers write their specifications in the Perfect Language which is then used to automatically generate code (e.g. Java or C++). In this respect, Perfect is akin to model-driven development (MDD) methods. Perfect has a notion of refinement that can be used to improve the efficiency of the generated code.

We have examined the Java code and found that the generated code is much longer and more complex than the original contract-based specification. The MDD approach is useful if there is never a need to deal with the generated code. However, Perfect specifications are not directly executable nor there is a debugger at the model level. Thus our preference is to write code in Eiffel. Eiffel has a mature industrial strength contracting mechanism with the full set of tools such as debuggers, profilers, documentation and browsing capabilities. The language is admired for its clear syntax and expressive use of full range of object-oriented constructs such as multiple inheritance.

Our approach is to write the code in Eiffel and thus retaining the simple but expressive use of the language constructs. The Eiffel code is then translated to Perfect using (a) the refinement constructs of Perfect for the feature implementations and (b) the Perfect contracting mechanism for Eiffel contracts. The Eiffel model library (ML) was designed in order to avoid impedance mismatches between itself and the Perfect data structures. Theorem proving program involving genericity, loops (and loop invariants) is a non-trivial task and this work shows that model libraries (such as ML) must be designed with the target theorem prover in mind. In the sequel we will use the abbreviation PD both for the Perfect specification language and for the Perfect theorem prover.

2 Models via ML

As explained in [13] with reference to Z, formal specifications use mathematical notation to describe, in a precise way, the properties which a software product must have, without unduly constraining the way in which these properties are achieved. We may call the mathematical description an abstract *model* of the system under development. The model describes *what* the system must do without



(a) BON Diagram of STACK

(b) *put* feature of STACK

(c) Stack LIFO property

Figure 1: STACK[G] modelled by ML_SEQ[G]

saying *how* it is to be done. Models allow questions about what the system does to be answered confidently, without the need to disentangle the information from a mass of detailed program code, or to speculate about the meaning of phrases in an imprecisely-worded prose description.

In Z, the mathematical models are based on predicate logic and the set theory and thus obey a rich collection of mathematical laws which makes it possible to reason effectively about the way a specified system will behave, but these models are not oriented towards computer representation.

The model library (ML) described in this paper encode predicate logic acting on sets, sequences, bags and maps (as in Z), but the mathematical theories are structured as classes (producing immutable objects needed for mathematical specification) whose features (e.g. $\forall, \exists, \in$, set comprehension etc.) are pure functions executable in the object-oriented style[1].

The classes of ML are shown in Fig. 2. Contracts may be specified using ML and these contracts are executable. When runtime assertion checking is turned on, contract violations (if any) are signalled via exceptions, thus indicating an inconsistency between the implementation and the specification. The complete specification of a system and its implementation can be provided in the same compilable and executable Eiffel text (e.g. see class `STACK[G]` in Fig. 4). The immutable ML classes will be inefficient by comparison to the mutable classes in the Eiffel base library (such as `ARRAY` and `LIST`), but this is acceptable as contract checking may be turned off in the final delivered code which will use the efficient base library for implementation.

As a simple example, consider the BON [15] contract view of a stack as shown in Fig. 1a. The model of the stack consists of an `ML_SEQ[G]` (i.e. a sequence of items of type `G`, where `G` is a generic parameter) and *count* (the number of items in the stack). The contracts of all the other features of the stack can be described in terms of the sequence and *count*. In the absence of a sequence to model the stack (i.e. with just the model attribute *count*), the best postcondition for the stack push operation `put` is

[1]The Eiffel agent mechanism for iteratively applying a supplied expression to a collection is much used.

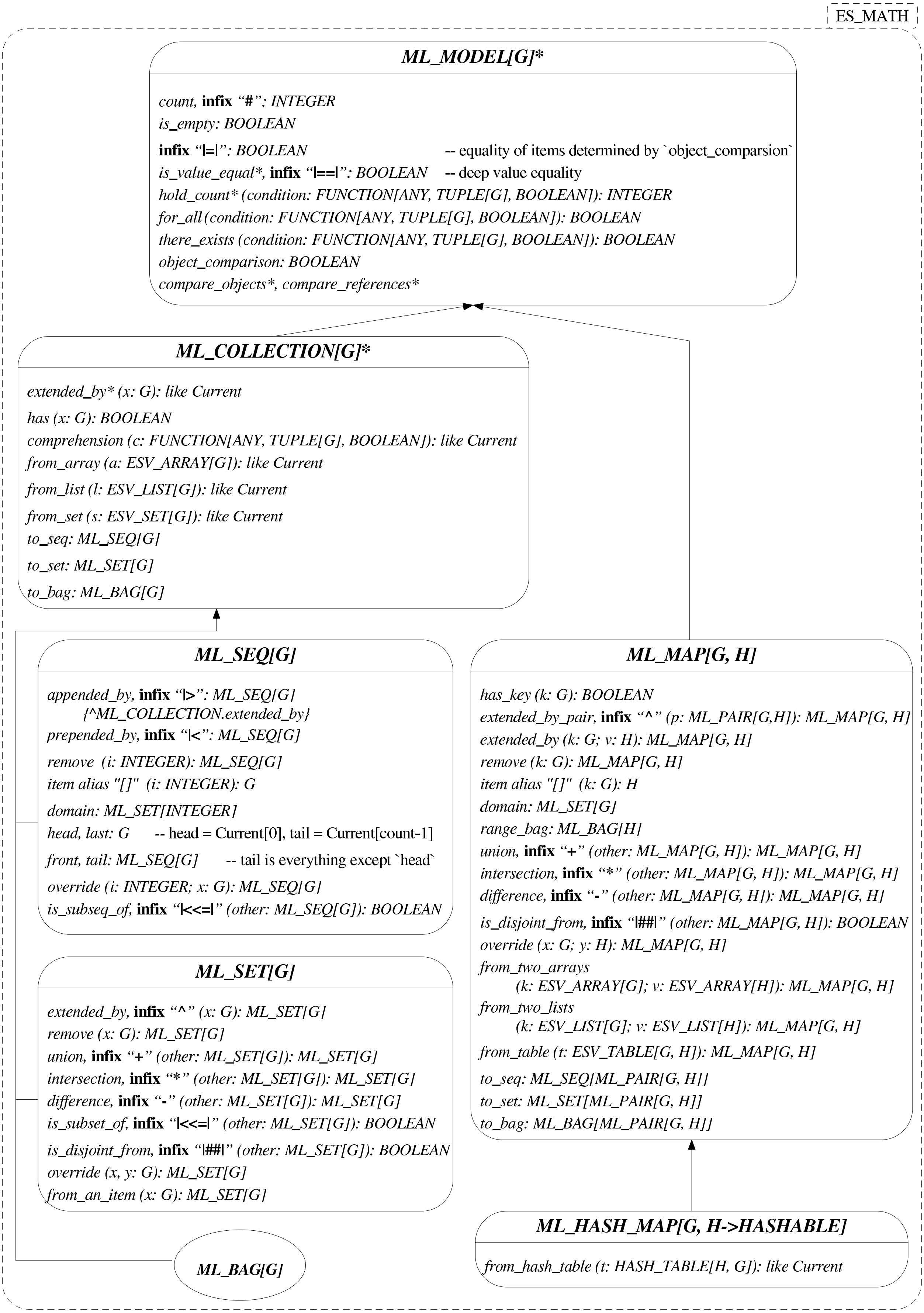


Figure 2: Core Classes in the Mathematical Library (ML) for Model-based Specification

$$count = \mathbf{old}\, count + 1 \wedge item = x \tag{1}$$

This abstract specification violates Einstein's maxim to "make everything as simple as possible, but not simpler" because the specification is incomplete. For example, an implementor can satisfy the above specification yet change old values of the stack that are not at the top (we need a frame condition that says the old part of the stack remains unchanged). However, by adding a sequence to the model we can now express the complete contract as

$$model = \mathbf{old}\, model \blacktriangleright x \tag{2}$$

where $\blacktriangleright$ is the `appended_by` (pure) function of a mathematical sequence which returns a new sequence the same as the old one, but with the argument appended to the end. Since (2) $\Rightarrow$ (1), there is also no need to write the abstract postcondition as it is entailed by the model postcondition. In addition, with the full model we can provide the complete contract for the query `item` that returns the top of the stack.

The Eiffel notation follows the BON notation quite closely as shown in Fig. 1b. For $\blacktriangleright$, we may use the `appended_by` function or alternatively the infix operator `|>` as shown in class `ML_SEQ` in Fig. 2.

Model classes such as `ML_SEQ` hold items that may be stored by reference or by value (Eiffel has the **expanded** construct for constructing value semantics). We thus introduce the notion of model equality (infix operator `|=|`) which depends on what type of comparison is requested (see `ML_MODEL` in Fig. 2). The default is that two model sequences (say $s1$ and $s2$) are compared via reference equality (i.e. $s1 \mid = \mid s2$ iff the two sequences have the same size and the items stored at each index refer to the same object). A specifier may invoke feature `compare_objects` (see `ML_MODEL`), in which case the items stored at each index are compared based on how the inherited feature `is_equal` (of the instantiated generic type `G`) is defined[2].

With our contracts complete, and even in the absence of implementation details, we may already begin to validate our specification based only on the model. For example, the last-in-first-out (LIFO) property of the stack can be specified as shown in Fig. 1c. In the abscence of implementation we cannot execute or unit test the LIFO property. However, with the translator and theorem prover (see sequel), the LIFO property will prove with a warning that the body of `put` and `remove` must be refined to an implementation.

We must now refine the specification to an efficient implementation. There are two steps. First choose an efficient representation such as an array or linked list. Then define the abstraction relation between the concrete representation and the mathematical model. The contracts of all features remain the same as they are all described in terms of the model.

We may use `ARRAY` from the Eiffel base library or the efficient (mutable) class if a value semantics rather than a reference semantics is preferred (i.e. we would declare `imp:ESV_ARRAY[G]`). The prefix "ESV" in class `ESV_ARRAY` stands for an "ESpec Value" array, which is part of the ESpec base library (built on top of the Eiffel base library) for implementing code using a value semantics.

Next we need to define the relationship between the abstract space in which the abstract program is written (`model`), and the space of the concrete representation (`imp`). This can be accomplished by giving an abstraction function which maps the concrete variables into the abstract objects which they represent. We may do this as follows. The body of the query `model: ML_SEQ[G]` for the stack in Fig. 1 could be a loop that iterates through the implementation array and returns an equivalent sequence with the same elements as the array (i.e. we "lift" the mutable array into a mathematical immutable sequence). The abstraction function [10] is captured by the postcondition of query `model` as follows:

[2] `is_equal` in Eiffel is similar to `equals` in Java

$$Result = \langle i : INTEGER \mid 0 \leq i < imp.count \bullet imp[i] \rangle \tag{3}$$

where the angle brackets $\langle\,\rangle$ stand for sequence comprehension in the same way that $\{\,\}$ stands for set comprehension. For example, $\{i : INT \mid 0 \leq i \leq 2 \bullet i+1\} = \{1,2,3\}$. Set, bag, sequence and map comprehension present expressive notation for abstraction functions which is supported in ML. The Eiffel ML library uses the agent construct for writing comprehension (see Fig. 2). However, for the postcondition of `model` we may use one of the pre-defined functions `from_array` that "lifts" an efficient mutable array into a mathematical sequence, so that the postcondition (3) writen in ML becomes:

```
Result |=| Result.make.from_array(imp.subarray(0,count-1))
```

Function `from_array` returns a new sequence whose items refer to the same items as in the array `imp` between $0 \cdots count - 1$. Thus, the above assertion says that the resulting sequence returned by the model is model-equal to the items of the implementation array treated as a sequence.

2.1 The Birthday Book example – ML specifications and loop invariants

The author of [14] reports that a web-enabled database system, consisting of 35,799 lines of Perfect, generated 9810 proof obligations which were proven automatically in 4.5 hours (1.6 seconds per proof) on a modest laptop. We believe that the above performance is sustainable for reasonable chunks of code where there is minimal refinement and PD does the code generation. However, in our case where there is refinement from high level models to more complex constructs (e.g. loops with loop variants and invariants), then the demands on PD are much greater. Nevertheless, by careful matching of ML to PD facilities and tuning of the translator, we can achieve proofs of the vast majority (if not all) verification conditions.

The birthday book example [13] nicely illustrates refinement to loops and more intensive use of ML as shown by the BON diagram in Fig. 3a. The model for the birthday book is the combination of the number of name-and-date pairs stored (i.e. `count`) together with an `ML_MAP[NAME, DATE]`, i.e. a set of pairs of name and date. Alternatively, this map is a function whose domain is a set of names and whose range is a bag of dates. The features of the birthday book include the ability to add a new pair (e.g. $[Peter, (March\ 1)]$), find a birthday given a name, and a `remind` function that for a given date d returns the set of names whose birthday is on d.

The `remind` function returns a set of names (`SET[NAME]`) where `SET` is an efficient mutable collection in the Eiffel base library. The birthday book is implemented as two arrays one for names and the other for dates. The postcondition of the `remind` query is

$$\{n : NAME \mid Result.has(n) \bullet n\} = \{n \in model.domain \mid model[n] = d \;\bullet\; n\} \tag{4}$$

Thus, in the postcondition for the provided date d, the RHS expression $\{n \in model.domain \mid model[n] = d \;\bullet\; n\}$ means the set of all names in the domain of the model who have birthdays on the date d. This must be equal to the LHS which is the set of all names returned by the `remind` function. The Eiffel notation is shown in Fig. 3b. The postcondition of the remind query (4) is:

```
model_set.from_set(Result)  |=|  model.comprehension( agent date_matches (?, ?, d)).domain
```

The agent function used in the postcondition (and loop invariant) is:

```
date_matches (x: NAME; y, date: DATE): BOOLEAN is
    do
        if y.is_equal (date) then
            Result := true else Result := false
        end
    end
```

BIRTHDAY_BOOK

add_birthday (n:NAME; d: DATE)
  require $\neg$ model.has_key(n)
  ensure count = **old** count + 1 **and** model = (**old** model) ^ [n, d]

find_birthday (n:NAME): DATE
  require model.has_key(n)
  ensure Result = model[n] **and** model = **old** model

remind (d: DATE): SET[NAME]
  ensure {n: NAME | Result.has(n) $\bullet$ n} = {n $\in$ model.domain | model[n] = d $\bullet$ n}
  model = **old** model

MODEL

count: INTEGER

model: MAP[NAME, DATE]
  ensure Result = **[** i: INTEGER | names.lower $\le$ i $\le$ names.upper $\bullet$ [names[i], dates[i]] **]**

NONE

names: ARRAY[NAME]
dates: ARRAY[DATE]

Invariant

count = #model
names.count = dates.count **and** names.is_unique

(a) BON Diagram of BIRTHDAY_BOOK

```
class BRITHDAY_BOOK feature

  remind (n: NAME ; d: DATE): SET[NAME] is
   local
     i : INTEGER
   do
     create Result.make
     from
       i := dates.lower
     invariant
       pd_modify ("i, Result")
       i >= 0 and then i <= names.count
       i < names.count implies names.valid_index (i)
       inv: -- see text
     variant
       dates.count - i
     until
       i = dates.count
     loop
       if dates.item (i).is_equal (today) then
         Result.extend (names[i])
       end
       i := i + 1
     end
   ensure
     model_set.from_set (Result) |=|
     model.comprehension (agent date_matches (?, ?, d)).domain
   end
   ...
end
```

(b) *remind* feature of BIRTHDAY_BOOK

Figure 3: Birthday Book

The loop invariant can now be constructed to approximate the postcondition by defining a slice of the model to the loop counter i as follows:

$$\text{modelslice}(i, names, dates) \;\widehat{=}\; \langle\!\langle j : INTEGER \mid 0 \le j < i \bullet [names[j], dates[j]] \rangle\!\rangle \tag{5}$$

The loop invariant for the remind query is similar to the postcondition:

$$\begin{aligned}\{n : NAME \mid Result.has(n) \bullet n\} \quad = \quad & \{n \in \text{modelslice}(i, names, dates).domain \mid \\ & model[n] = d \;\bullet\; n\}\end{aligned} \tag{6}$$

The equivalent Eiffel loop invariant (`inv` in Fig. 3b) is

```
model_set.from_set (Result) |=|
model.from_two_arrays(names.subarray (0, i-1),dates.subarray(0,i-1)).
comprehension( agent date_matches (?, ?, today).domain).
```

3 The Eiffel to PD Translator

Underlying Theorem Prover

Our goal is to automatically verify Eiffel code specified via ML as in the stack and birthday book examples. The question would be, which theorem prover do we use? The *Perfect Developer* (PD) specification language and theorem prover [5] is a technically mature product that is aligned with the object-orientation and design by contract paradigms. PD theorem prover has about the same level of power and automation as *Simplify* [6] (used for program checking in Spec# and ESC/Java). *Simplify* handles integers and booleans at the primitive level while PD has a greater repertoire (e.g. reals, characters, and strings). PD specification language also has a library of generic sequences, sets, bags and maps well-suited to ML [7]. A limitation of PD is that it discourages reference semantics. It is well-known that the presence of multiple references to a common object causes aliasing and makes sound and complete static verification problematic. Therefore PD, unlike say Java and Eiffel, adopts value semantics by default and discourages the use of reference semantics [3]. Despite these limitations, we have adopted PD for automated deduction in our ES-Verify tool, and we are in the process of constructing a library of base Eiffel classes in value semantics (see Introduction) using the Eiffel **expanded** construct. As a future goal we intend to expand our tool to full reference semantics

The theoretical foundations of PD are Floyd-Hoare logic and Dijkstra's weakest precondition calculus and it has the power of first-order predicate calculus, as well as a few higher-order constructs. The prover generates verification conditions and aims for verifying the total correctness (termination and refinement satisfying specification) of the input code. It delivers either a proof, upon success in discharging all verification conditions, or otherwise a list of warnings, possibly accompanied by useful fix suggestions. Output from the prover can be in formats such as HTML or Tex [4]. From an academic point of view, there is a lack of information about the inner workings of the PD theorem prover (as opposed to an interactive theorem-proving system such as *Isabelle* [3]). Ideally, the logical rules used in correctness proofs, should be open for inspection so that independent trust can be established. However, the PD theorem prover does provide the complete proof, and thus the product is robust and suitable for engineering use [8]. Fig. 4 shows how the Eiffel stack example is translated into a PD specification.

Outline of Class Translation

The translator assumes that all Eiffel classes to be translated have already been compiled and type checked. On the Eiffel side (left of Fig. 4), there are three different **feature** declarations: the *public* feature declaration[4], the *model* feature declaration[5], and the *implementation* feature declaration[6]. And on the PD side, there are also three different sections: **abstract**, **internal** and **interface**.

We first consider the Eiffel public feature declaration. Each Eiffel public attribute (e.g. `count`) becomes a variable (i.e. **var** declaration) in the PD abstract section. In order to allow client classes to access this variable, it must also be redeclared as a function in the PD interface section (hence the first line in the PD interface section reads `function` count). Each Eiffel public command (e.g. `put`) becomes a **schema** in the PD interface section. Each Eiffel public query (e.g. `item`) becomes a **function** in the PD interface section.

We then consider the Eiffel model feature declaration. In stack we only have the query `model`, but in general we may have attributes and queries (but no commands) in this declaration. Each Eiffel model attribute becomes a variable in the PD abstract section. Each Eiffel model query (which is essentially the abstraction function), not only becomes a variable in the PD abstract section, but also becomes two functions in the PD internal section. The first PD function uses the same name as the Eiffel model query and its definition (expression following symbols ˆ =, i.e. is-defined-as) corresponds to the translated postcondition[7] of the that query. The second PD function is a twin function[8] with a `_verification` name suffix. This twin function has the same definition but with a

[3]In PD, if reference semantics is adopted, then, roughly speaking, a `heap` declaration, e.g. `heap` MyHeap, would be required. For a reference entity v of type `T`, its declaration would be `v: ref T on MyHeap`. And its call to an applicable method m would be `v.value.m`, where `value` is the de-reference operator. Although we have several simple PD examples on basic aliasing effect, we have not yet experienced much the power of the prover on handling reference semantics. Escher Technologies Ltd. is in the process of developing a new beta intending to properly handle the issue.

[4]The part under the label **feature**{ANY}.

[5]The part under the label **feature**{ML_MODEL, ANY}.

[6]The part under the label **feature**{ML_MODEL}.

[7]More precisely, RHS of the first assertion clause which is a matching type with it of that query.

[8]This twin function is needed because future versions of PD will disallow refinement/implementations of abstraction functions. Since we desire to verify that the `model` implementation satisfies its postcondition, we need this twin function.

refinement (`via...end` segment) underneath which is the translated body of the Eiffel model query. In stack the Eiffel query `model` becomes (a) a variable in the PD abstract section, and (b) a function `model` and its twin refined function `model_verification` in the PD internal section.

Now we consider the Eiffel implementation feature declaration. All features under this declaration appear in the PD internal section in the obvious way, i.e. Eiffel attributes become PD variables, Eiffel queries become PD functions, and Eiffel commands become PD schemas. Moreover, since Eiffel agent expressions in loop invariants are private, they should be declared in this feature declaration; however, agent expressions in pre/postconditions may be declared in either the public or model feature declaration part. One such example is the agent function `date_matches` occurring in the loop invariant and postcondition of remind feature in birthday book.

Finally we consider the Eiffel class invariants: those clauses that only refer to public or model attributes become equivalent invariants in the PD abstract section; otherwise, they become equivalent invariants in the PD internal section.

Outline of Routine Translation: Eiffel commands and queries become PD schemas and functions, respectively. For a command that may modify the current object, frame constraints are needed. In order to specify frame constraints, PD supports a **change** clause[9]. For translation into PD, we use in Eiffel specification a `pd_modify`[10] declaration with its string argument become a list of attributes that the PD schema may change. For an Eiffel command or query, its require clause and ensure clause[11] appear as equivalent PD **pre** and **satisfy** clauses, respectively. For Eiffel command, its ensure clause (with its modify declaration) appears as the equivalent PD change and satisfy clauses under a **post** declaration[12]. Moreover, the Eiffel **old** notation for the value of expressions in a prestate is converted into the equivalent PD primed notation. Finally, the body of an Eiffel command or querry appears as an equivalent PD **via ... end** refinement segment.

4 Conclusion

When the PD translator is applied to the Eiffel code for the birthday book example, the theorem prover generates 158 verification conditions which are *all* automatically discharged. This includes proof of termination via the loop variant and invariant. For the two implementation arrays we used the value semantics class `ESV_ARRAY`. Preliminary experience with other examples indicates that the vast majority of verification conditions are quickly and automatically discharged, including loop variants and invariants, without any interaction with the user. The user may add axioms (with the danger of introducing inconsistencies) or assertions to help the theorem prover, but this is mostly unnecessary.

We have presented in this paper a system where we make use of the mathematical but executable ML library and the translator to convert clean and expressive Eiffel code into PD for automated verification of the implementations. The translation process translates each Eiffel construct into an equivalent PD construct so that this one-to-one relation between Eiffel and PD constructs allows us to assign the semantics of the PD language to that of Eiffel (rather than the use of traditional semantic methods such as operational or Action Semantics.). Of course such a semantics depends upon the soundness of PD. Future work aims to extend the verification to the full reference semantics.

Acknowledgements: We deeply appreciate the help we have received from David Crocker of Escher Technologies with the Perfect toolset. Likewise we would like to acknowledge helpful feedback

[9] The new ECMA specification for Eiffel has a somewhat equivalent **only** clause.

[10] A boolean function that takes as argument a string and always returns true, hence can always pass the run-time contract checking. Expression `pd_modify("*")` is an abbreviation meaning all attributes may change.

[11] pd_modify declaration in the ensure clause is replaced with true in PD.

[12] An Eiffel query is translated in the same way as it for a command except there is no modify declaration in its postcondition, and thus there exists no change list and post declaration for its translation in PD.

```
class  MY_STACK[G] create
   make

feature {ANY} -- public feature declaration

   make is -- constructor
      do
         create imp ; count := 0
      ensure
         pd_modify ("*")
         # model = 0 and  count = 0
      end

   count: INTEGER

   item: G is
      require count > 0
      do
         Result := imp [count - 1]
      ensure Result = model.last
      end

   put(x: G) is
      do
         if imp.is_empty then imp.force (x, 0)
         else
            if count = imp.count then
               imp.grow (imp.count * 2)
            end
            imp.put (x, count)
         end
         count := count + 1
      ensure
         pd_modify ("*")
         count = old count + 1 and then  model |=| (old model |> x)
      end

feature {ML_MODEL} -- implementation feature declaration
   imp: ESV_ARRAY[G]

feature {ML_MODEL, ANY} -- model feature declaration
   model: ML_SEQ[G] is
      do
         create Result.make; Result := Result.from_array(imp.subarray (0, count -1))
      ensure
         Result |=| Result.from_array (imp.subarray (0, count -1 ))
      end

invariant
   count >= 0 and then count <= imp.count  and then count <= # model
end
```

```
import
   "ESV_ARRAY.pd",  "ML_COLLECTION.pd";

class MY_STACK of ( G ) ^=
   abstract
      var model:  ML_SEQ of ( G ),
            count: int;
      invariant count <= #model;

   internal         //refinement
      var imp: ESV_ARRAY of ( G );
      invariant count >= 0 & count <= #imp;

      function model
         ^= (for i :: 0 .. <#imp.slice(0, (count-1)-0+1) yield imp.slice(0, (count-1)-0+1)[i]);

      function model_verification: ML_SEQ of ( G )
         ^= (for i :: 0 .. <#imp.slice(0, (count-1)-0+1) yield imp.slice(0, (count-1)-0+1)[i])
         via
            var Result:  ML_SEQ of ( G ); Result! =  ML_SEQ of ( G ){} ;
            Result! =
               (for i :: 0 .. <#imp.slice(0, (count-1)-0+1) yield imp.slice(0, (count-1)-0+1)[i]);
            value Result;
         end;

   interface        //public methods
      function count;

      build {}  //constructor equivalent to Eiffel `make'
         post
            change model, count satisfy #model' = 0 & self'.count = 0
         via imp! =  ESV_ARRAY of ( G ){} ; count! = 0 end;

      schema! put(x : G)
         post
            change
               model, count
            satisfy
               self'.count = count + 1, model' = model.append( x )
         via
            if [imp.empty]: imp! = force @ ESV_ARRAY_HELPER of  G (imp, x, 0) ;
               []:
                  if [count = #imp]:
                        imp! = grow @ ESV_ARRAY_HELPER of  G (imp, #imp*2);
                     []: pass
                  fi ; imp! = put @ ESV_ARRAY_HELPER of  G (imp, x, count)
            fi ; count! = count + 1
         end;

      function item: G
         pre count > 0 satisfy result = model.last
         via var Result:  G ; Result! = imp[count - 1] ; value Result end;

end;
```

Figure 4: STACK example: The Translation Layout from Eiffel into Perfect Language

from Bertrand Meyer and Bernd Schoeller of ETH Zurich. This work was funded by a Discovery Grant from NSERC.

References

[1] Jean-Raymond Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996. ISBN 0 521 49619 5 (hardback).

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume LNCS 3362. Springer Verlag, 2004.

[3] Achim D. Brucker and Burkhart Wolff. A Proposal for a Formal Ocl Semantics in Isabelle/Hol. In *Theorem Proving in Higher Order Logics*, volume LNCS 2410. Springer-Verlag, 2002.

[4] David Crocker. Perfect Developer: A tool for Object-Oriented Formal Specification and Refinement. In *Tools Exhibition Notes at Formal Methods Europe*, 2003.

[5] David Crocker. Safe Object-Oriented Software: The Verified Desing-By-Contract Paradigm. In F.Redmill & T.Anderson, editor, *Twelfth Safety-Critical Systems Symposium*, pages 19–41. Springer-Verlag, London, 2004.

[6] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[7] Escher Technologies. *Perfect Developer Language Reference Manual*, 3.0 edition, December 2004. Available from www.eschertech.com.

[8] Ingo Feinerer. Formal Program Verification: a Comparison of Selected Tools and Their Theoretical Foundations. Master's thesis, Vienna University of Technology, January 2005.

[9] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[10] C. A. R. Hoare. Proof of Correctness of Data Representations. In *Acta Informatica*, volume 1, pages 271–281. Springer-Verlag, February 1972.

[11] Gary T. Leavens, K. Rustan M. Leino, and Peter Mller. Specification and verification challenges for sequential object-oriented programs. TR 06-14, Department of Computer Science, Iowa State University, May 2006.

[12] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997. 0-13-629155-4.

[13] J.M. Spivey. *The Z Notation: A Reference Manual (2nd edition).* Prentice-Hall, Englewood Cliffs, N.J., 1992.

[14] Brian Stevens. Implementing Object-Z with PerfectDeveloper. *Journal of Object Technology*, 6(2):189–202, March-April 2006.

[15] Kim Walden and Jean-Marc Nerson. *Seamless Object Oriented Software and Architecture.* Prentice Hall, 1995. Seamless Object Oriented Software and Architecture.