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 object-oriented Eiffel programs in an integrated environment. The suite includes unit testing tools (ES-Test) and Fit tables (ES-Fit 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 and allows the developer to use the clean syntax and object-oriented structures of Eiffel, together with its mature industrial-strength design by contract (DbC) 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, post-conditions and class invariants [19].

A variety of object-oriented languages have followed this contracting approach to software quality such as Eiffel [19], Spec# [4, 3], JML [17] tools like ESC/Java2 [10, 7], and UML/OCL [5]. A "lightweight" formal approach to checking the correctness of code works by runtime assertion checking, i.e. the contracts are checked as the code is executed and an exception is raised if there is a contract violation. However, we would also like to reason formally about the correctness of programs and to mechanize such process. Automated verification of object-oriented code has been pursued in systems such as Spec# and JML tools like ESC/Java2.

ESpec (Eiffel Specification) toolset is a unified environment allowing software developers to combine Fit tables (ES-Fit for customer requirements and acceptance tests) with contracts and unit testing tools (ES-Test). 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 a combination of lightweight validation and automated deductive verification.

In this paper we describe the automated model-based verification for a significant subset of Eiffel. The following three components, which together we call the ES-Verify, are under development as part of the ESpec suite:

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

- An Eiffel Model Library (ML) for specifying the abstract state of a program without exposing its implementation details. This library is similar to the model-based specifications as in B [1] and Z [20], except that it is object-oriented. ML contains classes such as ML_SEQ, ML_SET, ML_BAG, and ML_MAP. These classes are both immutable and executable. They are immutable so that software properties specified in the pre- and post- conditions as well as the class invariants can be based on them. They are executable so that contract violations will be reported (if any). This mathematical library is thus useful for lightweight verification even in the absence of a theorem prover.

- An Eiffel base library (ES_BASE) of data structures (classes such as ESV_ARRAY, ESV_LIST, ESV_SET, and ESV_TABLE) for the efficient implementation of software products. The prefix "ESV" stands for an "ESpec Value" structure, which is part of the ESpec base library (built on top of the Eiffel base library via inheritance) for implementing code. These ESV classes apply a value semantics [12], but for efficiency they are mutable. While class features are contracted via ML (which are executable but inefficient due to their mathematical immutability), their bodies are implemented via the ES_BASE classes (which are mutable and hence efficient, but not as suitable for specifications as ML ones).

- A translator that will convert Eiffel code implemented via ES_BASE and specified via ML into an equivalence written in a specification language Perfect [14]. The advantage of this translator is that there is, associated with the Perfect language, a fully-automated reasoning tool - Perfect Developer (PD) - that fits well for our source Eiffel code. PD supports object-oriented, model-driven, and DbC software development as well as its verification [11]. PD converts its specification (written in the Perfect Language) into complete verification conditions and attempts to automatically discharge their proofs.

As stated, ES-Verify uses the PD tools (the Perfect language and its associated theorem prover). Although we are impressed by the expressiveness and power of the PD tools, we have not used them in the intended fashion. The intended use of PD tools is that developers write their specifications in the Perfect Language, which is then used to automatically generate executable code (e.g. Java or C++). In this respect, Perfect is akin to model-driven development (MDD) methods. Perfect also 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 - much longer and more complex than the original contract-based specification - is not intended to be read. The MDD approach is useful if there is never a need to deal with the generated code. However, Perfect specifications are neither directly executable nor is there a debugger at the model level. As a result, our preference is to write code in Eiffel. Eiffel has a mature industrial-strength contracting mechanism with a full set of tools such as debuggers, profilers, documentation, and browsing capabilities. The language is admired for its clear syntax and expressive use of a 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 its language constructs. The Eiffel code is then translated into Perfect using (a) the Perfect refinement constructs for Eiffel feature implementations and (b) the Perfect contracting mechanism for Eiffel contracts. The Eiffel model library (ML) was designed in order to avoid mismatches between itself and the Perfect data structures. Theorem proving program involving genericity and loops (with their 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 for the combination the Perfect specification language and its associated theorem prover.

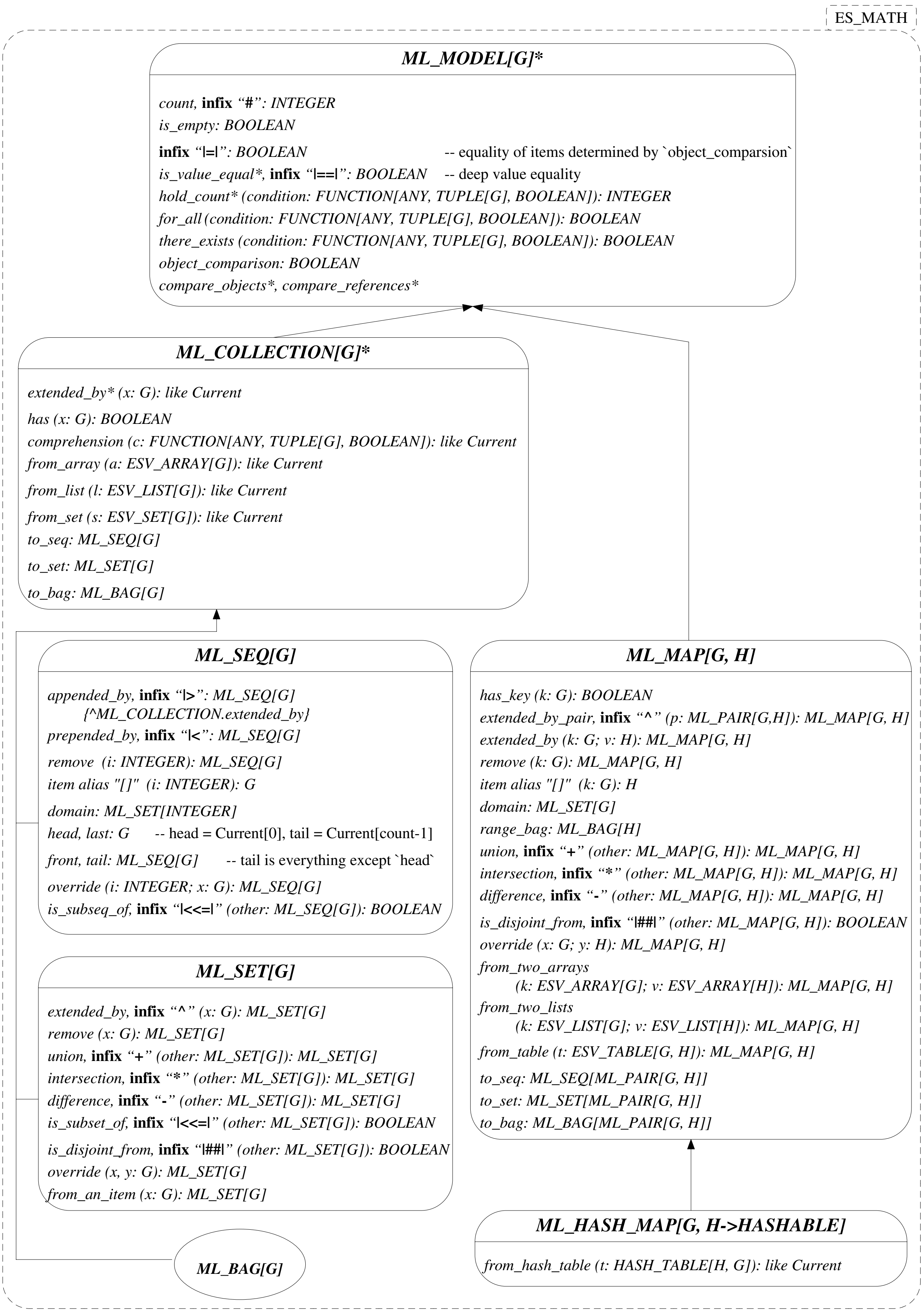
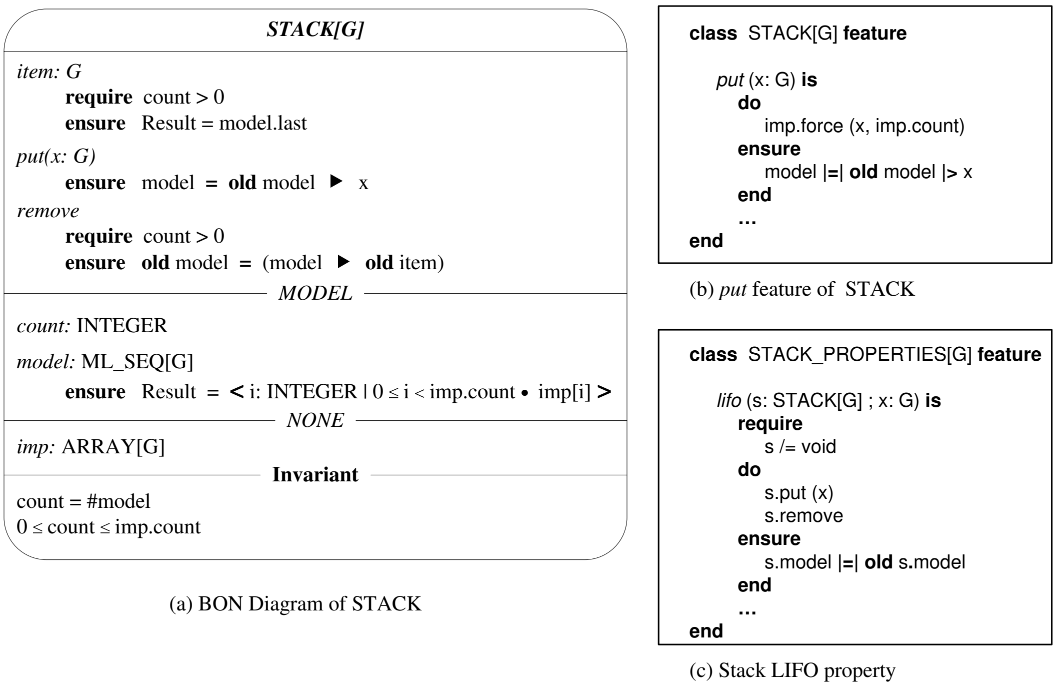

ES_MATH
ML_MODEL[G]*
count, infix “#”: INTEGER
is_empty: BOOLEAN
infix “|=|”: BOOLEAN -- equality of items determined by `object_comparsion`
is_value_equal*, infix “|==|”: BOOLEAN -- deep value equality
hold_count* (condition: FUNCTION[ANY, TUPLE[G], BOOLEAN]): INTEGER
for_all (condition: FUNCTION[ANY, TUPLE[G], BOOLEAN]): BOOLEAN
there_exists (condition: FUNCTION[ANY, TUPLE[G], BOOLEAN]): BOOLEAN
object_comparison: BOOLEAN
compare_objects*, compare_references*
ML_COLLECTION[G]*
extended_by* (x: G): like Current
has (x: G): BOOLEAN
comprehension (c: FUNCTION[ANY, TUPLE[G], BOOLEAN]): like Current
from_array (a: ESV_ARRAY[G]): like Current
from_list (l: ESV_LIST[G]): like Current
from_set (s: ESV_SET[G]): like Current
to_seq: ML_SEQ[G]
to_set: ML_SET[G]
to_bag: ML_BAG[G]
ML_SEQ[G]
appended_by, infix “|>”: ML_SEQ[G]
{^ML_COLLECTION.extended_by}
prepended_by, infix “|<”: ML_SEQ[G]
remove (i: INTEGER): ML_SEQ[G]
item alias "[]" (i: INTEGER): G
domain: ML_SET[INTEGER]
head, last: G -- head = Current[0], tail = Current[count-1]
front, tail: ML_SEQ[G] -- tail is everything except `head`
override (i: INTEGER; x: G): ML_SEQ[G]
is_subseq_of, infix “|<<=|” (other: ML_SEQ[G]): BOOLEAN
ML_SET[G]
extended_by, infix “^” (x: G): ML_SET[G]
remove (x: G): ML_SET[G]
union, infix “+” (other: ML_SET[G]): ML_SET[G]
intersection, infix “*” (other: ML_SET[G]): ML_SET[G]
difference, infix “-” (other: ML_SET[G]): ML_SET[G]
is_subset_of, infix “|<<=|” (other: ML_SET[G]): BOOLEAN
is_disjoint_from, infix “|##|” (other: ML_SET[G]): BOOLEAN
override (x, y: G): ML_SET[G]
from_an_item (x: G): ML_SET[G]
ML_BAG[G]
ML_MAP[G, H]
has_key (k: G): BOOLEAN
extended_by_pair, infix “^” (p: ML_PAIR[G,H]): ML_MAP[G, H]
extended_by (k: G; v: H): ML_MAP[G, H]
remove (k: G): ML_MAP[G, H]
item alias "[]" (k: G): H
domain: ML_SET[G]
range_bag: ML_BAG[H]
union, infix “+” (other: ML_MAP[G, H]): ML_MAP[G, H]
intersection, infix “*” (other: ML_MAP[G, H]): ML_MAP[G, H]
difference, infix “-” (other: ML_MAP[G, H]): ML_MAP[G, H]
is_disjoint_from, infix “|##|” (other: ML_MAP[G, H]): BOOLEAN
override (x: G; y: H): ML_MAP[G, H]
from_two_arrays
(k: ESV_ARRAY[G]; v: ESV_ARRAY[H]): ML_MAP[G, H]
from_two_lists
(k: ESV_LIST[G]; v: ESV_LIST[H]): ML_MAP[G, H]
from_table (t: ESV_TABLE[G, H]): ML_MAP[G, H]
to_seq: ML_SEQ[ML_PAIR[G, H]]
to_set: ML_SET[ML_PAIR[G, H]]
to_bag: ML_BAG[ML_PAIR[G, H]]
ML_HASH_MAP[G, H->HASHABLE]
from_hash_table (t: HASH_TABLE[H, G]): like Current

Figure 1: Core Classes in the Mathematical Library (ML) for Model-based Specification



```
STACK[G]
item: G
    require count > 0
    ensure Result = model.last
put(x: G)
    ensure model = old model ▶ x
remove
    require count > 0
    ensure old model = (model ▶ old item)
—————— MODEL ——————
count: INTEGER
model: ML_SEQ[G]
    ensure Result = < i: INTEGER | 0 ≤ i < imp.count • imp[i] >
—————— NONE ——————
imp: ARRAY[G]
—————— Invariant ——————
count = #model
0 ≤ count ≤ imp.count
```

(a) BON Diagram of STACK

```
class STACK[G] feature

    put (x: G) is
        do
            imp.force (x, imp.count)
        ensure
            model |=| old model |> x
        end
        ...
end
```

(b) *put* feature of STACK

```
class STACK_PROPERTIES[G] feature

    lifo (s: STACK[G] ; x: G) is
        require
            s /= void
        do
            s.put (x)
            s.remove
        ensure
            s.model |=| old s.model
        end
        ...
end
```

(c) Stack LIFO property

Figure 2: STACK[G] modelled by ML_SEQ[G]

2 Models via ML

As explained in [20] 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 saying *how* it is to be done. Models allow questions about what the system does to be answered confidently, without the need to either disentangle the information from a mass of detailed program code, or 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 effectively reason 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 encodes predicate logic acting on sets, sequences, bags, and maps (as in Z), but the mathematical theories are structured as classes (instantiated to 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. The Eiffel agent mechanism for iteratively applying a supplied expression to a collection is much used.

The classes of ML are shown in Fig. 1. 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 its 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 (due to its re-construction of a new ML object every time a feature such as `appended_by` is invoked), by comparison to the mutable classes in the Eiffel or ES 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 only use the efficient base library for implementation.

As a simple example, consider the BON [22] contract view of a generic stack as shown in Fig. 2a.

The model of the stack consists of a `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 post-condition for the stack push operation `put` is

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

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

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

where ▶ is the `appended_by` (pure) function of a mathematical sequence that returns a new sequence same as the old one, but with the argument item appended to the end. Since (2) ⇒ (1), there is then no need to write (1) as it is entailed by the model post-condition. With the full model we can then provide the complete contracts for the pop operation `remove` and the query `item` that returns the top of the stack. The Eiffel notation follows the BON notation quite closely as shown in Fig. 2b. For ▶, we may use either the `appended_by` function or alternatively the infix operator `|>` as shown in class `ML_SEQ` in Fig. 1.

Model classes such as `ML_SEQ` hold items that may be stored either by reference or by value. Eiffel has the **expanded** construct for constructing a 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. 1). The default is that two model sequences (say $s1$ and $s2$) are compared for their stored items via reference equality (i.e. $s1$ |=| $s2$ iff the two sequences have the same size and the items stored at each index both refer to the same object). A specifier may invoke feature `compare_objects` (see `ML_MODEL`), in which case the items stored at each index will be compared based on how the inherited feature `is_equal` (of the actual generic type `G`) is defined [1].

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. 2c. In the absence of implementation, we cannot execute or unit test the LIFO property. However, with the translator and theorem prover, the LIFO property will prove with a warning that the body of `put` and `remove` must be refined with an implementation.

We must now refine the specification to an efficient implementation. We choose mutable structures such as an array or linked list. We may use `ARRAY` from the Eiffel base library, or from the ES base library if a value semantics rather than a reference semantics is desired (i.e. by declaring `imp:ESV_ARRAY[G]`).

Next we need to define the abstraction relation between the abstract space in which the abstract program is written (i.e. `model`) and the space of the concrete representation (i.e. `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` (a `ML_SEQ[G]`) for the stack in Fig. 2 could be a loop that iterates through the implementation array and returns an equivalent sequence with the same elements as the array. That is, we "lift" the mutable array into a mathematical immutable sequence. The abstraction function [16] is captured by the post-condition of query `model` as follows:

[1] `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 or map comprehension presents expressive notation for abstraction functions and is supported in ML. The Eiffel ML library uses the agent construct for writing comprehension (see Fig. 1). However, for the post-condition of `model` we may use one of the pre-defined ML functions `from_array` that "lifts" an efficient mutable array to a mathematical sequence. Function `from_array` returns a new sequence whose items refer to the same items as in the array `imp` between $0 \cdots count - 1$. So the post-condition (3) writen in ML becomes:

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

which asserts that the resulting sequence returned by the model is model-equal to the implementation array treated as a sequence. The contracts of all other features remain the same as they are all described in terms of `model`.

2.1 The Birthday Book example – ML specifications and loop invariants

The author of [21] reports that a web-enabled database system, consisting of 35,799 lines of Perfect, generated 9810 proof obligations and proved 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 but there is minimal refinement and PD does the code generation. However, in our case there is refinement from high level models to more complex constructs (e.g. loops their variants and invariants), and thus the demands on PD are much greater. Nevertheless, by means of careful matching between ML and PD data structures as well as tuning of the translator, we can achieve proofs of the vast majority (if not all) verification conditions.

The birthday book example [20] 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 a combination of the number of name-and-date pairs stored (i.e. `count`) and a `ML_MAP[NAME, DATE]` (i.e. a set of name-and-date pairs). 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.

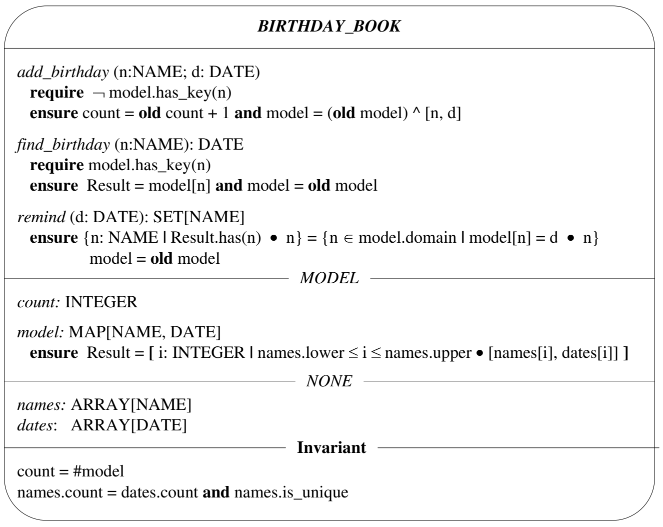
The `remind` function returns a set of names (`SET[NAME]`) where `SET` is an efficient mutable structure from either the Eiffel or ES base library. The birthday book is implemented as two arrays: one for names and the other for dates. The post-condition of the `remind` query is

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

where the RHS expression means the set of all names, from the domain of the model map, whose birthday is on the provided date d. And this must be equal to the LHS expression which represents the set of all names returned by the `remind` function. The Eiffel notation for the `remind` function is shown in Fig. 3b. The Eiffel post-condition of the `remind` query in (4) now becomes:

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

The agent function used in the post-condition (and loop invariant) of the `remind` query is:



```
BIRTHDAY_BOOK

add_birthday (n:NAME; d: DATE)
  require ¬ 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) • n} = {n ∈ model.domain | model[n] = d • n}
          model = old model
                              MODEL
count: INTEGER

model: MAP[NAME, DATE]
  ensure  Result = [ i: INTEGER | names.lower ≤ i ≤ names.upper • [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

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

By defining a slice of the model map, according to the current loop counter i as well as arrays *names* and *dates*, as follows:

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

we can show that the loop invariant for the `remind` query has been constructed to approximate and hence similar to its post-condition:

$$\{n : NAME \mid Result.has(n) \bullet n\} = \{n \in \mathrm{mSlice}(i, names, dates).domain \mid model[n] = d \bullet n\} \tag{6}$$

And 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 [12] 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* [13] that is used for static verification in Spec# and ESC/Java2. *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 [14]. A limitation of PD is that it discourages reference semantics [12]. 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 a value semantics by default and discourages the use of reference semantics [2]. 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 with a value semantics (see Introduction) using the Eiffel **expanded** construct. As a future goal we have to expand our tool to handle verification of reference aliasing and inheritance.

The theoretical foundations of PD are Floyd-Hoare logic and Dijkstra's weakest pre-condition calculus and it has the power of first-order predicate calculus, as well as a few higher-order constructs [11]. 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. 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* [5]). 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 [15].

Outline of Class Translation

Fig. 4 shows how the Eiffel generic stack example is translated into its equivalent PD specifications. 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[3], the *model* feature declaration[4], and the *implementation* feature declaration[5]. And on the PD side, there are three corresponding 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`) and public query (e.g. `item`) become a **schema** and a **function** in the PD interface section, respectively.

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 ^ =,

[2]In PD, if a reference semantics is adopted, then, roughly speaking, a `heap` declaration, e.g. `heap` MyHeap, would be required. 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.

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

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

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

i.e. is-defined-as) corresponds to the translated post-condition[6] of the that query. The second PD function is a twin function with a `_verification` name suffix. This twin function has the same definition but with a refinement (`via...end` segment) underneath which is the translated body of the Eiffel model query. 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 post-condition, we need this twin function. In stack the Eiffel query `model` becomes (a) a variable `model` 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-/post-conditions may be declared in either the public or model feature declaration part for access from clients. One such example is the agent function `date_matches` occurring in the loop invariant and post-condition of the `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:

As stated, Eiffel commands and queries become PD schemas and functions, respectively. For an Eiffel command that may modify the current object, frame constraints are needed. In order to specify frame constraints, PD supports a **change** clause[7]. For translation into PD, we use in Eiffel specification a `pd_modify`[8] declaration with its string argument passed as a list of attributes that the PD schema may change. For an Eiffel command or query, its require clause (for pre- condition) and ensure clause (for post-condition) appear as equivalent PD **pre** and **satisfy** clauses, respectively. For Eiffel command, its ensure clause (with its pd_modify declaration) appears as the equivalent PD change and satisfy clauses under a **post** declaration. For Eiffel query, it is translated in the same way as it for a command except there is no pd_modify declaration in its post-condition, and thus there exists no change list and post declaration for its translation in PD. Moreover, the Eiffel **old** notation for the value of expressions in a pre-state is converted into the equivalent PD primed notation. Finally, the body of an Eiffel command or query appears as an equivalent PD **via ... end** refinement segment.

4 Comparison with other tools

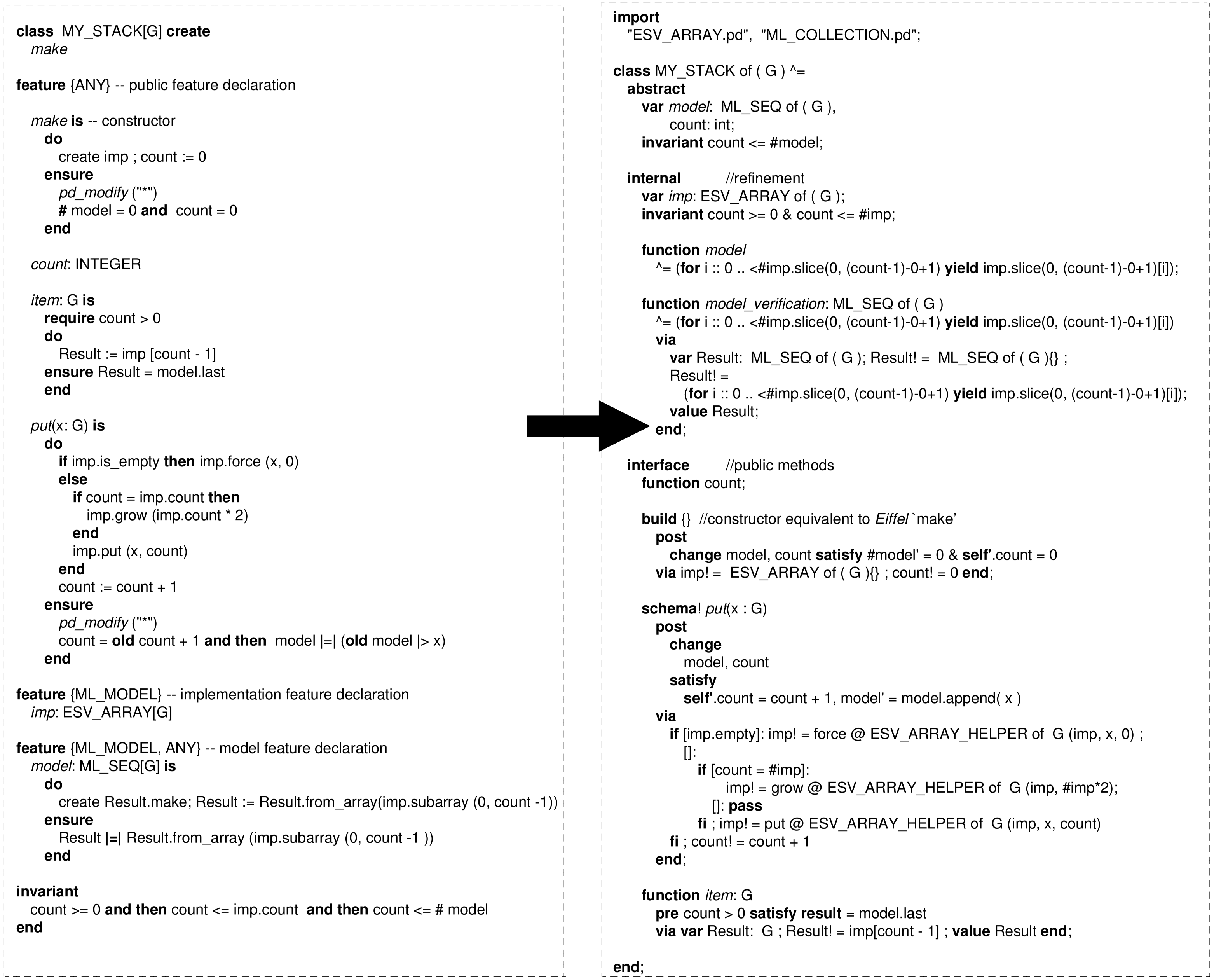
We compare ES-Verify with the other two similar software verification tools: ESC/Java2 and Spec#.

Tools like ESC/Java2 and Spec# allow the developer to increase the confidence of already existing Java and C# code following an Annotated Development approach by adding specifications as annotations [12]. Spark Ada [2] is a successful example of Annotated Development, but the specifications are usually only partial (in particular, expressing data refinement is difficult)[12]. When applied to an object-oriented language that uses reference semantics or makes heavy use of pointers, correctness has to be sacrificed in order to allow more potential bugs to be spotted, otherwise very little can be verified. Spark Ada instead preserves correctness by subsetting the Ada language. This is the strategy that ES-Verify has assumed where references and inheritance are for now not used.

[6]More precisely, RHS of the first post-condition clause which has a matching type with it of that query.

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

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



```
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

ES-Verify takes advantage of the built-in Eiffel DbC constructs, enabling it to both immediately use the run-time debugging as well as the ES-Test tool and formalize these specifications by translating them into PD code. JML tools like ESC/Java2 and `jmlc` [8] exist separately to support static verification and runtime assertion checking. The current release of ESC/Java2 claims it now runs in the JML runtime assertion checker. The Spec# system supports both static verification and runtime assertion checking in Microsoft Visual Studio. That is, the code accompanied with its specifications in ESC/Java2, Spec#, and ES-Verify are immediately executable. However, the goal of ESC/Java2, Spec#, and ES-Verify is to find bugs rather than prove total correctness. An interesting property of these tools is that they neither warn about *all* errors nor do they warn *only* about actual errors [6] and may raise false alarms due to the nature of logical proof.

The PD theorem prover has approximately the same level of proving power as the B theorem prover. It is capable of dealing with all the primitive types including reals, quantification, and set theory. Also, PD is used to verify itself with about 130,000 verification conditions. In proving data-intensive applications like the Birthday Book example (with loop invariants and model contracts in both pre- and post- conditions), PD has been able to discharge all the verification conditions. When we attempted to code the Birthday Book example in Spec#, with the same set of model specifications, it did not even translate the code into the intermediate one and start verifying. A number of specifications had to be removed in order to make it verify. Unlike ESC/Java2 and Spec#, ES-Verify can reason about generic classes. The updated version of ESC/Java2 is compatible with Java version 1.4, but not 1.5 which includes generic types. Spec# compiles generic types but did not yet verify them at the time we attempted to code the Stack example in it.

ES-Verify follows an Abstract Specification and Refinement approach as in the B-method. This approach requires a notation that can adequately express both an abstract specification and an implementation. Data refinement is a key feature, i.e. you develop the specification using an abstract data model (in our case ML), then refine the data model if necessary with an efficient implementation. The notation and the semantics are designed for correctness and provability unlike traditional programming languages, so there is no need to sacrifice correctness. Our declared model, if not deferred, is defined in terms of other fields, and thus is used as a model field not accessible to clients. ESC/Java2 and Spec# (following JML) include the ability to declare specification-only model fields [9] and abstraction functions for representing the relationship between the value of the model field and the implementation so that refinements can be proved. The authors in [18] claim the existing techniques for JML model variables suffer from soundness, modularity, expressiveness, or practical problems. They present a simpler but more expressive methodology for model fields, but it has not yet been implemented.

ESC/Java2 and Spec# provide precise feedback as to where errors occur. Our tool does not yet provide such precise feedback. However, the output html file produced by the PD theorem prover is informative. That line number easily associates with the Eiffel feature having the same name or assertion tag, so that it is relatively easy to track back to where the problem was. We have to improve this feedback reporting in future versions.

5 Conclusion

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. The translation process transforms each Eiffel construct into an equivalent PD one 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. Of course such semantics depends upon the soundness of PD.

When the ES-Verify translator is applied to the Eiffel code for the birthday book example, the

PD theorem prover generates 158 verification conditions which are *all* automatically discharged. This includes proof of termination via the loop variant. We used a value semantics class `ESV_ARRAY` for the two implementation arrays. 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. Future work aims to extend the verification to handle the issue of reference aliasing and inheritance.

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

References

[1] J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

[2] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003. With Praxis Critical Systems Limited.

[3] Mike Barnett, Robert DeLine, Bart Jacobs, Manuel Fhndrich, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The Spec# Programming System: Challenges and Directions. *Position paper at VSTTE*, 2005.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. 2004.

[5] 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.

[6] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[7] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In Springer-Verlag, editor, *Formal Methods for Components and Objects (FMCO'2005)*, LNCS, 2006.

[8] Yoonsik Cheon. A runtime assertion checker for the java modeling language. TR 03-09, Department of Computer Science, Iowa State University, April 2003.

[9] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[10] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2. Technical Report NIII-R0413, Nijmegen Institute for Computing and Information Science, May 2004.

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

[12] 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.

[13] 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.

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

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

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

[17] 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.

[18] K. Rustan M. Leino and Peter Mller. A verification methodology for model fields. ESOP 2006.

[19] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

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

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

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