Verifying Executable Object-Oriented Specifications with Separation Logic

Stephan van Staden[1], Cristiano Calcagno**[2 3], and Bertrand Meyer[1]

[1] ETH Zurich, Switzerland
{Stephan.vanStaden,Bertrand.Meyer}@inf.ethz.ch
[2] Monoidics Ltd
[3] Imperial College, London
ccris@doc.ic.ac.uk

Abstract. Specifications of Object-Oriented programs conventionally employ Boolean expressions of the programming language for assertions. Programming errors can be discovered by checking at runtime whether an assertion, such as a precondition or class invariant, holds. In this work, we show how separation logic can be used to verify that these executable specifications will always hold at runtime. Both the program and its executable assertions are verified with respect to separation logic specifications. A novel notion called *relative purity* embraces historically problematic side-effects in executable specifications, and verification boils down to proving *connecting implications*. Even model-based specifications can be verified. The framework is also well-suited to separation logic proof tools and now implemented in jStar. Numerous automatically verified examples illustrate the framework's use and utility.

Key words: Object-orientation, Specification, Verification, Separation logic, Executable assertions, Contracts

1 Introduction

Many conventional Object-Oriented (O-O) program specification approaches, such as Eiffel [1], Spec# [2] and JML [3], use Boolean expressions of the programming language to specify routine preconditions and postconditions, class invariants, and other assertions which hold at particular points such as loop invariants. Explaining the operational meaning of such executable assertions to programmers is easy, as they already know the meaning of programming language expressions. Even if such specifications are not formally verified, their runtime checking and value for testing provide significant benefits. Their popularity in software development is therefore not surprising.

Verifying that executable specifications will always hold at runtime is not easy. Expressions used for assertions typically include calls to methods which depend on the heap and which may also cause side-effects such as heap mutation. Consider for example a class SLIST which implements a sorted list of

** This work was done while visiting ETH Zurich.

integers. The postcondition of method insert(i) can be specified as has(i), meaning that the list has or contains an element i. Yet has(i) might allocate a new iterator, traverse the linked structure, and temporarily affect bookkeeping of active iterators. Translating expressions such as has(i) into logical formulae for static verification is hard. Moreover, programming language expressions might have preconditions and are generally not guaranteed to terminate. This further complicates matters for verification.

In contrast to executable specifications, separation logic predicates are not computations. They specify the effects of computations in terms of program states, i.e. the contents of the stack and heap, and not in terms of outcomes of further computations. Proof systems based on separation logic (e.g. [4, 5]) are promising for O-O verification and can successfully verify common programming patterns such as the Visitor [6] and Composite [7]. Unfortunately, the semantics of separation logic is not yet widely known by O-O programmers.

How separation logic specifications can be used to tame executable ones is the topic of this paper. The guarantee is simple: if a program satisfies its separation logic specification and the executable assertions are verified with the separation logic ones, then all executable assertions are guaranteed to hold at runtime. We do not verify the program with respect to its executable assertions, we verify the program *and* its executable assertions with respect to a separation logic specification. For example, if a program containing class SLIST satisifies its separation logic specification, and the separation logic postcondition of insert(i) is Q, then we are guaranteed that the executable postcondition, has(i), will always hold at runtime if we can prove the *connecting implication* $\mathrm{Q} \Rrightarrow \mathrm{has(i)}$. A connecting implication connects the world of separation logic predicates with the one of Boolean programming language expressions. It is proved in this case by deriving the Hoare-style triple $\{\mathrm{Q}\}\mathrm{v} := \mathrm{has(i)}\{\mathrm{Q} \wedge \mathrm{v} = \mathrm{True}\}$, where v is a fresh variable. A novel notion called *relative purity* is embedded in connecting implications and embraces side-effects in expressions such as has(i). The logical framework depends on the non-faulting semantics of separation logic triples and its $*$-connective, as we shall see.

We believe that executable and separation logic specifications can be complementary in O-O software development. The proposed formal framework accommodates both ordinary programmers and proof experts. Programmers can express their intentions in the form of executable assertions and use runtime checking to identify faults. While software designs are still evolving, it is probably also easier to change executable assertions than more elaborate specifications. We envisage that in a second phase, once the software has stabilized and many faults have been removed, proof experts would annotate the critical parts with separation logic for verification. At this point the executable assertions do not have to be discarded: our framework integrates the specification approaches and can verify whether the expectations recorded in executable assertions are fulfilled. Problems in the verification of executable assertions can indicate discrepancies between the two types of specifications, e.g. a misunderstanding by the separation logic specifier as to when a routine may be called. Executable

and separation logic specifications can therefore complement each other even in verification.

This paper makes several contributions in the area of O-O specification and verification:

1. It describes a simple technique based on connecting implications for verifying executable preconditions and postconditions. It gives executable assertions a semantics based on their semantics as expressions. The formalization can even be applied to contracts for non-O-O non-garbage collected languages such as C.
2. It presents simple techniques to verify class invariants. If an invariant is specified with a separation logic predicate, then properties of the invariant, such as the fact that it holds in all visible states [8], often follow as a consequence. The framework can help to devise flexible and sound class invariant protocols.
3. It illustrates the framework's applicability to model-based specifications [9], where model classes and model queries are used to strengthen contracts [10].
4. It shows how the novel notion of relative purity tolerates side-effects in executable assertions to a high degree.

The techniques are well-suited to separation logic proof tools. We implemented them in the jStar tool [6] and verified the paper's examples automatically.

A discussion of background material follows in Section 2. Precondition and postcondition verification are the topics of Sections 3 and 4 respectively. Section 5 contains an exposition on class invariants. The verification techniques are then applied to model-based specifications in Section 6. Relative purity and predicate extraction are considered in Section 7. A discussion of the jStar implementation follows in Section 8. Finally, Section 9 concludes and mentions related work.

2 Background

The paper presents the verification framework in an abstract setting, which can be instantiated with concrete languages and proof systems. The abstract setting is not committed to a particular separation logic, proof system or language type. It is even applicable to non-O-O non-garbage-collected languages such as C. Our presentation uses an O-O language and proof system to illustrate the abstract ideas with concrete examples.

2.1 The abstract setting: triples and footprints

An abstract triple is of the form {P}s{Q}. The partial correctness meaning of a triple is as follows: statement s does not fault when executed in a state satisfying P, and if it terminates then the resulting state satisfies Q. Faulting occurs when s accesses unallocated memory. In O-O terms, 'unallocated memory' means heap

storage which is not necessarily present in the initial state[4] and not allocated by s before being accessed. O'Hearn [11] uses the term *footprint* of s to describe the minimal state from which s can be executed safely. So s when executed in a state satisfying P will not fault if P describes at least the footprint of s. The notion of footprint has been expanded since [12], yet this is of little concern here.

2.2 The concrete setting: an O-O language and proof system

O-O language The programming language notation is based on Eiffel [1]. **Void** corresponds to 'null' in other languages. Two reserved program variables **Current** and **Result** denote the current object ('this') and the result of a function call respectively. **Current** is never **Void**. The term *feature* describes methods and fields, sometimes called the 'members' of a class. Feature overloading, including method overloading, is not allowed. We drop empty argument lists in method declarations and calls, and adhere to the uniform access principle [1] where queries can be implemented by fields or result-returning methods.

Predicates Predicates have the usual intuitionistic separation logic semantics [13, 11, 14, 5]. Informally, the predicate $x.f \hookrightarrow e$ means that the f field of object x has value e, and $P * Q$ means that P and Q hold for disjoint portions of the heap. The predicate $x : C$ means x references an object whose dynamic type is exactly C, and $x <: C$ means x references an object whose dynamic type is a subtype of C. In both cases $x \neq$ **Void**, and $x : C \Rightarrow x <: C$. Within a context, if x is declared of type C then $x <: C$ whenever $x \neq$ **Void**.

As customary in a Hoare-style logic, we allow auxiliary variables[5] in predicates. $FV(P)$ denotes the set of free variables, i.e. free program and auxiliary variables, in predicate P.

Abstract predicate families [15, 5] facilitate O-O abstraction in predicates. An abstract predicate family (abbreviated *apf*) provides an abstract predicate p for which each class C can define an entry p@C. The first argument of an apf predicate or entry is called the *root*. The root followed by a dot is written before the apf predicate or entry name. The whole prefix is omitted if the root is **Current**. An apf predicate's root can never be **Void**. Since the meaning of an apf predicate depends on the dynamic type of the root object, it can be seen as mirroring dynamic dispatch of object-orientation in the logic. The other arguments of an apf predicate or entry are a set of tagged arguments, where tag names provide useful hints about the purpose of tagged values. Apfs offer high levels of abstraction in specifications and proofs: apf predicates and entries are treated abstractly by clients, unless information about them is made explicitly available [4].

Finally, inductive data types and functions involving them are used in predicates. This paper employs only sequences of integers, where α and β are sequence-valued variables. ϵ denotes the empty sequence, and [e] denotes the singleton

[4] Separation logic systems do not rely on well-formedness of the heap.
[5] Also called *logical* or *ghost* variables.

sequence whose only element is e. The length of α is written $|\alpha|$, the i'th element of α is α_i, and $\alpha{+\!\!+}\beta$ denotes the sequence obtained by appending α and β.

Separation logic proof system The proof system for partial correctness we use is described in [4]. An understanding of Parkinson and Bierman's proof system [5] also suffices, since the extensions which [4] provides are exercised sparingly in the presentation.

Judgments about statements are of the form $\Delta; \Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}$, where s is a statement and P and Q are separation logic predicates. The predicate Δ contains assumptions about the class in which s appears, and the environment Γ contains specifications of all methods in a program. In examples we leave Δ and Γ implicit, since the presented proofs do not rely on Δ and method specifications appear in program listings.

The proof rules for most statements are standard (see e.g. [14, 4]). Verification of a dynamically dispatched call uses the called feature's dynamic separation logic specification. Provided x is not y and x is not in $\overline{\mathrm{z}}$, the rule for a dynamically dispatched result-returning call is:

$$\frac{\Gamma(\mathrm{C.f}) = (\overline{\mathrm{u}},\{\mathrm{P}\}\_\{\mathrm{Q}\})}{\begin{array}{c}\Delta; \Gamma \vdash_s \{\mathrm{P}[\mathrm{y}, \overline{\mathrm{z}}/\mathbf{Current}, \overline{\mathrm{u}}] * \mathrm{y} <: \mathrm{C}\} \\ \mathrm{x} := \mathrm{y.f}(\overline{\mathrm{z}}) \\ \{\mathrm{Q}[\mathrm{y}, \overline{\mathrm{z}}, \mathrm{x}/\mathbf{Current}, \overline{\mathrm{u}}, \mathbf{Result}]\}\end{array}}$$

where $\overline{\mathrm{u}}$ are the formal arguments of feature f in class C.

Structural rules for manipulating statement judgments include the rule of Consequence and the Auxiliary variable elimination rule. Separation logic also has the Frame rule, which facilitates local reasoning with high levels of abstraction in the presence of aliasing. Provided s modifies no variable in $FV(\mathrm{T})$:

$$\frac{\Delta; \Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}}{\Delta; \Gamma \vdash_s \{\mathrm{P} * \mathrm{T}\}\mathrm{s}\{\mathrm{Q} * \mathrm{T}\}}\,\mathrm{Frame}$$

Informally, the Frame rule states that a statement does not modify disjoint portions of the heap, since the statement can never access storage outside its footprint.

Programming language expressions The kernel language [4] for which the proof system is defined uses simple statements and expressions. To facilitate formal reasoning about executable assertions, the function $(\!|\mathrm{E}|\!)_\mathrm{v}$ translates a programming language statement v := E to an equivalent kernel language statement. We assume v has the same static type as expression E and that t, t_1, ... t_n are fresh variables:

$$\begin{aligned}
(\!|\mathrm{x}|\!)_\mathrm{v} &\stackrel{\mathrm{def}}{=} \mathrm{v} := \mathrm{x}, \text{ if x is a variable, } \mathbf{Void} \text{ or True.} \\
(\!|\mathrm{e_0.f}(\mathrm{e_1}, \ldots, \mathrm{e_n})|\!)_\mathrm{v} &\stackrel{\mathrm{def}}{=} (\!|\mathrm{e_0}|\!)_{\mathrm{t_0}}; \ldots; (\!|\mathrm{e_n}|\!)_{\mathrm{t_n}}; \mathrm{v} := \mathrm{t_0.f}(\mathrm{t_1}, \ldots, \mathrm{t_n}) \\
(\!|\mathbf{not}\ \mathrm{e}|\!)_\mathrm{v} &\stackrel{\mathrm{def}}{=} (\!|\mathrm{e}|\!)_\mathrm{t}; \mathrm{v} := \mathbf{not}\ \mathrm{t} \\
(\!|\mathrm{e_1 op\ e_2}|\!)_\mathrm{v} &\stackrel{\mathrm{def}}{=} (\!|\mathrm{e_1}|\!)_{\mathrm{t_1}}; (\!|\mathrm{e_2}|\!)_{\mathrm{t_2}}; \mathrm{v} := \mathrm{t_1\ op\ t_2}, \text{ if op} \in \{\mathbf{and}, +, -, =, >=\}.
\end{aligned}$$

The translation function preserves the semantics of complex programming language expressions. For example, if the source language used short-circuit evaluation of e_1 **and** e_2 instead, then its translation would have involved a conditional statement.

3 Precondition verification

Given a separation logic predicate P and an executable assertion B which should hold at the same program point, the question is whether B somehow follows from P. To this end we define the connecting implication $P \Rrightarrow B$, which informally means that P is sufficient to evaluate B to True:

$P \Rrightarrow B \stackrel{\text{def}}{=} \{P\}v := B\{P \wedge v = \text{True}\}$, where v is a fresh variable.

The intuition behind the definition is that in every state satisfying P, we can evaluate B into v without faulting, and if this computation terminates, then P will hold in the resulting state and v will have value True. The routine body relies on P, so it must be re-established by the evaluation of B. This enforces a notion of purity on B, i.e. it prevents B from performing certain kinds of side-effects while allowing others. Section 7.1 contains more about this.

In the concrete setting of the O-O proof system, there is a sufficient condition[6] for $P \Rrightarrow B$:

$$\boxed{\text{If } \Delta;\Gamma \vdash_s \{P\}(\!|B|\!)_v\{P * v = \text{True}\}, \text{ then } P \Rrightarrow B \text{ under } \Delta \text{ and } \Gamma.}$$

In intuitionistic separation logic, which is used for garbage-collected O-O languages such as our concrete one, $(P * e = e') \Leftrightarrow (P \wedge e = e')$. The $\Delta;\Gamma$ used to verify a B appearing in class C is the same $\Delta;\Gamma$ under which statements and methods of C are verified. In the presented examples we omit explicit reference to Δ and Γ, since none of them uses the additional logical assumptions in Δ, and method specifications in Γ can be read directly from code listings. For a formal treatment of Δ and Γ, the reader is referred to [4].

Examples. Consider the method preconditions of class SLIST in Figure 1.

1. The executable precondition of the constructor *SLIST* and features *insert*, *has*, *count* and *is_empty* is True. For any separation logic predicate P it holds that $P \Rrightarrow \text{True}$:

$$\dfrac{\dfrac{\dfrac{}{\Delta;\Gamma \vdash_s \{\text{True} = \text{True}\}v := \text{True}\{v = \text{True}\}}\text{Assigment axiom}}{\Delta;\Gamma \vdash_s \{\text{True}\}v := \text{True}\{v = \text{True}\}}\text{Consequence}}{\dfrac{\Delta;\Gamma \vdash_s \{\text{True} * P\}v := \text{True}\{v = \text{True} * P\}}{\Delta;\Gamma \vdash_s \{P\}v := \text{True}\{P * v = \text{True}\}}\text{Consequence}}\text{Frame rule}$$

[6] Whether or not this condition is also necessary depends on the completeness of the proof system.

class SLIST
define x.L@SLIST(l: α) **as** ...
axiom
    L_sorted: $\forall\alpha\cdot$ L(l: α) $\Rightarrow$ [$\forall$i,j$\in$ (1..$|\alpha|$)$\cdot$ i $<$ j $\Rightarrow$ $\alpha_{\mathrm{i}} \leq \alpha_{\mathrm{j}}$]
feature
    SLIST
    dynamic {True}_{L(l: ϵ)}
    executable {True}_{is_empty}

    insert(i: INT)
    dynamic {L(l: α)}_{$\exists\alpha_{\mathrm{F}}, \alpha_{\mathrm{S}}\cdot$ L(l: $\alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}$) $*$ $\alpha_{\mathrm{F}}{+\!\!+}\alpha_{\mathrm{S}} = \alpha$}
    executable {True}_{has(i) **and** count = **old**(count) + 1}

    remove_first
    dynamic {L(l: $[e]{+\!\!+}\alpha$)}_{L(l: α)}
    executable {**not** is_empty}_{count = **old**(count) $-$ 1}

    first: INT
    dynamic {L(l: $[e]{+\!\!+}\alpha$)}_{L(l: $[e]{+\!\!+}\alpha$) $*$ **Result** = e}
    executable {**not** is_empty}_{**not** is_empty}

    has(i: INT): BOOL
    dynamic {L(l: α)}_{L(l: α) $*$ **Result** = ($\exists j \in 1..|\alpha| \cdot \alpha_j$ = i)}
    executable {True}_{True}

    count: INT
    dynamic {L(l: α)}_{L(l: α) $*$ **Result** = $|\alpha|$}
    executable {True}_{True}

    is_empty: BOOL
    dynamic {L(l: α)}_{L(l: α) $*$ **Result** = ($\alpha = \epsilon$)}
    executable {True}_{True}
invariant
    count_non_negative: {L(l: α)} count >= 0
    empty_definition: {L(l: α)} is_empty = (count = 0)
end

Fig. 1. The interface of a sorted list class.

The side-condition of the Frame rule is satisfied: $\mathit{modifies}(\mathrm{v} := \mathrm{True}) = \{\mathrm{v}\}$ and $\{\mathrm{v}\} \cap \mathrm{FV}(\mathrm{P}) = \emptyset$ (remember that v is fresh). The second application of Consequence used the commutativity of $*$ and the fact that $\mathrm{P} \Rightarrow (\mathrm{P} * \mathrm{True})$ in intuitionistic separation logic.
Instead of such detailed proofs, the rest of the paper uses proof outlines where statements are interspersed between assertions. Explicitly mentioning the freshness of certain variables is omitted if it is clear from the context. The above proof looks as follows in outline form:

$\{\mathrm{P}\}$
    $\mathrm{v} := \mathrm{True}$
$\{\mathrm{P} * \mathrm{v} = \mathrm{True}\}$

2. The preconditions of *remove_first* and *first* are identical. Here is a proof of $\mathrm{L}(\mathrm{l}\colon [e]{+\!\!+}\alpha) \Rrightarrow$ **not** is_empty:

$\{\mathrm{L}(\mathrm{l}\colon [e]{+\!\!+}\alpha)\}$
    $\mathrm{t} := \mathrm{is\_empty}$
$\{\mathrm{L}(\mathrm{l}\colon [e]{+\!\!+}\alpha) * \mathrm{t} = ([e]{+\!\!+}\alpha = \epsilon)\}$
$\{\mathrm{L}(\mathrm{l}\colon [e]{+\!\!+}\alpha) * \mathrm{t} = \mathrm{False}\}$
    $\mathrm{v} :=$ **not** t
$\{\mathrm{L}(\mathrm{l}\colon [e]{+\!\!+}\alpha) * \mathrm{t} = \mathrm{False} * \mathrm{v} = \neg\mathrm{t}\}$
$\{\mathrm{L}(\mathrm{l}\colon [e]{+\!\!+}\alpha) * \mathrm{v} = \mathrm{True}\}$

3. The executable precondition does not always follow from the separation logic one. Suppose that the executable precondition of *has* is not True but instead **not** is_empty. A proof attempt of $\mathrm{L}(\mathrm{l}\colon \alpha) \Rrightarrow$ (**not** is_empty) proceeds as follows:

$\{\mathrm{L}(\mathrm{l}\colon \alpha)\}$
    $\mathrm{t} := \mathrm{is\_empty}$
$\{\mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{t} = (\alpha = \epsilon)\}$
    $\mathrm{v} :=$ **not** t
$\{\mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{t} = (\alpha = \epsilon) * \mathrm{v} = \neg\mathrm{t}\}$
$\{\mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{v} = (\alpha \neq \epsilon)\}$

Since $\alpha \neq \epsilon$ is not a consequence, the proof cannot conclude with $\mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{v} = \mathrm{True}$. The separation logic specification is too weak to demonstrate the executable one.[7]
Such a clash of specifications has several potential causes. The executable assertion is maybe overly restrictive, or it captures important semantic properties of the domain which the separation logic one ignores. If the issue is not resolved, there is no guarantee that the executable assertion will hold at runtime. □

The rest of the paper contains more examples of the form $\mathrm{P} \Rrightarrow \mathrm{B}$.

[7] In fact we have shown that (**not** is_empty) evaluates to False if $\alpha = \epsilon$, so the only way it can also evaluate to True is when it loops forever. A verified implementation of SLIST (and classes transitively used by it) for which (**not** is_empty) terminates when executed in some initial state satisfying $\mathrm{L}(\mathrm{l}\colon \epsilon)$ comprises a counterexample for $\mathrm{L}(\mathrm{l}\colon \alpha) \Rrightarrow$ (**not** is_empty).

4 Postcondition verification

Executable postconditions for routines may contain so-called *old-expressions*. The old-expression **old**(E) denotes the value of expression E in the state right before the routine is executed, i.e. the state which satisfies the precondition.

Given two separation logic predicates P and Q and an executable assertion B which may contain old-expressions, the connecting implication P,Q $\Rrightarrow$ B informally means that any pre-state satisfying P and any post-state satisfying Q are sufficient to evaluate B to True:

Let **old**(E_1),...,**old**(E_n) be the list of old-expressions in B and let v, v_1,...,v_n be fresh variables.

$$P,Q \Rrightarrow B \stackrel{\text{def}}{=} \exists R\cdot \{P\}(v_1 := E_1, \ldots, v_n := E_n)\{P * R\} \text{ and } \{Q * R\}v := B[v_1; \ldots; v_n]\{Q \wedge v = \text{True}\}$$

$$B[v_1;\ldots;v_n] \stackrel{\text{def}}{=} B[v_1/\textbf{old}(E_1),\ldots,v_n/\textbf{old}(E_n)]$$

Intuitively, R contains the result of evaluating (v_1:=E_1,...,v_n:=E_n) in the pre-state where P holds. Old-expression evaluation must re-establish P because the routine body relies on it. The existence of old-expression results depends on runtime checking, so R is not contained in Q and the body is not allowed to access state described by R. The body is verified only with respect to P and Q, which guarantees that it will establish Q upon termination and never touch R. So Q $*$ R holds right after the body's execution. Evaluating B[v_1;...;v_n] in this state should yield True and re-establish Q, since clients depend on the fact that Q holds. Figure 2 summarizes the proof obligations of a routine.

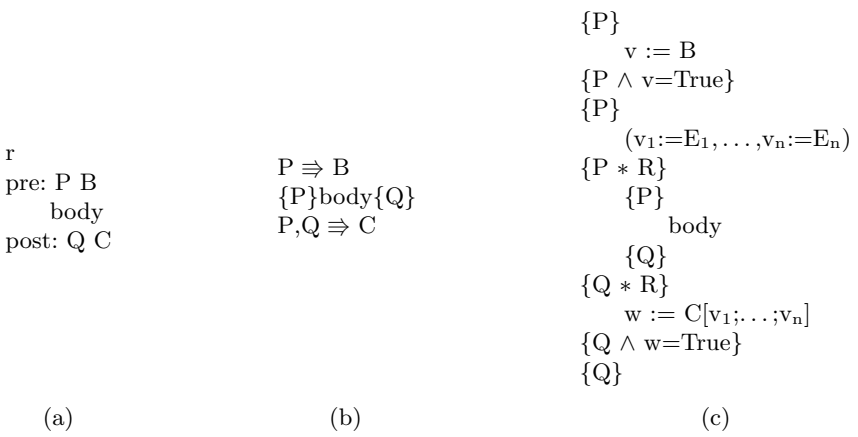

Fig. 2. (a) Routine r and its specification. (b) Resulting proof obligations. (c) Proof obligations as triples in a proof outline.

The predicate R will almost certainly have $v_1,\ldots,v_n$ as free variables, but should not have free variables which are modified by the routine body.[8]

An assertion language's semantics will stipulate how to prove a triple of the form $\{P\}(v_1 := E_1, \ldots, v_n := E_n)\{Q\}$. For example, if old-expressions are evaluated in an unspecified order, then $\{P\}s\{Q\}$ must be proved for every permutation s of $(v_1{:=}E_1,\ldots,v_n{:=}E_n)$.

In our concrete O-O setting, the assertion language specification states that old-expressions will be evaluated in an arbitrary order. The following condition is sufficient (but not necessary[9]) for concluding P,Q $\Rrightarrow$ B:

> If $\forall i \in (1..n)\cdot\ \Delta;\Gamma \vdash_s \{P\}(\!|E_i|\!)_{v_i}\{P * R_i\}$ with $\mathbf{Result} \notin FV(R_i)$
> and $\Delta;\Gamma \vdash_s \{Q * R_1 * \ldots * R_n\}v := B[v_1;\ldots;v_n]\{Q * v = \mathrm{True}\}$
> then P,Q $\Rrightarrow$ B under Δ and Γ.

Once again, an assertion B appearing in class C is verified with the same $\Delta;\Gamma$ as the statements and methods of C.

Relationships between P,Q $\Rrightarrow$ B and Q $\Rrightarrow$ B exist, which are convenient in proofs because many executable postconditions contain no old-expressions.

> 1. If P,Q $\Rrightarrow$ B then Q $\Rrightarrow$ B whenever B contains no old-expression.
>
> 2. If Q $\Rrightarrow$ B then P,Q $\Rrightarrow$ B for any P.
>
> So P,Q $\Rrightarrow$ B iff Q $\Rrightarrow$ B whenever B contains no old-expression.

Examples. Consider class SLIST in Figure 1.

1. For *remove_first*, it is the case that L(l: $[e]{+}{+}\alpha$),L(l: α) $\Rrightarrow$ (count = **old**(count) − 1). The executable postcondition contains one old-expression and E_1 = count. Here is the first part of the proof:

   $\{L(l: [e]{+}{+}\alpha)\}$
   t1 := count
   $\{L(l: [e]{+}{+}\alpha) * t1 = |[e]{+}{+}\alpha|\}$

   Notice that FV(t1 = $|[e]{+}{+}\alpha|$) = {t1,e,α}. Using (t1 = $|[e]{+}{+}\alpha|$) for R_1 completes the proof:

   $\{L(l: \alpha) * t1 = |[e]{+}{+}\alpha|\}$
   t2 := count
   $\{L(l: \alpha) * t2 = |\alpha| * t1 = |[e]{+}{+}\alpha|\}$
   $\{L(l: \alpha) * t2 = |\alpha| * t1 = |\alpha| + 1\}$
   t3 := t1 − 1
   $\{L(l: \alpha) * t2 = |\alpha| * t1 = |\alpha| + 1 * t3 = |\alpha|\}$

[8] This allows the transfer of R over the routine body with the Frame rule.

[9] Suppose we take $(\exists e\cdot\ o.f \hookrightarrow e * even(e))$ for P and o.f++ for both E_1 and E_2. Since there exists no R_i such that $\Delta;\Gamma \vdash_s \{P\}(\!|E_i|\!)_{v_i}\{P * R_i\}$ for $i \in 1..2$, the rule cannot prove P,True $\Rrightarrow$ odd(**old**(E_1)+**old**(E_2)).

$\{\mathrm{L(l:}\ \alpha) * \mathrm{t2} = |\alpha| * \mathrm{t3} = |\alpha|\}$
  v := t2 = t3
$\{\mathrm{L(l:}\ \alpha) * \mathrm{t2} = |\alpha| * \mathrm{t3} = |\alpha| * \mathrm{v} = (\mathrm{t2} = \mathrm{t3})\}$
$\{\mathrm{L(l:}\ \alpha) * \mathrm{v} = \mathrm{True}\}$

2. The executable postcondition of the constructor *SLIST* contains no old-expression, so it suffices to prove $\mathrm{L(l:}\ \epsilon) \Rrightarrow$ is_empty:

$\{\mathrm{L(l:}\ \epsilon)\}$
  v := is_empty
$\{\mathrm{L(l:}\ \epsilon) * \mathrm{v} = (\epsilon = \epsilon)\}$
$\{\mathrm{L(l:}\ \epsilon) * \mathrm{v} = \mathrm{True}\}$

3. The executable postcondition of *first* contains no old-expression, so we only need $(\mathrm{L(l:}\ [e]{+\!\!+}\alpha) * \mathbf{Result} = e) \Rrightarrow \mathbf{not}$ is_empty. Example 2 in the previous section established the inner triple of the proof:

$\{\mathrm{L(l:}\ [e]{+\!\!+}\alpha) * \mathbf{Result} = e\}$
  $\{\mathrm{L(l:}\ [e]{+\!\!+}\alpha)\}$
    $(\!|\mathbf{not}\ \mathrm{is\_empty}|\!)_{\mathrm{v}}$
  $\{\mathrm{L(l:}\ [e]{+\!\!+}\alpha) * \mathrm{v} = \mathrm{True}\}$
$\{\mathrm{L(l:}\ [e]{+\!\!+}\alpha) * \mathrm{v} = \mathrm{True} * \mathbf{Result} = e\}$
$\{\mathrm{L(l:}\ [e]{+\!\!+}\alpha) * \mathbf{Result} = e * \mathrm{v} = \mathrm{True}\}$

   The Frame rule is key to the proof.
4. Consider the postcondition of *insert*. It contains one old-expression, namely **old**(count):

$\{\mathrm{L(l:}\ \alpha)\}$
  t1 := count
$\{\mathrm{L(l:}\ \alpha) * \mathrm{t1} = |\alpha|\}$

   The second part of the proof consists of small pieces which are put together.

$\{\mathrm{L(l:}\ \alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}})\}$
  t2 := has(i)
$\{\mathrm{L(l:}\ \alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}) * \mathrm{t2} = (\exists j \in 1..|\alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}| \cdot (\alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}})_j = \mathrm{i})\}$
$\{\mathrm{L(l:}\ \alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}) * \mathrm{t2} = \mathrm{True}\}$

   Applying Frame and Consequence to this triple yields Piece 1:

$\{\mathrm{L(l:}\ \alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}) * \alpha_{\mathrm{F}}{+\!\!+}\alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha|\}$
  t2 := has(i)
$\{\mathrm{L(l:}\ \alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}) * \alpha_{\mathrm{F}}{+\!\!+}\alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha| * \mathrm{t2} = \mathrm{True}\}$

   Next, we prove

$\{\mathrm{L(l:}\ \alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}) * \alpha_{\mathrm{F}}{+\!\!+}\alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha|\}$
  t3 := count
$\{\mathrm{L(l:}\ \alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}) * \mathrm{t3} = |\alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}| * \alpha_{\mathrm{F}}{+\!\!+}\alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha|\}$
$\{\mathrm{L(l:}\ \alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}) * \mathrm{t3} = |\alpha| + 1 * \alpha_{\mathrm{F}}{+\!\!+}\alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha|\}$
  t4 := t1 + 1
$\{\mathrm{L(l:}\ \alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}) * \mathrm{t3} = |\alpha| + 1 * \alpha_{\mathrm{F}}{+\!\!+}\alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha| * \mathrm{t4} = \mathrm{t1} + 1\}$
$\{\mathrm{L(l:}\ \alpha_{\mathrm{F}}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_{\mathrm{S}}) * \mathrm{t3} = |\alpha| + 1 * \alpha_{\mathrm{F}}{+\!\!+}\alpha_{\mathrm{S}} = \alpha * \mathrm{t4} = |\alpha| + 1\}$

t5 := t3 = t4
{L(l: α_F++[i]++α_S) * t3 = $|\alpha|$ + 1 * α_F++α_S = α * t4 = $|\alpha|$ + 1 * t5 = (t3=t4)}
{L(l: α_F++[i]++α_S) * α_F++α_S = α * t5 = True}

Applying Frame establishes Piece 2:

{L(l: α_F++[i]++α_S) * α_F++α_S = α * t1 = $|\alpha|$ * t2 = True}
t3 := count
t4 := t1 + 1
t5 := t3 = t4
{L(l: α_F++[i]++α_S) * α_F++α_S = α * t5 = True * t2 = True}

Here is Piece 3:

{L(l: α_F++[i]++α_S) * α_F++α_S = α * t5 = True * t2 = True}
v := t2 **and** t5
{L(l: α_F++[i]++α_S) * α_F++α_S = α * t5 = True * t2 = True * v = (t2 and t5)}
{L(l: α_F++[i]++α_S) * α_F++α_S = α * v = True}

The proof is completed by putting the three pieces together, eliminating α_F and α_S with the Auxiliary variable elimination rule, and applying Consequence to move (t1 = $|\alpha|$) and (v = True) out from under the quantifiers to the top level in the precondition and postcondition respectively. □

5 Class invariant verification

Consider an executable assertion B which should hold at program point *pp*. If the separation logic predicate P holds at *pp* and P ⇛ B, then B is verified for *pp*. This technique can be used to verify executable assert statements and loop invariants, for example.

Class invariants[10] (henceforth simply called invariants) are verified in a similar way. Since an invariant protocol [16] always specifies the points in a program where an invariant, say B, should hold, B is verified if it is verified for all such points.

Another technique is to annotate an invariant B with a separation logic predicate P which characterizes the states in which the invariant should hold. If P ⇛ B, then B is verified. This offers a flexible scheme if annotations are given on the level of individual invariant clauses.

Examples. Consider the following invariant clauses, reproduced from the bottom of class SLIST in Figure 1:[11]

invariant
count_non_negative: {L(l: α)} count >= 0
empty_definition: {L(l: α)} is_empty = (count = 0)

[10] Also called *object invariants*.

[11] These are public invariants, as opposed to private representation invariants (which is how the term 'invariant' is used in the Spec# literature, e.g. [17]). Private invariants can be verified in a similar way.

1. For the clause named *count_non_negative*, $\mathrm{L(l:}\ \alpha) \Rrightarrow (\mathrm{count} >= 0)$ holds:

$\{\mathrm{L(l:}\ \alpha)\}$
  $\mathrm{t} := \mathrm{count}$
$\{\mathrm{L(l:}\ \alpha) * \mathrm{t} = |\alpha|\}$
  $\mathrm{v} := \mathrm{t} >= 0$
$\{\mathrm{L(l:}\ \alpha) * \mathrm{t} = |\alpha| * \mathrm{v} = (\mathrm{t} \geq 0)\}$
$\{\mathrm{L(l:}\ \alpha) * \mathrm{v} = \mathrm{True}\}$

2. The invariant clause *empty_definition* is verified similarly:

$\{\mathrm{L(l:}\ \alpha)\}$
  $\mathrm{t1} := \mathrm{is\_empty}$
$\{\mathrm{L(l:}\ \alpha) * \mathrm{t1} = (\alpha = \epsilon)\}$
  $\mathrm{t2} := \mathrm{count}$
$\{\mathrm{L(l:}\ \alpha) * \mathrm{t2} = |\alpha| * \mathrm{t1} = (\alpha = \epsilon)\}$
  $\mathrm{t3} := \mathrm{t2} = 0$
$\{\mathrm{L(l:}\ \alpha) * \mathrm{t2} = |\alpha| * \mathrm{t1} = (\alpha = \epsilon) * \mathrm{t3} = (\mathrm{t2} = 0)\}$
$\{\mathrm{L(l:}\ \alpha) * \mathrm{t1} = (\alpha = \epsilon) * \mathrm{t3} = (|\alpha| = 0)\}$
  $\mathrm{v} := \mathrm{t1} = \mathrm{t3}$
$\{\mathrm{L(l:}\ \alpha) * \mathrm{t1} = (\alpha = \epsilon) * \mathrm{t3} = (|\alpha| = 0) * \mathrm{v} = (\mathrm{t1} = \mathrm{t3})\}$
$\{\mathrm{L(l:}\ \alpha) * \mathrm{v} = ((\alpha = \epsilon) = (|\alpha| = 0))\}$
$\{\mathrm{L(l:}\ \alpha) * \mathrm{v} = \mathrm{True}\}$

□

The fact that the structural rules of separation logic are sound provides a simple way to show that a verified invariant clause holds at a particular program point. The following rules can also be used when verifying other executable assertions:[12] [13]

$$\frac{\mathrm{P} \Rrightarrow \mathrm{B}}{(\mathrm{P} * \mathrm{Q}) \Rrightarrow \mathrm{B}}\,\mathrm{Frame}'$$

$$\frac{\mathrm{P} \Leftrightarrow \mathrm{Q} \qquad \mathrm{P} \Rrightarrow \mathrm{B}}{\mathrm{Q} \Rrightarrow \mathrm{B}}\,\mathrm{Conseq}'$$

$$\frac{\mathrm{P} \Rrightarrow \mathrm{B}}{(\exists \mathrm{x} \cdot \mathrm{P}) \Rrightarrow \mathrm{B}}\,\mathrm{AuxVarElim}'$$

$$\frac{\mathrm{P} \Rrightarrow \mathrm{B} \qquad \mathrm{Q} \Rrightarrow \mathrm{B}}{(\mathrm{P} \vee \mathrm{Q}) \Rrightarrow \mathrm{B}}\,\mathrm{Disj}'$$

and others.

[12] Similar rules can be given for P,Q ⇛ B.

[13] These rules are independent of the structure of B. Rules also exist which do depend on its structure. For example, if $\mathrm{P}_1 \Rrightarrow \mathrm{B}_1$ and $\mathrm{P}_2 \Rrightarrow \mathrm{B}_2$, then $(\mathrm{P}_1 * \mathrm{P}_2) \Rrightarrow (\mathrm{B}_1$ **par_and** $\mathrm{B}_2)$. This rule allows concurrency in executable assertions.

The Frame′ rule does not need any side-condition, since the translation of B will modify only fresh variables, i.e. variables not free in Q. Note the way Conseq′ is written: if P ⇛ B and Q ⇒ P, then Q ⇛ B does *not* necessarily hold (example 3 in Section 7.1 provides a counterexample). In the AuxVarElim′ rule, the variable x may not be free in B.

Example. If a program containing class SLIST from Figure 1 is verified, then it follows as a consequence of the separation logic specification that both invariant clauses hold in the visible states [8] of SLIST. In other words, both clauses are established by the constructor and hold on entry and exit of all other public features. Since each clause B is verified, we know $\mathrm{L(l:}\ \alpha) \Rrightarrow \mathrm{B}$. In the following proofs the '⇛ B' part is omitted.

1. Upon exit from *SLIST*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{L(l:}\ \alpha)}{\mathrm{L(l:}\ \alpha) * \alpha = \epsilon}\,\mathrm{Frame'}}{\mathrm{L(l:}\ \epsilon) * \alpha = \epsilon}\,\mathrm{Conseq'}}{\exists\alpha\cdot\ \mathrm{L(l:}\ \epsilon) * \alpha = \epsilon}\,\mathrm{AuxVarElim'}}{\mathrm{L(l:}\ \epsilon) * \mathrm{True}}\,\mathrm{Conseq'}}{\mathrm{L(l:}\ \epsilon)}\,\mathrm{Conseq'}$$

2. Upon entry to *remove_first*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{L(l:}\ \alpha)}{\mathrm{L(l:}\ \alpha) * \alpha = \beta}\,\mathrm{Frame'}}{\mathrm{L(l:}\ \beta) * \alpha = \beta}\,\mathrm{Conseq'}}{\exists\alpha\cdot\ \mathrm{L(l:}\ \beta) * \alpha = \beta}\,\mathrm{AuxVarElim'}}{\mathrm{L(l:}\ \beta)}\,\mathrm{Conseq'}}{\mathrm{L(l:}\ \beta) * \beta = [e]{+\!\!+}\alpha}\,\mathrm{Frame'}}{\mathrm{L(l:}\ [e]{+\!\!+}\alpha) * \beta = [e]{+\!\!+}\alpha}\,\mathrm{Conseq'}}{\exists\beta\cdot\ \mathrm{L(l:}\ [e]{+\!\!+}\alpha) * \beta = [e]{+\!\!+}\alpha}\,\mathrm{AuxVarElim'}}{\mathrm{L(l:}\ [e]{+\!\!+}\alpha)}\,\mathrm{Conseq'}$$

3. For the exit of *insert*, we start with $\mathrm{L(l:}\ \beta)$ which was derived in the previous proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{L(l:}\ \beta)}{\mathrm{L(l:}\ \beta) * \beta = (\alpha_\mathrm{F}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_\mathrm{S}) * \alpha_\mathrm{F}{+\!\!+}\alpha_\mathrm{S} = \alpha}\,\mathrm{Frame'}}{\exists\beta\cdot\ \mathrm{L(l:}\ \beta) * \beta = (\alpha_\mathrm{F}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_\mathrm{S}) * \alpha_\mathrm{F}{+\!\!+}\alpha_\mathrm{S} = \alpha}\,\mathrm{AuxVarElim'}}{\mathrm{L(l:}\ \alpha_\mathrm{F}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_\mathrm{S}) * \alpha_\mathrm{F}{+\!\!+}\alpha_\mathrm{S} = \alpha}\,\mathrm{Conseq'}}{\exists\alpha_\mathrm{S}\cdot\ \mathrm{L(l:}\ \alpha_\mathrm{F}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_\mathrm{S}) * \alpha_\mathrm{F}{+\!\!+}\alpha_\mathrm{S} = \alpha}\,\mathrm{AuxVarElim'}}{\exists\alpha_\mathrm{F},\alpha_\mathrm{S}\cdot\ \mathrm{L(l:}\ \alpha_\mathrm{F}{+\!\!+}[\mathrm{i}]{+\!\!+}\alpha_\mathrm{S}) * \alpha_\mathrm{F}{+\!\!+}\alpha_\mathrm{S} = \alpha}\,\mathrm{AuxVarElim'}$$

□

In effect we create fine-grained invariant protocols by annotating invariant clauses: the predicate abstractly specifies the program points where they should

hold, and invariants can be verified, i.e. guaranteed, to hold there. One can refine this idea to create more sophisticated invariant protocols.

Example. Here is a sound invariant protocol for single-inheritance programs where subclasses can refine individual invariant clauses. Γ maps a class name and invariant clause name to the invariant specification predicate and Boolean expression. Two rules distinguish the case where class C introduces an invariant clause named inv from the case where C inherits and refines inv from its parent C':

$$\frac{\forall i \in (1..n) \cdot (P * \textbf{Current} : C) \Rrightarrow B_i}{\Gamma \vdash_i \textbf{introduce}\ \text{inv: } \{P\}\ \langle B_1 \rangle\ \textbf{and}\ \ldots\ \textbf{and}\ \langle B_n \rangle\ \text{in } C}$$

$$\frac{\begin{array}{l} C' \prec_1 C \\ \Gamma(C.\text{inv}) = \{P\}\ B \\ B = \langle B_1 \rangle\ \textbf{and}\ \ldots\ \textbf{and}\ \langle B_n \rangle \\ P \Leftrightarrow (P' * R) \\ \forall i \in (1..n) \cdot (P' * \textbf{Current} : C') \Rrightarrow B_i \\ \forall j \in (1..m) \cdot (P' * \textbf{Current} : C') \Rrightarrow B'_j \end{array}}{\Gamma \vdash_i \textbf{inherit}\ \text{inv: } \{P'\}\ B\ \textbf{and}\ \langle B'_1 \rangle\ \textbf{and}\ \ldots\ \textbf{and}\ \langle B'_m \rangle\ \text{in } C'}$$

The above invariant protocol guarantees that whenever 'inv: {P} B' appears in class C, then $\forall x <: C \cdot P[x/\textbf{Current}] \Rrightarrow B[x/\textbf{Current}]$ where x is fresh. The soundness proof uses the fact that if $P \Rrightarrow B$ and $P \Rrightarrow B'$, then $P \Rrightarrow (B\ \textbf{and}\ B')$. Enclosing an expression in angle-brackets provides it with a side-effect scope. □

6 Model-based specifications

Since executable specifications are frequently not very expressive, model classes and model-based contracts are sometimes used to strengthen them [9, 10].

Example. Consider the interface of model class SEQUENCE in Figure 3. It provides an abstraction of immutable sequences for specification purposes. Class SLIST can be specified in terms of SEQUENCE, as the interface extract in Figure 4 shows. Note that SLIST now has a model query, namely *model*, which returns the immutable sequence abstraction of an SLIST instance at the point when it is called. A comparison of *remove_first*'s specification in Figures 1 and 4 shows that the model-based specification involves the element values stored in an SLIST instance and not just their number. □

The following three problems are typically not easy to solve with conventional techniques [18, 19]:

1. Devising semantics for model classes and proving their implementations correct.
2. Giving a semantics to model queries, such as *model* in Figure 4, and proving their implementations correct.

```
class SEQUENCE
define x.SEQ@SEQUENCE(s: α) as ...
feature
    SEQUENCE
    dynamic {True}_{SEQ(s: ϵ)}

    cons(i: INT): SEQUENCE
    dynamic {SEQ(s: α)}_{SEQ(s: α) ∗ Result.SEQ(s: [i]++α)}

    head: INT
    dynamic {SEQ(s: [e]++α)}_{SEQ(s: [e]++α) ∗ Result = e}

    tail: SEQUENCE
    dynamic {SEQ(s: [e]++α)}_{SEQ(s: [e]++α) ∗ Result.SEQ(s: α)}

    is_nil: BOOL
    dynamic {SEQ(s: α)}_{SEQ(s: α) ∗ Result = (α = ϵ)}

    eq(o: SEQUENCE): BOOL
    dynamic {SEQ(s: α) ∗ o.SEQ(s: β)}_
            {SEQ(s: α) ∗ o.SEQ(s: β) ∗ Result = (α = β)}
end
```

Fig. 3. The interface of model class SEQUENCE.

```
class SLIST
    ...
feature
    ...
    model: SEQUENCE
    dynamic {L(l: α)}_{L(l: α) ∗ Result.SEQ(s: α)}
    executable {True}_{True}

    remove_first
    dynamic {L(l: [e]++α)}_{L(l: α)}
    executable {not model.is_nil}_{model.eq(old(model).tail)}
    ...
invariant
    ...
    empty_inv: {L(l: α)} is_empty = model.is_nil
end
```

Fig. 4. An extract from class SLIST which uses model-based contracts.

3. Verifying model-based specifications, such as the model-based contract of *remove_first* and the model-based invariant clause *empty_inv* in Figure 4.

The first two problems can be solved with separation logic. Figures 3 and 4 show separation logic specifications for a model class and model query. Conventional separation logic proof systems can be used to verify their implementations. The third problem can then be addressed with the framework of this paper.

Examples. Consider the model-based specifications of class SLIST in Figure 4. Suppose that omitted features have the same separation logic specifications as in Figure 1.

1. For the invariant clause *empty_inv*:

   $\{\mathrm{L(l:}\ \alpha)\}$
   t1 := is_empty
   $\{\mathrm{L(l:}\ \alpha) * \mathrm{t1} = (\alpha = \epsilon)\}$
   t2 := model
   $\{\mathrm{L(l:}\ \alpha) * \mathrm{t2.SEQ(s:}\ \alpha) * \mathrm{t1} = (\alpha = \epsilon)\}$
   t3 := t2.is_nil
   $\{\mathrm{L(l:}\ \alpha) * \mathrm{t2.SEQ(s:}\ \alpha) * \mathrm{t3} = (\alpha = \epsilon) * \mathrm{t1} = (\alpha = \epsilon)\}$
   $\{\mathrm{L(l:}\ \alpha) * \mathrm{t3} = (\alpha = \epsilon) * \mathrm{t1} = (\alpha = \epsilon)\}$
   v := t1 = t3
   $\{\mathrm{L(l:}\ \alpha) * \mathrm{t3} = (\alpha = \epsilon) * \mathrm{t1} = (\alpha = \epsilon) * \mathrm{v} = (\mathrm{t1} = \mathrm{t3})\}$
   $\{\mathrm{L(l:}\ \alpha) * \mathrm{v} = \mathrm{True}\}$

2. We next verify the postcondition of *remove_first* which contains one old-expression:

   $\{\mathrm{L(l:}\ [e]{+\!\!+}\alpha)\}$
   v1 := model
   $\{\mathrm{L(l:}\ [e]{+\!\!+}\alpha) * \mathrm{v1.SEQ(s:}\ [e]{+\!\!+}\alpha)\}$

   The proof is completed by using $\mathrm{v1.SEQ(s:}\ [e]{+\!\!+}\alpha)$ for R_1:

   $\{\mathrm{L(l:}\ \alpha) * \mathrm{v1.SEQ(s:}\ [e]{+\!\!+}\alpha)\}$
   t1 := model
   $\{\mathrm{L(l:}\ \alpha) * \mathrm{t1.SEQ(s:}\ \alpha) * \mathrm{v1.SEQ(s:}\ [e]{+\!\!+}\alpha)\}$
   t2 := v1.tail
   $\{\mathrm{L(l:}\ \alpha) * \mathrm{t1.SEQ(s:}\ \alpha) * \mathrm{v1.SEQ(s:}\ [e]{+\!\!+}\alpha) * \mathrm{t2.SEQ(s:}\ \alpha)\}$
   $\{\mathrm{L(l:}\ \alpha) * \mathrm{t1.SEQ(s:}\ \alpha) * \mathrm{t2.SEQ(s:}\ \alpha)\}$
   v := t1.eq(t2)
   $\{\mathrm{L(l:}\ \alpha) * \mathrm{t1.SEQ(s:}\ \alpha) * \mathrm{t2.SEQ(s:}\ \alpha) * \mathrm{v} = (\alpha = \alpha)\}$
   $\{\mathrm{L(l:}\ \alpha) * \mathrm{v} = \mathrm{True}\}$

□

7 Relative purity and predicate extraction

7.1 Relative purity

Side-effects in executable specifications conventionally complicate verification, since the logical predicates extracted from them cannot be imperative. Most

techniques therefore impose some form of purity, i.e. side-effect freeness, on specification expressions. Purity comes in many flavors, such as strong purity, weak purity and observational purity [20, 21].

The verification techniques of this paper tolerate side-effects in executable specifications to a high degree. In fact, they use a novel notion of purity, namely *relative* purity. An executable assertion is pure or impure with respect to a given separation logic specification:

$$\text{B is pure relative to P} \stackrel{\text{def}}{=} \{\text{P}\}\text{v} := \text{B}\{\text{P}\}, \text{ where v is a fresh variable.}$$

This means informally that B can be evaluated in a state satisfying P, and that the evaluation will preserve P. Similarly, for an executable assertion B containing the list of **old**-expressions **old**(E_1),...,**old**(E_n):

$$\begin{aligned}\text{B is pure relative to P,Q} \stackrel{\text{def}}{=} \exists \text{R}\cdot\ & \{\text{P}\}(\text{v}_1 := \text{E}_1, \ldots, \text{v}_\text{n} := \text{E}_\text{n})\{\text{P} * \text{R}\} \text{ and} \\ & \{\text{Q} * \text{R}\}\text{v} := \text{B}[\text{v}_1; \ldots; \text{v}_\text{n}]\{\text{Q}\}\end{aligned}$$

where v, v_1,...,v_n are fresh variables and B[v_1;...;v_n] is defined as before.

The following lemmas follow from the definitions by the rule of Consequence:

> If P $\Rrightarrow$ B, then B is pure relative to P.
>
> If P,Q $\Rrightarrow$ B, then B is pure relative to P,Q.

Examples. Consider the specification of class SLIST in Figure 1.

1. Suppose we replace the executable postcondition of *insert* with an assertion B for which $(\!|\text{B}|\!)_\text{v}$ is given by:

   t := remove_first
   insert(t)
   v := has(i)

   B is pure relative to the separation logic postcondition of *insert* because one can prove[14] $(\exists \alpha_\text{F}, \alpha_\text{S}\cdot\ \text{L(l: } \alpha_\text{F} +\!\!+ [\text{i}] +\!\!+ \alpha_\text{S}) * \alpha_\text{F} +\!\!+ \alpha_\text{S} = \alpha) \Rrightarrow \text{B}$.
2. Suppose B is an assertion for which $(\!|\text{B}|\!)_\text{v}$ is:

   insert(4)
   v := **not** is_empty

   B is not pure relative to L(l: α). We cannot complete the following proof attempt, since the needed implication does not necessarily hold:

   $\{\text{L(l: } \alpha)\}$
   insert(4)
   $\{\exists \alpha_\text{F}, \alpha_\text{S}\cdot\ \text{L(l: } \alpha_\text{F} +\!\!+ [4] +\!\!+ \alpha_\text{S}) * \alpha_\text{F} +\!\!+ \alpha_\text{S} = \alpha\}$
   v := **not** is_empty
   $\{\exists \alpha_\text{F}, \alpha_\text{S}\cdot\ \text{L(l: } \alpha_\text{F} +\!\!+ [4] +\!\!+ \alpha_\text{S}) * \alpha_\text{F} +\!\!+ \alpha_\text{S} = \alpha\}$
   // An implication is needed to establish:
   $\{\text{L(l: } \alpha)\}$

[14] The proof uses axiom information [4] expressed in L_sorted. The rule of Disjunction can combine subproofs arising from a case split on $\alpha_\text{F} = \epsilon$.

A counterexample showing that B is not pure relative to P in the context of class C comprises:

(a) a verified implementation of C and all classes transitively used by it;
(b) an execution trace where B is evaluatated in an initial state satisfying P that either faults or terminates in a state not satisfying P.

3. We use B from the previous example and show $(\exists\alpha\cdot \mathrm{L}(\mathrm{l}\colon \alpha)) \Rrightarrow \mathrm{B}$:

$\{\exists\alpha\cdot \mathrm{L}(\mathrm{l}\colon \alpha)\}$
  insert(4)
$\{\exists\alpha, \alpha_\mathrm{F}, \alpha_\mathrm{S}\cdot \mathrm{L}(\mathrm{l}\colon \alpha_\mathrm{F} +\!\!+ [4] +\!\!+ \alpha_\mathrm{S}) * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha\}$
  v := **not** is_empty
$\{\exists\alpha, \alpha_\mathrm{F}, \alpha_\mathrm{S}\cdot \mathrm{L}(\mathrm{l}\colon \alpha_\mathrm{F} +\!\!+ [4] +\!\!+ \alpha_\mathrm{S}) * \mathrm{v} = \mathrm{True} * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha\}$
$\{\exists\alpha\cdot \mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{v} = \mathrm{True}\}$

So B is pure with respect to $\exists\alpha\cdot \mathrm{L}(\mathrm{l}\colon \alpha)$.
Since $\mathrm{L}(\mathrm{l}\colon \alpha) \Rightarrow (\exists\alpha\cdot \mathrm{L}(\mathrm{l}\colon \alpha))$, this example and the previous one show that if $\mathrm{P} \Rightarrow \mathrm{Q}$ and $\mathrm{Q} \Rrightarrow \mathrm{B}$, then $\mathrm{P} \Rrightarrow \mathrm{B}$ does not hold in general.

4. Let B be any executable assertion without old-expressions. B is always pure relative to False. If B never faults, then it is also pure relative to True. □

Any notion of purity which classifies side-effects as intrinsically harmful or not will rule out executable assertions that programmers might want to write. Relative purity judges whether a side-effect is harmful or not only with respect to the properties one wants to establish. For example, the side-effects in B of examples 2 and 3 are harmless with respect to $\exists\alpha\cdot \mathrm{L}(\mathrm{l}\colon \alpha)$, but harmful with respect to $\mathrm{L}(\mathrm{l}\colon \alpha)$. This is so because the side-effect of inserting 4 in a sorted list will maintain a list structure (what the first purity specification states), but not a list structure with the same elements (what the second specification demands). The concept of relativity pervades verification already, since code is only correct or incorrect relative to a specification. We adopt it also for purity.

Relative purity guarantees soundness of reasoning and does not impose unnecessary constraints on executable assertions. An executable assertion is free to perform any side-effect as long as nothing happened from the perspective of the separation logic specification. Runtime assertion checking does nothing relative to the specification of a verified program. An executable assertion can even print 'Hello World!' if the specification permits it. Purity is in the eye of the asserter.

7.2 Predicate extraction

Programmers are conventionally encouraged to specify sets of states with computations. Hoare-style verification cannot do much with computations – it needs predicates. So conventional verification approaches, including the Spec# system [2] and JML toolset [22], have to extract predicates from computations. This can be achieved in various ways: as long as there is an agreed-upon mapping from computations to predicates, the computations can be seen as syntactic sugar for predicates.

When given an executable assertion B without old-expressions[15], one possibility is to extract a predicate P which follows from B. This is described by the connecting implication $B \Rrightarrow P$, which informally means that B evaluating to True is sufficient to conclude P. Here is a formal definition:

$B \Rrightarrow P \stackrel{\text{def}}{=} \forall Q \cdot$ if $Q \Rrightarrow B$ then $Q \Rightarrow P$, where v is fresh.

If $B \Rrightarrow P$, then $P \Rrightarrow B$ does not necessarily hold. However, the best predicate that can be extracted in this way, i.e. the strongest P such that $B \Rrightarrow P$, is precisely the weakest P such that $P \Rrightarrow B$. So if we define

$$\mathit{Best}(B) \stackrel{\text{def}}{=} \bigvee_{Q \in \{R \mid R \Rrightarrow B\}} Q$$

then the following lemmas hold:

1. $\mathit{Best}(B) \Rrightarrow B$ and $B \Rrightarrow \mathit{Best}(B)$, i.e. $\mathit{Best}(B) \Leftrightarrow B$
2. $B \Rrightarrow P$ iff $\mathit{Best}(B) \Rightarrow P$
3. If $P \Rrightarrow B$ then $P \Rightarrow \mathit{Best}(B)$

Since $\mathit{Best}(B) \Leftrightarrow B$, the predicate $\mathit{Best}(B)$ can be seen as the exact logical counterpart of B.

Examples. Let us ignore inheritance and suppose an implementation of class SLIST in Figure 1 maintains its stored elements in a list of linked nodes.

1. If *is_empty* is implemented as a field in class SLIST, then $\mathit{Best}(\text{is\_empty}) = (\text{is\_empty} \hookrightarrow \text{True})$.
2. If *is_empty* tests whether a field named 'head', which points to the first node, is **Void**, then $\mathit{Best}(\textbf{not}\ \text{is\_empty}) = (\exists x \cdot \text{head} \hookrightarrow x * x \neq \textbf{Void})$.
3. If *has* is implemented by a simple linear traversal of the nodes, then the predicate $\mathit{Best}(\text{has(i)})$ denotes that i is encountered inside a node before the next node pointer becomes **Void**. Note that $\mathit{Best}(\text{has(i)})$ is too weak to rule out a cyclic or frying-pan [23] list. Such a list does not even have to contain i in order to satisfy $\mathit{Best}(\text{has(i)})$, because the definition does not demand termination of has(i). □

As these examples show, the logical counterparts of executable assertions are often very weak. Predicate extraction techniques will have to map computations into stronger predicates in order to verify method bodies. Such predicates are similar in flavor to the ones used in this paper, and do not follow from the executable ones.

Our approach uses predicates rather than computations to characterize states. It views the separation logic specification as the primary one for verification.

[15] The treatment can be generalized to executable assertions with old-expressions.

There is no extraction of predicates, and verification treats executable assertions as computations which should evaluate to True. We do not verify the program *using* its executable assertions, we verify the program *and* its executable assertions.

8 Implementation

We extended jStar [6] to provide fully automatic verification of executable O-O assertions with respect to separation logic specifications. The implementation leverages mechanisms which are present in jStar and most separation logic proof tools: symbolic execution, implication checking and frame inference. Symbolic execution and implication checking are the basic ingredients for proving connecting implications. For example, when proving $P \Rrightarrow B$, jStar executes $v := B$ in the symbolic state P to obtain a symbolic state P'. It then checks whether $P' \Rightarrow (P * v = \mathrm{True})$. For executable postconditions, the results of old-expressions are inferred automatically with frame inference and not specified manually. Suppose jStar must prove the connecting implication $P,Q \Rrightarrow C$, where C contains a single old-expression **old**(E). Executing $v_1 := E$ in the symbolic state P yields a resulting symbolic state P'. Frame inference next determines the symbolic result R of E in $P' \vdash P * R$. Then $w := C[v_1]$ is executed in the symbolic state $Q * R$ and yields a symbolic state Q'. Finally, jStar checks whether $Q' \Rightarrow (Q * w = \mathrm{True})$.

The implementation can easily handle the examples presented in this paper, which are available online at [24]. We do not foresee any fundamental obstacles in the broader application of this new verification method.

9 Conclusions and related work

The presented framework offers a sound and simple way to verify various executable specifications with separation logic. The notion of relative purity is central to the framework and embraces side-effects in executable assertions, thereby allowing programmers more freedom of expression compared to conventional verification approaches. The framework is well-suited to separation logic proof tools and implemented in jStar.

Regarding related work, separation logic [11, 13] offers local reasoning for heap-manipulating programs and is central to the framework. Its adaptation to object-orientation [14, 5, 4] is especially relevant. Abstract predicate families [14, 15, 5] provide high levels of abstraction in specifications and proofs of O-O programs. Our presentation builds upon proof systems [5, 4] which incorporate them.

Executable specifications are embodied in programming languages such as Eiffel [1] and Spec# [2]. Dedicated specification languages such as JML [3] also use them. JML includes model classes and allows model-based contracts [9]. Another library with model classes is MML [10]. Executable assertions offer several benefits in software development, including runtime checking [1, 27] and automated testing [28, 29].

The problem of obtaining the strongest P such that B $\Rrightarrow$ P amounts to finding the weakest footprint of B preserved by B which ensures B evaluates to True. This problem appears to be similar in flavor but more general than abduction [30, 31].

Conventional approaches to verification, including the Spec# system [2] and JML toolset [22], extract predicates from assertions in a different way [17, 20]:

1. They do not use footprints. Well-formedness of the heap is exploited and any chain of references can be followed. Since a method can potentially modify any reachable object, the absence of footprints makes reasoning more global. Specifications must describe which objects are modified, because there might be external references to reachable objects. The *aliasing problem* arises because objects maintaining such references, or aliases, most likely depend on properties of the aliased objects. Approaches to the problem typically restrict or prevent aliases and/or operations on references. They include confinement, sharing and access control, ownership, immutability, uniqueness, information flow and escape analyses [32].
2. They impose purity constraints on the methods used in assertion expressions. Several notions of purity exist which prevent methods from changing the state [20, 21]. A strongly pure method has no side-effect at all. A weakly pure method does not change existing objects, but might allocate and modify new ones. An observationally pure method may modify existing objects, provided that the change is sufficiently encapsulated such that no other class can observe a change. Proving method purity becomes harder as the notion becomes more permissive. Weak purity can be proved with a combination of pointer and escape analysis [33], while a method's observational purity can be shown by proving that it simulates a weakly pure one from the perspective of other classes [21].
3. They encode pure methods and their contracts in first-order logic as uninterpreted functions and axioms respectively [20, 17, 34]. Checking well-formedness of pure method specifications is vital in this step, because inconsistent axiomatizations can result from unsatisfiable specifications or recursive specifications which are ill-founded [35]. A pure method call in an assertion is encoded as an application of the corresponding uninterpreted function. A loop, written in the stylized form of quantification over a finite domain (as in JML and Spec#), is directly encoded as a quantified formula. The reader is referred to [17] for assertion encoding details. Model class and model-based contract encoding is discussed in [19].

This paper shows that a marriage between executable specifications and separation logic-based reasoning is possible.

Acknowledgements
Special thanks to Matthew Parkinson and Sebastian Nanz for feedback on this work, and Matthew Parkinson for helping with the jStar implementation. Van Staden was supported by ETH Research Grant ETH-15 10-1. Calcagno was partially funded by EPSRC.

References

1. ECMA International: Standard ECMA-367. Eiffel: Analysis, Design and Programming Language. 2nd edn. (June 2006)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: CASSIS '05. Volume 3362 of LNCS., Springer (2005) 49–69
3. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes **31**(3) (2006) 1–38
4. Van Staden, S., Calcagno, C.: Reasoning about multiple related abstractions with MultiStar. Technical Report 666, ETH Zurich (2010)
5. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2008) 75–86
6. Distefano, D., Parkinson J, M.J.: jStar: towards practical verification for Java. In: OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, New York, NY, USA, ACM (2008) 213–226
7. Jacobs, B., Smans, J., Piessens, F.: Verifying the composite pattern using separation logic. SAVCBS Composite pattern challenge track (2008)
8. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming **62**(3) (2006) 253–286
9. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: cleanly supporting abstraction in design by contract: Research articles. Softw. Pract. Exper. **35**(6) (2005) 583–599
10. Schoeller, B., Widmer, T., Meyer, B.: Making specifications complete through models. In: Architecting Systems with Trustworthy Components. Volume 3938 of LNCS., Springer (2006) 48–70
11. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: CSL '01. Volume 2142 of LNCS., Springer (2001) 1–19
12. Raza, M., Gardner, P.: Footprints in local reasoning. Logical Methods in Computer Science **5**(2:4) (2009) 1–27
13. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, IEEE Computer Society (2002) 55–74
14. Parkinson, M.J.: Local reasoning for Java. PhD thesis, University of Cambridge, Computer Laboratory. Technical Report UCAM-CL-TR-654 (November 2005)
15. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL '05: Proceedings of the 32nd annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2005) 247–258
16. Summers, A.J., Drossopoulou, S., Müller, P.: The need for flexible object invariants. In: IWACO '09: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, New York, NY, USA, ACM (2009) 1–9
17. Leino, K.R.M., Schulte, W.: A verifying compiler for a multi-threaded object-oriented language. Software System Reliability and Security **9** (2007) 351–416
18. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing **19**(2) (2007) 159–189
19. Darvas, Á., Müller, P.: Faithful mapping of model classes to mathematical structures. IET Software **2**(6) (2008) 477–499

20. Darvas, A., Müller, P.: Reasoning about method calls in interface specifications. Journal of Object Technology (JOT) **5**(5) (June 2006) 59–85
21. Naumann, D.A.: Observational purity and encapsulation. Theor. Comput. Sci. **376**(3) (2007) 205–224
22. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) **7**(3) (2005) 212–232
23. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2008) 101–112
24. Van Staden, S.: jStar sources of the examples. Available online at `http://se.inf.ethz.ch/people/vanstaden/executablespecs.tgz`.
25. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: ESOP '08. Volume 4960 of LNCS., Springer (2008) 307–321
26. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Enhancing modular OO verification with separation logic. In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2008) 87–99
27. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java modeling language (JML). In: Proceedings of the international conference on Software engineering research and practice (SERP '02), Las Vegas, CSREA Press (2002) 322–328
28. Meyer, B., Ciupa, I., Leitner, A., Liu, L.L.: Automatic testing of object-oriented software. In: SOFSEM '07. Volume 4362 of LNCS., Springer (2007) 114–129
29. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Experimental assessment of random testing for object-oriented software. In: ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis, New York, NY, USA, ACM (2007) 84–94
30. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2009) 289–300
31. Gulavani, B.S., Chakraborty, S., Ramalingam, G., Nori, A.V.: Bottom-up shape analysis. In: SAS '09. Volume 5673 of LNCS., Springer (2009) 188–204
32. Clarke, D., Drossopoulou, S., Müller, P., Noble, J., Wrigstad, T.: Aliasing, confinement, and ownership in object-oriented programming (IWACO). In: Object-Oriented Technology. ECOOP 2008 Workshop Reader. Volume 5475 of LNCS., Springer (2008) 30–41
33. Salcianu, A., Rinard, M.C.: Purity and side effect analysis for Java programs. In: VMCAI '05. Volume 3385 of LNCS. (2005) 199–215
34. Darvas, A., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: FASE '07. Volume 4422 of LNCS., Springer (2007) 336–351
35. Rudich, A., Darvas, Á., Müller, P.: Checking well-formedness of pure-method specifications. In: FM '08. Volume 5014 of LNCS., Springer (2008) 68–83