Lightweight Type-Like Hoare-Separation Specs for Java

Tiago Santos

Departamento de Informática FCT/UNL, Lisboa, Portugal
tiago.santos@fct.unl.pt

Abstract. Type systems are effective but not very precise, while program logics tend to be very precise, but undecidable. The aim of this work is extend the expressiveness of more familiar type-based verification towards more informative logical reasoning, without compromising soundness and completeness. We thus investigate a lightweight specification language based on propositional logic for Java and describe a prototype implementation on top of Polyglot. The verification process is modular and based on Dijkstra's weakest precondition calculus, which we extend to a large fragment of the Java object-oriented language. A distinguishing aspect of our approach is a novel "dual" separation logic formulation, which combines Hoare logic with separation logic reasoning in a unified way, allowing us to handle aliasing through a separation of pure from linear properties.

Keywords: Lightweight Specifications, Static Analysis, Verifying Compiler, Hoare Logic, Separation Logic, Weakest Precondition Calculus

1 Introduction

Over the past decades, specification, verification and validation of software present a very important role in software development, since they guarantee correctness and the absence of runtime errors statically, reducing maintenance and development costs. Last year marked the fortieth anniversary of Hoare's article that contributed to the revolution of this subject [1, 2].

The use of formal methods for verifying program properties has witnessed an impulse recently, with tools and programming languages (e.g. ESC/Java2 [3], JACK [4], Spec# [5]) that have great expressiveness power and allow static verification of programs. However, most of them require user interaction and have very complex specification language, which are obstacles for their use.

Lightweight specification languages, on the other hand, thought presenting less expressiveness, still allow reasoning about interesting properties of a system with less effort, thus making its usage compelling in software development.

Figure 1 illustrates a simple example of how to specify the absolute value of a number, ensuring that the result is never negative (`ensures !return:neg`). Other motivating examples are the specification of a buffer, where one can write only if there are free positions and read if the buffer has data; protocols that

INForum 2010 - II Simpósio de Informática, Luís S. Barbosa, Miguel P. Correia (eds), 9-10 Setembro, 2010, pp. 43–54

```
public int abs(int x)
    ensures !return:neg
{
    if (x > 0) return x;
    else return -x;
}
```

Fig. 1. Specification – Absolute Value of a Number

follow a similar approach, such as specifying an FTP session; and simpler situations, like ensuring that a given variable is not null, avoiding invalid dereferences. With our solution, these checks can be specified in a simple way and with little impact on the compilation process.

This paper presents a lightweight specification language and its integration into the Java programming language, by extending its type system and creating a verifying compiler. This extension, called SpecJava, allows the use of assertions in the Java language, providing developers a way to write correct programs according to their specifications. The specifications can be expressed in a simple way when compared to existing tools and the verification process is fully automatic. The main contributions of this paper are:

- a lightweight specification language for Java and the underlying logic (Sect. 2);
- the technique used to verify a SpecJava program (Sect. 3);
- the implementation of SpecJava (Sect. 4).

2 Lightweight Specification Language

SpecJava's specification language is similar to JML [6] and Spec# [5], but is lightweight and based on a monadic dual logic. It is simpler, not presenting, for example, quantifiers and reference to the value of an expression in its precondition and uses a novel approach to handle aliasing by separating pure from linear properties. Like JML and Spec#, we use the reserved keywords `requires` and `ensures` to describe, respectively, pre and postconditions of a procedure. As for class and loop invariants, we use also the keyword `invariant`.

2.1 Dual Hoare-Separation Logic

In this section we present the underlying logic of the developed specification language denoted dual logic. The purpose of defining a new logic comes from the need of having aliasing control on our specification language, since we are extending an object-oriented language where alias can occur. The original Hoare Logic was designed for an imperative programming language with only simple values, where therefore aliasing does not cause any problems.

As the name suggests, a dual logic formula is composed by two components (Fig. 2): the left side (ϕ), named pure formula, composed by a single formula in propositional logic, states properties of immutable objects, or objects that cannot

$$
\begin{array}{lll}
\psi ::= & \phi + \varphi & \text{(Dual Formula)} \\
\varphi ::= & \emptyset \mid \phi \mid \phi * \varphi & \text{(Linear Formula)} \\
\phi ::= & & \text{(Classic Formula)} \\
& \bot & \text{(Bottom)} \\
\mid & \phi \; lc \; \phi & \text{(Binary Formula)} \\
\mid & \neg\phi & \text{(Negation)} \\
\mid & (\phi) & \text{(Parenthesized Formula)} \\
\mid & P(t_1, t_2, \cdots, t_n) & \text{(Predicate Symbols)} \\
lc ::= & \vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow & \text{(Logical Connectives)} \\
t ::= & & \text{(Terms)} \\
& c & \text{(Constants)} \\
\mid & x & \text{(Variables)} \\
\mid & f(t_1, t_2, \cdots, t_n) & \text{(Function Symbols)}
\end{array}
$$

Fig. 2. Dual Logic Syntax

be aliased (e.g. primitive types of Java) and the right side (φ), named linear formula, composed by a set of classic formulas in propositional logic, models the linear part of the heap (inspired by separation logic [7]). The formulas in the linear side are disjoint, in the sense that two formulas can not refer to a same linear object, and each formula only talks about a single linear object. Note the existence of predicates and functions that allow, respectively, to express properties (such as relations between two terms for e.g. $>(2,3)$) and compose constants and variables with operators to do certain computations (e.g. $-(2,3)$).

We can also state specific zones of the heap on which we want to mention properties by using heap restriction, defined as follows.

Definition 1 (Heap Restriction). *Let φ be a linear formula and $x_1, x_2, \cdots, x_n$ linear variables, then $\varphi \downarrow \{x_1, x_2, \cdots, x_n\}$ is named restricted linear formula to $x_1, x_2, \cdots, x_n$, that is, the subset of formulas contained in φ that correspond to the variables $x_1, x_2, \cdots, x_n$.*

In addition to this definition, we can exclude parts of the heap over which we do not intend to refer properties.

Definition 2 (Heap Exclusion). *Let φ be a linear formula and $x_1, x_2, \cdots, x_n$ linear variables, then $\varphi - \{x_1, x_2, \cdots, x_n\}$ is named linear formula excluding $x_1, x_2, \cdots, x_n$, that is, the subset of formulas contained in φ that do not contain information of variables $x_1, x_2, \cdots, x_n$.*

As an example, in Fig. 3, we define a linear formula (φ) and apply the two previous definitions.

$$
\begin{array}{c}
\varphi \equiv P_1(x) * P_2(y) * P_3(z) \\
\varphi \downarrow \{x, z\} = P_1(x) * P_3(z) \\
\varphi - \{x, z\} = P_2(y)
\end{array}
$$

Fig. 3. Heap Restriction and Exclusion Example

2.2 Assertions

In this section we present the abstract syntax of SpecJava's assertions. As we can see in Fig. 4, this allows to describe the state in which certain objects or primitive types are, specifically, fields (fn), procedures parameters (pn) and methods return (`return`), or the state of the class itself (`this`). States are composed by a set of basic states, which apply to primitive types. For primitive boolean variables, we associate the states `true` and `false` and for numeric variables, we associate `pos`, `neg` or `zero`. With regard to object references, these can be null references (`null`), or refer to states defined in the class of the object type (sn).

D	::=	CF `+` SLF	(Dual Formula)
SLF	::=	CF | CF `*` SLF	(Sep. Formula)
CF	::=	`true` | `false` | CF bop CF | `!` CF | b : S	(Classic Formula)
bop	::=	`&&` | `||` | `=>` | `<=>`	(Logical Connectives)
b	::=	fn | `this` | `return` | pn	(Properties/States – Target)
S	::=	`true` | `false` | `pos` | `neg` | `zero` | `null` | sn	(Properties/States)

$pn, sn, fn \in$ parameter/state/field names

Fig. 4. Abstract Syntax – Assertions

2.3 Classes

In this section we present the abstract syntax for classes. As we can see in Fig. 5 the novelty is the class specification. An invariant declared as `invariant` D express a property that all classes instances must satisfy. A class preserves its invariants if all methods preserve those invariants. However, contrarily to Spec#, where we can only temporarily break invariants via an explicit statement, in our solution invariants can be broken during a method's execution, as long as they are restored at the end. The constructors must guarantee also, in addition to their postconditions, the invariants of the class.

In addition to class invariants, class level specifications are composed by two other constructions, to define states/properties associated to the class. These can

classDecl	::=	`class` cn { *classMember** }	(Class Declaration)
classMember	::=	... | *field* | *method* | *constructor* | *classSpec*	(Class Member)
classSpec	::=		(Class Specification)
		`define` sn`;`	(Abstract Definition)
		| `define` sn `=` D`;`	(Concrete Definition)
		| `invariant` D`;`	(Class Invariant)
field	::=	T fn (`=` E)? `;`	(Instance Variable)

$mn, cn, sn, fn \in$ method/class/state/field names

Fig. 5. Abstract Syntax – Classes

be concrete or abstract, and are declared as `define` sn = D and `define` sn, respectively. Concrete definitions are built based on states/properties observed in class instance variables and/or other definitions at class level (abstract or concrete). Abstract definitions represent class abstract states/properties, without using other definitions and/or states observed in class instance variables.

2.4 Procedures

In this section we present the abstract syntax for procedures. Methods and constructors specification consists of two formulas concerning to preconditions and postconditions, declared as `requires` D, `ensures` D, respectively (Fig. 6).

method	::=	*modifier* T $mn(\overline{arg})$ $\overline{spec}$ { ST }	(Method Declaration)
constructor	::=	*modifier* $cn(\overline{arg})$ $\overline{spec}$ { ST }	(Constructor Declaration)
modifier	::=	`static` | ... | `pure`	(Modifiers)
spec	::=		(Procedure Specification)
		`requires` D	(Precondition)
	|	`ensures` D	(Postcondition)
ST	::=	...	(Statement)
	|	`assume` D	(Assume)
	|	`sassert` D	(Static Assert)

$mn, cn \in$ method/class names

Fig. 6. Abstract Syntax – Procedures

The preconditions of a procedure specify conditions that must be true at the beginning of its execution. In these conditions we can refer to properties from the class, fields, and method parameters. With respect to postconditions, they designate the object state after performing the operation, and may involve, in their conditions method's return state.

In addition to these specifications, assume and assert statements are also supported, with the usual meaning of assuming or verifying a condition at a given point by using `assume` D and `sassert` D, respectively. Last but not least, to specify that a procedure do not change the state of an object, Java modifiers are extended with the keyword `pure`.

As an example, in Fig. 7 we create a buffer class with the states full and empty (Fig. 7a) and the respective constructor with a specification to ensure that the buffer is created empty (Fig. 7b).

3 Program Verification

This section presents the verification process of a SpecJava program. Our approach is based on the weakest precondition calculus (wp-calculus, Definition 3), initially proposed by Dijkstra [8, 9], that extended Hoare Logic [1] by creating

```
public class Buffer {

   define empty = count:zero;
   define full;

   private int buffer[];
   invariant + !buffer:null;
   ...

   private int count;
   invariant !count:neg;
   ...

}
```

(a) Class Specification

```
public Buffer(int size)
   requires size:pos
   ensures + empty && !full
   ...
{ ...
  assume + empty && !full;
}

public pure int dataSize()
   ensures return:zero
             || return:pos
{  sassert !count:neg;
   return count;
}
```

(b) Procedures Specification

Fig. 7. SpecJava – Buffer Specification Example

a method to define the semantics of an imperative programming language, assigning to each statement a predicate transformer, allowing validity verification of a Hoare Triple. In this work, we propose an extension to that calculus, for an object-oriented language, in this case Java. According to the properties referred in [9, pp. 18:19], to prove that a SpecJava program is correct against its specification, it is necessary to associate to each statement its predicate transformer.

Definition 3 (Weakest Precondition). *Let S be a sequence of statements and R its postcondition, then, the corresponding weakest precondition is represented as:*

$$wp\,(S,\ R)$$

Figures 8 and 9 illustrate the more relevant weakest preconditions rules, which concern to loops, object creation and non-void method invocation.

Figure 8 presents wp-calculus for pure statements. In loops, the loop condition (ε) must be pure, composed only by constants or variables of primitive types. Regarding to object creation, these have to be immutable or pure. In this approach an object is considered pure if all methods of the object class are pure. For non-void method invocation the return value is also pure. We can see that the weakest precondition rule for loop is quite simple and corresponds to its invariant, since the invariant must hold in every iteration of the loop and it also must be valid at the beginning of the loop, corresponding to the premises of the rule.

Concerning to object creation, it is necessary to have as its weakest precondition, on the pure side of the result, in respect to a postcondition C + S the class constructor precondition of the object that we are instantiating, and also that we end in a state whose constructor postcondition implies C. For the linear part of the result, since we can have linear arguments we must assure as weakest precondition the constructor precondition, remaining all the facts of elements

$$\text{[loop]}\quad \frac{(I \wedge \neg\varepsilon) \Rightarrow R \quad (I \wedge \varepsilon) \Rightarrow wp(ST, I)}{wp\left(\begin{array}{l}\texttt{while } (\varepsilon)\\ \quad \texttt{invariant } I \texttt{ , } R\\ ST\end{array}\right) = I}$$

$$\text{[pure creation]}\quad \frac{S{\downarrow}\{\overline{z}\} = \emptyset}{wp(x = \texttt{new } cn(\overline{y},\, \overline{z}),\, C{+}S) = \left(\begin{array}{c} Q_{cn_A}[\overline{p_1}/\overline{y}] \\ \wedge \\ \left(\left(\begin{array}{c} f \neq \texttt{null} \\ \wedge \\ R_{cn_A}[this/f, \overline{p_1}/\overline{y}] \end{array}\right) \Rightarrow C[x/f]\right) \end{array}\right) + \left(\begin{array}{c} Q_{cn_B}[\overline{p_2}/\overline{z}] \\ * \\ S - \{\overline{z}\} \end{array}\right)}$$

$$\text{[pure non-void call]}\quad \frac{S{\downarrow}\{\overline{z}\} = \emptyset}{wp(x = k.mn(\overline{y},\, \overline{z}),\, C{+}S) = \left(\begin{array}{c} k \neq \texttt{null} \wedge Q_{mn_A}[this/k,\ \overline{p_1}/\overline{y}] \\ \wedge \\ \left(\left(\begin{array}{c} k \neq \texttt{null} \\ \wedge \\ R_{mn_A}[this/k,\ \overline{p_1}/\overline{y}, return/f] \end{array}\right) \Rightarrow C[x/f]\right) \end{array}\right) + \left(\begin{array}{c} Q_{mn_B}[\overline{p_2}/\overline{z}] \\ * \\ S - \{\overline{z}\} \end{array}\right)}$$

$Q_{mn}/Q_{cn} \in$ method/constructor precondition
$R_{mn}/R_{cn} \in$ method/constructor postcondition
$\varepsilon \in$ pure expression
$\overline{p_1}/\overline{y} \in$ pure formal/concrete parameters
$\overline{p_2}/\overline{z} \in$ linear formal/concrete parameters
$f \in$ fresh name

Fig. 8. WP Calculus – Pure Rules

that are not affected by the procedure call on the heap, through heap exclusion (cf. Definition 2) of the constructor linear parameters. For non-void method invocation the weakest precondition is similar to object creation, but in addition we must guarantee that we are not doing a null reference invocation ($k \neq$ `null`). We can see also that we replace x and the return variable by a fresh name f in the pure side, because the result is pure and we are assigning a new value to the variable x that corresponds to the newly created object or the result of the method. Last but not least, we also need to have the auxiliary condition in the premise, which allow us to check that the postcondition S does not refer to states of the linear parameters, assuring the linearity of our calculus.

Figure 9 shows weakest precondition calculus for linear statements. As we can see these rules are quite similar to pure statements weakest precondition calculus. For procedures we must exclude variable x from the heap (cf. Definition 2) because now x is linear and we are assigning it a new value that does not exist at the precondition state. For methods we must also guarantee that we exclude information about the object where we are calling the method because since the call is not pure we assume that the state of the object changes. In respect to linear assignment, this is similar to Hoare assignment rule and we must guarantee as precondition that variable y acquire x properties by replacing all occurrences of x by y.

As for the auxiliary conditions in the premises, we must check in object creation and method call that after the invocation we end in a state where the

[linear assign]

$$\frac{S{\downarrow}\{y\} = \emptyset}{wp(x = y,\, C{+}S) = C{+}S[x/y]}$$

[linear creation]

$$\frac{S{\downarrow}\{\overline{z}\} = \emptyset \qquad \texttt{true}{+}\Big(\big(x \neq \texttt{null} \wedge R_{cn_B}[this/f]\big) \Rightarrow S{\downarrow}\{x, \overline{z}\}[x/f]\Big)}{wp(x = \texttt{new}\ cn(\overline{y}, \overline{z}),\, C{+}S) = \left(\begin{array}{l} Q_{cn_A}[\overline{p_1}/\overline{y}] \wedge \\ \big(R_{cn_A}[\overline{p_1}/\overline{y}] \Rightarrow C\big) \end{array}\right){+}\big(Q_{cn_B}[\overline{p_2}/\overline{z}] * S{-}\{x, \overline{z}\}\big)}$$

[linear non-void call]

$$\frac{S{\downarrow}\{\overline{z}\} = \emptyset \qquad \texttt{true}{+}\Big(\big(k \neq \texttt{null} \wedge R_{mn_B}[this/k, return/f]\big) \Rightarrow S{\downarrow}\{k, x, \overline{z}\}[x/f]\Big)}{wp(x = k.mn(\overline{y}, \overline{z}),\, C{+}S) = \left(\begin{array}{l} Q_{mn_A}[\overline{p_1}/\overline{y}] \wedge \\ \big(R_{mn_A}[\overline{p_1}/\overline{y}] \Rightarrow C\big) \end{array}\right){+}\left(\begin{array}{l} \big(k \neq \texttt{null} \ \wedge Q_{mn_B}[this/k,\ \overline{p_2}/\overline{z}]\big) * \\ S - \{k,\ x,\ \overline{z}\} \end{array}\right)}$$

Q_{mn}/Q_{cn} ∈ method/constructor precondition
R_{mn}/R_{cn} ∈ method/constructor postcondition
$\overline{p_1}/\overline{y}$ ∈ pure formal/concrete parameters
$\overline{p_2}/\overline{z}$ ∈ linear formal/concrete parameters
f ∈ fresh name

Fig. 9. WP Calculus – Linear Rules

procedure's postcondition implies the linear zone of the heap modified by the procedure, by restricting the heap (cf. Definition 1) and that the postcondition S does not refer to states of the linear parameters. It is still necessary to verify, in linear assignment that we do not have information of variable y on the linear side of the heap, since our calculus is linear and the state of y is transfered to x after the assignment, removing all the facts of y from the heap.

To verify that a program is correct according to its specification, the following Hoare Triples must be valid:

$$\forall_{mn} : \{\ Q_{mn} \wedge I_c\ \}\ ST\ \{\ R_{mn} \wedge I_c\ \}$$

$$\forall_{cn} : \{\ Q_{cn}\ \}\ ST\ \{\ R_{cn} \wedge I_c\ \} \quad ,$$

where ST is the procedure body, composed by a sequence of statements and I_c corresponds to class invariants. Thus, methods must preserve class invariants and constructors in addition to its postconditions must assure also the class invariants.

Supposing that we have a SpecJava program, we want to verify that it is valid against its specification using the above mentioned weakest precondition calculus. The verification process is modular, that is, only one procedure at a time is verified. Considering the body of a procedure of this language as the statements sequence $s_1, s_2, \ldots, s_n$ with precondition $\{\ Q\ \}$ and postcondition $\{\ R\ \}$, by applying wp-calculus rules, we obtain as a final result the more general precondition ($\{\ Q_0\ \}$). After this process, for the program to be valid according to its specification, the precondition of the procedure has to imply the more general one ($Q \Rightarrow Q_0$).

The formulas obtained in the verification conditions generation process are formulas in propositional logic, unlike other situations where are generated formulas in first order logic, due to quantifiers, which are not present in the developed specification language, therefore reducing them to a propositional calculus. A formula in propositional logic is said to be satisfiable if we can assign logical values to its variables so that the formula is true. This boolean satisfiability problem is NP-complete, thought is decidable and can be solved using a SAT-Solver. However, the formulas obtained by our calculus contain, uninterpreted predicates and functions that require specific background theories to be solved. For obtaining solutions to these problems, is common to use SMT-Solvers [10].

Despite of an NP-complete problem, there are finite algorithms for obtaining solutions to these problems. However these algorithms execution time may be too high due to the size of the formulas to be verified. Although, we think that this is not a problem to this solution, since it is modular, procedure by procedure, and the size of the formulas is not very high.

4 Implementation

This section presents some implementation details of our system. The extension of the Java language was implemented using the Polyglot tool [11], which implements an extensible compiler for Java 1.4. This tool is also implemented in Java and, in its simplest form only, performs semantics verification of Java. However, it can be extended in order to define changes in the compilation process, including changes in the abstract syntax tree (AST) and semantic analysis.

Polyglot has been used in several projects and has proved to be quite useful when developing compiler extensions to Java-like languages. This work's specification language was fully integrated in the Java compiler, extending the Java syntax with the language proposed in Sect. 2.

The verification process of a program was integrated into Polyglot compilation passes. In addition we extended the semantics verification and typification with new conditions, like not allowing to state a linear property on a pure formula, properties that are not declared, assuring that primitive types only refer to base properties, etc. We developed an internal representation for propositional and dual logic formulas aiming an independent format that can be used by any SMT-Solver. We have also implemented a visiting architecture over formulas to perform operations needed by wp-calculus, such as variable substitution, conversion to conjunctive normal form, obtaining pure and linear variables of a formula, etc. As for the wp-calculus, it is realized in a new pass, after type checking, that goes through each procedure generating the corresponding verification conditions. After this step another pass translates the generic formulas to the SMT-Solver representation and submits it for validity proof. Thus by using this architecture the verification process is independent from a particular SMT-Solver until the submission point, allowing further extensions to any SMT-Solver just by adding a class that translates the generic formulas to the solver's input format or the recently used SMT-lib format. The current SMT-Solver used is

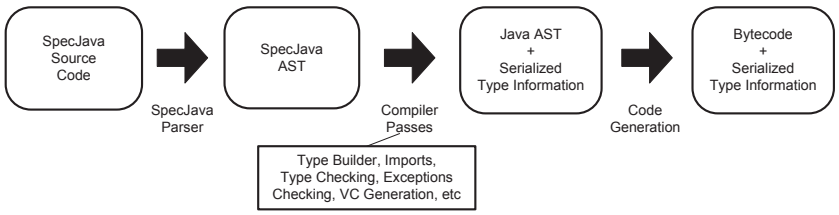

Fig. 10. SpecJava Compiler Architecture

CVC3, it is efficient and has the necessary built-in theories (equality over uninterpreted function and predicate symbols, real and integer linear arithmetic) to prove our verification conditions.

We illustrate the compiler architecture of the developed language in Fig. 10. First, the source code is parsed, generating the corresponding AST. Next, several passes are performed over the AST, including the passes described above. In these passes, if any error occurs (e.g. typing, invalid specification), the compilation process terminates showing the cause of the error, the location in the source code and a counter-example, in case of invalid specification. In a next stage, the AST is translated into a Java AST with type information serialized, preserving new types created by the extension. Finally, the Java code obtained from the previous pass is compiled to bytecode by a standard Java code compiler (e.g. `javac`).

5 Related Work

There are lots of tools and programming languages that support program verification according to its specification (e.g. ESC/Java2 [3], Eiffel [12], Spec# [5]). Some of them, like Eiffel, transform program specification into executable code and perform those verifications at runtime, while others also support formal verification of a program with static analysis (e.g. Spec#). There are four properties that characterize these tools: specification language used, programming language coverage, verification techniques and verification mode.

Regarding the specification language used, tools like ESC/Java2, LOOP [13], JACK [4] and Forge [14] use JML. In KeY [15] specifications are written in OCL. Spec# programming language and jStar [16] tool, have their own specification language. Spec#'s specification language is similar to JML and jStar's is farmost different from the others. In our approach, the specification language is closer to Spec# and JML, but with the novelty of separating pure from linear properties, modeling the heap on the linear side of a formula to track aliasing problems, that cannot be expressed on those languages. Spec#, instead, uses a ownership model to deal with aliasing and specifies frame conditions explicitly by using a `modifies` clause denoting which pieces of the program state a method is allowed to modify.

As for programming language coverage, Spec# and ESC/Java2 cover most of their respective target languages' constructions. In ESC/Java2 it is also possible

to detect synchronization errors such as race conditions and deadlock situations at compile time. Certain tools do not support some features of Java such as dynamic class loading or multithreading (e.g. KeY, Forge, LOOP). In this version of our solution some features of Java are not supported, like inheritance, exceptions, break, continue and switch statements, interface specifications.

With regard to verification techniques, there are several approaches. ESC/-Java2, LOOP, JACK and Spec# use Dijkstra weakest precondition calculus or variants to generate verification conditions. KeY tool uses dynamic logic, where deduction is based on symbolic execution of the program. Forge uses a technique named limited verification that uses symbolic execution and reduces the problem to boolean variables satisfiability. As for jStar, it combines abstract predicate family with symbolic execution and abstraction using separation logic. LOOP defines a denotational semantics in PVS, in contrast to the approaches followed by ESC/Java2, JACK and Spec# that depend directly on an axiomatic semantics. Our approach is based on a variant of Dijkstra wp-calculus to object-oriented languages resembling ESC/Java2, JACK and Spec# and also depends on an axiomatic semantics.

Finally, with regard to verification mode, in ESC/Java2, Forge, jStar and Spec# the verification process is fully automatic, as our SpecJava. LOOP tool needs user intervention. KeY and JACK support both modes.

With these tools we can verify a program against its specification. They have high expressiveness level and allow very complex specification. However, this is the main reason for its rejection by developers, who in general do not have high expertise in logic, nor intend to deal with all the complex mechanisms that are associated with most of these tools. Another point is the fact that most tools are not native to the respective programming language, which forces the use of separated tools in the software development process. On the other hand, our work has less expressive power, but still allows interesting specifications to be written, and its simplicity is appealing to developers.

6 Concluding Remarks and Future Work

In this paper we presented a lightweight specification language for Java, its integration in the compilation process, extending the checks carried out by the type system, and the underlying calculus of the verification process of a SpecJava program. The lightweight specification language developed is closer to JML and Spec#, it is based on propositional logic and is quite intuitive, allowing developers to specify their programs easily and check them automatically at compile time.

This work contributes for a better and easier integration of program verification during its development by augmenting the programming language design. For future work we highlight the following points:

- extend wp-calculus to support inheritance and exception mechanisms;
- support specification at interface level and in separated files, allowing core Java classes specification;

– support `break` and `continue` statements, that change the execution flow of a program.

Acknowledgments. I would like first to thank Prof. Luís Caires for his guidance throughout the development of this work. I thank to Mario Pires and Luísa Lourenço for their comments on previous versions of this paper.

References

1. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.
2. C. A. R. Hoare. Retrospective: An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 52(10):30–32, 2009.
3. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, et al. Extended Static Checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM, New York, NY, USA, 2002.
4. G. Barthe, L. Burdy, J. Charles, et al. JACK – a tool for validation of security and behaviour of Java applications. *Lecture Notes in Computer Science*, 4709:152, 2008.
5. Mike Barnett, Leino, and Wolfram Schulte. *The Spec# Programming System: An Overview*, volume 3362/2005 of *Lecture Notes in Computer Science*, pages 49–69. Springer, Berlin / Heidelberg, January 2005.
6. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design, 1999.
7. John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. *Logic in Computer Science, Symposium on*, 0:55–74, 2002.
8. Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, August 1975.
9. Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Inc., October 1976.
10. L. De Moura and N. Bjørner. Satisfiability Modulo Theories: An Appetizer. *Formal Methods: Foundations and Applications*, pages 23–36, 2009.
11. Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.
12. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, second edition, March 2000.
13. B. Jacobs and E. Poll. Java Program Verification at Nijmegen: Developments and Perspective. *Lecture Notes in Computer Science*, pages 134–153, 2004.
14. G.D. Dennis. *A Relational Framework for Bounded Program Verification.* PhD thesis, Massachusetts Institute of Technology, 2009.
15. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, et al. The KeY tool. *Software and Systems Modeling*, pages 32–54, April 2004.
16. Dino Distefano and Matthew. jStar: Towards practical verification for Java. *SIGPLAN Not.*, 43(10):213–226, 2008.