Testing Automation of Context-Oriented Programs Using Separation Logic

Mohamed A. El-Zawawy

1 College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh 432, Saudi Arabia
2 Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt

Correspondence should be addressed to Mohamed A. El-Zawawy; maelzawawy@cu.edu.eg

Received 12 July 2014; Accepted 2 December 2014; Published 29 December 2014

1. Introduction

To support and dynamically control the modularization of crosscutting concerns [1], a new programming style, context-oriented programming (COP) [2], has appeared. COP can be defined as a programming approach that produces software that is dynamically adaptable. COP was invented to treat programming problems when the behavior of the required software changes at runtime depending on the execution conditions. This is not easily achieved by classical approaches of programming. COP was established on languages like Smalltalk [3], Java [4], and JavaScript [5]. While the idea of homogeneous crosscutting is to execute the same source code at the join points of concerns, the idea of heterogeneous crosscutting concerns is to execute different source code at the join points. Heterogeneous crosscutting [1] is the type of crosscutting concerns adopted by COP. Partial functions declarations adapting common functions to their new style are used in COP to implement these crosscutting methodologies.

Main components of COP include (a) layers of variant functions for providing performance alterations and (b) a tool for layer activation/deactivation. A variant function is a function that can be executed after, before, and around the same (variant) function included in another layer. Hence a layer is a group of variant functions.

Layer declarations are used to encapsulate partial function declarations. The semantics of classes and that of crosscutting can be combined at the runtime. COP [2] provides a block statement defining a group of layers to enable runtime layer composition. Determined by the statement block, an execution scope is also provided by COP. In this scope layers are typically combined with the base system. Another statement enables stopping execution of specified layers. The keywords denoting these two statements are with and without statements [2]. COP appears to be a very convenient technique for encapsulation of homogeneous crosscutting concerns as proved by many applications.

However, so far research did not provide appropriate logic to reason about context-oriented programs. A main concern in COP is the treatment of layers activations/deactivations. Usually, the set of active layers changes from one program point to another. Therefore the efficiency of any logic for
COP strongly depends on using correct layers. In spite of the concept of activating/deactivating a layer is simple, and its logical treatment is a bit complex.

To reason about resources, Reynolds [6] and independently Samin and O’Hearn [7] presented new logic (separation logic). This was based on Burstall’s early work. A logical operator (separation conjunction) enabling localizing operation actions on shared resources is among the main contributions of separation logic. Sound form of local reasoning was reached by the separation conjunction which ensures the invariance of resources not included in preconditions.

To model the meanings of COP concepts, this paper introduces operational semantics. Accurate meanings for basics of COP languages are provided by the semantics. Axiomatic semantics for COP is the second contribution of this paper. This semantics is an extension of separation logic to establish and ensure partial correctness specifications for COP programs. Rather than other logic, separation logic seems most suitable for verifying COP. This is so as the concept of context changing is explicitly “treated” by the special operators of separation logic. One of the challenges in this paper is to expose this explicit relationship in the form of formal definitions and inference rules. Judgments of the proposed logic have the form $L = [P]S(Q)$ meaning that

(i) if layers of $L$ are active, then executing $S$ in a state satisfying $P$ is ensured not to abort,

(ii) if layers of $L$ are active and the execution of $S$ in a state satisfying $P$ ended at some state, then this last state satisfies $Q$.

Assertions $P$ and $Q$ may reason about the program control locations as well as upon the values of local and global variables. A system of axioms and inference rules constituting the proposed logic is introduced in the paper which presents a partial correctness specification, for example, COP-program, as well. Against the proposed operational semantics, the mathematical soundness of the logical system is shown in this paper.

Contributions of this paper are as follows:

1. new operational semantics to provide accurate meanings for COP concepts,
2. novel axiomatic semantics to construct and validate partial correctness specifications for COP programs.

The rest of the paper is organized as follows. Section 2 presents in detail the programming language J-COP and its operational semantics. The assertion language, axioms, and inference rules constituting the logical system are presented in Section 3 which also presents the mathematical proof for the correctness of the system and an example of its use. Related research is presented in Section 4.

2. Programming Language and Operational Semantics

The syntax and semantics of the programming language (dubbed J-COP) used in this paper are presented in this section. Basic aspects of object-oriented programming like inheritance and subtyping are modeled in J-COP which follows Java syntax for comparable structures. Algorithm 1 presents the syntax of J-COP.

J-Cop is a rich model for COP. The model includes basic language constructs and is extendable directly over well-studied Java features. Although the model is expressive enough to include more language features, it is simple. In addition to typical Java constructs, the model allows layers activation/deactivation, overriding variant functions, and a technique for executing proceed and super.

We let $C$ denote the typical member of the set of names of classes $\mathcal{C}$ which is included in the set of types ($\mathcal{C}$). In addition to $\mathcal{C}$, $\mathcal{T}$ includes function and reference types. The typical element of $\mathcal{T}$ is denoted by $r$. As it is common in programming, J-COP’s functions contain local variables whose scopes are the same as their functions. Also parameters for functions are local variables which are denoted by $LVar$. Instance variables model internal states of classes. We let $LVar_C$ denote the set of instance variables of the class $C$. The symbols $s$ and $v$ denote typical elements of $LVar$ and $LVar_C$, respectively. Sequences of layers activation/deactivation constitute layer expressions ($LayerExpr$) with $le \in LayerExpr$. $FunNames$ and $LayerNames$ denote the sets of function and layer names, respectively, with $f \in FunNames$ and $l \in LayerNames$.

A class in J-COP includes function definitions and a group of layers containing function definitions. A function is defined via a parameter, a statement, and an expression. The expression models the returned value of the function. A set of classes and a main function (marks the start of program execution) are the components of a J-COP program.

The operational semantics of J-COP presented in this section is defined using state representations and a subtype relation on classes. If $C$ inherits $D$ by definition of $C$, then $C$ is a subclass of $D$ (denoted by $C \ll D$) and $D$ is a superclass of $C$. The reflexive transitive closure of $\ll$ is denoted by $\le$. States of the operational semantics are presented in Definition 1 where $\alpha$ denotes an infinite set of memory addresses with $\alpha \in \mathcal{A}$ and $\mathcal{Z}$ is the set of integers.

**Definition 1.** One has the following:

1. $\mathcal{D} = \mathcal{L} \cup \mathcal{A}$;
2. $Layer_C$ and $Fun_C$ denote the sets of function and layer names of $C$, respectively;
3. $Stacks = \{s \mid s : LVar \rightarrow p\mathcal{Z}\};$
4. $ObjCont = \{O_C | O_C : LVar_C \rightarrow \mathcal{Z}, C \in \mathcal{C}\};$
5. $Heaps = \{h \mid h : \mathcal{A} \rightarrow p\text{ObjCont}\};$
6. $States = \text{Stacks} \times \text{Heaps}.$

The set $\mathcal{D}$ includes model values. States of the operational semantics are pairs of a stack and a heap. A special variable (called this), to point at the object being executed, is included in the set of local variables. For each class $C$, we assume two functions $F_C$ and $L_C$. The map $F_C$ determines for every function $f \in Fun_C$ its components $(p_f, S_f, e_f)$, the parameter variable of the function, the function statement, and the
### Algorithm 1: The programming language J-COP

```
AExpr ::= n | o | e \cdot V | e_1 \cdot o \cdot e_2 | (C)e | this

Expr ::= with l | without l | v | (C)e | true | false

LayerExpr ::= le

Stat ::= S | S; S

Prog ::= main {S}

S ::= [e](s, h)
```

### Algorithm 2: Semantics of J-COP expressions.

```
[C(o)](s, h) = \begin{cases} 
  undefined, & \text{if } [e](s, h) \in \text{dom}(h), h([e](s, h)) = O_D, \text{ and } D \notin C; \\
  [e](s, h), & \text{otherwise.} 
\end{cases}

[e \cdot v](s, h) = \begin{cases} 
  O_D(v), & \text{if } h([e](s, h)) = O_D \text{ and } v \in \text{dom}(O_D); \\
  undefined, & \text{otherwise.} 
\end{cases}
```

The returned expression of the function. The map \( I_C \) determines for every layer \( l \in \text{Layer}_C \) the parts of the layer’s function \((f, p, S_f, e_f)\). We also assume a function \( \text{SerLay}(L, le, f) = L' = \{l_1, \ldots, l_m\} \) whose inputs are a set of layers, a layer expression, and a function name. \( \text{SerLay} \) adds (removes) layers activated (deactivated) by \( le \) to (from) \( L \). Then \( \text{SerLay} \) returns the set of layers in \( L \) containing definitions for \( f \).

The semantics of arithmetic and boolean expressions of J-COP is similar to the case of simple imperative language. The cases of \((C)e\) and \(e \cdot v\) are shown in Algorithm 2. Some comments on this algorithm are in order. The expression \( e \cdot v \) denotes the variable \( v \) of the class referenced by \( e \). In line with common OOP concepts, variables of \( C \) and that of its ancestors are included in domain of \( \text{IVar}_C \). Therefore the semantics of \( e \cdot v \) is defined only if \( v \) is a local variable of the class referenced by \( e \) or any of the class’s ancestors. The semantics of \( e \) (in \( e \cdot v \)) is supposed to be an address containing a class object. The semantics of \((C)e\) is undefined if \( e \) is the address of an object of a class \( D \) that is not a descendant of \( C \).

Algorithm 3 presents the semantics of J-COP’s statements. The judgments of the semantics have the form \( S \vdash L : (s, h) \rightarrow (s', h') \) meaning that executing \( S \) in the state \((s, h)\) results in the state \((s', h')\) provided that \( L \) is the set of the currently active layers. Some comments on the rules are as follows. The local variable \( v \) of the object pointed to by \( e_i \) is updated in the rule \((\cdot =_v e \cdot f)\). This amounts to modifying the function \( O_C \) in the image of \([e_i](s, h)\) under \( h \). With input \( e \), the rule \((\cdot =_e e_\cdot f)\) provides semantics for running the function \( f \) of the object pointed to by \( o_1 \). The semantics of \( le \) in \( o_1 \) \( := le \cdot o_2 \cdot f(e) \) is to activate/deactivate certain layers. The rule \((\cdot =_{o_1} e \cdot f)\) provides operational semantics for this statement. This rule does the call \( \text{SerLay}(L, le, f) \) to add (remove) layers activated (deactivated) via \( le \) to (from) \( L \). This call then returns the subset of \( L \) with definitions for \( f \). The rule then executes the bodies of layer functions sequentially. The execution number \( i \) builds on its previous execution by updating \( o_1 \) to the value \([e_{i-1}](s_i, h_i)\). The statement \( o := \text{super} \cdot f(e) \) running the function \( f \) of an ancestor of the current object is given semantics in the rule \((\text{sup}'\cdot)\). The rule assumes a function \( \text{super} \) that finds the ancestor of the class pointed to by \( o_2 \) such that this ancestor includes a definition for \( f \). Semantics for \( o_1 := \text{proceed} \cdot o_2 \cdot f(e) \) is given in the rule \((\text{pro}'\cdot)\). This statement runs all functions \( f \) in active layers of the object referenced by \( o_2 \).

### 3. Assertion Language and Logical System

Extended separation logic to cover context-oriented programs is presented in this section. The section introduces the assertion language followed by the logical system’s axioms and inference rules. Against the operational semantics presented in the previous section, this section also outlines a formal mathematical proof for the correctness of the logical system. Finally an example of a derivation for a partial correctness specification using the logical system is included in this section as well.

**3.1. Assertion Language**

Boolean expressions, classical connectives, first order quantifications, and assertions specific to separation logic are the components of separation logic assertions. The following version of assertions is used in this section.

(i) \( \text{emp} \) denotes an empty heap.

(ii) \( e \mapsto C \) denotes that the heap has a unique memory cell whose address is \( e \) and whose content is an instance of the class \( C \).
\[
O_C = h([e_1](s, h)) \quad O'_C = O_C \cup \{v \mapsto [e_2](s, h)\}
\]
\[
L \vdash e_1 \cdot v ::= e_2 : (s, h) \quad \text{if layers of } L \text{ are active and } S \text{ is executed in a state satisfying } P, \text{ then the final state will satisfy } Q. \text{ Definition 3 formalizes the }
\]

Algorithm 3: Inference rules of the operational semantics for J-COP constructs.

(iii) this \( \rightarrow C \) denotes a special case of the previous assertion where \( e = \text{this} \).

(iv) \( P \neq Q \) dubbed separating conjunction.

(v) \( P \neq \neg Q \) dubbed separating implication.

(vi) \( \forall e_1 \vdash P_1 \) is a general form of separating conjunction.

Algorithm 4 presents the specific definition of the language of assertions. A group of states is modeled by every assertion via a modeling relation \((s, h) \models P\). The semantics of this relation is that the state \((s, h)\) satisfies the assertion \( P \). Definition 2 formalizes the semantics of the modeling relation.

**Definition 2.** One has the following:

(i) \((s, h) \models \text{fine}(e_1, \ldots, e_n)\) \[\text{def}\] \(\text{for all } i, [e_i](s, h) \neq \text{abort}\);

(ii) \((s, h) \models \text{emp} \[\text{def}\] \(\text{dom}(h) = \emptyset\);

(iii) \((s, h) \models e \[\text{def}\] \(\text{dom}(h) = \{[e](s, h)\}\) and \(h([e](s, h)) = O_C\);

(iv) \((s, h) \models P_1 * P_2 \[\text{def}\] \exists h', h'' \text{ s.t. } h = h' * h'', (s, h') \models P_1, \text{ and } (s, h'') \models P_2,

(v) \((s, h) \models P_1 \rightarrow P_2 \[\text{def}\] \text{for all } h'((h' \neq h \text{ and } (s, h') \models P_1) \Rightarrow (s, h') \models P_2).\)

The fact that \(\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset\) is denoted by the expression \(h_1 \neq h_2\). The union of heaps is denoted by \(h_1 \cdot h_2\) and is defined only if \(h_1 \neq h_2\).

A judgment of the proposed logical system is of the form \(L \models [P]S\{Q\}\) where \(P\) and \(Q\) are the pre- and postconditions, respectively. The semantics of this judgment is that if layers of \(L\) are active and \(S\) is executed in a state satisfying \(P\), then the final state will satisfy \(Q\). Definition 3 formalizes the
soundness concept of specifications produced by the logical system. Axioms and inference rules of the logical system are introduced in Algorithm 5.

We fix the abortion notion for Definition 3. A statement $S$ is aborting at $(s, h)$ (denoted by $S : (s, h, L) \rightarrow \text{abort}$) if (a) a state $(s', h')$ such that $S : (s, h) \rightarrow (s', h')$ is not available and (b) $S$ is not stuck in an infinite loop.

**Definition 3.** A specification $L \vdash \{ P \}\{ Q \}$ is sound if, for each $(s, h)$ such that $(s, h) \models P$, the following is satisfied:

1. $(L \vdash S : (s, h) \rightarrow \text{abort})$,
2. if $L \vdash S : (s, h) \rightarrow (s', h')$, then $(s', h') \models Q$.

Some of the rules in Algorithm 5 are certainly noteworthy. In many rules like $(\models_{Q}^0)$ and $(\models_{O}^0)$, the expressions of relevant statements contribute to the preconditions of the rules. This has the advantage of ensuring that the meant memory addresses are indeed allocated and hence evaluations of expressions do not abort. The preconditions of $(\models_{Q}^0)$ and $(\models_{O}^0)$ also ensure that appropriate variables reference objects in heap. The side-conditions of $(\models_{Q}^0)$ and $(\models_{O}^0)$ consider that executing these statements involves executing certain function bodies. We let $\mod(S, L)$ denote the set of locations modified by $S$ provided that layers of $L$ are active. The symbol $f \nu(R)$ denotes the set of free variables of $R$. The rule (share) enables the composition of specifications having disjoint sets of activated layers if the preconditions of the specifications describe disjoint regions of the heap. The rule (frame) enables avoiding a frame of the heap, $R$, that does not affect the statement. This rule also ensures that $R$ is satisfied by the postcondition.

**Soundness.** The soundness of the logical system is guaranteed by the following theorem.

**Theorem 4.** Specifications $L \vdash \{ P \}\{ Q \}$ that resulted using the logical system of Algorithm 5 are sound (Definition 3).

**Proof.** Structure induction on axioms and inference rules of Algorithm 5 achieves the proof. In the following, basic cases are outlined.

(i) The case of the rule $(\models_{Q}^0)$: in this case

(a) $S = e_1 \cdot v \models e_2$,
(b) $P \models e_1 \rightarrow C \land \text{fine}(e_1, v) \land \text{fine}(e_2) \land Q[e_2/e_1, v]$.

Suppose that, for state $(s, h), (s, h) \models P$. Hence $O_C = h[e_1[s, h))$ and $e_2[s, h]) \neq \text{abort}$. Therefore $O_C = O_C[v \mapsto e_2[s, h)]$ is defined and so does $(s, h)[e_1[s, h)) \rightarrow O_C]$. Hence $(L \vdash S : (s, h) \rightarrow \text{abort})$. Now we suppose $L \vdash S : (s, h) \rightarrow (s', h')$. In this case $(s', h') \models Q$ because $(s, h) \models Q[e_2/e_1, v]$. This completes the proof for this case.

(ii) The case of the rule $(\models_{O}^0)$: in this case

(a) $S = o_1 := o_2 \cdot f(e)$,
(b) $F_C(f) = (p_f, S_f, e_f)$,
(c) $L \vdash \{ e[f/this, e/p_f] \} S_f \{ Q[e_f/o_1] \}$,
(d) $L \vdash S_f : (s[\text{this} \rightarrow s(o_2), p_f \rightarrow \text{fine}(e[s, h]), h) \rightarrow (s'', h'')$,
(e) $P \models o_2 \rightarrow C \land R \land \text{fine}(e, e_f)$.

Suppose that, for state $(s, h), (s, h) \models P$. This implies $h[e_1[s, h)] = O_C$. The condition fine($e, e_f$) implies $(s[\text{this} \rightarrow s(o_2), p_f \rightarrow \text{fine}(e[s, h]), h) \models R[o_f/this, e/p_f]$. By induction hypothesis $S_f$ does not abort at $(s[\text{this} \rightarrow s(o_2), p_f \rightarrow \text{fine}(e[s, h]), h])$ and therefore $o_1 := o_2 \cdot f(e)$ does not abort at $(s, h)$. Now we suppose that $o_1 := o_2 \cdot f(e) : (s, h) \rightarrow (s', h')$. By induction hypothesis $(s', h') \models Q[e_f/o_1]$ implying $(s', h') \models Q$. This completes the proof for this case.

(iii) The case of the rule $(\models_{\nu}^1)$: in this case

(a) $S = o_1 := le \cdot f(e)$,
(b) $\text{SerLay}(L, le, f) = L' = \{ l_1, \ldots, l_m \}$, for all $1 \le i \le m, L(c_i) = (f, p_i, S_i, e_i)$,
(c) $L' \vdash S_1 : (s[\text{this} \rightarrow s(o_2), p_i \rightarrow \text{fine}(e[s, h]), h) \rightarrow (s_2, h_2)$,
(d) for all $i > 1, L' \vdash S_i : (s_i[o_1 \mapsto e_1[s, h_i]), h_i) \rightarrow (s_{i+1}, h_{i+1})$,
(e) $L' \vdash \{ R[o_i/this, e/p_i] \} S_i \{ Q[i/e_1[o_i] \}$,
(f) for all $i > 1, L' \vdash \{ Q[e_i/o_1/this, e/p_i] \} S_i \{ Q[e_i/o_1] \}$,
(g) $P \models o_2 \rightarrow C \land R \land \text{fine}(e, e_1, \ldots, e_n)$.

Suppose that, for state $(s, h), (s, h) \models P$. This implies $h[e_1[s, h)] = O_C$. The condition fine($e_1, e_1, \ldots, e_n$) implies $(s[\text{this} \rightarrow s(o_2), p_i \rightarrow \text{fine}(e[s, h]), h) \models R[o_i/this, e/p_i]$. By induction hypothesis $S_j$, does not abort at $(s[\text{this} \rightarrow s(o_2), p_i \rightarrow \text{fine}(e[s, h]), h)]$. The symbol $\nu(R)$ denotes the set of free variables of $R$. The rule (share) enables the composition of specifications having disjoint sets of activated layers if the preconditions of the specifications describe disjoint regions of the heap. The rule (frame') enables avoiding a frame of the heap, $R$, that does not affect the statement. This rule also ensures that $R$ is satisfied by the postcondition.
Algorithm 5: The inference rules of the proposed logical system.

Now we suppose that $S_1 : (s, h) \rightarrow (s_2, h_2)$. By induction hypothesis $(s_2, h_2) \equiv Q[e_j, o_j]$ implying $(s_2, h_2) \equiv Q[e_j, o_j]$. Therefore by induction hypothesis $S_3$ does not abort at $(s_2, h_2) \equiv Q[e_j, o_j]$, this $\rightarrow (s_3, h_3)$. By induction hypothesis $(s_3, h_3) \equiv Q[e_j, o_j]$, this $\rightarrow (s_3, h_3)$. Hence a simple induction on $m$ completes the proof for this case.

(iv) The case of the rule (sup): in this case

(a) $s = o : = \text{super} \cdot f(e)$,
(b) $\text{super}(E,f) = D$,
(c) $F_0(f) = (p, S_j, e_f)$,
(d) $L \vdash s : (s,p_f \rightarrow [e][s, h], h) \rightarrow (s'', h'')$,
(e) $L \vdash R[e/p_f] S_f[Q[e/o]]$,
(f) $P \iff this \rightarrow C \land C \land E \land R \land \text{fine}(e,e_f)$. 

Now we suppose that $Q$ does not abort at $e$.

$L \vdash R[e/p_f] S_f[Q[e/o]]$.
Suppose that, for state \( (s, h) \), \( (s, h) \vDash P \). This implies \( h[\{\text{this}(s, h)\}] = O_C \). The condition \( \text{fine}(e, e_f) \) implies \( (s[p_f \mapsto \{e\}(s])_h, h) \) is defined. Therefore \( (s(\text{this} \mapsto s(\text{this}))_o, p_f \mapsto \{e\}(s], h), h) \vDash R[e/p_f]. \) By induction hypothesis \( S_f \) does not abort at \( (s[p_f \mapsto \{e\}(s], h), h) \) and therefore \( o := \text{super} \cdot f(e) \) does not abort at \( (s, h) \). Now we suppose that \( o := \text{super} \cdot f(e) : (s, h) \rightarrow (s', h'). \) By \((:=o \cdot f(e), (s', h') = (s')^{\circ \cdot o \cdot f} \cdot \{e\}(s', h'), h'). \) Also by induction hypothesis \( (s', h'') \vDash Q[e_f/o] \) implying \( (s', h'') \vDash Q. \) This completes the proof for this case.

(v) The case of the rule \((\text{pro}\,^i)\): it is similar to the case of \((\ast := \text{pro}\,^i, f)\).

4. Discussion

Related Work. The most related logic to our work is that of object-oriented programs. Specifically for Java, a huge literature on specification proof techniques for object-oriented languages exists. For example, for annotated Java programs in JML (Java modeling language), [8] presents operational semantics for a COP language, namely, c.l. For COP concepts as introduced in ContextJ [2] and ContextL, [9] introduces syntax-based semantics. Compared to the work mentioned above, a logical system that stops COP program from getting stuck is presented in the current paper. To model context-dependent behavior of COP programs, the proposed logical system is based on general calculi.

Future Work. Extending the language model of the current paper to allow executing candidate functions of \( \text{proceed} \) on priority bases is a direction for a future work. Extending the language to include layer inheritance and layer dependency is another direction for future work. Doing so allows one layer to assume the existence of another layer and allows expressing the assumption that two layers are not active at the same time.

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

References


