OESPA: A Theory of Programming that Support Software Engineering

Chongyi Yuan  
School Of Electronics Engineering  
And Computer Science  
Peking University  
Beijing, China  
lwyuan@pku.edu.cn

Wen Zhao, Sen Ma  
National Engineering Research Center for Software Engineering  
Peking University  
Beijing, China  
zhaowen@pku.edu.cn, masen@pku.edu.cn

Abstract

A new theory of programming is proposed. The theory consists of OE (Operation Expression), SP (Semantic Predicate) and A (Axiom), abbreviated as OESPA.

OE is for programming: its syntax is given by BNF formulas and its semantics is defined by axioms on these formulas. Similar to predicates in logic, SP is for describing properties of OE (i.e. programs) and for program property analysis. But SP is different from predicates, it directly relates the final values of variables upon termination of a given OE with initial values of these variables before the same OE. As such, it is feasible to prove or disprove whether a given SP is a property of a given OE by computation based on A (Axioms). SP calculus is proposed for program specification and specification analysis, that is missing in software engineering.

Key words-OE, SOE, SP, Axioms, SP Calculus, 3-step formalism, foundation of software engineering

1. INTRODUCTION

1.1 Software engineering need a foundation

Mathematics, physics, dynamics etc. provide a firm foundation for civil engineering. As such, a blueprint can be approved, prior to its construction, by computing expectable properties against user’s requirements.

Is there a way to compute program properties from given program text? Or, in other words, does software engineering have a foundation comparable to the foundation of civil engineering?

The answer is NO! Not yet for today. The fact is, software products rely on post-development testing for its approval. Formal methods do not help much in this regard. For example, model checking requires a formal model to be built from a given program in order to prove properties of it. The consequence is, properties proven by model checking are not necessarily properties of the program\cite{1,2}. They are just the properties of the formal model.

Software engineering needs a firm foundation so that properties of its products can be computed directly from the products.

1.2 An investigation: formal semantics is missing from programming languages in use

A programming language has a formal syntax for its compiler, but no formal semantics for property analysis. To be more precise, the BNF formulas that define a programming language have no concern about
formal semantics.

For example, the two appearances of \( x \) in “\( x := x + 1 \)” would be recognized by the compiler as a unique identifier, representing the same variable. But a single variable is a two-facet semantic object: it represents a memory location as well as a data stored in that location.

The way in which variables are used by programming languages has obscured this semantic difference. This is why a assignment like “\( x := x + 1 \)” has always been treated as a whole in the discipline of formal semantics, by saying that “the semantics of \( x := x + 1 \)” is “an operation”, “a function”, or “a denotation”. In Hoare logic\(^5\), the same semantics is given by assertions

\[
\forall a \in D : \{ x = a \} x := x + 1 \{ x = a + 1 \}
\]

where \( D \) is the domain of \( x \).

This observation explains why program properties can not be computed from program text. Formal semantics is missing in the BNF formulas that define programming languages in use.

### 1.3 Predicate and program semantics

Predicates are suitable for formal description of state properties of programs. But, state properties are not the only thing important to program semantics. An even more important aspect of program semantics is how two consecutive states are related with each other.

For example, let \( S \) be a program (or program segment) to exchange values of variables \( x \) and \( y \). As long as \( x \) and \( y \) hold valid data, \( S \) should do the exchange, regardless what exact values are there in \( x \) and \( y \). With the predicates as the means, this property has to be given indirectly by

\[
\forall a, b \in D : \{ x = a \land y = b \} S \{ x = b \land y = a \}
\]

where \( D \) is the domain of both \( x \) and \( y \).

A straightforward description of this property would be

\[
x' = y \land y' = x
\]

where \( x' \), \( y' \) represent respectively the final values of \( x \) and \( y \) upon termination of \( S \).[5]

It is clear at a glance that (2) is much simpler and more explicit than (1). The point is, how to relate (2) with a given program? Or, how to prove formally that (2) is a property of \( S \) when the text of \( S \) is given?

We have proposed a new way of programming in the last few years\(^6\), in which assignments are operations on physical object (one of the two facets of variables) and programs are expressions consisting of such operations and control operators. This new type of expressions is called Operation Expression, OE for short.

OE has a formal syntax as well as a formal semantics, and the former is given by BNF formulas while the latter is given by axioms based on BNF formulas.

Semantic axioms are given in terms of two read_operations. In addition to the conventional instant read_operation on variables, two new read_operations are introduced: to read a variable before or after an OE. What obtained by such a read operation is a mathematical expression, telling how a variable before or after the given OE is related to initial values. To read a variable before an OE is meaningful when that OE is a constituent portion of another OE.

Recall the above mentioned program \( S \). If \( S \) is given as an OE, then \( \underline{x}(S) \) and \( \underline{y}(S) \), where “\( \underline{\_} \)” is the read_after operator, tell respectively how \( x \) and \( y \) after \( S \) is related to variables before \( S \). So, in case

\[
\underline{x}(S) = y \land \underline{y}(S) = x \quad (3)
\]

is true, \( S \) does exchange the values of \( x \) and \( y \). Note that \( \underline{x}(S) \) and \( \underline{y}(S) \) are computable from \( S \) (i.e. OE) based on semantic axioms. Here \( S \) is considered a stand-alone program, not part of another OE.

If \( S \) is removed from (3), we get

\[
\underline{x} = y \land \underline{y} = x \quad (4)
\]
(4) is called a semantic predicate, SP for short, \( x \) and \( y \) are “functions” defined on OE. For any given OE, say \( P \), we can compute \( x(P) \) and \( y(P) \) based on semantic axioms on OE, and to check whether \( x(P) = y \land y(P) = x \) is true. (4) is a property of \( P \) if and only if \( x(P) = y \land y(P) = x \) is true. It is in this sense, we say that program properties can be computed.

Recall (2) given above, (4) is as direct as (2) and as formal as (2), but we know how to compute \( x \) and \( y \) for a given program while there is no formal connection between \( x', y' \) and a given program.

1.4 3-step formalism: how this paper is organized

A general understanding of formalism may be stated as “to express something in precise and rigorous mathematics”. But, this understanding does not bring much hint with it about how to do formalism. Here we proposed a 3-step approach as a guideline for practising formalism. The 3 steps are: abstraction, representation and analysis methods.

- **Abstraction**
  Physical objects should be abstracted as mathematical objects, based on the purpose of the formalism in question, by forgetting physical aspects that have on connection with the purpose.

- **Representation**
  This step focuses on how mathematical objects obtained by abstraction are related with each other. Such relations usually form a mathematical system like an equation system.

- **Analysis methods**
  To develop and/or to apply analysis methods for the mathematical system to solve problems raised by the purpose of the formalism.

Formalism has focused too much on the representation step so far, with the abstraction step more or less overlooked. The practice of defining programming languages is but one such example. The purpose of our study is to propose a theory of programming that may serve as the foundation of software engineering, and that is comparable to the foundation of civil engineering. To this end, we followed the 3-step approach in practice.

- **Abstraction**
  Programming languages in use have abstracted memory locations as mathematical variables. A mathematical variable allows a read operation applied on it: whenever it appears in a mathematical expression, the read operation is implied. This is an instant read (the returned value is always the value currently held by the variable) and needs no operator. To cope with the need of programming, assignments on the variables are introduced with “\( := \)” as the assign operator for most of programming languages. But, as an operator “\( := \)” is not well defined since its semantics is not formally given. That means, the abstraction is incomplete.

  We propose a complete abstraction of memory locations as a program variable in section 2 as the basis for BNF formulas that define the syntax of OE. Section 2 is about BNF formulas.

- **Representation**
  A mathematical expression relates mathematical variables with each other. An OE (Operation Expression) defined by BNF formulas given in section 2 relates program variables with each other. Thus, what an OE provides is a syntactic representation.

  All mathematical operators have well defined semantics so that every mathematical expression can be computed to yield a value. We defined semantics for OE operators with semantic axioms in section 3, so that every OE can be executed. Thus, semantic axioms provide a semantic representation of how program variables are related.

  This paper focuses on sequential operation expressions, SOE for short, while parallel operation expressions (POE) and reactive operation expressions (ROE) are left for separate discussion. Operators for
SOE include the write operator (i.e. assign operator), and control operator to form operation sequences, choices and loops

- Analysis methods

Section 4 is about theorems derived from semantic axioms, Section 5 discusses properties of SOE and Section 6 proposes a SP calculus for program specification and specification analysis. These 3 sections are about methods for SOE property analysis. Section 7 relates SP calculus with SOE specification, Section 8 points out how to derive SOE from specification, and section 9 talks about SOE and software engineering: why OESPA may serve as the foundation of software engineering. Section 10, the last section, plans our future work and expresses our thanks to those who have supported our research. References are given to end this paper.

2. BNF FORMULAS FOR OE SYNTAX

2.1 Where to start

We don’t start from scratch. Data types, scalar or structured, are well defined, syntactically and semantically, and shared by programming languages. OE assumes their definitions without redefining them. It is in this sense, OE is not a complete programming language.

Furthermore, we don’t make it explicit what data types are included in OE. It is left for implementation of OE to decide when it becomes mature enough for practical use. For structured data types, array is used as an example to show how to be included in OE. OE is open to everything: as long as it has a well-defined syntax and a well-defined semantics that fit in OE, it may be included, in a nature way, as a constituent of OE.

In the rest of this paper, the word “variable” will mean a scalar variable, an array element or an array as a whole, unless otherwise stated. The range of all arrays are assumed to be [0..N].

2.2 Preliminary

2.2.1 Operations on variables

Let V be the set of variables in question.

There are 3 operations on a variable, namely the write_operation, the read_after_operation and read_before_operation. The over_bar “-”, the under_bar “_”, and the curved under_bar “~” are respectively the operator as given below:

\[ - : V \times E_M \rightarrow E_0, \text{ i.e. for } x \in V \text{ and } e \in E_M, \]

\[ \overline{x}(e) \in E_0, \]

\[ _- : V \times E_0 \rightarrow E_M, \text{ i.e. for } x \in V \text{ and } p \in E_0, \]

\[ \underline{x}(p) \in E_M, \]

\[ ~ : V \times E_0 \rightarrow E_M, \text{ i.e. for } x \in V \text{ and } p \in E_0, \]

\[ \tilde{x}(p) \in E_M, \]

where \( E_M \) is the set of mathematical expression on \( V \) (i.e. every expression in \( E_M \) contains only variables in \( V \)), \( E_0 \) is the set of OE to be defined. These operators are binary; “×” represents Cartesian product.

Let \( x \in V \) and \( e \in E_M \), then \( \overline{x}(e) \in E_0 \), and \( \underline{x}(\overline{x}(e)) \in E_M \) and \( \tilde{x}(\overline{x}(e)) \in E_M \). For example, let be \( e \equiv x + y \), then we have, by intuition, \( \overline{x}(x + y) = x + y \), and \( \tilde{x}(\overline{x}(x + y)) = x \) when \( \overline{x}(x + y) \) is studied in isolation. Semantic axioms specify how these operators are related as suggested by above examples from intuition.

2.2.2 Variable functions and Semantic predicates

From the definitions of the read_after_operation “_-” and the read_before_operation “~”:

\[ _- : V \times E_0 \rightarrow E_M, \]

\[ ~ : V \times E_0 \rightarrow E_M, \]
We have, for every $v$ in $V$:

\[ v : E_0 \to E_M \]
and

\[ v : E_0 \to E_M \]

**Definition 1**

1. $v$ and $\bar{v}$ are called variable functions: $v$ is
   the final function and $\bar{v}$ is the initial function, $F\_function$ and $I\_function$ for short respectively.

2. $V = \{v | v \in V\}$, the set of all $F\_functions$, is called the $F\_set$,
   $\bar{V} = \{v | v \in V\}$, the set of all $I\_functions$, is called the $I\_set$.

3. For operation expression $p$,
   \[ \{v | v(p) \in V\} \]
   is called the $F\_set$ of $p$,
   \[ \{v | v(p) \in \bar{V}\} \]
   is called the $I\_set$ of $p$.

4. For operation expressions $p$ and $q$,
   \[ \forall v \in V : v(p) = v(q) \]
   if and only if $\forall v \in V : v(p) = v(q)$,
   \[ \forall v \in V : v(p) = v(q) \]
   if and only if $\forall v \in V : v(p) = v(q)$,
   \[ \forall v \in V : v(p) = v(q) \]
   if and only if $\forall v \in V : v(p) = v(q)$.

**Example 1**

Let $p = \bar{y}(x+1)$ and $q = \bar{x}(x+1)$ be operation expressions, where operator “$\bar{}$” takes $y$ and $x+1$,
respectively $x$ and $x+1$ as its first and second operands. Let $S = q; p$ where “;” is the sequential control operator shared by programming languages. We have, from intuition,

\[ \bar{y}(y(x+1)) = x+1 \text{ and } \bar{x}(x(x+1)) = x+1 \]

i.e. $\bar{y}(p) = x+1$ and $\bar{x}(q) = x+1$

In the context of $S$, we have

\[ \bar{x}(p) = \bar{x}(q) = x+1 \]

and

\[ \bar{y}(S) = \bar{y}(p) | (\bar{x}(p) = \bar{x}(q)) \]

\[ = \bar{x}(p) + 1 = (x+1) + 1 = x + 2 \]

where $\bar{y}(p) | (\bar{x}(p) = \bar{x}(q))$ indicates that the initial value of $x$ before $p$ is the final value of $x$ after $q$.

Note from this example, $x$ is the very initial value before $S$. $x+1$ is the final value after $q$ as well as the initial value before $p$. Thus, $x+2$ is the final value after $S$. The exact value of $x$ is irrelevant in above computation.

**Definition 2**

1. A predicate $R$ on $V \cup \bar{V}$ is called a semantic predicate if it contains at least one variable function from $V$;

2. A Boolean expression on $V \cup \bar{V}$ is called a semantic Boolean expression if it contains at least one variable function from $V$.

A semantic predicate is denoted by $R(V, \bar{V})$ and a semantic Boolean expression is denoted by $\bar{b}$. The domain of semantic predicates is $E_0$ since $V$ is the $F\_set$ defined on $E_0$.

**Example 2**

\[ \bar{R}_1(V, \bar{V}) \equiv \bar{x} = y \land \bar{y} = x \]
\[ \bar{R}_2(V, \bar{V}) \equiv \bar{x} \leq \bar{y} \]

are semantic predicates, $\bar{x} \leq \bar{y}$, $\bar{x} + \bar{x} = \bar{y}$ are semantic Boolean expressions.

$\bar{R} \equiv \bar{x} + \bar{y} + \bar{z} = 0$ is not a semantic predicate since it contains no variable functions. $\bar{x} < \bar{y}$ is a Boolean expression, but not a semantic one since it has nothing to do with $V$.

**2.2.3 Conditional expressions**

Let $e_1, e_2, ..., e_n$ be expressions of the same type and $b_1, b_2, ..., b_n$ be Boolean expressions, where $n$ is an integer with $n > 1$. 

Definition 3

1. \( e_i \) if \( b_i \sim e_i \) if \( b_1 \sim \ldots \sim e_n \) if \( b_n \) is called a conditional expression, if
\[
\forall i, j : 1 \leq i, j \leq n : (i \neq j \land b_i \land b_j \rightarrow e_i = e_j)
\]

2. Let \( l \) be the value of the above defined conditional expression, then
\[
\forall i : 1 \leq i \leq n : (b_i \rightarrow l = e_i)
\]

This definition ensures that every conditional expression has a unique value if every \( e_i, i = 1, 2, \ldots, n \), has a value at a given state.

A conditional expression is also called a table by some formal methods like PVS, where
\[
\forall i, j : 1 \leq i, j \leq n : (i \neq j \rightarrow \neg (b_i \land b_j))
\]
is required. We have this requirement relaxed a bit here.

Conditional expressions are mainly used in semantic axioms.

2.3 BNF Formulas

In the BNF formulas given below, \( \nu \) represents either a scalar variable like \( x \), or an array element like \( A[i] \), or an array as a whole like \( A \). All arrays are assumed to share \([0..N]\) as their index range. In addition, type match is always assumed whenever it is required.

Conventions we follow are:
\( i, j \) : variables for array index,
\( n, N \) : integer constants, \( n > 0 \), \( N > 1 \),
\( E \) : array expressions,
\( b \) : semantic Boolean expressions,
\( b, b_1, b_2, \ldots, b_n, b(i) \) : Boolean expressions,
\( e, e_1, e_2, \ldots, e_n, e(i) \) : mathematical expressions

BNF formulas
\[
(B1) = (B1.0) + (B1.1) + (B1.2) + (B1.3)

(B1.0) \ \text{simple}_\text{term} ::= <\text{simple_term}> | <\text{vs_term}> | \text{sequential}_\text{O_expression}
\]

\( (B1.1) \ \text{s}_\text{term} ::= \overline{x}(e) | \overline{A}[i](e) | \overline{A}(e) \)

\( (B1.2) \ \text{vs}_\text{term} ::= e | \overline{v}(e)^b \)

\( (B1.3) \ \text{es}_\text{term} ::= \overline{v}(e_i)^b \overline{v}(e_j)^b \ldots \overline{v}(e_n)^b \)

Remarks
- A simple_term applies write_operation on a single variable.
- A vs_term is a variation of s_term: it is either the special term \( e \), or a conditional term \( \overline{v}(e)^b \).
- An es_term is an extension of a s_term: a multiple choices for the same variable. It is required by semantic axioms to be given that
\[
i \neq j \land b_i \land b_j \rightarrow e_i = e_j
\]

\( (B2) = (B2.0) + (B2.1) + (B2.2) + (B2.3) \)

\( (B2.0) \ \text{multi}_\text{term} ::= <\text{m}_\text{term}> | <\text{vm}_\text{term}> | <\text{am}_\text{term}> \)

\( (B2.1) \ \text{m}_\text{term} ::= \text{simple}_\text{term} \cdot \text{simple}_\text{term} \)

\( (B2.2) \ \text{vm}_\text{term} ::= \text{m}_\text{term}^b \)

\( (B2.3) \ \text{am}_\text{term} ::= \overline{A}[i : b(i)](e(i)) \)

Remarks
- A multi_term applies write_operation on more than one variable.
- A vm_term is a conditional m_term.
- An am_term is a conditional m_term for elements of the same array.
- The operator “ \( \cdot \) ” between two simple_terms is often omitted, just like the operator for multiplication.

\( (B3) = (B3.0) + (B3.1) + (B3.2) + (B3.3) \)

\( (B3.0) \ \text{sequential}_\text{O_expression} ::= <\text{soe}> | <\text{v}_\text{soe}> | <\text{r}_\text{soe}> \)

\( (B3.1) \ \text{soe} ::= <\text{term}> | <\text{term}>;\text{soe} \)

\text{term} ::= \text{simple}_\text{term} | \text{multi}_\text{term} \)
(B3.2) \( \text{v}_\text{soe} ::= \text{soe}^b \)

(B3.3) \( \text{r}_\text{soe} ::= \text{soe}^n \mid \text{soe}^b \)

Remarks
- ";" is the sequential control operator
- \( \text{r}_\text{soe} \) is for loops
- SOE is used as an abbreviation of Sequential_O_expressions, and sometimes it is also used to denote the set of all Sequential_O_expressions when no confusion possible. Thus, SOE is a subset of \( E_0 \).

3. SEMANTIC AXIOMS ON SOE

Let \( p, q \) be SOE, i.e. \( p, q \in \text{SOE} \), and let \( V_p = \{ v | \overline{v}(e) \in p \} \) be the subset of \( V \), consisting of variables on which \( p \) applies write_operation.

(A1) = (A1.1) + (A1.2) + (A1.3) , Foundation
(A1.1) \( p = q \Leftrightarrow \forall v \in V : v(p) = v(q) \)
(A1.2) \( v \notin V_p \Rightarrow v(p) = v \)
(A1.3) \( v \in V \Rightarrow v(p) = v \) if \( p \) is in isolation.

(A2) = (A2.1) + (A2.2) + (A2.3) + (A2.4)

simple_term
(A2.1) \( v(\overline{v}(e)) = e \)
(A2.2) \( V_e = \Phi \)
(A2.3) \( \overline{v}(e)^b = \overline{v}(e) \) if \( b \sim e \) if \( \sim b \)
(A2.4) \( \overline{v}(e_1)^b \overline{v}(e_2)^b \ldots \overline{v}(e_n)^b = \overline{v}(e_1) \) if \( b_1 \sim \overline{v}(e_1) \) if \( b_2 \sim \ldots \sim \overline{v}(e_n) \) if \( b_n \)
\( \sim e \) if \( \sim (b_1 \lor b_2 \lor \ldots \lor b_n) \)

Distributive law on conditional operation expressions:
\( v(p \text{ if } b_1 \sim p_1 \text{ if } b_2 \sim \ldots \sim p_n \text{ if } b_n) \)
= \( v(p_1) \text{ if } b_1 \sim v(p_2) \text{ if } b_2 \sim \ldots \sim v(p_n) \text{ if } b_n \)
(A3) = (A3.1) + (A3.2) + (A3.3) multi_term

(A3.1) \( v(p \cdot q) = v(p) \text{ if } v \in V_p \sim v(q) \text{ if } v \in V_q \) for \( v \in V_{p \cdot q} \)

(A3.2) \( p \cdot q)^b = p^b \cdot q^b \)

(A3.3) \( \overline{A}[i : b(i)](e(i)) = \overline{A}[0](e(0))^{e(0)} \cdot \overline{A}[1](e(1))^{e(1)} \ldots \overline{A}[N](e(N))^{e(N)} \)

(A4) = (A4.1) + (A4.2) + (A4.3) + (A4.4) + (A4.5) + (A4.6)
\( p, q, p_1, p_2, p_3, p_4 \in \text{SOE} \)
(A4.1) \( v(p \cdot q) = (v(q) | V(q) = V_p(p)) \) for all \( v \in V \)
(A4.2) \( p_1; p_2; p_3 = (p_1; p_2; p_3) \)
(A4.3) \( p^b = p \text{ if } b \sim e \text{ if } \sim b \)
(A4.4) \( p^n = p \text{ if } n = 1 \sim p^{n-1}; p \text{ if } n > 1 \)
(A4.5) \( p^b = e \text{ if } b(e) \sim p^b \text{ if } \sim b(e) \)
\( \land \exists n : (b(p^n) \land \forall l: 0 < l < n: \sim b(p^l)) \sim p; p^b \text{ otherwise} \)
(A4.6) \( v(p; q) = v(p) \) for all \( v \in V \)

Remarks
- \( v(q) | V(q) = V_p(p) \) indicates that \( q \) takes \( u(p) \) as the initial value of \( u \) for every \( u \in V \).
- \( p; p^b \) in (A4.5) represents an endless repetition of \( p \). For SOE at the mean time, this is a semantic error. We keep it as it is since we will extend SOE to include reactive actions.

4. THEOREMS DERIVED FROM SEMANTIC AXIOMS

Theorem 1

All axioms given with conditional expressions, namely (A2.3),(A2.4),(A3.1),(A4.3) and (A4.4),(A4.5) are deterministic.

Proof.

Conclusions in this theorem are ensured by (A1.1). All those that do not fulfill this theorem, are not
semantically valid, and should be excluded from SOE.

**Definition 4**

A sequential operation expression \( p \), syntactically satisfies all BNF formulas given in subsection 2.3, is a valid SOE if and only if it is semantically deterministic, i.e. it leads to no ambiguity no matter which semantic axiom is applied.

In what follows SOE refers to only valid sequential operation expressions.

**Theorem 2**

\( (p_1; p_2; p_3 = p_1; (p_2; p_3) \) for \( p_1, p_2, p_3 \in SOE \)

**Proof.**

1. \( V((p_1; p_2); p_3) = V(p_1)|V(p_2); p_3 = V(p_1); p_2 = V(p_1) \) (A4.1)
2. \( V(p_1)|V(p_2); p_3 = V(p_2); p_1 = V(p_1) \) (A4.6)
3. \( V(p_2; p_2); p_3 = V(p_2; p_3); p_1 = V(p_1) \) (A4.1)
4. \( (p_1; p_2); p_3 = p_1; (p_2; p_3) \) (A1.1)

**Theorem 3**

\( (p^h)^b = p^h \cdot b \) (A4.3)

**Corollary 1** for terms

1. \( e \cdot p = p \cdot e = p \) (A3.1)
2. \( p \cdot p = p \) (A3.1)
3. \( p \cdot q = q \cdot p \) (A3.1)
4. \( (p_1 \cdot p_2) \cdot p_3 = p_1 \cdot (p_2 \cdot p_3) \) (A3.1)

5. **PROPERTIES OF SOE**

\( R(V, V) \) is a property of \( p \), \( p \in SOE \), if \( R(V(p), V) \), i.e. \( p \) makes \( R(V, V) \) true.

**Example 3**

1. \( x = y \) is a property of \( (x(y))^{+} \) since \( x((x(y))^{+}) = y \) by (A2.1).

2. \( x \leq y \) is a property of \( (x(y))^{+} \) since, by (A2.3),

\[
\chi((x(y))^{+}) = y \quad \text{if} \quad x > y \quad \wedge \quad x \leq y.
\]

3. \( x = y \wedge y = x \) is a property of

\[
p = \chi(x + y; \chi(x - y); \chi(x - y))
\]

**Detailed Computations for 2 and 3:**

2. \( p = \chi(x(y)^{+}) \),

\[
p = \chi(x(y)) \quad \text{if} \quad x > y \quad \wedge \quad x \leq y \quad \text{(A2.3)}
\]

3. \( p = \chi(x(y)^{+}) \),

\[
p = \chi(x(y)^{+}) \quad \text{if} \quad x > y \quad \wedge \quad y \leq x \quad \text{(A2.1),(A2.2),(A1.2)}
\]

Thus,

\[
\chi(p) = \chi((x(y))^{+}) \quad \text{if} \quad x > y \quad \wedge \quad y \leq x
\]

4. \( \chi(x + y; \chi(x - y); \chi(x - y)) \)

\[
\chi(x + y) = y \quad \wedge \quad \chi(x - y) = y \quad \text{(A2.1),(A2.2),(A1.2)}
\]

so,

\[
\chi((x(y))) = x + y, \chi((x(y))) = y
\]

Thus,

\[
\chi(p) = \chi((x(y))) \quad \text{if} \quad x = x + y, y = y
\]

Thus,

\[
\chi(p) = \chi((x(y))) \quad \text{if} \quad x = x + y, y = x
\]

Thus,

\[
\chi(p) = \chi((x(y))) \quad \text{if} \quad x = x + y, y = x
\]
Definition 6

Semantic predicate \( R(V, V) \) is bipartite if it takes the form \( R(V, V) = R(V) \) op \( R(V) \), where \( op \in \{=, <, \leq, \geq, \rightarrow, \leftarrow\} \) and \( R(V) \), \( R(V) \) share the same structure with variables from \( V \) and \( V \) respectively.

Note that \( op \) is a binary transitive operator, “\( \rightarrow \)” is the “implies” operator and “\( \leftarrow \)” is the “implied by” operator.

Example 4

\[
R_1(V, V) = x + y + z = x + y + z, \quad R_2(V, V) = x < x
\]

and \( R_1(V, V) = x + y > 0 \leftarrow x + y > 0 \) are bipartite.

\( R_1(V, V) = x < y \leftarrow x < z \) is not bipartite.

Definition 7

If bipartite \( R(V, V) = R(V) \) op \( R(V) \) is a property of \( p \), then \( R(V) \) is said to be, as shown by the table below, stable, inheritable, traceable, respectively, constant, decreasing, not increasing, increasing and not decreasing. Such \( R(V) \) will be denoted by, respectively, stable( \( R(V) \) ), inherit( \( R(V) \) ), traceable( \( R(V) \) ), const( \( R(V) \) ), dec( \( R(V) \) ), not_inc( \( R(V) \) ), inc( \( R(V) \) ) and not_dec( \( R(V) \) ).

| op       | \( R(V) \in E_s \) | \( R(V) \notin E_s \) |
|----------|-----------------|-----------------|
| =        | stable          | constant        |
| \( \leftarrow \) | inheritable    | decreasing      |
| \( \rightarrow \) | traceable      | \( \leq \) not increasing |
| \( E_s \): Set of logic expression | \( > \) increasing | \( \geq \) not decreasing |

Table 1 \( R(V) \) op \( R(V) \) is a property of \( p \)

Theorem 4

If \( R(V, V) = R(V) \) op \( R(V) \) is a property of \( p \), then

1. stable( \( R(V) \) ) \( \land \) \( R(V) \rightarrow R(V(p)) \)
2. inherit( \( R(V) \) ) \( \land \) \( R(V) \rightarrow R(V(p)) \)
3. traceable( \( R(V) \) ) \( \land \) \( \neg R(V) \rightarrow \neg R(V(p)) \)

The proof is straightforward from Definition 7.

Definition 8

A semantic predicate is a state predicate if it contains only variable functions. A state predicate \( R \) is denoted by \( R(V) \).

Example 5

\( R_1(V) = x + y + z = 100 \), \( R_2(V) = \leq y \land \leq z \) are state predicates.

Definition 9

1. \( R(V) \) is an invariant of \( p \), if \( R(V(c)) \land R(V(p)) \).
2. \( R(V) \) is a conditional property of \( p \) if there exists a state predicate \( Q(V) \) such that \( Q(V(c)) \rightarrow R(V(p)) \). \( Q(V) \) is called the condition of \( (R(V), p) \).

Example 6

1. Let be \( p = \overline{x}(x+1)\overline{y}(y-1) \), \( R_1(V) = x + y + z = 100 \) and \( R_2(V) = \overline{x} + \overline{y} + \overline{z} \), then \( R_1(V) \) is an invariant of \( p \) if \( x + y + z = 100 \) initially, and \( R_2(V) \) is a constant of \( p \) regardless \( x + y + z = 100 \) or not. Note \( R_2(V) \) is itself not a predicate.
2. \( Q(V) = \overline{z} \leq \bar{y} \land \overline{z} \leq \bar{z} \) is the condition for

\[
((\overline{y(z)}\overline{z(y)})^{\overline{yz}}, \overline{z} \leq \bar{y} \land \bar{z} \leq \bar{z}).
\]

Definition 10

State predicate \( R(V) \) is irrelevant to \( p \), \( p \in SOE \),
if and only if \( V_R \cap V_p = \emptyset \) where \( V_R = \{ v \mid v \in R(V) \} \) and \( V_p = \{ v \mid v(e) \in p \} \).

**Example 7**

\( R(V) \equiv x \leq y \) is irrelevant to \( p = z(x+y) \), since
\( V_R = \{ x, y \} \) and \( V_p = \{ z \} \).

**Corollary 2**

If \( R(V) \) is irrelevant to \( p \), then \( \forall v \in V_q \rightarrow \forall(p) = \forall \) and thus \( R(V(p)) = R(V) \), where \( R(V) \) is obtained for \( R(V) \) by replacing every variable function \( v \) with the variable \( v \).

**Example 8**

For \( R(V) \equiv x \leq y \) and \( p = z(x+y) \), we have
\( \forall(p) = x \) and \( \forall(p) = y \). Thus, \( R(V(p)) \equiv x \leq y \), i.e. \( R(V(p)) \equiv x \leq y \).

**Theorem 5**

1. If \( R(V) \) is a conditional property of \( p \) with \( Q(V) \) as the condition of \( (R(V), p) \), \( R(V) \) is irrelevant to \( q \) and \( Q(V) \) is a property of \( q \), then \( R(V) \) is a property of \( q; p \).

2. If \( R(V) \) is a property of \( q \) and it is irrelevant to \( p \), then \( R(V) \) is a property of \( q; p \).

3. If \( R(V) \) is an invariant of both \( q \) and \( p \), then \( R(V) \) is a property of \( q; p \), or to be more precise, \( R(V) \) is also an invariant of \( q; p \).

4. For \( v \in V_p \), if \( \forall(p) \) contains no variable in \( V_q \) then \( \forall(q; p) = \forall(p) \).

**Proof.** 1, 2 and 3 are direct consequences of relevant definitions.

For conclusion 4,

\( \forall(q; p) = \forall(p) \) by (A4.1)

For variable \( x \) that appears in \( \forall(p) \), we know \( x \in V_q \) thus \( \forall(q) = \forall \), i.e. \( \forall(p) = \forall \). Since \( x \) is an arbitrary variable appearing in \( \forall(p) \), we know
\( \forall(q(p)|V(p)) = \forall(q) = \forall(p) \)
so, \( \forall(q; p) = \forall(p) \).

**Theorem 6**

1. If \( R(V) \) is an invariant of \( p \), then \( R(V) \) is also an invariant of \( p^n \) for \( n > 1 \).

2. If there exists \( n, n \geq 0 \), such that \( p^n = p^n \) by (A4.5), then \( b \) is a property of \( p^n \), and every invariant of \( p \) is also invariant of \( p^n \).

**Proof.** Direct consequence of Theorem 5.

**Example 9**

Let \( p = i(i+1)\overline{m}(m+A[i]) \), then \( R(V) = m = \sum_{i=0}^{i-1} A[i] \) is an invariant of \( p^n \) for \( n, 0 < n \leq N \), if initially
\( i = 1 \land m = A[0] \). It’s easy to compute that \( i(p^n) = N+1 \) and \( m(p^n) = \sum_{i=0}^{N} A[i] \), i.e. \( q; p^n \) is a program to compute the sum of array elements for array \( A \), where
\( q = i(1)\overline{m}(A[0]) \).

Note that array \( A \) is not changed by \( p \), i.e. for any index \( I \), \( A[I](p) = A[I] \) (or \( A[p][I] = A[I] \)). As such \( R(V) \) can be written as \( R(V, V) = m = \sum_{i=0}^{i-1} A[i] \), and say
\( R(V, V) \) is an invariant of \( p \) and \( p^n \). The difference between \( R(V) \) and \( R(V, V) \) in this example is that the former is a state predicate and the latter is not. From now on we relax the definition of an invariant to
include semantic predicates like $R(V, V)$.

**Example 10**

Let $p, q$ and $R(V, V)$ as given in example 9. By theorem 6 we know $R(V, V)$ is also an invariant of $p^k$ where $b = i = N + 1$, and $m(p^k) = \sum_{i=0}^{N} A[i]$, as long as $i = 1 \land m = A[0]$ initially, i.e. $i(p^k) = 1 \land m(p^k) = A[0]$.

So, $q; p^k$ is also a program to compute the sum of all elements of array $A$.

### 6. SP CALCULUS

SP calculus is developed mainly for program specification and specification analysis based on semantic axiom (A 1.2), (A 4.1) and (A 4.6).

A noticeable fact is, the write_operation (i.e. assignment) is the only operation that changes a program state. Thus for a complete program specification, there must be an expression $e_v$ for an arbitrary variable $v$ such that $v = e_v$ is derivable from it, where $e_v$ contains only variables in $V$, no variable function from $V$. SP calculus tries to capture the process of deriving $v = e_v$ for all $v$ in $V$, from a given specification. A specification is called a SP formula in this calculus.

#### 6.1 Semantic Predicates (SP)

The concept of semantic predicates was defined earlier by Definition 2. But, axiom (A 4.1) indicates that $I_functions$ $v$ for $v \in V$ are needed in the process of reasoning. Thus, we will extend the definition to serve the need of SP calculus.

**Definition 11**

A predicate $R$ defined on $V \cup V \cup V$ is a semantic predicate, SP for short, if it contains no contradiction.

Apparently, all semantic predicates satisfying Definition 2 remain to be semantic predicates by Definition 11. All semantic predicates takes SOE as their domain since $V, V$ are respectively the $F_set$ and $I_set$ defined on SOE.

**Example 11**

$x < y$, $x < x \land x < x$ are contradictions.

We write $v \in R$ for semantic predicate $R$ if $v$ appears in $R$. Similarly, we write $v \in R$ and $v \in R$.

Let be $V_k = \{v \mid v \in R\}$, $V_s = \{v \mid v \in R\}$ and $V_r = \{v \mid v \in R\}$.

**Definition 12**

$R$ is said to be $V_free$, $V_free$ and $V_free$ respectively, if $V_k = \phi$, $V_s = \phi$ or $V_r = \phi$.

$R$ is said to be $V_complete$, $V_complete$ and $V_complete$ respectively, if $V_k = V$, $V_s = V$ and $V_r = V$.

#### 6.2 SP formulas and computation Rules

SP is the simplest formula.

**Definition 13**

$SP_formula ::= SP | SP; SP_formula$

In what follows, a SP_formula will be called a formula.

For $R$, $R$ is not $V_complete$, we have, by Axiom (A 1.2),

**Computation Rule 1: Extension**
\[ y \in R \rightarrow R \equiv (R \land \underline{y} = \nu) \,.
\]

We now develop computation Rules for structured formulas, i.e. for \((R_1; R_2)\) and \((R; F)\) where \(R_1, R_2\) are SP while \(F\) is itself a formula.

For given \(R_1, R_2\) is a SP without saying, let \(R(V_1, V_2)\) denote what is obtained by replacing every appearance of \(y\) in \(R\) with \(y\) for all \(y \in V_1\);
\(R(V, V_2)\) denote what is obtained by replacing every appearance of \(v\) in \(R\) with \(v\) for all \(v \in V\);
\(R(V', V)\) denote what is obtained by replacing every appearance of \(v\) in \(R\) with \(v\) for all \(v \in V\);
\(R(V_1, V_2)\) denote what is obtained by replacing every appearance of \(v\) in \(R\) with \(v\) for all \(v \in V\);

Example 12
For \(R_1 \equiv x = x^2\), we have \(R(V_1, V_2) = x = x^2\), \(R(V, V_2) = x = x^2\) and \(R(V_1, V) = x = x^2\).

Theorem 7
\(R(V_1, V_2)\) and \(R(V, V_2)\) are both SP if \(R\) is \(V\) _free_.

Proof
\(V_1 = \phi\) ensures that both \(R(V_1, V_2)\) and \(R(V, V_2)\) are contradiction free.

Let \(R_1, R_2\) be both \(V\) _free_ SP.

Definition 14
1. \(R_1\) is said to be independent of \(R_2\) if \(V_1 \cap V_2 = \phi\).
2. If \(R_1\) is not independent of \(R_2\), then \(R'\) is said to be the independent portion of \(R_1\) with respect to \(R_2\) if \(R'\) is the maximum portion of \(R_1\) such that \(V_1 \cap V_2 = \phi\). This fact is denoted by \(R'_1 = ind(R_1, R_2)\).

Example 13
\(R_1 \equiv y = \sin x \land y = \cos x\), \(R_2 \equiv y = \sqrt{x}\), then \(ind(R_1, R_2) \equiv y = \cos x\).

Definition 15
Let \(R_1, R_2\) be \(V\) _free_. \(R_1\) is said to be complete with respect to \(R_2\) if \(V_1 \supseteq V_2\) and for every \(y \in V_1\), an equation \(y = e_1\) is derivable from \(R(V_1, V_2)\) such that \(e_1\) contains only variable from \(V\) and \(R_1(V_1, V_2) \mid R(V_1, V_2)\) represents a substitution in \(R(V_1, V_2)\) that every appearance of \(y\) is replaced by \(e_1\) for every \(y \in V_1\).

Example 14
Let be \(R_1 \equiv \{x, y, z\} = \{x, y, z\}\), \(R_2 \equiv \{x, y, z\} = \{x, y, z\}\), we have:
\(V_{R_1} = V_{R_2} = \{x, y, z\}\)
and \(R_1(V_1, V_2) = \{x, y, z\}\).

Computation Rule 2: Resolution
If \(R_1\) is complete with respect to \(R_2\), then \(R_1 \land (R_2(\overline{V_1}, \overline{V_2}) \mid R_1(V_1, V_2))\) where \(R'_1 = ind(R_1, R_2)\).

Semantic axiom(A 4.1) is the foundation of this rule.

An obvious conclusion is:

Theorem 8
If \(R_1\) is complete with respect to \(R_2\), then \(R'_1 \land (R_2(\overline{V_1}, \overline{V_2}) \mid R_1(V_1, V_2))\) is \(V\) _free_.


Example 15

Let be \( R_1 \equiv x = x + y \wedge y = y \) and \( R_2 \equiv y = x - y \), we have \( \text{ind}(R_1, R_2) \equiv x + y, \quad R_1(V, V') \equiv x + y \wedge y = y, \) \( R_2(V, V') \equiv y = x - y \) and \( R_1(V, V') \mid R_2(V, V') \)

\( \equiv y = (x + y) - y \equiv y = x \). By Computation Rule 2:

\( R_1 \wedge R_2 \rightarrow \text{ind}(R_1, R_2) \wedge (R_2(V, V') \mid R_1(V, V')) \),

i.e. \( x = x + y \wedge y = y; \quad y = x - y \rightarrow x = x + y \wedge y = x \).

In this way we can prove \( x = x + y \wedge y = x \);

\( x = x - y \rightarrow x = x + y \wedge y = x \).

This example concludes that \( x = x + y \wedge y = y; \quad y = x - y \);

\( x = x - y \rightarrow x = x + y \wedge y = x \),

i.e. \( x = x + y \wedge y = y; \quad y = x - y; x = x - y \) is a specification of a program that exchanges the values of \( x \) and \( y \).

But, to derive this conclusion formally, we need the Computation Rule below:

Computation Rule 3: Association

\( R_1; R_2; R_3 \equiv (R_1; R_2); R_3 \)

\( R_1; R_2; R_3 \equiv R_1; (R_2; R_3) \)

Theorem 2 in Section 4 is the foundation of this rule.

People would prefer to take \( R' \equiv x = x + y \) instead of \( R_1 \equiv x = x + y \wedge y = y \) as the first SP in the specification in Example 15. But, \( R_1 \) is not complete with respect to \( R_1 \). To prove \( R_1; R_2; R_3 \) is as good as \( R_1; R_2; R_3 \) as a specification of exchanging values of \( x \) and \( y \), we need the next computation rule:

Computation Rule 4: substitution

\( R_1 \equiv R_1 \wedge R_2 \equiv R_2 \rightarrow R_1; R_2 \equiv R_1; R_2 \)

By Computation Rule 1 we know \( R_1 \equiv R_1 \), by Computation Rule 4 we know \( R_1; R_2; R_3 \equiv R_1; R_2; R_3 \), so \( R_1; R_2; R_3 \equiv R_1; R_2; R_3 \). That is, \( R_1; R_2; R_3 \) is as good as \( R_1; R_2; R_3 \).

Theorem 9

\( R_1; R_2; R_3; R_4 \equiv (R_1; R_2); (R_3; R_4) \)

Proof

\( R_1; R_2; R_3 \equiv (R_1; R_2); R_3 \) \quad (Rule 3)

\( R_1; R_2; R_4 \equiv (R_1; R_2); R_4 \) \quad (Rule 4)

\( (R_1; R_2); R_3 \equiv (R_1; R_2); (R_3; R_4) \) \quad (Rule 3)

Thus, \( R_1; R_2; R_3; R_4 \equiv (R_1; R_2); (R_3; R_4) \).

Note that Rule 2 reduces \( R; R \) to a single SP, Rule 3 allows to apply Rule 2 more than once to reduce any formula to a single \( V \)-free SP. Rule 4 let us do the reduction in parallel. It is important to notice that all actual reductions always reduces two SP’s to a single \( V \)-free one, and the two SP’s to be reduced must satisfy the requirement that “\( R_1 \) is complete with respect to \( R_2 \)”.

To ensure such completeness, we define the concept of reducible formulas.

Definition 16

A SP formula \( R_1; R_2; \ldots; R_n \), \( n \geq 2 \), is reducible if \( R_1; R_2 \) can be reduced to a single SP and \( (R_1; R_2); R_3; \ldots; R_n \) is reducible.

With this concept, we have

Theorem 10

1. All reducible formulas can always be reduced to a single \( V \)-free SP.
2. Reducible formulas comply with associative law.

The second part of this theorem can be proved in the way Theorem 9 is proved. The first part is a direct conclusion from Definition 16.

Example 16

\( f \equiv x \leq y; \quad x \leq z; \quad y \leq z \) may be taken, by many, as a
specification of sorting $x, y, z$ into ascending order. But this formula is not complete. The OE program

$$\bar{x}(1)y(2); \bar{x}(3)y(4); y(1)z(2)$$

satisfies $f$. But this OE ends up at $x = 3 \land y = 1 \land z = 2$ and $x, y, z$ have nothing to do with their initial values and they are not in ascending order.

Example 18 in next section will suggest a correct specification for sorting $x, y, z$ into ascending order.

Example 17

For $R_1 = x = \sin x$, $R_2 = x = x^3$ and $R_3 = \sqrt{x}$, we have

$$R_1; R_2 = x = (\sin x)^3, \quad (R_1; R_2); R_3 = x = \sqrt{(\sin x)^3}$$

and

$$R_2; R_3 = x = \sqrt{x^3}, \quad (R_2; R_3); R_1 = x = \sqrt{(\sin x)^3}.$$

7. SP CALCULUS AND SPECIFICATION ANALYSIS

Examples are used in this section to illustrate how to do specification analysis with SP calculus.

Examples 15-17 given in section 6 have in fact shown the function of SP calculus in specification analysis as long as all those formulas are considered as program specifications.

Specification process includes a phase for refinements. $x = y \land y = x$ can be refined to $x = x + y; y = x - y; x = x - y$. The former indicates “WHAT” is wanted and the latter tells “HOW” to do it.

Example 18

Let be $R = \{x, y, z\} = \{x, y, z\}$, $R_1 = x \leq y \land z = z$, $R_2 = x \leq z \land y = y$ and $R_3 = y \leq z \land x = x$, we first prove that $f' = R_1 \land R_2 \land R_3$ is a complete SP formula.

Since $R_1(V_x V_y) \land R_2(V_x V_y) = x \leq y \land z = x \land \{x, y, z\} = \{x, y, z\}$

and $R_2(V_x V_y) \land R(V_y V_x) = x \leq z \land y = y \land \{x, y, z\} = \{x, y, z\}$.

by Rule 2:

$$f' \rightarrow x \leq z \land y = y \land \{x, y, z\} = \{x, y, z\}$$

and

$$x \leq z \land y = y \land \{x, y, z\} = \{x, y, z\} \land z \leq y.$$

where $x \leq y$ is obtained in the following way:

$$x \leq z \land y = y \land \{x, y, z\} = \{x, y, z\}$$

and

$$x \leq z \land \{x, z\} \rightarrow x \leq y \land \{x, z\} = \{x, y, z\}$$

by Rule 2:

$$f' \rightarrow x \leq z \land y = y \land \{x, y, z\} = \{x, y, z\}$$

$$x \leq z \land x = x \land \{x, y, z\} = \{x, y, z\} \land x \leq y.$$

That is, $f'$ is complete, and

$$f' \rightarrow x \leq y \land z = z \land R.$$

Similarly we can prove that

$$R_1 \land R; R_2 \land R; R_3 \land R \rightarrow x \leq y \land z \leq z \land R.$$

This is a proper specification of a sorting program.

Example 19

$$f(n) = i = 1 \land f = 1; (i = i + 1 \land f = f \cdot i)$$

is a specification for the program to compute $n!$ for $n \geq 1$, since

$$f(n) \rightarrow f = n! \land i = n + 1.$$

If we take the Boolean constant True as a SP, and for $R$, $R$ is an arbitrary SP, define $R^0 = \text{True}$, then $f(n)$ can be relaxed to allow $n = 0$. We have, by Rule
\[ f(0) \to i = 1 \land f = 1 \quad \text{since} \quad i = 1 \land f = 1 \] is independent of True. i.e. \( f(0) \to f = 0! \)

The prove of \( f(n) \to f = n! \land i = n + 1 \) requires the application of mathematical induction.

**Example 20**

A specification in terms of SP calculus will be proposed for the 8 Queen Problem.

The first step towards a formalization of this problem is to do abstraction: to find a mathematical representation of physical objects involved, namely the chess board and the queens on the board.

To keep data structure as simple as possible, we use an array as the abstraction of the board, i.e. array \( A[0..7] \) whose elements take values from \{0,1,2,3,4,5,6,7\}, and \( A[i] = j \) denotes that a queen is placed at \((i,j)\) on the board.

The second step is to represent in terms of \( A[0..7] \) what the problem is required: to place 8 queens on board such that they stay in peace, i.e. no one is in a position to kill anyone else. For two queens \( A[i] = l \) and \( A[j] = k \), they can not kill each other if and only if

\[
l \neq k \land l \neq k \land i \neq j \land l \neq k \neq j - i
\]

i.e. \( A[i] \neq A[j] \land A[i] - A[j] \neq i - j \land A[i] - A[j] \neq j - i \).

We denote this with peace \((A,i,j)\), i.e.

\[
\text{peace}(A,i,j) = A[i] \neq A[j] \land A[i] - A[j] \neq i - j \land A[i] - A[j] \neq j - i
\]

The third step is to propose a solution.

There are two ways to understand this problem: to find one solution for the 8 queens, or to find all solutions for the 8 queens. The auxiliary variables proposed next aim at all solutions.

Auxiliary variables:

- \( i, l \): integer with \(-1 \leq i \leq 7\) and \( 0 \leq l \leq 8 \).
- \( P \): array\([0..7]\) of integer whose element set is \{0,1,2,3,4,5,6,7\}.
- \( Q \): constant array\([0..7]\) with all elements being \( 0 \).

intended meaning:

\[
i < 9 \land i \geq 0 \to A[i] = l \quad \text{is a candidate for the} \quad (i+1)\text{th} \quad \text{queen.}
\]
\[
l = 8 \to A[0..l-1] \quad \text{is not part of next solution.}
\]
\[
i = -1 \to \quad \text{no more solution}
\]
\[
i \geq 0 \to A[0..l-1] \quad \text{may be part of next solution.}
\]

We need two predicates:

\[
R(A,i) = \forall j, k : 0 \leq j, k < i :: \text{peace}(A,j,k) \quad \text{for} \quad i, 0 \leq i \leq 8
\]
\[
H(A,i,l) = (A[i] = l \to R(A,i+1))
\]

We have

\[
R(A,8) \to (A \text{ is a solution})
\]

Specification for finding one solution:

Let be

\[
R_0 = i = 1 \land l = 0 \land P = Q \land A = Q
\]
\[
R_1 = l = 8 \lor (l < 8 \land H(A,l,l))
\]
\[
R_2 = l < 8 \to A[l] = l
\]
\[
R_3 = (i = 7 \land l < 8 \to P = A)
\]
\[
\oplus(i < 7 \land l < 8 \to i = i + 1 \land l = 0)
\]
\[
\oplus(i > 0 \land l = 8 \to i = i - 1 \land l = A[i-1] + 1)
\]
\[
\oplus(l = 8 \land i = 0 \to i = -1)
\]

The solution is

\[
R_0; (R_1; R_2; R_3)^{P \lor \neg i = -1}
\]

This solution includes a formula \((R_1; R_2; R_3)^{P \lor \neg i = -1}\), and this formula can not be derived from the definition of SP_formulas. We may extend the definition to include this, but we didn’t. We have treated \( R^* \) as an abbreviation of \( R; R; \cdots; R \) for simplicity. \( R^x \), where \( \overline{x} \) is a SP, can also be viewed as an abbreviation of \( R; R; \cdots; R \), with a flexible times of repetition of \( R \). If this repetition does not lead to \( \overline{x} \), \( R^x \) is not a right formula. For one solution, \( P \not\equiv Q \lor \neg i = -1 \) can be reached, since when \( A \) is considered as an octal number, \( A < A \) is derivable from \((R_1; R_2; R_3)^n\) for some \( n \) unless \( P \not\equiv Q \lor \neg i = -1 \).
\( P \neq Q \) denotes the fact that \( P \) is a solution, and \( i = -1 \) means that there is no solution.

Specification for finding all solutions:

Let be

\[
R_i \equiv (i = 7 \land l < 8 \Rightarrow P = A \land l = A[6] + 1 \land i = 6) \\
\land (i > 0 \land l = 8 \Rightarrow i = i - 1 \land l = A[i - 1] + 1) \\
\land (i = 8 \land i = 0 \Rightarrow i = -1)
\]

The solution is

\[
R_0; (R_1; R_2; R_3)^{i-1}
\]

and each time \( P \) is changed, a new solution is found.

Note that the difference between a specification and a program is that a program consists of details of operations while a specification contains only the result of these operations. For example, \( R_i \) demands \( l \) to be either \( l = 8 \), or \( l < 8 \land H(A,i,l) \), it does not tell how to achieve the goal.

8. HOW TO DEVELOP SOE FROM GIVEN SPECIFICATIONS

The SOE for \( x = y \land \neg y = x \) is \( \overline{x(y)\overline{y(x)}} \)

The SOE for \( x = x + y; \overline{y = x - y}; \overline{x = x - y} \) is

\( \overline{x(x + y); \overline{y(x - y); \overline{x(x - y)}}} \)

The SOE for \( x \leq y \land R; x \leq z \land R; y \leq z \land R \), where

\[
R \equiv \{x, y, z\} = \{x, y, z\}, \text{ is}
\]

\[
(x(y)\overline{y(x)})^{\forall x, y}; (x(z)\overline{z(x)})^{\forall x, z}; (y(z)\overline{z(y)})^{\forall z}
\]

The SOE for \( i = 1 \land f = 1; (i = i + 1 \land f = f + i)^n \) is

\[
\tilde{i}(1)\tilde{f}(1); (\tilde{i}(i + 1)\tilde{f}(f + i))^n.
\]

The SOE for the 8_Queen problem is:

SOE for \( R_0 : P_i = \tilde{i}(i)(0)\overline{P(Q)}\overline{A(Q)} \)

SOE for \( R_1 : P_i = \tilde{i}(i + 1)\overline{H(A,i,l)} \)

SOE for \( R_2 : P_i = \overline{A[i](i)}^{\forall i} \)

SOE for \( R_3 : P_i = \overline{P(A)^{=\forall i}} \)

\[
(\tilde{i}(i - 1)\overline{A[i - 1] + 1})^{\forall i}; \tilde{i}(i - 1)^{=\forall i};\tilde{i}(i - 1)^{\forall i}
\]

SOE for \( R_4: \)

\[
(\overline{P(A)^{=\forall i}})\overline{i}(i)(6)\overline{A[i][6] + 1})^{=\forall i};\tilde{i}(i - 1)^{=\forall i};\tilde{i}(i - 1)^{=\forall i}
\]

SOE for one solution \( P_0; (P_1; P_2; P_3)^{F=\forall i} \)

SOE for all solutions \( P_0; (P_1; P_2; P_3)^{F=\forall i} \)

Apparently, it is not difficult to develop a SOE from a given SP formula since for \( R (R \) is a SP), \( V_R \) is the set of variables that require a simultaneous write operation.

The analysis of the specifications for the 8_Queen problem has been omitted here to save space. In fact, this problem is among those that are used for demonstration of prototype tools developed in our lab for SOE analysis.

The prototype tools are in fact a SOE property verifier capable to do semi automatic SOE analysis (human interfere is needed when, say, mathematical induction is required). A SOE may be restructured as a tree, called analysis tree, whose intermediate nodes are control operators for operation sequences, choices and loops, while leaf nodes are SOE Terms. We go no further here on these tools since this paper focuses on OESPA, the theory itself.

Advantages of SOE and SP calculus include:

- SOE is syntactically formal as well as semantically formal. As such, properties can be
computed from SOE texts.

- SP, i.e. semantic predicates, and SP formulas provide a firm foundation for program specification, since direct descriptions of how consecutive program states are related become feasible. SP calculus has made specification analysis practical.

- Specification in SP formulas let users focus on WHAT instead of HOW, since operation details are left for later consideration.

To end this section, we have the following definition:

**Definition 17**

A SOE developed from a given reducible SP formula is correct if and only if the given formula is a property of it.

Note that, in our theory OESPA, a SOE is a program, a reducible SP formula is a program specification, and a reducible SP formula can always be reduced to a single SP.

9. SOE AND SOFTWARE ENGINEERING

The software industry depends heavily on software engineering. But software engineering relies on post-development testing for the quality of its products.

It is impossible, at least for the time being, to have SOE as the main means of programming. But SOE and SP_calculus provide a formal foundation for software engineering. With the help of SOE and SP calculus, programmers may better understand what programming is about.

We wish to have a programming language that has a BNF definition similar to programming languages in wide use, but potentially equivalent to the BNF definition of SOE. In this way, software qualities may be guaranteed with testing being of second importance only.

We have applied our theory to develop tools for the analysis of C pointers. It has turned out to be very helpful. It leads us to a systematical way of thinking, and makes it possible to develop a C-pointer analysis prototype tool within half a year.

10. FUTURE WORK

This paper focused on SOE, sequential operation expression. We will have POE and ROE proposed in the future: POE is an abbreviation for Parallel Operation Expression while ROE, an abbreviation for Reactive Operation Expression.

Parallelism is the only means to promote software efficiency and to make full use of super computers. Reactive systems are important for computer applications in the area of management and control.

ACKNOWLEDGMENTS

We are grateful to the National Engineering Research Center For Software Engineering of Peking University, Beijing Beida Software Engineering Development Company for their continuous support. Thanks to everyone in our lab among them Mr. Zhang Hang has helped us with typesetting. The authors are sponsored by the National Basic Research Program of China (973) No.2009CB320706.

REFERENCES

[1] Emerson Clarke, Formal Methods: State of the Art and Future Directions [R]. VeriSign Inc, Mountain View, CA, 2004.

[2] E.M. Clarke, O. Grumberg, and D.A. Peled, Model Checking[C]. MIT Press, 1999.

[3] Hoare C A R. Communicating sequential processes[M]. Prentice Hall, 1985.

[4] Hoare C A R. An axiomatic basis for computer programming[J]. Communications of the ACM, 1969, 12(10): 576-580.

[5] C.A.R.Hoare, HeJifeng. Unifying Theories of Programming[M]. Prentice Hall International, 1998.

[6] Chongyi Yuan, Assignment : Operation on a
physical object[J]. Journal of Frontiers of Computer Science and Technology, 2008, 2(5): 487-499.

[7] Chongyi Yuan, Huang Yu, Zhao Wen. 2009. Program: Expressions of Operations On Physical Objects. Journal of Frontiers of Computer Science and Technology. 2009 Vol.3(2):144-153.

[8] Chongyi Yuan, Zhao Wen, Gao Xin, Huang Yu. 2010. Definitions and Specifications of O-expression properties. Journal of Frontiers of Computer Science and Technology. 2010 Vol.4(1):20-28.

[9] Chongyi Yuan, Wen Zhao, Yu Huang. Operation Expression: A Way to Verified Software. The 2010 International Conference on Foundations of Computer Science (FCS2010, WORLDCOMP’10), Las Vegas Nevada, USA, 2010, 91-96.

[10] Dijkstra E W. Letters to the editor: go to statement considered harmful[J]. Communications of the ACM, 1968, 11(3): 147-148.