An Overview of Formal Specification Languages and their Adequacy for Formalizing the Definition of Function Points

Project FP-Formal
Technical report 1
version 1.1
February 18, 1999

Marc Frappier

Département de mathématiques et d'informatique
Faculté des sciences
Université de Sherbrooke

Abstract

Function point is the most widely used functional size measure in the software industry. It serves several purposes like software cost estimation, productivity management, quality management, benchmarking, outsourcing management, and several others. The goal of project FP-Formal, a research project funded by Bell Canada, is to formalize the function point measure in order to automate its calculation and to reduce measurement variance and costs. In this report, we analyse a set of formal specification languages and evaluate their adequacy for specifying industrial information systems and for formalizing the function point measure.

Résumé

La mesure de taille fonctionnelle la plus répandue de l'industrie du logiciel est le point de fonction. On l'utilise pour diverses activités de gestion telles l'estimation du coût d'un projet, la gestion de la productivité, de la qualité, de l'impartition et la définition de bancs d'essais. Le projet FP-Formal, financé par Bell Canada, a pour objectif de formaliser la mesure des points de fonction. Cette formalisation permettra d'automatiser le comptage des points de fonction, ce qui devrait réduire les coûts et les erreurs de comptage. Dans ce rapport, nous analysons un ensemble de langages de spécification formels et nous évaluons leur adéquation pour la spécification de systèmes d'information et la formalisation des points de fonction.

Contents

1 Introduction

The use of size measures for functional user requirements has rapidly spread in the software industry over the last two decades. Function point is probably the best known of these measures [3, 36]. Function points are used for critical aspects of software management such as estimating software effort, calculating productivity and defect density, and managing outsourcing contracts [2, 25, 46, 53].

Function point has several weaknesses as a method for evaluating functional size. It lacks a formal model: the counting rules are given in plain natural language and are subject to interpretation by experts. This problem introduces some variances on function point count depending on the rater. Studies noted a difference varying from 11 % to 30 % [25, 41, 44, 45][1], which is significant if function points are used for managing outsourcing contracts. Moreover, function point calculation is mostly a manual process which is quite expensive. Typically, an expert can count 57 function points per hour. The Bell Canada software portfolio is estimated at 325 000 function points, which means that it would take around three person-years to precisely measure this portfolio.

Recognizing the importance of these issues, the objective of the FP-Formal project is to formalize the IFPUG definition of the function point measure. A formal foundation for function point should increase its objectivity and reduce measurement variance between counters. It should also provide a basis for automating its calculation from a formal requirements specification, thereby reducing measurement cost and variance.

A first step to achieve this objective is to select a formal requirements specification language. In this report, we review a set of formal specifications languages and evaluate their adequacy for specifying industrial information systems and formalizing the function point measure. We warn the reader that such an evaluation is bound to be partial, because the array of languages to study is too vast to provide a detailed analysis. Nevertheless, this evaluation is useful and necessary, because it allows us to make a more rigorous selection.

We characterize information systems as follows. They typically use several large data structures which are modified and queried by transactions submitted by several users in concurrency. The difficulty of information systems mainly resides in complex relationships between data structures, in complex calculation involving several data structures, in processing large volume of data, and in preserving the integrity of the data structures through concurrent updates by several users. Information systems typically have little hard real-time constraints or inter-process communication.

In Sect. 2, we present a brief overview of the function point measure and discuss the issues related to its formalization. Sect. 3 provides a classification of formal specifications languages, offering the reader an overview of the domain. In Sect. 4, we evaluate each class of languages, describing strengths and weaknesses for function point formalization and industrial information system specification. This evaluation is summarized in Sect 4.7. We conclude with some remarks in Sect 5.

[1]These figures hold when the count boundary is well-defined. Larger variations are observed when boundaries are not clearly delineated

2 Overview of Function Points

Function points were proposed by Albrecht in 1983 [3]. Its user community has constantly grown since then, and it is now organized into a user group (IFPUG - International Function Point User Group) which proposes international standards for measurement and analysis [36]. Another version of function points, Mark II Function Point Analysis, has been proposed by Symons [74], but it is mostly used in the United Kingdom. In this report, we focus on the IFPUG definition of function points.

The following equation describes how function points, denoted by FP, are counted.

$$FP = UFP * VAF$$

Variable UFP denotes the number of unadjusted function points and variable VAF denotes the value adjustment factor. Variable UFP is determined using two main categories of components: data components (e.g., files, database tables, etc) and function components (e.g., updates, inquiries, reports). Data components are further divided in two categories: external interface files (EIF) and internal logical files (ILF). Function components are divided into three categories: external inputs (EI), external outputs (EO), and external inquiries (EQ). To compute UFP, each component c is assigned a weight w_c according to its category (EIF, ILF, EI, EO, or EQ) and other parameters: for data components, these parameters are the number of data element types (DET) and the number of record element types (RET); for function components, they are the number of DET and the number of EIF and ILF referenced. The UFP is simply the sum of the component weights.

$$UFP = \sum_c w_c$$

The Counting Practice Manual of IFPUG [36] provides rules for identifying, categorizing and weighing components. There are 14 rules for data components and 41 rules for function components. It is these rules that we want to formalize.

Variable VAF (Value Adjustment Factor) is determined by evaluating 14 factors on a scale of 0 to 5. The following equation describes the calculation of VAF from each factor f_i

$$VAF = 0.65 + 0.01 * \sum_{i=1}^{14} f_i$$

There also are rules for evaluating each factor, but they are not easily formalizable. Moreover, the effort required to evaluate VAF only represents a small percentage of the total effort for function point count. Hence, we will not invest effort for their formalization.

2.1 Formalization of Function Points

We define a formalized rule as a mathematical formula with a precise semantics. The problem of implementing a formal rule with a computer algorithm is a separate issue. Some formalization are easy to implement efficiently whereas others are more difficult to implement or are too algorithmically complex to be used effectively for automatic function point count. We will not restrict ourselves to the former, as our primary goal is to provide a precise and clear definition of counting rules. If some rules are too difficult to mechanize, then we may either simplify them or require human expertise for their evaluation.

It is clear that some rules of the Counting Practice Manual cannot be formalized. In [63], the authors argue that five steps require human judgement. We hope that we can reduce this number using a formal specification language, because the formal syntax and semantics allow more precision in the definition of rules. For instance, the issue of *uniqueness* in data components and function components seems to be amenable to formalization if we select a model-based specification language like Z [73].

A formal rule should only depend on the requirements specification document. We assume no input from a human expert that would clarify part of the specification, or supplement the specification with annotations to cater for specification language limitations.

3 A Classification of Formal Specification Languages

Research in formal specification languages started in the early sixties. Numerous languages resulted from this research endeavor. Our goal here is not to recapitulate all of them, but more to focus on the ones which retained the attention of the research community over the years, and had successful trials in some industrial applications.

We distinguish the following classes of specification languages.

State Machine Languages A state machine specification has a set of states, a set of inputs, a set of outputs, a transition function, an output function and an initial state. A machine starts in its initial state. When an input is received from the environment, the transition function determines the next state, and the output function determines the output returned to the environment. Examples of state-based languages are Moore machines [61], Mealy Machines [55], statecharts [30], Z [73], VDM [42], B [1], OCL [67], Cleanroom [57, 59], SDL [22], ROOM [72].

Trace Languages A specification is defined as a set of traces (also called input-output sequences). A trace describes a sequence of interactions between the system and its environment. An interaction is either the reception of an input, the production of an output, or both (synchronization on a value). Traces abstract from state transition; they only focus on observable interactions. Examples of trace-based languages are Cleanroom [57, 59], the trace-assertion method [77], the inductive method [58] and the entity black-box notation [24].

Algebraic Languages A specification is given as a set of functions, whose signatures are defined using sorts, and a set of conditional equations. A subset of the functions, the generators, defines the normal forms, i.e., the set of values (also called terms) defining the state space of the specification. Examples of algebraic languages are CLEAR [12], Larch [28], OBJ [27], ACT ONE [21], CASL [13] and PLUSS [26].

Process Algebra Languages A specification is a process which may communicate with its environment through gates. The behavior of a process is given by an expression defining an ordering of actions (also called events, interactions) occurring at gates. Expressions are constructed using operators like sequential composition, choice, parallel composition with synchronization, interleave, and nondeterministic choice. Process algebras were designed for specifying distributed, concurrent systems. Examples of process algebra languages are CCS [60], CSP [33], LOTOS [7] and ACP [6].

Modal Logics Modal logics denotes a class of logics using modalities describing the modes of truth of formulas. Temporal logic is a subclass of modal logics which deals with the evolution of truth values of a formula over time. Example of modalities are always, which says that a formula is always true, or eventually, which says that a formula will eventually become true. Modal logics are useful for specifying properties of systems, especially distributed, concurrent, reactive systems. They may be used in conjunction with state machine languages or process algebra languages to defines properties about a specification and prove them. Examples of modal logics are Manna and Pnueli's temporal logic of reactive and concurrent systems [51], CTL [15], and TLA [48].

Object-Oriented Notations These notations emerged from object-oriented programming languages like Simula and Smalltalk. The word "object" bears so many different meanings in the software literature that it is difficult to propose a widely agreed upon definition. In short, an object has a state and an interface which describes the operations (methods) that can modify its state. The attributes of an object, which describes its state, and the implementation of operations are encapsulated, so that a user of an object only sees what is exported (published) in the interface. A class is a collection of objects. Objects of a class have the same state structure and the same operations. Sharing of properties between classes may be described using inheritance. UML (Unified Modeling Language) [65] is the result of a joint effort from several organisations to propose a standard notation for object-oriented models. It is inspired from work by several authors like Booch [8], Jacobson [39], Rumbaugh [70] and others. A formal definition of the UML syntax and semantics [66] is defined using OCL (Object Constraint Language) [67].

Wide-Spectrum Notations Most specification notations are typically suitable for a specific class of problems (e.g., sequential systems, distributed systems, concurrent systems, real-time systems) or for a specific phase of the software life cycle (specification, design, implementation). Consequently, software engineers may have to use several notations to develop a set of related systems. It is not always easy to couple different notations, because of paradigm or semantic clashes. To circumvent these problems, wide-spectrum notations covering several specification paradigms and unified under a common semantic framework were developed. The most notable examples are RAISE [64], COLD-1 [14] and SPECTRUM [10].

As an example, consider the RAISE notation. It unifies features of several specification languages (VDM, CSP and ACT-ONE). Another example is COLD-1, which covers all phases of the software life cycle, and includes specification paradigm like state machine specification, algebraic specification, inductive definitions, algorithmic definitions in functional as well as imperative style. Moreover, it offers facilities for the modular structuring of specifications. COLD stands for "Common Object-oriented Language for Design". It is used within Philips in an important project in consumer electronics, and in the software and information technology center of Philips Research Laboratories.

In the next section, we provide a detailed evaluation of each language class. Because wide-spectrum notations consist of a mixture of several languages from several classes, they are not specifically covered in the next section.

4 Detailed Evaluation of Languages Classes

An evaluation should analyze and compare the following:

- specification language characteristics;
- measures of specification language "experiments" (actual language uses for industrial-size systems).

Unfortunately, very few reports in the literature describe the industrial usage of any formal specification language for *information systems*. Therefore, the emphasis of our evaluation is on language characteristics. For space considerations, we restrict our analysis to salient features of each class.

4.1 State Machine Languages

4.1.1 Language Characteristics

In basic state machine notations (e.g., Moore machines [61] or Mealy Machines [55]), the input set, the output set and the transition function are defined by enumeration (i.e., set extension). In more elaborate notations, these sets are defined by set comprehension using first-order logic, set theory and other basic data types like relations and sequences; the transitions are defined using operations which map an input and the current state to an output and a next state. These notations are often called *model-based* notations.

For the sake of simplicity, let us use the general, abstract model of Cleanroom [57, 59] to represent state machines. Most of the state-machine notations (or a large useful portion of them) can be mapped to this general model. The description of this mapping involves too many details, so it is far beyond the scope of this document. We will use this general model throughout the rest of this document to compare languages from different classes.

In Cleanroom, a state machine is a tuple (X, Y, S, R, s_0) where X denotes the input set, Y denotes the output set, S denotes the state space, s_0 denotes the initial state and R denote the transition relation. Relation R satisfies the following constraint:

$$R \subseteq (X \times S) \ \times \ (Y \times S) \ .$$

A transition is triggered by the reception of an input from the environment. Given an input x and a current state s, the transition relation R provides the set of possible next states and outputs delivered by the system when the processing of x is completed. Note that we use a transition *relation* instead of a transition *function* to cater for non-deterministic specifications. We may denote the set of next states by $(x, s).R$:

$$(x, s).R \triangleq \{(y, s') \in Y \times S \mid ((x, s), (y, s')) \in R\} \ .$$

The essence of languages like B, Z, VDM, and OCL is to provide syntactic constructs to define X, Y, S, R and s_0.

Usually, R is defined as a union of relations R_i:

$$R \triangleq R_1 \cup \ldots \cup R_m \ .$$

These relations R_i are often called *operations*. A simple analogy with information systems is to consider that operations R_i are specifications of modules interacting with the users.

The state space S is typically defined as the Cartesian product of sets:

$$S \triangleq S_1 \times \ldots \times S_n \ .$$

We will refer to sets S_i as *dimensions* of the state space. Pursuing our analogy with information systems, the reader may consider that a dimension S_i is an abstract specification of a table in a relational database.

Sets X and Y are often implicitly defined using the signatures of operations R_i. For the sake of simplicity, we may define the signature of an operation R_i as an expression of the form

$$(X_i \times (S_{j_1} \times \ldots \times S_{j_p})) \ \times \ (Y_i \times (S_{k_1} \times \ldots \times S_{k_q})) \ .$$

This expression simply states that operation R_i takes an input from set X_i and delivers an output from set Y_i. The operation reads state information from dimensions S_{j_1} to S_{j_p} of the state space S and modifies dimensions S_{k_1} to S_{k_q}; the other dimensions are preserved by the execution of the operation.

In turn, sets X_i, Y_i, and S_i are typically defined using Cartesian products of sets. We call these sets *attributes*.

$$X_i \triangleq A_1 \times \ldots \times A_{m_i}$$

$$Y_i \triangleq A_1 \times \ldots \times A_{n_i}$$

$$S_i \triangleq A_1 \times \ldots \times A_{p_i}$$

In our information systems analogy, attributes of input sets or output sets correspond to fields in a form or a report; attributes of a state space dimension correspond to columns in a table.

Sets X and Y are implicitly defined as the union of input sets and output sets, respectively, appearing in operation signatures.

$$X \triangleq \bigcup_{i=1}^{m} X_i \qquad Y \triangleq \bigcup_{i=1}^{n} Y_i$$

4.1.2 Adequacy for the Formalization of Function Points

There is a natural mapping between state machine languages and function points. An operation R_i corresponds to an *elementary process.* Consequently, the *boundary* of a function point count could be defined as a set of operations. Given that we consider each R_i as an elementary process, each R_i is a function component.

Operation signatures may be used to classify function components. An external input is an operation whose signature contains a non-empty set of modified dimensions S_{k_j}. An external output is an operation whose signature contains a non *trivial* output set Y_i. Dually, an external inquiry is an operation whose signature contains a *trivial* output set Y_i. The meaning of "trivial" could be defined by syntactic heuristics on R_i, depending on the specific language used for the specification. Informally, a trivial output is derived by performing a straightforward copy from state dimensions $S_{j_1}, \ldots, S_{j_p}$ to Y_i.

A data component is a dimension S_i of the state space. It is included in the count if there exists an operation signature referring to it. It is classified as an external interface file if there does not exist an operation signature where it appears as a modified dimension; otherwise, it is classified as an internal logical file.

It seems reasonable to conjecture at this point that component weights could also be formalized using operation signatures and dimension definitions. DET are required to determine component weights. Attributes A_i appearing in the definition of input sets, output sets and dimensions correspond to DET.

Function components are weighed according to the number of data components appearing in the signature and the number of DET appearing in the definition of an input set or in a modified dimension.

From this preliminary analysis, it is also clear that some parts of the IFPUG function point counting manual can not be formalized using state machine languages, because they are subjective (i.e., they depend on human judgement). For instance, the identification of data components, record types and data element types *according to the user point of view* is subjective.

4.1.3 Adequacy for Specifying Information Systems

Basic state languages like Mealy and Moore's finite state machines are not appropriate for writing realistic, complete information system specifications. They require an enumeration of the state space and of state transitions which quickly becomes unmanageable, even for the simplest information system.

State machine languages using set theory and set comprehension are more powerful. They have the following strengths for specifying information systems.

1. The state of an information system is typically large and composed of several data structures. Languages based on set theory can easily model these complex data structures.

2. Operation preconditions in information systems are typically large and involve several data structures. Preconditions can be stated in a straightforward manner.

3. The next state in a transition can be described by computing modification to the current state.

4. Invariant properties (safety properties) of the state can be specified in a straightforward manner.

5. They are closer in style to programming languages.

State machine languages have the following weaknesses.

1. The specification of complex ordering constraints on transactions (i.e., similar to telephony systems or telecommunication protocols) is difficult, because the history of transactions is encrypted into state dimensions.

2. The dynamic of the state space is difficult to grasp. To understand the behavior of an operation, one must understand how the state dimensions used by the operation evolve; therefore, one must also look at every operation that modifies the state dimensions.

4.2 Trace-Based Languages

4.2.1 Language Characteristics

We may describe trace-based languages using the concepts introduced for state machine languages. Let A^+ denote the set of non empty finite sequences constructed using elements of set A. A trace specification is a triple (X, Y, R) such that R is a subset of $X^+ \times Y^+$. The signature of an operation Op_i is given by an expression of the form

$$(X_i \times Y_i)$$

It is easy to understand what a trace specification represents by looking at a state machine specification. Let $s \lhd R \rhd s'$ be an abbreviation for $(s, s') \in R$. We may compute the trace specification $T = (X, Y, R)$ of a state machine $M = (X, Y, S, R', s_0)$ by stating that for any sequence of transitions

$$\begin{array}{l} (x_1, s_0) \lhd R' \rhd (y_1, s_1) \\ (x_2, s_1) \lhd R' \rhd (y_2, s_2) \\ \ldots \\ (x_n, s_{n-1}) \lhd R' \rhd (y_n, s_n) \end{array}$$

in M, we have

$$x_1.x_2. \ldots .x_n \lhd R \rhd y_1.y_2. \ldots .y_n$$

in T. In other words, if the state machine M accepts, starting from the initial state s_0, a sequence $s \triangleq x_1.x_2. \ldots .x_n$ of inputs, and produces the sequence $s' \triangleq y_1.y_2. \ldots .y_n$ of outputs, then $(s, s') \in R$ is a trace in specification T.

Trace specifications abstract from state transitions by describing only the *history* of inputs and outputs. Trace specifications are often called *black-box specifications*, because they completely hide the internal description of the state machine.

4.2.2 Adequacy for the Formalization of Function Points

Trace specifications are less adequate than state machine specifications for the formalization of function points. Because they abstract from states, it is not possible to identify data components. Consequently, the calculation of weights for function components cannot be fully formalized, because data components are required.

As in state machine specifications, we may define elementary processes as operations and the boundary as a set of operations. We may classify function components by looking at their effect on the traces. When the condition 1 below is satisfied, operation OP_i is either an external inquiry or an external output. Let $x \in X_i$ be an input on which operation Op_i is called, and let $y \in Y_i$ be the result of this operation call.

$$\begin{array}{l} \forall x, y, s_1, s_2, s'_1, s'_2 : x \in X_i \land y \in Y_i : \\ s_1.x.s_2 \lhd R \rhd s'_1.y.s'_2 \\ \Rightarrow \\ s_1.s_2 \lhd R \rhd s'_1.s'_2 \end{array} \tag{1}$$

In other words, the execution of operation Op_i does not affect the behavior of subsequent operations in any trace. If this condition is not satisfied, then the operation is an external input. The DET of a function component are represented by the attributes of the operation signature.

We do not see at this point how we could distinguish between external outputs and external inquiries. We may have to recourse to syntactic heuristics, depending on the language used.

Data components could also be defined using syntactic heuristics. For instance, entities from the entity black box notation [24] could represent data components; DET would be calculated by taking the list of all attributes of all operations appearing in an entity structure diagram.

4.2.3 Adequacy for Specifying Information Systems

Trace specifications have been used for over ten years by several Cleanroom practitioners in organizations like IBM, the US Federal Government, US military contractors and Ericsson, a telecommunications manufacturer. Typically, trace specifications are written in a semi-formal notation.

This large user community represents evidence, as good as state machine experiments, that trace specification are useful for specifying systems. However, as for state machine languages, we lack precise data specific to information systems.

Trace specifications have the following strengths for specifying information systems.

1. The abstraction from states forces the specifier to focus on specification issues rather than design issues, providing a clearer understanding of the expected system behavior.

2. If an appropriate language is used, trace specifications may be shorter in length than state machine specifications [24].

In counterpart, we have the following weaknesses.

1. Cleanroom practitioners report that trace specifications can be quite lengthy, with 80% of the specification effort dedicated to handling invalid input sequences and errors. As we stated in the previous paragraph, small scale experiments show that an appropriate notation can alleviate that.

2. Complex calculations or validations are sometimes cumbersome to describe with traces. In these cases, one must mimic the state machine specification style by defining functions that compute state-like information from the traces.

4.3 Algebraic Languages

4.3.1 Language Characteristics

An algebraic specification is a triple $(Sorts, Functions, Axioms)$. Functions are defined over the sorts using axioms (equational or conditional) in first-order logic. The system state is represented by terms which are elements of sorts. Algebraic specifications are as abstract as trace specifications, because the state space needs not to be defined in terms of elementary theory like sets and relations. In practice however, most large size specifications make pervasive uses of elementary theories to represent the state space.

For the sake of simplicity, the reader may consider that functions correspond to operations in trace specifications, and that their signatures may be defined in a similar manner.

4.3.2 Adequacy for the Formalization of Function Points

Algebraic specifications are less adequate than trace specifications for formalizing function point. Operations are not always elementary processes. Some of them may be auxiliary operations used in the definition of operations required by the user. We do not see any systematic way of distinguishing them.

There is a similar problem with data components. Sorts may represent data components and DET, without any systematic way of distinguishing between them.

4.3.3 Adequacy for Specifying Information Systems

We do not have any data on the industrial usage of algebraic specifications. The basic academic examples of specifications that we have analyzed leads us to the following conclusions regarding their strengths and weaknesses.

1. Algebraic specifications are very good for defining abstract specifications of basic data structures. In fact, model-based specifications implicitly use an algebraic-like definitions of sets, relations and other basic data types.

2. Algebraic specifications are executables through term rewriting.

On the other hand, there are the following weaknesses.

1. Axioms for moderately complex specifications are not easy to write.

2. It is not easy to determine if the set of axioms is complete (sufficient) for the problem at hand.

3. Specifications are not easy to validate.

In his book on Larch [28], Guttag, one of the pioneers in algebraic specifications, advocates the use of algebraic specifications in conjunction with model-based specifications. In this approach, data types are defined using algebraic specifications; state space definitions and operations use these data types.

4.4 Process Algebra Languages

4.4.1 Language Characteristics

Process algebras share several similarities with trace specifications and algebraic specifications. They may be used at the same level of abstraction, that is, by describing observable events between the system and its environment.

In contrary to trace specifications, process algebraic specifications do not use predicates on sequences to describe ordering constraints; rather, the specification is a *term* of an algebra. Operators of a process algebra are specifically designed to express ordering constraints between events and parallel compositions of processes with synchronisation. Events are described using algebraic specifications.

Another distinction is that process algebras do not distinguish between input events and output events. In other words, events are not described as data submitted by the environment to the system, or data produced by the system for the environment. An event is simply a synchronization between the system and the environment. To classify events, one may use *heuristics* based on event decoration symbols like "?" and "!" (in CSP and LOTOS) to identify inputs and outputs, respectively. However, these heuristics may fail in several occasions.

4.4.2 Adequacy for the Formalization of Function Points

Because process algebras do not distinguish between inputs and outputs, it seems difficult to identify function components. There is a similar problem with data components. In process algebras, event parameters may either be input parameters, output parameters or state information. Hence, it is difficult to identify data components and weight them. Because of these difficulties, process algebra are not adequate for formalizing function points.

4.4.3 Adequacy for Specifying Information Systems

Process algebras were designed for specifying distributed systems and reactive systems, where the complexity resides in complex ordering constraints. It is feasible to specify information systems with them, but it is less convenient than model-based languages.

Strengths

1. Suitable for specifying ordering on events;

2. Suitable for specifying synchronization between processes.

Weaknesses

1. Cumbersome to specify data-intensive systems.

4.5 Modal Logic

4.5.1 Language Characteristics

As in process algebras, modal logics do not make a distinction between input events, output events, and state modification. Typically, modal logics are used in conjunction with other formalisms, like state machines or process algebras, to express safety properties or liveness properties. One may then use model checking or theorem proving to ensure that a specification satisfies these properties.

4.5.2 Adequacy for the Formalization of Function Points

If they are used alone, modal logics suffer the same limitations as process algebras. If they are used in conjunction with other formalism, then function points can be counted from the other formalism. Hence, there is no benefit to use modal logic to formalize function points.

4.5.3 Adequacy for Specifying Information Systems

Strengths

1. Suitable for specifying safety properties and liveness properties.

Weaknesses

1. Cumbersome to specify data-intensive systems;
2. Difficult to express complex ordering constraints.

4.6 Object-Oriented Notations

4.6.1 Language Characteristics

Among all the notations presented in this document, object-oriented notations are the most widely used in industry. UML becoming a *de facto* industry standard, and having a formal syntax and semantics, we will restrict our analysis to this language.

UML is a quite comprehensive and complex notation. It includes several types of diagrams for which a formal semantics has been defined. These diagrams are typically supplemented with formal texts (e.g., in OCL) or, most likely in practice, plain natural languages texts.

use case diagram : it defines the behavior of an entity, like a system or a subsystem, without specifying its internal structure.

In industry, specifiers typically use plain English to describe use cases. The UML semantics [66] suggests that more formal notations may be used to described a use case, like operations and state machines.

class diagram : it defines attributes, operations, methods, and relationships of the class objects. There are several possible relationships between objects: inheritance, aggregation, composition, association, dependency, etc.

behavior diagrams

statechart diagram : it shows the sequences of states that an object goes through during its life in response to inputs (method calls), together with its outputs and actions.

activity diagram : it is a kind of flowchart. Nodes are actions; arrows represent sequential execution of actions.

interaction diagrams

sequence diagram : it shows the sequence of messages exchanged between the environment and the objects composing the system. A message is represented by an horizontal arrow. Messages are temporally ordered by listing them from top to bottom.

collaboration diagram : like the sequence diagram, it shows the sequence of messages exchanged between the environment and the system; however, a number is assigned to each message to describe the ordering between messages.

4.6.2 Adequacy for the Formalization of Function Points

There is a natural mapping between function points and object-oriented notations, but it is difficult to formalize. Data components corresponds to classes of a class diagram. Note that not every class is a data components. For instance, control classes, that are responsible for managing the execution sequence, or interface classes, that are responsible for managing interaction with the environment, are not data components; they do not hold information relevant for the user; they represent design decisions rather than requirements information.

Relationships between classes like aggregation and generalization could be used to define ILF, EIF and RET. In [79], Whitmire suggests that leafs in a generalization hierarchy are ILF or EIF, and that aggregates are RET. Other view points could also be considered, like considering each aggregate or specialization as a RET.

Function components can be identified from sequence diagrams or collaboration diagrams. They correspond to messages exchanged between the environment and the system. However, we do not see a systematic way of classifying them into inputs, outputs or inquiries.

The main difficulty with object-oriented notations is that they mix requirements issues and technical design issues. Moreover, behavior diagrams are not precise enough to properly classify function components.

4.6.3 Adequacy for Specifying Information Systems

Object-oriented notations are widely spread in the industry. There is no doubt that they can be used for specifying information systems. However, there is no experimental evidence that they are the most cost-efficient notations for specifying systems.

Although we said that UML has a formal syntax and a formal semantics, its diagrams are not as expressive as the other formal specification languages presented in the previous sections. To write complete specifications, one must supplement the diagrams with a language like OCL.

Strengths

1. good abstraction mechanisms: classification, encapsulation, aggregation, specialization;
2. close to object-oriented programming languages;
3. graphical notation.

Weaknesses

1. relationship and consistency between diagrams are not well defined;
2. the notion of correctness between a specification and an implementation is not defined;
3. behavioral diagrams do not provide a complete specification of the system behavior.

4.7 Summary of Evaluation

Table 1 presents a summary of our evaluation for function point formalization and information systems specification. Each specification language class is given an overall rating in terms of three levels (good, fair, or weak), which provides a crude ordering. We also list dominant strengths, identified with a "+", and weaknesses, identified with a "-". The following acronyms are used:

- DC : data component in function points
- DS : data structure in information systems
- EO : external output in function points
- EQ : external inquiry in function points
- FC : function component in function points
- RET : record type in function points

Specification Language Class	Function Points Formalization	Information Systems Specification
State Machine	Good + identify FC and DC − distinguish EQ and EO − identify RET in DC	Good + model DS − event ordering
Traces	Fair + identify FC − distinguish EQ and EO − no DC	Good + state abstraction − complex DS management
Algebraic	Weak − identify FC − identify DC	Fair + basic DS − axiom derivation − validation
Process Algebras	Weak − identify FC − identify DC	Fair + event ordering + parallelism − DS management
Modal Logics	Weak − identify FC − identify DC	Fair + express properties − DS management
Object-oriented	Fair + identify FC − classify FC − identify DC	Fair + DS abstraction mechanisms − behavior specification

Table 1: Summary of Language Class Evaluation

5 Conclusion

It stems from our evaluation that state machines are the most appropriate languages for formalizing function points. Hence, they should be the first candidates. If we consider the adequacy for specifying information systems, then trace specifications are also good candidates; for some applications, they are even more appropriate than state machines. On the other hand, they are weaker than state machines for formalizing function points.

Another factor that we can take into account for our selection is industrial usage. A formalization of function points based on the most widely used notation would facilitate its integration into industrial practice. Object-oriented languages are predominant in industry, but they are less adequate for the formalization.

Considering all these issues, we propose the following research directions:

- formalize function points using state machine languages;
- define heuristics to estimate function points from trace specifications;

- adapt object-oriented notations so that they can be used to formalize function points.

References

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

[2] Abran, A., Robillard, P. N.: Reliability of Function Points Productivity Model for Enhancement Projects (A Field Study), *Conference on Software Maintenance*, Montreal, Quebec, Canada, 1993, 80–87.

[3] Albrecht, A.J., Gaffney, J.E.: Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, **SE-9**(6) (1983) 639–648.

[4] A. Arnold. MEC: a system for constructing and analyzing transition systems. In J. Sifakis, ed., *Automatic Verification of Finite State Systems*, *Lecture notes in Computer Science*, Vol. 407, 117–132, Springer, 1989.

[5] M. von der Beeck. A comparison of statecharts variants. In H. Langmaack, W.-P. de Roever and J. Vytopil, eds., *Formal Techniques in Real-Time and Fault-Tolerant Systems*, *Lecture Notes in Computer Science*, Vol. 863, 128–148, Springer, 1994.

[6] Bergstra, J.A., J.W. Klop: Algebra of Communicating Processes with Abstraction, *Theoretical Computer Science* **37**(1) (1985) 77–121.

[7] Bolognesi, T., E. Brinksma: Introduction to the ISO Specification Language LOTOS, *Computer Networks ISDN Systems* **14**(1) (1987) 25–59.

[8] G. Booch. *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin-Cummings, 1994.

[9] Broy, M., F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, R. Weber: The Design of Distributed Systems — an Introduction to FOCUS, Technische Universität München, Institut für Informatik, TUM-I9203, January 1992

[10] Broy, M., C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, K. Stølen: The Requirement and Design Specification Language SPECTRUM. An Informal Introduction, Version 1.0, Technical Report TUM-I9311, Institut für Informatik, Technische Universität München, Germany, 1993.

[11] Broy, M., Coleman, D., Maibaum, T., Rumpe, B.: *PSMT — Workshop on Precise Semantics for Software Modeling Techniques*, Technical Report TUM-I9803, Institut für Informatik, Technische Universität München, Germany, 1998.

[12] Burstall, R.M., J.A. Goguen: The Semantics of CLEAR, a Specification Language, *Proc. Advanced Course on Abstract Software Specification*, Lecture Notes in Computer Science, Vol. 86, Springer-Verlag, 1980, 192–232.

[13] CoFi Task Group on Language Design: CASL — The CoFi Algebraic Specification Language — Design Proposal, `http://www.brics.dk/Projects/CoFI/index.html`, 1998.

[14] Feijs, L.M.G., H.B.M. Jonkers, C.A. Middelburg: Notations for Software Design. FACIT Series, Springer-Verlag, 1994.

[15] Clarke, E.M., E.A. Emerson: Design and Synthesis of Synchronisation Skeletons using Branching Time Temporal Logic, *Proc. Workshop of Logics of Programs*, Lecture Notes in Computer Science, vol. 131, 1981, 52–71.

[16] P. J. Denning, J. B. Dennis and J. E. Qualitz. *Machines, Languages and Computation*, Prentice Hall, 1978.

[17] Desharnais, J.-M., St-Pierre, D., Maya, M., Abran, A.: Full Function Points: Counting Practices Manual, Rules and Procedures, Université du Québec à Montréal, Montréal, Technical Report no. 1997–06, November 1997.

[18] J. Desharnais, M. Frappier, R. Khédri and A. Mili. Integration of sequential scenarios. In M. Jazayeri and H. Schauer, ed., *6th European Software Engineering Conference / 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lect. Notes in Comp. Sci.*, Vol. 1301, 310–326, Springer, 1997.

[19] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, ed., *Programming Languages: NATO Advanced Study Institute*, 43–112, Academic Press, 1968.

[20] Dolado, J.J.: A Study of the Relationships among Albrecht and Mark II Function Points, Lines of Code 4GL and Effort, *The Journal of Systems and Software*, **37**(2) (1997) 161–173.

[21] Ehrig, H., B. Mahr: *Fundamentals of Algebraic Specification 1*, Springer-Verlag, 1985.

[22] J. Ellsberger, D. Hogrefe and A. Sarma. *SDL — Formal Object-Oriented Language for Communicating Systems*, Prentice Hall, 1997.

[23] J. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13, 219–236, 1989.

[24] Frappier, M., St-Denis, R.: A Specification Method for Cleanroom's Black Box Description. *31st Hawaii International Conference on System Sciences*, IEEE Computer Society Press, 1998.

[25] Furey, S., Kitchenham, B.: Point/Counterpoint Function Points, *IEEE Software*, **(**14)2 (1997) 28–33.

[26] Gaudel, M-C.: Structuring and Modularizing Algebraic Specifications: the PLUSS Specification Language, Evolutions and Perspectives, Proc of the 9th Symposium on Theoretical Aspects of Computer Science, Lecture Notes In Computer Science, vol. 577, Springer-Verlag, 1992.

[27] Goguen, J.A., T. Winkler: Introducing OBJ3, SRI International, 1988.

[28] Guttag, J.V., J.H. Horning: *LARCH: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[29] Allemand, M., Attiogbé, C. Habrias, H., editors (1998) *Proc. of International Workshop on Comparing Systems Specification Techniques*, Institut de recherche en informatique de Nantes, Nantes, (France), March 26–27, 1998.

[30] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 231–274, 1987.

[31] D. Harel. On the formal semantics of statecharts. *Proc. 2nd IEEE Symposium on Logic in Computer Science*, 54–64, Ithaca, NY, 1987.

[32] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5), 514–530, May 1988.

[33] Hoare, C.A.R. (1985) *Communicating Sequential Processes*, Prentice Hall.

[34] G. J. Holzmann. *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[35] J. J. M. Hooman, S. Ramesh and W.-P. de Roever. A compositional axiomatization of statecharts. *Theoretical Computer Science*, 101, 289–335, 1992.

[36] IFPUG (International Function Point Users Group): Function Point Counting Practices Manual, Release 4.0. Westerville. Ohio: International Function Point Users Group, 1994.

[37] i-Logix Inc. Andover, MA 01810, USA `http://www.ilogix.com/`.

[38] ISBSG (International Software Benchmarking Standards Group): Benchmarking Repository, Victoria, Analysis, Release 3, June, 1996.

[39] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

[40] Jackson, M. (1983) *System Development*, Prentice Hall.

[41] Jeffery, D.R., Low, G.C.: Comparison of Function Point Counting Techniques, *IEEE Transactions on Software Engineering*, **SE-19**(5) (1993) 529–532.

[42] Jones, C.B.: *Systematic Software Development using VDM*, second edition, Prentice Hall, 1989.

[43] Jones, E.L.: Automated Calculation of Function Points, *Proc. of the 4th Software Engineering Research Forum*, Boca Raton, Florida, 1995.

[44] Kemerer, C.F., Porter, B.S.: Improving the Reliability of Function Point Measurement: An Empirical Study, *IEEE Transactions on Software Engineering*, **18**(11) (1992) 1011–1024.

[45] Kemerer, C.F.: Reliability of function points measurement: a field experiment, *Communications of the ACM* **36**(2) (1993) 85–97.

[46] Kitchenham, B.: Using Function Points for Software Cost Estimation — Some Empirical Results, *Tenth Annual Conference of Software Metrics and Quality Assurance in Industry*, Amsterdam, 1993.

[47] Kitchenham, B., Kansala, K.: Inter-item correlations among function points, *Proceedings First International Software Metrics Symposium* , IEEE Computer Society Press, (Cat. No.93TH0518-1), Baltimore, MD, USA, 1993, 11–14.

[48] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16, 872–923, May 1994.

[49] L. Lamport. TLA in pictures. *IEEE Transactions on Software Engineering*, 21 (9), 768–775, September 1995.

[50] Lano, K.:*Formal object-oriented development*, Springer-Verlag, 1995.

[51] Manna, Z., A. Pnueli: *The temporal logic of reactive and concurrent systems*, Springer-Verlag, 1992.

[52] Matson, J.E. , Mellichamp, J.M.: An object-oriented tool for function point analysis, *Expert Systems*, **10**1 (1993) 3–14.

[53] Matson, J.E., Barrett, B.E., Mellichamp, J.M.: Software development cost estimation using function points, *IEEE Transactions on Software Engineering*, **SE-20**(4) (1994) 275–287.

[54] Mazzucco, F.A.: IEF—Automatic Function Point Count, *Proc. of Proc. of the Conference on Software Measurement and Management (IFPUG'92)*, Baltimore, Maryland, 1992, 169–181.

[55] G. H. Mealy. A method for synthesising sequential circuits. *Bell System Tech. J.*, 34(5), 1045–1079, September 1955.

[56] Mendes, O., Abran, A., Bourque, P.: Function Point Tool Market Survey, Research Report, Université du Québec à Montréal, Software Engineering Management Laboratory, Montréal, Canada, December, 1996.

[57] Mills, H.D., A.R. Hevner, R.C. Linger: *Principles of information systems analysis and design*, Academic, 1986.

[58] Skuce, D.R., Mili, A.: Behavioral Specifications in Object Oriented Programming, *Journal of Object Oriented Programming*, January, 1995, 41–49.

[59] Mills, H.D., V.R. Basili, J.D. Gannon, R.G. Hamlet: *Principles of Computer Programming: A Mathematical Approach.* Allyn and Bacon, 1987.

[60] Milner, R.: *Communication and Concurrency* Prentice Hall, 1989.

[61] E. F. Moore. Gedanken-experiments on sequential machines. *Annals of Mathematics Studies, Vol. 34, Automata Studies*, 129–153, Princeton University Press, Princeton, NJ, 1956.

[62] ObjecTime Corporation Limited, Kanata, Ontario, Canada, `http://www.objectime.on.ca/`.

[63] Paton, K., Abran, A.: A Formal Notation for the Rules of Function Point Analysis, Research Report, Université du Québec à Montréal, Software Engineering Management Laboratory, Montréal, Canada, April, 1995.

[64] RAISE Method Group: *The RAISE Development Method.* BCS Practitioner Series, Prentice Hall, 1995.

[65] Rational Software (1997) Unified Modeling Language Summary, version 1.1, September 1st, `http://www.rational.com/uml/` .

[66] Rational Software (1997) UML Semantics version 1.1, September 1st, `http://www.rational.com/uml/` .

[67] Rational Software (1997) Object Constraint Language Specification, version 1.1, September 1st, `http://www.rational.com/uml/` .

[68] Rask, R.: Algorithms for Counting Unadjusted Function Points from Dataflow Diagrams, University of Joensuu, Joensuu, Finland, Research report A-1991-1, September 30, 1991.

[69] Rask, R.: Counting function points from SA descriptions, *Proc. of Annual Oregon Workshop on Software Metrics*, Silver Falls, Oregon, 1991.

[70] J. Rumbaugh. *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[71] Schilling, M.: Counting Function Points From Entity Relationship Models, *Proc. of the Conference on Software Measurement and Management (IFPUG'96)*, February 5–9, Rome, Italy, 1996.

[72] B. Selic, G. Gullekson, P. T. Ward. *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.

[73] Spivey, J.M.: *Understanding Z : A Specification Language and its Formal Semantics*, vol. 3 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (UK), 1988.

[74] Symons, C.R.: *Software Sizing and Estimating: Mk II FPA*, John Wiley, Chichester, U.K., 1991.

[75] Telelogic, Malmö, Sweden, `http://www.telelogic.se/`.

[76] VERILOG, Toulouse CEDEX, France, `http://www.verilogusa.com/`.

[77] Wang, Y., D.L. Parnas (1994) Simulating the Behavior of Software Modules by Trace Rewriting. *IEEE Transactions on Software Engineering*, **20**(10) 750–759.

[78] Wittig, G.E., Finnie, G.R.: Software design for the automation of unadjusted function point counting, *Proc. of Business Process Re-Engineering: Information Systems Opportunities and Challenges*, B.C. Glasson *et al*, eds, IFIP TC8 Open Conference, Australia, Elsevier Science, 1994.

[79] Whitmire, S.A.: Applying function points to object-oriented software models, in *Software engineering productivity handbook*, J. Keyes, Ed.: McGraw-Hill, 1992, 229–244.