An Adaptive Framework for the Design of Software Specification Languages

J. J. Neto, P. S. Muniz Silva

Department of Computer and Digital Systems Engineering,
Polytechnic School of the Sao Paulo University, Brazil
E-mail: {joao.jose, paulo.muniz}@poli.usp.br

Abstract

Software specifications may be viewed as theories elaborated for application domains. A useful strategy for building specifications is the incremental extension of an initial theory, in which increments add new terms and notions not considered in previous extensions. Given an increment, the corresponding theory is stated in a corresponding specification language. The next increment – or extension of the theory – typically requires a related language extension. Adaptive devices naturally support such scheme, whose instances should reflect the impact of extension variations on the specification language. This paper describes an adaptive framework for the design of a class of software specification languages supporting the incremental process of elaborating software specifications.

1 Introduction

Software engineering strives to provide practitioners with principles that help building reliable software systems. One of the best-known principles concerns the elaboration of software specifications to bridge the gap between software requirements and their implementation in some programming language. It is not the case of discussing the rich debate about this issue, but it is worth mentioning some lessons learned from the use of specification languages in the traditional sequential program construction viewpoint[1]. Firstly, the use of a formal specification language, guided by some formal method, does not guarantee the correctness of the software system under construction [1], but specification languages with a formal basis increase our understanding of the specification by allowing the detection of inconsistencies and ambiguities. Secondly, a specification should support extensions, and should also exhibit operational capabilities [2]. We are interested in these latter features as the main drivers of specification language design. The novelty of our approach is the use of an adaptive device [3] in the specification language design, supporting the realization of those desirable features. The motivation for using the adaptive device approach is that such devices naturally solve the realization problem. Section 2 presents the adopted viewpoint for specifications and specification languages. Sections 3, 4 and 5 describe our strategy to extend a specification (programming) language, and present simple examples. Section 6 draws some conclusions.

[1] In this paper, we are not considering the component based software construction approach, i.e., the reuse in-the-large.

2 Specifications and Specification Languages

Software specifications may be viewed as theories presentations elaborated for application domains. It is expected that the resulting working program be derived from its specification through a finite series of step-by-step transformations extending the base theory, i.e. the original specification [4]. Fundamental definitions underlying this approach may be found elsewhere [5, 6], to mention a few. We will briefly quote and synthesize the core definitions of [4], with respect to theory and language extensions.

- Since specifications and programs are linguistic constructs, they must be expressed in a defined linguistic system. A linguistic system consists of two parts: a collection of well-formed sentences, and a code of reasoning. Formally, $LS = \langle L_{LS}, \vdash_{LS} \rangle$, where L_{LS} denotes the linguistic system language and $\vdash_{LS}$ denotes the rules of reasoning of the *LS*.
- A theory *T* in a linguistic system *LS* is a set of *LS* formulae, which is closed under $\vdash_{LS}$. A theory presentation is an axiomatization of the theory, i.e. a set of formulae from which all formulae of *T* can be derived by means of the derivability relation $\vdash_{LS}$. Formally, $T = \langle LS_T, A_T \rangle$, where LS_T is the linguistic system of *T* and A_T is the presentation of *T*.
- An $LS' = \langle L_{LS'}, \vdash_{LS'} \rangle$ is an extension of $LS = \langle L_{LS}, \vdash_{LS} \rangle$, iff $L_{LS} \subseteq L_{LS'}$ e $\vdash_{LS} \subseteq \vdash_{LS'}$. If two theories are expressed in the same *LS*, their respective languages and presentations, leaving the underlying linguistic system implicit, can characterize them. Formally, $T = \langle L_T, A_T \rangle$ e $T' = \langle L_{T'}, A_{T'} \rangle$.
- A theory *T'* is an extension of *T*, i.e. $T \subseteq T'$, iff the properties defined by *T* for the symbols of L_T are still there in *T'*, and *T'* allows the proofs of some new properties.

- An extension $T \subseteq T'$ is conservative iff for all formulae A of L_T, if $A_{T'} \vdash_{SL'} A$, then $A_T \vdash_{SL} A$.

A program can also be viewed as a theory presentation with an underlying linguistic system, but due to the operational nature of programming languages there is a bias of the theory presentation towards a particular interpretation (a particular implementation) [4]. In other words, a program is a specification of another program written, say, in some machine language.

In brief, the program construction process is a series of conservative incremental extensions of an initial theory, in which increments add new terms and notions not considered in previous extensions, while preserving the properties included at each step. Given an increment, the corresponding theory is stated in a corresponding specification language. The next increment – or extension of the theory – typically requires a related language extension. To meet this requirement, the specification language should be *extensible*. Does this property imply writing a new compiler? Or would it be better using a purely syntactic extension based on the semantics of an appropriate existing language? We take the latter approach as the main strategy for the design of software specification languages.

In this paper, we illustrate this strategy with a very simple example. Let the following operation be a fragment of a theory presentation on integers, written in an imperative Pascal-like programming language:

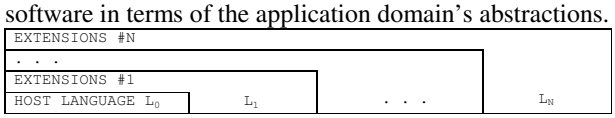
```
FUNCTION divide (n: INTEGER; d: INTEGER): INTEGER;
      START
      divide:= n/d
      END;
```

Let us suppose we want to extend the theory with the notion of *preconditions* as of the *design by contract* approach [7]. This approach states that the function invocations at any point in a program must satisfy the precondition specified for the function. That means the programmer will not write explicit defensive code in order to check the precondition in the function body, because the respect for the precondition is an obligation for the client of certain service, not for the supplier of that service. The precondition specification for the above fragment may be written as: `PRE d <> 0`, where `PRE` denotes a precondition. This theory extension implies a language extension.

3 Extending Programming Languages

Let us sketch a proposal for extending programming languages as an intermediate goal towards the extension of software specifications. Our main intent is keeping the specifications executable. Starting from an available programming language, one is allowed to employ in the desired software specification strictly the abstractions provided by the host language's syntactic constructs. When this is a syntactically extensible language, users may create new syntactic constructs for representing abstractions not included in the original notation, so users may represent new abstractions in terms of existing ones. By proceeding in a hierarchical way, successive abstraction layers of increasing complexity may be built in order to bring the expressiveness of the language closer and closer to the domain of the particular software being specified. Since each layer's abstractions are stated strictly as combinations of previously defined abstractions, at each specification step any software will be easily translated into the immediately preceding layer's abstractions, until reaching the lowest level abstraction layer, represented by the host programming language.

With such a simple approach, specifying a software is reduced to specifying a sequence of successive abstractions, which are directly mapped into corresponding language syntactical extensions and their associate translating scheme into the abstractions defined in previously defined abstraction layers. This process proceeds until the set of available abstractions match the expressiveness requirements for defining the application software in terms of the application domain's abstractions.



The main feature of this approach is keeping the specifications executable all along the development process. No semantic gaps are introduced between the specification and its implementation, since all specification steps may be immediately converted into executable code, and no abstraction is created unless it be strictly stated in terms of existing, executable ones.

By adequately choosing the host programming language, one may significantly reduce the need for supporting computing resources for implementing this approach. For instance, if L_0 is an already existing extensible language (e.g. LISP) all one must do is to define the abstractions needed as extensions of L_0 (in the case of LISP, the extensions take the form of either macros or functions). On the other hand, when L_0 is not already extensible, then an extension feature must be added in order to allow following the proposed approach.

For usual non-extensible procedural host languages, it is possible to implement a language extension additional layer in order to provide the needed facilities for programmers to provide their language extensions as definitions for new abstractions. A powerful way to allow the inclusion of extra abstractions in a given language is to offer some meta-linguistic feature for allowing to define new syntactical constructs. In our case, Wirth's context-free extended BNF notation has been chosen [8]. That is

enough for defining new syntax. Processing and incorporating such user-defined syntactical extensions as a preprocessor for the host language compiler is quite straightforward [9]. However, syntax is not all we need. It is necessary to state all extensions in terms of already existing language constructs. In order to overcome this problem, we adopted another well-known solution, e.g. typifying the extensions and using classical operational semantics for interpreting each new construct in terms of existing ones: each new extension is declared as a context-free grammar, and its meaning is also declared as a text stated as a program using the basic host language syntax enriched with previously declared syntactical constructs.

In this way, the compiler is informed on the new construct to be further accepted, as well as on how exactly it must be translated into lower-level abstractions. After accepting such an extension definition, the compiler extends its acceptor in order to recognize the new syntax, and associates the declared translation procedure to the syntactical recognition of the new construct. Whenever further input text contain excerpts that follow the new defined syntax, its related translation procedure is followed, converting the input text in the extended notation into another text written in the previous abstraction level.

4 A Very Simple Extension Layer

In this section we present a simplified proposal of a nucleus for an extension mechanism, to be used as a preprocessor for procedural, originally non-extensible languages. For space reasons, the host language has been reduced to a minimum: block structure has been removed, declarations have been reduced to simple untyped variables, and commands have been eliminated, except for if's, go to's and assignments of simple expressions.

The following context-free grammar, stated in modified Wirth's notation, defines, in its first part, the (non-extensible host language) nucleus L_0 we are going to use as the starting version of our extensible language, and in its second part, the proposed extension mechanism, represented by **EXT**. Each time the non-terminal **EXT** is instantiated, it extends the previous version of the language by adding a new non-terminal (**NEWNTERM**) to its grammar, and incorporates the corresponding abstraction to the language syntax. Note that **TERM** represents any terminal in the language, including identifiers (**id**), integers (**int**) and other elementary language components.

```
//* HOST LANGUAGE (EXTREMELY SIMPLIFIED) *//
PROG = "BEGIN" ( DECL \ ";" ) "START" ( COM \ ";" ) "END" .
DECL = "VAR" ( id \ "," ) ":" "INTEGER" | PROCEDURE | EXTENSION .
COM  =  LABEL ":" PROG  | id := EXPARIT | "GOTO" LABEL |
        "IF" EXPARIT ( ">" | "=" | "<" | "<>" ) EXPARIT
           "THEN" PROG ( "ELSE" PROG | ε ) | PREVIOUSNTERM .
EXPARIT = (( id | int | CALL ) \ ( "+" | "-" | "*" | "/" )) .
CALL = id "(" ( id | int | CALL  \ "," ) ")" .
```

```
PROCEDURE = "FUNCTION" id "(" ( id ":" "INTEGER" \ ";" ) ")"
                       ":" "INTEGER"  ";"
                       "START" ( COM \ ";" ) "END" ";" .
LABEL = id .
```

```
//* PROPOSED EXTENSION MECHANISM *//
PREVIOUSNTERM  =  Ø .
EXTENSION = "DEFINE" NEWNTERM ":" "NEW" NTERM "AS" WIRTHMOD
                     "MEANING" PREVIOUSWIRTHMOD "ENDDEFINE" .
NTERM = "PROG" | "DECL" | "COM" | "EXPARIT" | "EXTENSION"
      | "LABEL" | "NTERM" | "CALL" | "PROCEDURE" | "NEWNTERM"
      | "WIRTHMOD" | "PREVIOUSWIRTHMOD" | PREVIOUSNTERM .
NEWNTERM = id .
WIRTHMOD = ((( TERM | NTERM | NEWNTERM | "ε"
          | "(" WIRTHMOD ( "\" WIRTHMOD  | ε ) ")" )
            ( "#" int | ε ) \ ( "|" | ε  ) ) .
PREVIOUSWIRTHMOD = ((( TERM | NTERM | "ε"
        | "(" PREVIOUSWIRTHMOD ( "\" PREVIOUSWIRTHMOD | ε ) ")" )
          ( "#" int | ε ) \ ( "|" | ε  ) ) .
```

The interpretation of the above grammar is almost conventional, except for the meta-symbol Ø that refers to the empty set: initially there are no **PREVIOUSNTERM**s. After the full handling of the declaration of an **EXTENSION** the name corresponding to the **NEWNTERM** being declared is added to the **PREVIOUSNTERM**s set of already defined non-terminals of the grammar. So, **WIRTHMOD** refers to some syntactical definition involving any terminals or non-terminals, while **PREVIOUSWIRTHMOD** represents a syntax strictly stated in terms of the non-terminals representing abstractions known at the previous extension's abstraction layer. The extension pre-processor must adequately update the set of **PREVIOUSNTERM**s in order to keep the integrity of this mechanism.

5 Illustrating Case Studies

For illustration purposes, let us first work the small situation referred to at the end of section 2. Let us restrict the extension to including in the language the declaration of preconditions. The suggested syntax has been starting the construct with the word "PRE" followed by a condition (in the case of our language, conditions may be defined in terms of the relation between two arithmetic expressions), exercising some of the previously described features. In order to explore the extensibility feature introduced by the preprocessor, the programmer should declare the desired new syntax as shown below.

In words, the denominator of the division will be automatically checked against zero every time the function `divide` is called, and an error report will be generated whenever that condition succeeds.

```
DEFINE PRECONDITION: NEW COM AS
   "PRE" EXPARIT # 1 ( "=" | "<" | ">" | "<>" ) EXPARIT # 2
MEANING
   "IF NOT (" EXPARIT # 1 ( "=" | "<" | ">" | "<>" ) EXPARIT # 2
         ") THEN ERROR() ELSE"
ENDDEFINE;
...
FUNCTION divide (n: INTEGER; d: INTEGER): INTEGER;
      START
      PRECONDITION d <> 0;
      divide:=n/d
      END;
...
```

In terms of our preprocessor, its behavior may be identified in the generated expanded code as follows:

```
FUNCTION divide (n: INTEGER; d: INTEGER): INTEGER;
      START
      IF NOT ( d <> 0 ) THEN ERROR () ELSE divide:=n/d
      END;
```

The next is another simple illustrating one referring to the creation of new commands. Through such a feature, it is easy to create new abstractions from already existing ones. For instance, the next declaration adds a `WHILE` statement to our host language:

```
DEFINE WHILESTATEMENT: NEW COM AS
   "WHILE" EXPARIT # 1 ( "=" | "<" | ">" | "<>" ) EXPARIT # 2
   "REPEAT" PROG # 3
MEANING
 "LOOP#: IF"  EXPARIT # 1 ( "=" | "<" | ">" | "<>" ) EXPARIT # 2
        "THEN BEGIN" PROG # 3 "; GO TO LOOP# END"
ENDDEFINE;
```

In this case, note the meta-label `LOOP#` that must be instantiated each time a `WHILESTATEMENT` is called, in order to avoid label duplication in the resulting program.
For instance, the `WHILESTATEMENT` below:

```
WHILE x < 3 REPEAT BEGIN x := x+1; y:= y-1 END
```

is expanded into the following equivalent program, according to the template previously defined:

```
LOOP0001: IF x < 3 THEN BEGIN
                        BEGIN x := x+1; y:= y-1 END;
                        GO TO LOOP0001
                        END
```

The host language must at least provide the full set of primitive constructs needed to specify all operations in the program we intend to build. If it is not the case, then it would not be possible to express the application program facts in terms of the available host language's basic syntactic constructs, unless some extra effort be made in order to provide the missing facilities for the host language before the desired extensions are created and used.

6 Conclusions

Obviously, there is much more to say about using extensible features of a programming language in order to ease the specification of software projects than what one is allowed to fit into a four-page paper. Programming language extensibility and operational semantics have not received adequate attention in recent scientific works. However, as we tried to show in this paper, their features allow an easy way for implementing software directly from specifications, in a bottom-um fashion, always keeping the specifications executable, and eliminating the deep gap between specifications and implementation, usually found in current software engineering practices.
Using adaptive technology in the implementation of compilers and preprocessors substantially reduces the difficulty of implementing languages representing multi-layers of abstractions by allowing the user to interact with the kernel of the host language's compiler without opening its source code. Such great feature is accomplished by the unification got when using adaptive automata as the language's run-time abstract machine.
In this way, adaptive automata may be generated both as object-code, and as a syntax recognizer, so programs may be compiled, their syntax may be modified and they also may be easily executed in the same unified framework provided by the underlying adaptive environment.

References

[1] Clarke, E.M., Wing, J.M. (1996) Formal methods: state of the art and future directions. ACM Computing Surveys 28(4): 626-643.

[2] Balzer, R., Goodman, N. (1986) Principles of good specification and their implication for specification languages. In Gehani, N., McGretick, A. (eds.) Software specification techniques. Addison-Wesley, pp. 25-39.

[3] Neto, J.J. (2001) Adaptive rule-driven devices – general formulation and case study. LNCS v.2494, Springer-Verlag, pp. 234-250.

[4] Turski, W.M., Maibaum, T.S.E. (1987) The specification of computer programs. Addison-Wesley, London, UK.

[5] Smith, D.R. (1999) Designware: software development by refinement. In Proc. 8th Internat. Conf. on Category Theory and Computer Science (CTCS '98), Edinburgh, UK.

[6] Maibaum, T.S.E. (2003) On what exactly goes on when software is developed step-by-step, II: the sequel. Information Processing Letters 88: 45-51.

[7] Meyer, B. (1997) Object-oriented software construction, 2nd. Ed. Prentice-Hall, New Jersey, USA.

[8] Wirth, N. What can we do with the unnecessary diversity of notation for syntactic definitions? CACM 20 (11): 822-823.

[9] Neto, J.J., Pariente, C.B., Leonardi, F. (1999) Compiler Construction – a Pedagogical Approach. V Int. Congress on Informatics Engineering – Buenos Aires.