Converting BPSL Behavioral Specification to FSP Using A Java-Based Parser Generator

Toufik Taibi
Multimedia University,
Faculty of Information Technology,
Jalan Multimedia, 63100 Cyberjaya,
Selangor, Malaysia
Email: toufik.taibi@mmu.edu.my

Abstract: *Balanced Pattern Specification Language (BPSL) can be used to formally specify design patterns and their combination. BPSL uses a subset of First Order Logic (FOL) to specify the structural aspect of design patterns and a subset of Temporal Logic of Actions (TLA) to specify their behavioral aspect. BPSL as any other language requires a lexical analyzer (lexer) and a parser in order to allow its users to check the lexical and syntactic correctness of their specification. Writing lexers and parsers from scratch can be a tedious and error prone process. As such, lexers and parsers generation tools have been developed to automate this task. This paper describes how both Java-based Lexer (JLex) and Constructor of Useful Parsers (CUP) were successfully used to generate highly optimized Java-based lexer and parser for BPSL. Moreover CUP was used to convert BPSL behavioral specifications to the well-known Finite State Processes (FSP) specifications in order to use Labeled Transaction System Analyzer (LTSA) model checking tool.*

Keywords: *BPSL, Lexer, Parser, JLex, CUP, FSP, LTSA*

Received: June 25, 2003 | ***Revised:*** January 05, 2004 | ***Accepted:*** February 15, 2004

1. Introduction

A design pattern describes a successful solution to a recurring problem within a context [6]. Since design patterns have been extensively tested and used in many development efforts, reusing them yields better quality software within reduced time frame. In the early stages of their evolution, design patterns were described using a combination of text, Object-Oriented (OO) graphical notations and sample code fragments. However as soon as the number of design patterns has grown, and problems requiring pattern combination have surfaced, users started to realize that textual description can be ambiguous and sometimes misleading in understanding and applying design patterns.

Hence, there was a need for a formal means of accurately describing design patterns. Formal specification of design patterns is meant to achieve well-defined semantics, allow rigorous reasoning and facilitate tool support. We propose a Balanced Pattern Specification Language (BPSL) [16] [17], meant to accurately convey the essence of design patterns in a balanced way. BPSL was inspired by our views on *"why and how should design patterns be formalized?"* [15]. BPSL combines the formal specification of structural and behavioral aspects of design patterns by using one subset of First Order Logic (FOL) [2] and one subset of Temporal Logic of Actions (TLA) [9] respectively.

BPSL as any other language needs a lexical analyzer (lexer) and a parser in order to allow users to check the lexical and syntactic correctness of their pattern specification using BPSL. Writing a lexer and a parser from scratch can be a tedious and error prone process. As such, lexers and parsers generation tools have been developed to automate this task. Java-based Lexer (JLex) and Constructor of Useful Parsers (CUP) were successfully used to generate highly optimized Java-based lexer and parser for BPSL. JLex takes a specification file similar to that accepted by *lex* [10], then creates a Java source file for the corresponding lexer.

CUP is a system for generating Look Ahead Left Right (LALR) [1] [5] parsers. CUP offers most of the features of *yacc* [10], however, CUP is written in Java, uses specifications including embedded Java code, and produces parsers, which are implemented in Java. Using CUP involves

creating a specification based on the grammar for which a parser is needed.

The possibility of embedding Java code into the CUP specification of BPSL grammar allowed the checking of all possible lexical, syntactic and semantic errors in a BPSL specification of a given pattern. Moreover Java code was added to convert a BPSL behavioral specification (which is based on a subset of TLA) to a Finite State Processes (FSP) specification in order to use the Labeled Transition System Analyzer (LTSA) model checking tool [11].

The rest of the paper is organized as follows. Section 2 summarizes BPSL's concepts and constructs. Section 3 describes how JLex and CUP were used to generate a lexer and a parser for BPSL. Section 4 discusses how BPSL behavioral specifications were converted to FSP. Section 5 shows how to run BPSL's parser, while section 6 concludes the paper.

2. Balanced Pattern Specification Language (BPSL)

Following are the building blocks of BPSL. They reflect entities (participants) and relations (collaborations) between them in a design pattern.

1. Classes, attributes, methods, objects, and untyped values make the *primary* entities, which are considered irreducible units. Untyped values are values of variables of any type.
2. Relations express the way entities collaborate. Relations can either be permanent or temporal. Permanent relations between entities once defined cannot be changed while temporal relations may change throughout the execution of actions. BPSL defines a set of *primary* permanent relations based on which other permanent relations can be built (see Table 1).
3. Actions are atomic units of execution, which can be understood as multi-object methods used to embody the behavioral aspect of design patterns. Actions associate and disassociate objects through temporal relations.
4. Any newly defined entity or permanent relation must derive from the *primary* entities and *primary* permanent relations respectively.

The subset of FOL used by BPSL to formally specify the structural aspect of patterns consists of constant and variable symbols, connectives (mainly $\wedge$), quantifiers (mainly $\exists$) and predicate symbols acting upon constant or variable symbols. Constant symbols are used in the case of formally specifying instances of patterns. Variable symbols represent classes, attributes, methods, objects and untyped values while the predicate symbols represent permanent relations among them. The domains (sets) of *primary* entities that are classes, attributes, methods, objects, and untyped values are designated *C, A, M, O,* and *V* respectively. Table 1 depicts the *primary* permanent relations, their domain and their intent. These relations straightforwardly derive from OO concepts. It is the smallest set (in terms of number of elements) on top of which any other permanent relation can be built.

For patterns that have a significant behavioral aspect, it is necessary to understand how objects collaborate to achieve the expected behavior. The subset of TLA used by BPSL to formally specify the behavioral aspect of pattern deals with behaviors $(S_0,S_1,...)$ defined as infinite sequences of states. Each state S_i is a collection of values of state variables (attributes) and the temporal relations that exist between objects. A pair of consecutive states (S_i,S_{i+1}) in a behavior is called a *transition*. The system starts in some initial state. As time passes, actions are executed, changing the system's state accordingly. Actions are selected for execution *non-deterministically*, the only restriction being that the precondition of an action must be true in order for the action to be executed. The execution of an action is *atomic,* meaning that once the execution of an action has been started, it cannot be interrupted or interfered with by other actions. The computational model is *interleaving*, that is, only one action at a time is being executed.

Name	Domain	Intent
Defined-in	***MxC***	Indicates that a method is defined in a certain class.
	AxC	Indicates that an attribute is defined in a certain class.
Reference-to-one (*-many*)	***CxC***	Indicates that one class defines a member whose type is a reference to one (many) instance(s) of the second class.
Inheritance	***CxC***	Indicated that the first class inherits from the second.
Creation	***MxC***	Indicates that a method contains an instruction that creates a new instance of a class.
	CxC	Indicates that one of the methods of a class contains an instruction that creates a new instance of another class.
Invocation	***MxM***	Indicates that the first method invokes the second method.
	CxM	Indicates that a method of a class invokes a specific method of another class.
	MxC	Indicates that a specific method of a class invokes a method of another class.
	CxC	Indicates that a method of a class invokes a method of another class.
Argument	***CxM***	Indicates that a reference to a class is an argument of a method.
	AxM	Indicates that an attribute is an argument of a method.
	VxM	Indicates that an untyped value is an argument of a method.
Return-type	***CxM***	Indicates that a method returns a reference to a class.
	OxM	Indicates that a method returns an object.
Instance	***OxC***	Indicates that an object is an instance of a certain class.

Table 1: Primary Permanent Relations and Their Intent**.**

A temporal relation can be defined as follows: *TR(C1[cardinality],C2[cardinality])*, where *TR* is the name of the temporal relation, *C1* and *C2* are classes, and cardinality represents the number of instances of each class that participate in the relation. Cardinality can be represented as either a closed interval *[n..m],* where *n* and *m* represent any two positive integers or *[*]* to depict any positive integer. When used in actions, *TR(o1,o2)* depicts that an object *o1* of a class *C1* is currently associated through *TR* with an object *o2* of a class *C2*, while *¬TR(o1,o2)* depicts that objects *o1* and *o2* are no longer associated through *TR*. *TR(o1,C2)* depicts that object *o1* is associated with all instances (objects) of the class *C2*. *¬TR(o1,C2)* depicts that object *o1* is not associated through *TR* with any object of class *C2*. *¬TR(C1,C2)* depicts that none of the objects of class *C1* is associated through *TR* with any object of class *C2*.

An action consists of a list of parameters (object and untyped values), a precondition and a body. The body is a definition of a state change caused by an execution of the action. For example, if *TR* is a temporal relation between two classes *C1* and *C2*, an action *A* may be defined as follows: $A(o1,o2,p) : TR(o1,o2) \wedge o1.x != p \rightarrow \neg TR'(o1,o2) \wedge o1.x'=p$, where *x* is an attribute of the class *C1*, *o1* is an object of the class *C1*, *o2* is an object of the class *C2* and *p* denotes an untyped value. The symbol *":"* means "by definition". Expression $TR(o1,o2) \wedge o1.x!= p$ is the precondition under which the action can be executed and $\neg TR'(o1,o2) \wedge o1.x'=p$ is the body of the action. The precondition may contain a set of conjunctions and/or disjunctions while the action body may contain a set of conjunctions. Unprimed and primed attributes refer to the values of attributes before and after the execution of the action, respectively. Unprimed and primed temporal relations refer to temporal relations before and after the execution of the action, respectively.

Objects and untyped values that participate in an action are *non-deterministically* selected from those that are suitable. For example, the above action is enabled for all objects having $TR(o1,o2) \wedge o1.x != p$. Semantically, an action is a Boolean expression that is true or false with regard to a pair of states. For example the action *A* defined earlier is true for a pair of states *(S,T)* if and only if the value that state *S* assigns to *x* is different from *p* and *o1* is associated with *o2* through *TR* and the value that state *T* assigns to *x* is equal to *p* and *o1* is not longer associated with *o2* through *TR*. Unlike permanent relations which can be used for many patterns, temporal relations and actions are specific to each pattern.

The sequence of states describing the execution of actions is potentially infinite. Properties of a given system (a set of behaviors) can be divided into *safety* and *liveliness*. Informally safety means that nothing will go wrong with the system while liveliness means that some actions will be executed infinitely. Safety can be guaranteed by ensuring that invariants are true at all states of the system while liveliness is obtained by giving an explicit *fairness* requirement. Marking an action with asterisk *(*)* denotes a fairness requirement, stating that if its precondition is true, the action

will be executed infinitely often [12]. In BPSL, invariants are defined by the keyword "*Invariants:"*, followed by a set of conditions on temporal relations and/or on attribute values. Likewise, initial state is given by the keyword *"Initially:"*, followed by a condition based on temporal relations and/or on attribute values.
A formula in BPSL has the following form:

$$\exists (x_1, \ldots, x_n): \begin{cases} \wedge_i PR_i\ (a_i,b_i)\ \ (Permanent\ \ relations) \\ [\wedge_j TR_j\ (c_j,d_j)\ \ (Temporal\ relations) \\ [Invariants:\ invariant\ conditions] \\ Initially:\ initial\ conditions \\ \vee_k A_k\ (\ldots)\ \ (Actions)] \end{cases}$$

PR_i are permanent relation symbols, TR_j are temporal relation symbols and A_k are action symbols, while $x_1,..,x_n$ are variable symbols representing the pattern primary entities (classes, attributes, methods, objects and untyped values).

In the notation $PR_i\ \ (a_i,b_i)$, a_i and b_i could represent classes, attributes, methods, objects or untyped values as per Table 1, while in the notation $TR_j\ (c_j,d_j)$ c_j and d_j represent classes. The notation $A_k\ (...)$ means that actions can have any number of arguments, which should be either objects or untyped values. Any argument of PR_i, TR_j and A_k must be subset of $\{x_1,..,x_n\}$. Variables $x_1,..,x_n$ are typed, each represents an entity as expected. In all specifications, we follow a convention in which only relations and actions start with a capital letter. Temporal relations, initial conditions, invariants and actions are put between square brackets because they are optional. Patterns that have no significant behavioral aspect are only specified using permanent relations. When patterns have a significant behavioral aspect, initial conditions are compulsory but invariants are optional.

BPSL uses four parts to specify a pattern. The first declares the variables and their type, the second defines permanent relations, the third defines temporal relations and the fourth defines actions. The $\vee$ connective is used to connect actions because the action to be executed is non-deterministically chosen from those enabled (their precondition is evaluated to true). This leads to many possible behaviors (infinite sequence of states).

3. BPSL's Parser

3.1 BPSL's Grammar in ABNF

Table 2 depicts BPSL grammar using ABNF [4]. There are slight changes in this grammar as compared to BPSL's mathematical definition given in the previous section. The colon *(:)* after the declaration of variables is no longer needed. Moreover a dot *(.)* is needed to separate permanent relations, temporal relations, invariants, initial condition and actions.

```
BPSL      = Structural_Part [Behavioral_Part]
Structural_Part = "∃" Declaration Permanent_Relations
Declaration = Variables Domain ";" [Declaration]
Variables = Variable ("," Variables) / "∈")
Variable = ((ALPHA/"_")*(ALPHA/DIGIT/"_"))
ALPHA=%x41-5A/%x61-7A                ; A-Z/a-z
DIGIT=%x30-39                        ;0-9
Domain = "C"/"A"/"M"/"O"/"V"
Permanent_Relations = Primary_PR Participants (("∧" Permananent_Relations )/".")
Primary_PR="Defined-in"/"Reference-to-one"/"Reference-to-many"/"Inheritance"/"Creation"/"Invocation"/"Argument"/"Return-type"/"Instance"
Participants = "("Variable "," Variable ")"
Behavioral_Part = Temporal_Relations [Invariants] Initially Actions
Temporal_Relations = Variable "(" Variable Cardinality "," Variable Cardinality ")" (("∧" Temporal_Relations)/".")
Cardinality = "[" ((DIGIT ".." DIGIT)/"*") "]"
Invariants= "Invariants" ":" Precond "."
Initially= "Initially" ":" Precond "."
Actions = Variable ["*"]"(" Action ":" Precond "→" Result (("∨" Action )/".")
Action=Variable (("," Action)/")")
Precond = ((["¬"] Variable Participants ) / (Variable "." Variable ("="/"!=")Variable)/ (Variable "." Variable ("="/"!=") Variable "." Variable) /( Variable ("="/"!=") Variable "." Variable))  [("∧"/"∨") Precond]
Result = ((["¬"] Variable "'" Participants )/ (Variable "." Variable "'" "=" Variable)/ (Variable "." Variable "'" "=" Variable "." Variable)) ["∧" Result ]
```

Table 2: ABNF Description of BPSL Grammar**.**

3.2 Using JLex to Generate BPSL's Lexer

JLex was developed by Elliot Berk at Princeton University [3]. To install the version of JLex (1.2.5) used, a *Main.java* file needs to be downloaded and compiled to generate a *JLex.Main.class* file along with many *.class* files to be used by the *JLex.Main.class*. JLex can be run using *java JLex.Main filename.lex* where *filename.lex* is a specially formatted specification file containing the details of a lexer. If there are no errors in the *.lex* file a lexer named *filename.lex.java* is generated. To ensure compatibility between JLex and CUP it is required to rename this file to *Yylex.java*. A JLex

specification file is organized into three sections, separated by double-percent directives *("%%")* as follows:

user code
%%
JLex directives
%%
regular expression rules

The user code section is copied directly into the generated Java-based lexer. The JLex directives section can include macros definitions and declare state names. The third section contains the rules of lexical analysis, each of which consists of three parts: an optional state list, a regular expression, and an action. Table 3 depicts the JLex specification of BPSL. It describes the tokens to be passed to the parser by the lexer.

```
package bpsl;
import java_cup.runtime.Symbol;
%%
%cup
%line
%%
"E" { return new Symbol(sym.EXIST); }
"@" { return new Symbol(sym.BELONG); }
"C" { return new Symbol(sym.C); }
"A" { return new Symbol(sym.A); }
"M" { return new Symbol(sym.M); }
"O" { return new Symbol(sym.O); }
"V" { return new Symbol(sym.V); }
";" { return new Symbol(sym.SEMI); }
"[" { return new Symbol(sym.LSQUARE); }
"]" { return new Symbol(sym.RSQUARE); }
"*" { return new Symbol(sym.TIMES); }
"."{2} { return new Symbol(sym.DOTS); }
":" { return new Symbol(sym.COLON); }
"~" { return new Symbol(sym.NOT); }
"v" { return new Symbol(sym.OR); }
"->" { return new Symbol(sym.ARROW); }
"'" { return new Symbol(sym.APOS); }
"=" { return new Symbol(sym.EQUAL); }
"!'"="  { return new Symbol(sym.DIFF); }
"^" { return new Symbol(sym.AND); }
"(" { return new Symbol(sym.LPAREN); }
")" { return new Symbol(sym.RPAREN); }
"," { return new Symbol(sym.COMMA); }
"." { return new Symbol(sym.DOT); }
"Defined-in" { return new Symbol(sym.DEFINED); }
"Reference-to-one" { return new Symbol(sym.REF1); }
"Reference-to-many" { return new Symbol(sym.REFN); }
"Inheritance" { return new Symbol(sym.INHERITANCE); }
"Creation" { return new Symbol(sym.CREATION); }
"Invocation" { return new Symbol(sym.INV); }
"Argument" { return new Symbol(sym.ARG); }
"Return-type" {return new Symbol(sym.RTN);}
"Instance" { return new Symbol(sym.INST); }
"Invariants" {return new Symbol (sym.INVARIANTS);}
"Initially" {return new Symbol (sym.INITIALLY);}
[a-zA-Z_][a-zA-Z_0-9]* { return new Symbol(sym.VAR, new String(yytext())); }
[0-9] { return new Symbol(sym.DIGIT, new String(yytext())); }
[ \t\r\n\f] { /* ignore white space. */ }
. { Editor.Errors.append("Illegal character "+yytext()+" in line "+(Yylex.yyline+1)+"\n"); }
```

Table 3: JLex Specification of BPSL.

The instruction *"package bpsl"* in the user code section makes the generated *Yylex.java* file part of a package called *bpsl*. In the user code section we also import the *Symbol* class, which is defined in the *java_cup.runtime* package. This is done because we need to return objects of this class in the action code of the regular expression section. In the JLex directives section we used *%cup* to activate Java CUP compatibility and *%line* to put the zero-based line index in a variable named *yyline*. The above two directives are turned off by default. In the third section we defined the regular expressions and their corresponding actions that need to be taken in case a match is found. Whenever a keyword terminal is found in the action code we return a new object of the class *Symbol* by providing to the constructor of the class *Symbol* the corresponding *public static final int* data member of the class *sym*.

The last two terminal symbols *VAR* and *DIGIT* are treated specially because they are the only terminal symbols that could take a variety of forms. *VAR* represents the variables declared at the beginning of a BPSL specification while *DIGIT* is used in the cardinality of temporal relations. *[a-zA-Z_][a-zA-Z_0-9]** means that variables declared in BPSL must start with a character or underscore then they can have any number of characters, numbers or underscores. If a variable or a digit is found, the constructor of the class *Symbol* is called with two parameters, the first being the same as the other keyword terminals and the second being the return value of the method *yytext()* defined in the generated *Yylex* class after executing JLex. In our case, the method *yytext()* returns a string representing the variable or digit defined. If an illegal character is typed then *Editor.Errors.append("Illegal character "+yytext()+" in line "+(Yylex.yyline+1)+"\n");* is called. *Errors* is a Java *non-editable TextArea* defined in the editor of BPSL's parser to display error messages. In Table 3, the following non-ASCII keyword symbols of BPSL ABNF grammar were replaced as follows*: "∃" by "E", "∈" by "@", "∧" by "^", "¬" by "~", "→" by "->" and "∨" by "v"*.

3.3 Using CUP to Generate BPSL's Parser

CUP was originally written by Scott Hudson while at Princeton University [8]. The version of CUP used is 0.10k and can be downloaded as a compressed tar file. After this, *java_cup* and its runtime system need to be compiled from the installation directory using *javac java_cup/*.java java_cup/runtime/*.java*. Java CUP can be run using *java java_cup.Main filename.cup* where *filename.cup* is the grammar of the target language written using CUP grammar specification syntax. If there are no errors in the *.cup* file then two files will be generated, the parser called *parser.java* and a public class called *sym* which has as *public static final int* data members all the terminal symbols of the grammar of the target language. A CUP specification file is organized into five sections as follows:

- Package and import specifications (optional).
- User code components (optional).
- Symbol (terminal and non-terminal) list.
- Precedence and associativity of terminals declaration (optional).
- The grammar.

Any *package* or *import* declaration that appear in the specification will also appear in the source code of the generated parser *(parser.java)*. The section *"user code components"* contains an optional *action code {: ...:}* that allows user code (variables and routines inserted between *{:* and *:}*) to be included as part of the generated parser file (*parser.java*) in a separate non-public class named *CUP$parser$actions*. In our case, all developed routines for lexical and semantic analysis of BPSL specifications and for conversion of BPSL behavioral specifications to FSP have been included in the action code of BPSL's CUP specification file, which is not shown here because it spans over six pages. The section *"user code components"* also contains an optional *parser code* declaration. This declaration allows methods and variables to be placed directly within the generated parser class. In our case, we put the main function of the parser in the parser code section. The symbol list declarations are responsible for naming and supplying a type for each terminal and non-terminal symbol that appears in the grammar. Each terminal and non-terminal symbol is represented at runtime with a *Symbol* object. In the case of terminals, these are returned by the lexer and placed on the parse stack. In the case of non-terminals these replace a series of *Symbol* objects on the parse stack whenever the right hand side of some production is recognized. The section *"precedence and associativity of terminals"* is useful for parsing ambiguous grammars. Each production in the grammar has a left hand side non-terminal followed by the symbol *"::=",* which is then followed by a series of zero or more actions (in the form of action code), terminal, or non-terminal symbols, followed by an optional contextual precedence assignment, and terminated with a semicolon. Each symbol on the right hand side can optionally be labeled with a name. Label names appear after the symbol name separated by a colon *(:)*. Label names must be unique within the production, and can be used within action code to refer to the value of the symbol. If there are several productions for the same non-terminal they may be declared together. In this case the productions start with the non-terminal and *"::="*. This is followed by multiple right hand sides each separated by a bar *(|)*. The full set of productions is then terminated by a semicolon. Actions appear in the right hand side as code strings (Java code inside *{: ... :}* delimiters). These are executed by the parser at the point when the portion of the production to the left of the action has been recognized (note that the lexer will have returned the token one past the point of the action since the parser needs this extra *look-ahead* token for recognition).

Besides the action code (outside and inside the production rules), the CUP specification of BPSL grammar is simply a translation of the ABNF grammar shown in Table 2 with some minor modifications. Firstly, since CUP grammar does not directly support optional sequence, *[f]* is replaced by *f|;* (*f* or nothing). Secondly, since CUP grammar does not directly support sequence grouping *A= a (b c | d)* is replaced by *A::= a B;* and *B::= b c | d;*

3.4 Lexical and Semantic Analysis

Following are the lexical and semantic cases that have to be checked in a BPSL specification:

- A variable cannot be declared more than once in the declaration part.
- Variables participating in permanent relations should belong to correct domain as follows: *Define-in (M∪A,C)*, *Reference-to-one(-many) (C,C)*, *Inheritance (C,C)*, *Creation(M∪C,C)*, *Invocation (M∪C,M∪C)*, *Argument (C∪A∪V,M)*, *Return-type (C∪O,M)* and *Instance(O,C)*.
- Variables participating in temporal relations should be declared as classes.
- Variables participating in actions should be declared as objects or untyped values.
- Variables participating in temporal relations (when used in actions) should be declared as objects or classes.
- If a *"."* (used to access attributes) appears in the precondition or body of an action then the following two cases are to be considered:

a) If the *"."* appears in a precondition, the domain of a variable depends on the following cases:

- *Variable "." Variable "="/"!=" Variable*. In this case the first variable should be an object, the second an attribute, and the third an untyped value.
- *Variable "." Variable "="/"!=" Variable "." Variable*. In this case the first variable should be an object, the second an attribute, the third an object and the fourth an attribute.
- *Variable "="/"!=" Variable "." Variable*. In this case the first variable should be an untyped value, the second an object and the third an attribute.

b) If the *"."* appears in the action body, the domain of a variable depends on the following cases:

- *Variable "." Variable "" "=" Variable*. In this case the first variable should be an object, the second an attribute and the third an untyped value.
- *Variable "." Variable "" "=" Variable "." Variable*. In this case the first variable should be an object, the second an attribute, the third an object and the fourth an attribute.

4. Converting BPSL Behavioral Specifications to FSP

A process is the execution of a sequential program. A process has a state modified by indivisible or atomic actions. Each action causes the transition from the current state to the next state. The order in which actions are allowed to occur is determined by a transition graph. Processes can thus be modeled as abstract state machines. Figure 1 depicts the state machine for a light switch that has the actions *on* and *off*. The initial state is always numbered 0 and transitions are always drawn in clockwise direction. In Figure 1, *on* causes a transition from state 0 to state 1 and *off* causes a transition from state 1 to state 0. This form of state machine is known as a LTS, because transitions are labeled with action names.

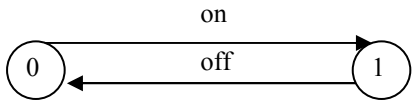

Figure 1: *Light Switch* State Machine**.**

Diagrams of this form can be displayed in LTSA. Although this representation of a process is finite, the behavior described need not be finite. For example the state machine of Figure 1 allows the following sequence of actions: *on->off->on->off->on->off->...* A form of this notation has been adopted in the Unified Modeling Language (UML) [14]. The graphical form of state machine description is excellent for simple processes however it becomes unreadable for large numbers of states and transitions. For this reason a simple algebraic notation called FSP is used to describe process models. Every FSP description has a corresponding state machine (LTS) description. Technically, FSP is a process calculus, one of a family of notations pioneered by Mihner's Calculus of Communicating Systems (CCS) [13] and Hoare's Communicating Sequential Processes (CSP) [7], for concisely describing and reasoning about concurrent programs. The difference from these notations is largely syntactic: FSP is designed to be easily machine-readable.

If *x* is an action and *P* a process then action prefix *(x->P)* describes a process that initially engages in

the action *x* and then behaves exactly as described by *P*. The action prefix operator "->" always has an action on its left and a process on its right. In FSP, identifiers beginning with a lowercase letter denote actions and identifiers beginning with an uppercase letter denote processes.

Repetitive behavior is described in FSP using recursion. The following FSP process describes the light switch of Figure 1.

SWITCH=OFF,
OFF=(on->ON),
ON=(off->OFF).

The process definitions for *ON* and *OFF* are part of and local to the definition for *SWITCH* (as indicated by *","*). In Figure 1, *OFF* defines state 0 and *ON* defines state 1. A more concise definition of *SWITCH* can be achieved by substituting the definition of *ON* in the definition of *OFF* as follows: *SWITCH=(on->off->SWITCH)*.

LTSA can draw state machines that correspond to any FSP definition. The definitions may also be animated using the LTSA animator to produce a sequence of actions. The animator lets the user control the actions offered by a model to its environment. Those actions that can be chosen for execution are ticked. We refer to the sequence of actions produced by the execution of a process as a *trace*.

If *x* and *y* are actions then *(x->P|y->Q)* describes a process which initially engages in either of the actions *x* or *y*. After the first action has occurred, the subsequent behavior is described by *P* if the first action was *x* and *Q* if the first action was *y*. The following example describes a drinks dispensing machine which dispenses hot coffee if the red button is pressed and iced tea if the blue button is pressed. Figure 2 depicts the graphical state machine description of the drinks dispenser. Choice is represented as a state with more than one outgoing transition. The order of elements of choice has no significance.

DRINKS=(red->coffee->DRINKS
|blue->tea->DRINKS).

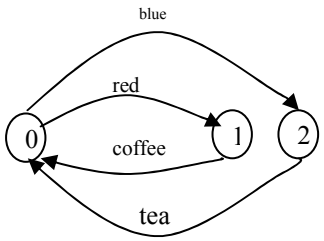

Figure 2: *DRINKS* State Machine.

The basic idea behind converting BPSL behavioral specifications to FSP is to consider objects participating in a temporal relation as being in one state. We follow a convention in which objects participate in at most one temporal relation at all times. Therefore, in all actions, we only allow a single temporal relation or disjunctions of temporal relations in preconditions and action bodies. As such, converting a BPSL behavioral specification to FSP is possible only if the above convention is satisfied. We call this convention the *non-cumulative* property of temporal relations as opposed to the *cumulative* property in which objects can participate in more than one temporal relation at a given time.

In order to handle the *cumulative* property of temporal relations, one could think of conjunctions of temporal relations as a super-state, made of each temporal relation conjunct (sub-states). However, this is not possible for two reasons:

- Some sub-states of a given super-state will be unrelated (they are not linked by an action). This does not comply with the reachability property of statecharts.
- Some sub-states will belong to more than one super-state (overlapping) which is not permitted in statecharts.

This shows indeed that the semantic powerfulness of BPSL in specifying the behavioral aspect of patterns as compared with FSP.

The following are the rules used for converting BPSL behavioral specifications to FSP:

- Ignore all parts of preconditions and action bodies that contain class attributes, as temporal relations are converted to states. The filename of the BPSL specification becomes the name of the main process (written in uppercase).
- The precondition of the first action becomes the initial state (local process)
- Each action definition is converted to FSP as follows:
    - The precondition becomes the local process (current state in uppercase).
    - The action name becomes the enabling action (in lowercase).
    - The action body becomes the next local process (next state in uppercase).
    - If there are many current states that appear in many actions then they are grouped together in the same local process (state) specification with an | between them.

5. Running BPSL's Parser

The following are the commands typed at the prompt (*c:\cup\bpsl*):

```
java JLex.Main bpsl.lex
rename bpsl.lex.java Ylex.java
java java_cup.Main bpsl.cup
javac *.java c:\cup\java_cup\runtime\lr_parser.java
java bpsl.parser
```

The first two commands create the *Ylex.java* file. The third command creates the file *sym.java* and *parser.java*. The fourth command compile all **.java* files (including the *editor.java* file), and the file *lr_parser.java* (the parser generator) because we have made changes in it in order to display syntactic errors in our editor. The fifth and final command launch BPSL's parser.

In the Appendix, we have shown the results of converting BPSL's behavioral specification of the *Observer* Pattern [6] into FSP notation, its LTS graph and how the behavior of the pattern was traced using LTSA tool.

6. Conclusion

BPSL is a formal specification language for accurately describing the structural as well as the behavioral aspects of patterns. In this paper, we have described how JLex and CUP were successfully used to generate highly optimized Java-based lexer and parser for BPSL. JLex was used to generate a lexer being able to tokenize an input file (BPSL specification file) and pass those tokens to the parser (generated by CUP) that checks them against the grammar of BPSL defined using CUP grammar specification. In addition to the syntactic checking done through the grammar, we have added action code to check for lexical and semantic errors in a BPSL specification. Moreover we are have added action code to convert the behavioral aspect specification into FSP notation. This has enabled us to use LTSA, to automatically check the safety, liveliness (progress) and reachability properties as well as the drawing the state machine of the behavioral aspect of a pattern and tracing it in order to check its correctness.

References

1. A. Aho, R. Sethi, and J. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Publishing: Reading, MA, 1986.
2. J. Barwise (Ed.), An Introduction to First-Order Logic, Handbook of Mathematical Logic, pp. 5-46, North-Holland, 1977.
3. E. Berk, *Jlex,* "A lexical analyzer generator for Java: version 1.2", Department of computer science, Princeton University, Tech. Rep., 2000.
4. D. Crocker and P. Overell, "Augmented BNF for syntax specifications: ABNF", RFC 2234, IETF Network Working Group, 1997.
5. C. Fischer and R. LeBlanc, Crafting a Compiler with C, Benjamin/Cummings Publishing, 1991.
6. E. Gamma, R. Helm, R Johnson. and J. Vlissides, Design Patterns: Elements Of Reusable Object-Oriented Systems, Addison-Wesley, 1995.
7. C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall International Series in Computer Science, 1985.

8. S.E. Hudson, "CUP, LALR parser generator for Java", Graphics Visualization and Usability Center, Georgia Institute of Technology, Tech. Rep., 1999.
9. L. Lamport, "The temporal logic of actions", ACM transactions on Programming Languages and Systems, Vol. 16, No.3, pp.872-923, 1994.
10. J. R. Levine, T. Mason and D. Brown, Lex and Yacc, O'Reilly, 1992.
11. J. Magee and J. Kramer, Concurrency, State Models and Java Programs, John Wiley, 1999.
12. T. Mikkonen, "Formalizing Design Patterns", in Proc. of ICSE'98, IEEE Computer Society Press, pp. 115-124.
13. R. Milner, Communication and Concurrency, Prentice-Hall International Series in Computer Science, 1989.
14. J. Rumbaugh, I. Jacobson and G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1998.
15. T. Taibi and D.C.L. Ngo, "Why and How Should Patterns Be Formalized", Journal of Object-Oriented Programming (JOOP), Vol. 14, No. 4, pp.8-9, 2001.
16. T. Taibi and D.C.L. Ngo, "Formal Specification of Design Patterns-A Balanced Approach", Journal of Object Technology (JOT), Vol. 2, No 04, pp. 127-140, 2003.
17. T. Taibi, D.C.L. Ngo, "Formal Specification of Design Pattern Combination Using BPSL", Elsevier Journal of Information and Software Technology, Vol. 45, No 03, pp. 157-170, 2003.

Appendix (Sample Screens)

Figure A.1: Sample Screen of BPSL's Parser (with the Formal Specification of the *Observer* Pattern)

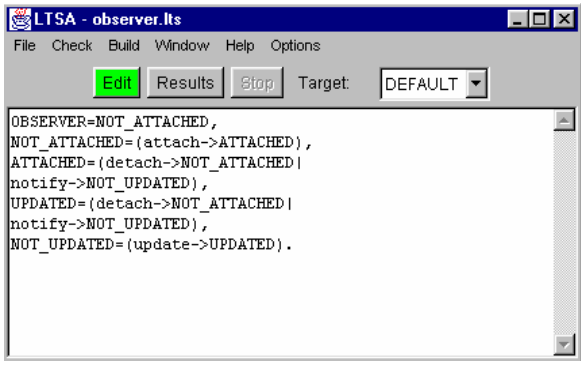

Figure A.2: Generated FSP Specification from BPSL's Behavioral Specification of the *Observer* Pattern in LTSA

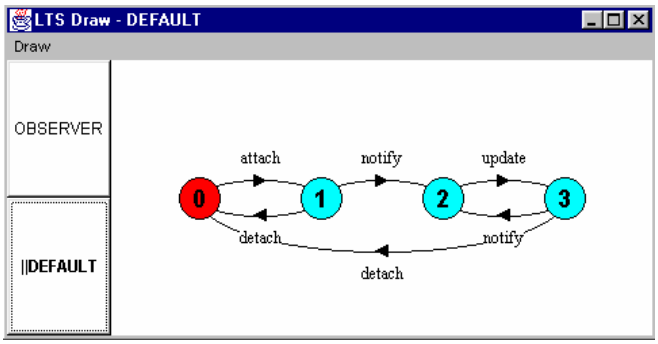

Figure A.3: State Machine of the *Observer* Pattern in LTSA

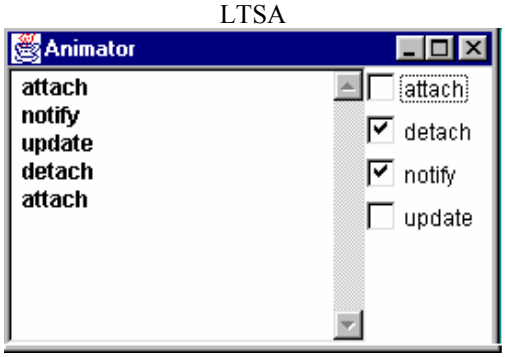

Figure A.4: Trace of the Behavioral Aspect of the *Observer* Pattern Using the LTSA Animator Window