The Xtext generated parsers to specify patterns and pattern languages: PSL and PLSL

Alireza Rouhi
Bahman Zamani
The Xtext generated parsers to specify patterns and pattern languages: PSL and PLSL

Alireza Rouhi and Bahman Zamani
Department of Software Engineering
University of Isfahan
Isfahan, Iran.
{rouhi, zamani}@eng.ui.ac.ir

Abstract: In this report, we aim at presenting the Xtext generated parsers named the Pattern Specification Language (PSL) and the Pattern Language Specification Language (PLSL) parsers to facilitate the tedious and error-prone process of writing and validating the specification of design patterns and PLs in our presented pattern specification scheme and PL formalism in UI-SE-MDSERG-2015-01. Utilizing the generated tools, we revised our earlier presented specification of the patterns of the Broker PL in UI-SE-MDSERG-2015-01 as a case study. The good news is that several typing and logical errors have been found and corrected by using the produced PSL parser. In this report, we present the Broker PL specification as a case-study to show the applicability of the generated PSL and PLSL parsers. Finally, the Xtext source code representing the PSL and PLSL parsers are presented.
The Xtext generated parsers to specify patterns and pattern languages: PSL and PLSL

Alireza Rouhi and Bahman Zamani
Department of Software Engineering
University of Isfahan
Isfahan, Iran.
{rouhi, zamani}@eng.ui.ac.ir

Abstract: In this report, we aim at presenting the Xtext generated parsers named the Pattern Specification Language (PSL) and the Pattern Language Specification Language (PLSL) parsers to facilitate the tedious and error-prone process of writing and validating the specification of design patterns and PLs in our presented pattern specification scheme and PL formalism in UI-SE-MDSERG-2015-01. Utilizing the generated tools, we revised our earlier presented specification of the patterns of the Broker PL in UI-SE-MDSERG-2015-01 as a case study. The good news is that several typing and logical errors have been found and corrected by using the produced PSL parser. In this report, we present the Broker PL specification as a case-study to show the applicability of the generated PSL and PLSL parsers. Finally, the Xtext source code representing the PSL and PLSL parsers are presented.

Contents

1 Introduction ........................................... 2
   1.1 Xtext ............................................ 2
   1.2 Eclipse Modeling Framework (EMF) Metamodeling .......... 3

2 The PSL parser in action ........................... 4
   2.1 The Broker Pattern ............................... 4
   2.2 The Wrapper Facade Pattern ..................... 8
   2.3 The Reactor Pattern ............................. 9
1 Introduction

To write and validate a given Pattern Language (PL) and its patterns with our presented formalism of a PL in [RZ16], we developed two languages named PLSL and PSL, respectively, to specify a given PL and its patterns. These languages have been generated automatically using the Xtext parser generator tool.

In this report, first we present a brief introduction to Xtext [Bet13] and EMF [SBMP08, Gro09]. This materials helps a designer to understand the specification of patterns and PLs in PSL and PLSL. As a case study, we present the PSL listings of the patterns of the Broker PL. Finally, we present the Xtext source code representing the PSL and PLSL parsers in Section A and Section B, respectively.

1.1 Xtext

Xtext is an Eclipse-based open-source framework to develop Domain-Specific Languages (DSLs) [Bet13] by a programmer in a quick manner. One of the most important and valuable advantages of using Xtext is its tool support on automatic generation of Eclipse-based Integrated Development Environment (IDE) plug-ins to deploy the produced DSL artifacts [EB10].

Since, Xtext supports a grammar notation similar to Extended BNF (EBNF) as its input which nearly matches with the revised and extended Graphic extension
of EBNF (GEBNF) notation, translating the proposed formal model of the pattern specification scheme [RZ16] to the Xtext’s input format is performed easily. Listing 10 in Section A demonstrates the PSL’s parser generation code in Xtext. Additionally, Listing 11 in Section B demonstrates the PSL’s parser generation code in Xtext. See [RZ15] for more information about the syntax definition and programming as well as running a sample example in Xtext.

In the following, we present a brief introduction to the EMF metamodeling.

1.2 EMF Metamodelling

A model is defined as an abstraction of a real system. A metamodel itself is a language or a model with a higher abstraction level than a model which conforms to the metamodel [JB06, BCW12]. A model conformance to a metamodel is much like as a program script respecting its grammar specification. These two development paradigms are known as Modelware and Grammarware, respectively [PKP14].

Xtext is compatible with the Eclipse EMF. EMF can be used to define a domain model in general. Here, to have a background in the EMF tool suite, we present a brief introduction [SBM08, Gro09].

The EMF metamodel itself has two base metamodels: the Ecore model and the Genmodel model. The Ecore model contains information regarding each model’s defined classifiers. In contrast, the Genmodel provides the required additional and control information for the code generation and direction; for example, the path of the stored model and related files information.

Each Ecore model defines the following elements:

- **EClass**: a class representation which can contain zero or more attributes as well as zero or more references.

- **EAttribute**: a named attribute representation which can have a type too.

- **EReference**: an association end representation between two classes. A reference can be a containment of or be contained in other class which is indicated by setting a flag.

- **EDatatype**: an attribute type representation, for example, int and boolean for primitive types and java.util.Date for an object type.
2 The PSL parser in action

In this section, the revised and validated version of patterns of the Broker PL [BHS07] are presented.

2.1 The Broker Pattern

Figure 1 and Figure 2 show the class diagram of the Broker pattern (changed from [BMR+96]) and a simple scenario of its sequence diagram, respectively.

Figure 3 displays the PSL Editor usage in practice. Listing 1 demonstrates the Broker pattern PSL code.

```
patternName: Broker

components:
{Broker, Client, C_Proxy, C_API, Bridge, S_API, S_Proxy, Server} SUBSETEQ classes();
call_server() IN opers(Client);
{send_request(), cpreturn()} SUBSETEQ opers(C_Proxy);
call_server() IN opers(C_API);
{find_server(), forward_request(), find_client(), forward_response()} SUBSETEQ opers(Broker);
{forward_message(), transmit_message()} SUBSETEQ opers(Bridge);
register_service() IN opers(S_API);
{call_service(), send_response()} SUBSETEQ opers(S_Proxy);
run_service() IN opers(Server);
```
Figure 2: The Broker pattern sequence diagram (changed from [BMR\textsuperscript{*}96])
Figure 3: The PSL Editor

```plaintext
staticConditions:
associates(Client, C_Proxy) ~
associates(C_Proxy, Client) ~
associates(Server, S_Proxy) ~
associates(S_Proxy, Server) ~
associates(Broker, Broker) ~
associates(Broker, Bridge) ~
associates(Bridge, Broker) ~
associates(Bridge, Bridge) ~
associates(C_Proxy, Broker) ~
associates(Broker, C_Proxy) ~
associates(S_Proxy, Broker) ~
associates(Broker, S_Proxy) ~
depends(Client, C_API) ~
depends(Server, S_API) ~
composes(Broker, C_API) ~
composes(Broker, S_API) ~
isInterface(C_API) ~
isInterface(S_API) ~

dynamicConditions:
FORALL M: Message @
(M IN msgs()) ~
sig(M) = call_server() ~
name(c(name(lifeline(sender(M)))) = Client ~
```
name(c(name(lifeline(receiver(M)))))) = Client

=>

(EXISTS M_c: Message | (M_c IN msgs())

  sig(M_c) = send_request() ~
  name(c(name(lifeline(sender(M_c)))))) = Client ~
  name(c(name(lifeline(receiver(M_c)))))) = C_Proxy

  => (precedes(M_5, M_c) ~ calls(call_server(), send_request())) ~

EXISTS M_b: Message | (M_b IN msgs())

  sig(M_b) = forward_request() ~
  name(c(name(lifeline(sender(M_b)))))) = C_Proxy ~
  name(c(name(lifeline(receiver(M_b)))))) = Broker) =>

  (precedes(M_c, M_b) ~
    calls(send_request(), forward_request()) ~

EXISTS M_sp: Message | (M_sp IN msgs())

  sig(M_sp) = call_service() ~
  name(c(name(lifeline(sender(M_sp)))))) = Broker ~
  name(c(name(lifeline(receiver(M_sp)))))) = S_Proxy) =>

  (precedes(M_c, M_sp) ~
    calls(forward_request(), call_service()) ~

EXISTS M_s: Message | (M_s IN msgs())

  sig(M_s) = run_service() ~
  name(c(name(lifeline(sender(M_s)))))) = S_Proxy ~
  name(c(name(lifeline(receiver(M_s)))))) = Server) =>

  (precedes(M_sp, M_s) ~
    calls(call_service(), run_service()) ~

EXISTS M_rsp: Message | (M_rsp IN msgs())

  sig(M_rsp) = send_response() ~
  name(c(name(lifeline(sender(M_rsp)))))) = Server ~
  name(c(name(lifeline(receiver(M_rsp)))))) = S_Proxy) =>

  (precedes(M_sp, M_rsp) ~
    calls(run_service(), send_response()) ~

EXISTS M_rb: Message | (M_rb IN msgs())

  sig(M_rb) = forward_response() ~
  name(c(name(lifeline(sender(M_rb)))))) = S_Proxy ~
  name(c(name(lifeline(receiver(M_rb)))))) = Broker) =>

  (precedes(M_rsp, M_rb) ~
    calls(send_response(), forward_response()) ~

EXISTS M_rbb: Message | (M_rbb IN msgs())

  sig(M_rbb) = find_client() ~
  name(c(name(lifeline(sender(M_rbb)))))) = Broker ~
  name(c(name(lifeline(receiver(M_rbb)))))) = Broker) =>

  precedes(M_rb, M_rbb) ~

EXISTS M_rcc: Message | (M_rcc IN msgs())

  sig(M_rcc) = cpreturn() ~
  name(c(name(lifeline(sender(M_rcc)))))) = Broker ~
  name(c(name(lifeline(receiver(M_rcc)))))) = C_Proxy) =>
2.2 The Wrapper Facade Pattern

Figure 4 and Figure 5 show the class diagram and sequence diagram representing this pattern, respectively.

Listing 2 demonstrates the Wrapper Facade pattern PSL code.

```plaintext
patternName:
WrapperFacade
components:
{WrapperFacade, Client, Functions} SUBSETEQ classes();
do_something() IN opers(Client);
service() IN opers(WrapperFacade);
func() IN opers(Functions);
staticConditions:
depends(Client, WrapperFacade) ~
depends(WrapperFacade, Functions)
// Each WrapperFacade must reference at least one Functions class
FORALL WR: WrapperFacade =>
EXISTS F: Functions => depends(WR, F)
```

Listing 1: The Broker pattern specification in PSL

```
<precedes(M_rbb, M_rcp) ~
calls(find_client(), cpreturn())
}
```
2.3 The Reactor Pattern

Figure 6 and Figure 7 show the class diagram and sequence diagram representing this pattern, respectively.

Listing 3 demonstrates the Reactor pattern PSL code.

Figure 6: The Reactor pattern class diagram (revised from [Sch95, BHS07])
Figure 7: The Reactor pattern sequence diagram (revised from [Sch95, BHS07])

```plaintext
patternName: Reactor

components:

{Reactor, AbstractEventHandler, ConcreteEventHandler} SUBSETEQ classes();
{registerHandler(), removeHandler(), dispatch(), waitForEvents()} SUBSETEQ opers(Reactor);
{handleEvent(), getHandle(), handleClose()} SUBSETEQ opers(AbstractEventHandler);
{handleEvent(), getHandle(), handleClose()} SUBSETEQ opers(ConcreteEventHandler);

staticConditions:

isAbstract(AbstractEventHandler) ~ depends(Reactor, ConcreteEventHandler) ~ aggregates(Reactor, AbstractEventHandler) ~ inherits(ConcreteEventHandler, AbstractEventHandler) ~

// Each abstract AbstractEventHandler must be inherited by at least one non-abstract ConcreteEventHandler
FORALL AEH: AbstractEventHandler @ isAbstract(AEH) => EXITS CEH: ConcreteEventHandler => (~isAbstract(CEH) ~ inherits(CEH, AEH))

dynamicConditions:

FORALL M: Message @
(M IN msgs()) ~
```

name(c(name(lifeline(sender(M))))) = Client ^
 name(c(name(lifeline(receiver(M))))) = Reactor ^
 sig(M) = registerHandler()

=>

EXISTS MC_2: Message |
 (MC_2 IN msgs() ^
  sig(MC_2) = getHandle() ^
  name(c(name(lifeline(sender(MC_2))))) = Reactor ^
  name(c(name(lifeline(receiver(MC_2))))) = ConcreteEventHandler) =>
  precedes(M, MC_2)

FORALL M: Message @
 (M IN msgs() ^
  name(c(name(lifeline(sender(M))))) = Client ^
  name(c(name(lifeline(receiver(M))))) = Reactor ^
  sig(M) = dispatch())

=>

EXISTS MR_2: Message |
 (MR_2 IN msgs() ^
  sig(MR_2) = waitForEvents() ^
  name(c(name(lifeline(sender(MR_2))))) = Reactor ^
  name(c(name(lifeline(receiver(MR_2))))) = Reactor) =>
  precedes(M, MR_2)

FORALL M: Message @
 (M IN msgs() ^
  name(c(name(lifeline(sender(M))))) = Reactor ^
  name(c(name(lifeline(receiver(M))))) = ConcreteEventHandler ^
  sig(M) = handleEvent())

=>

EXISTS MC_2: Message |
 (MC_2 IN msgs() ^
  sig(MC_2) = handleClose() ^
  name(c(name(lifeline(sender(MC_2))))) = Reactor ^
  name(c(name(lifeline(receiver(MC_2))))) = ConcreteEventHandler) =>
  precedes(M, MC_2)

FORALL M_1, M_2: Message @
 (M_1 IN msgs() ^
  M_2 IN msgs()) @
  (sig(M_1) = handleClose() ^
  sig(M_2) = removeHandler())

=>

precedes(handleClose(), removeHandler())

Listing 3: The Reactor pattern specification in PSL
2.4 The Acceptor and Connector Pattern

Figure 8, Figure 9, and Figure 10 show the class diagram and sequence diagrams representing this pattern, respectively.

Listing 4 demonstrates the Acceptor and Connector pattern PSL code.

```plaintext
patternName: AcceptorConnector

components: {Connector, Client, CServiceHandler, Reactor, Server, Acceptor, SServiceHandler} SUBSETEQ classes();
connect() IN opers(Client);
{connect(), handleEvents(), complete()} SUBSETEQ opers(Connector);
{open(), getHandle(), handleEvent(), service()} SUBSETEQ opers(CServiceHandler);
{select(), registerHandler(), handleEvents()} SUBSETEQ opers(Reactor);
{accept(), makeSrvHandler(), acceptSrvHandler(), operation(), handleEvent(), handleClose()} SUBSETEQ opers(Acceptor);
{open(), activateSrvHandler(), getHandle(), handleEvent(), service(), handleClose()} SUBSETEQ opers(SServiceHandler);
open() IN opers(Server);

staticConditions:
```

Figure 8: The Acceptor and Connector pattern class diagram (revised from [Sch96, BHS07])
Figure 9: The Acceptor pattern sequence diagram (adapted from [Sch96, BHS07])
Figure 10: The Connector pattern sequence diagram with the asynchronous initialization (adapted from [Sch96, BHS07])
depends(Client, Reactor) ~
depends(Client, Connector) ~
depends(Connector, Reactor) ~
depends(Connector, Acceptor) ~
depends(Server, Reactor) ~
depends(Server, Acceptor) ~
associates(CServiceHandler, Reactor) ~
associates(CServiceHandler, SServiceHandler) ~
associates(Acceptor, Reactor) ~
associates(SServiceHandler, Reactor) ~
associates(Acceptor, SServiceHandler)

dynamicConditions:

// Acceptor
(FORALL M: Message @
 (M IN msgs() ~
  name(c(name(lifeline(sender(M)))))) = Server ~
  name(c(name(lifeline(receiver(M)))))) = Acceptor ~
  sig(M) = open())
=>
((EXISTS MR_2, MA_3: Message |
  ({MR_2, MA_3} SUBSETEQ msgs() ~
   sig(MR_2) = registerHandler() ~
   sig(MA_3) = getHandle() ~
   name(c(name(lifeline(sender(MR_2)))))) = Acceptor ~
   name(c(name(lifeline(receiver(MR_2)))))) = Reactor ~
   name(c(name(lifeline(sender(MA_3)))))) = Reactor ~
   name(c(name(lifeline(receiver(MA_3)))))) = Acceptor) =>
  (precedes(M, MR_2) ^ precedes(MR_2, MA_3)) ~
(EXISTMR_4, MA_5: Message |
  ({MR_4, MA_5} SUBSETEQ msgs() ~
   sig(MR_4) = handleEvents() ~
   sig(MA_5) = select() ~
   name(c(name(lifeline(sender(MR_4)))))) = Server ~
   name(c(name(lifeline(receiver(MR_4)))))) = Reactor ~
   name(c(name(lifeline(sender(MA_5)))))) = Reactor ~
   name(c(name(lifeline(receiver(MA_5)))))) = Reactor) =>
  (precedes(M, MR_4) ^ precedes(MR_4, MR_5)) ~
(EXISTSM_6, MA_7, MA_8, MA_9: Message |
  ({MA_6, MA_7, MA_8, MA_9} SUBSETEQ msgs() ~
   sig(MA_6) = handleEvent() ~
   sig(MA_7) = makeSrvHandler() ~
   sig(MA_8) = acceptSrvHandler() ~
   sig(MA_9) = activateSrvHandler() ~
   name(c(name(lifeline(sender(MA_6)))))) = Reactor ~
   name(c(name(lifeline(receiver(MA_6)))))) = Acceptor ~
\[
\begin{align*}
\text{name}(c(name(lifeline(sender(MA_7))))) &= \text{Accepter} \land \\
\text{name}(c(name(lifeline(receiver(MA_7))))) &= \text{Accepter} \land \\
\text{name}(c(name(lifeline(sender(MA_8))))) &= \text{Accepter} \land \\
\text{name}(c(name(lifeline(receiver(MA_8))))) &= \text{Accepter} \land \\
\text{name}(c(name(lifeline(sender(MA_9))))) &= \text{Accepter} \land \\
\text{name}(c(name(lifeline(receiver(MA_9))))) &= \text{SServiceHandler} \Rightarrow \\
\text{precedes}(MR_5, MA_6) \land \\
\text{precedes}(MA_6, MA_7) \land \\
\text{precedes}(MA_7, MA_8) \land \\
\text{precedes}(MA_8, MA_9) \\
\text{EXISTS} MR_{10}, MSS_{11}, MSS_{12}, MSS_{13}, MSS_{14}, MA_{15}: Message \mid \\
\{\{MR_{10}, MSS_{11}, MSS_{12}, MSS_{13}, MSS_{14}, MA_{15}\} \subseteq \text{msgs}\} \land \\
\text{sig}(MR_{10}) = \text{registerHandler} \land \\
\text{sig}(MSS_{11}) = \text{getHandle} \land \\
\text{sig}(MSS_{12}) = \text{service} \land \\
\text{sig}(MSS_{13}) = \text{service} \land \\
\text{sig}(MSS_{14}) = \text{handleClose} \land \\
\text{sig}(MA_{15}) = \text{handleClose} \land \\
\text{name}(c(name(lifeline(sender(MR_{10})))))) = \text{SServiceHandler} \land \\
\text{name}(c(name(lifeline(receiver(MR_{10})))))) = \text{Reactor} \land \\
\text{name}(c(name(lifeline(sender(MSS_{11})))))) = \text{Reactor} \land \\
\text{name}(c(name(lifeline(receiver(MSS_{11})))))) = \text{SServiceHandler} \land \\
\text{name}(c(name(lifeline(sender(MSS_{12})))))) = \text{Reactor} \land \\
\text{name}(c(name(lifeline(receiver(MSS_{12})))))) = \text{SServiceHandler} \land \\
\text{name}(c(name(lifeline(sender(MSS_{13})))))) = \text{Reactor} \land \\
\text{name}(c(name(lifeline(receiver(MSS_{13})))))) = \text{SServiceHandler} \land \\
\text{name}(c(name(lifeline(sender(MSS_{14})))))) = \text{Reactor} \land \\
\text{name}(c(name(lifeline(receiver(MSS_{14})))))) = \text{SServiceHandler} \land \\
\text{name}(c(name(lifeline(sender(\text{MA}_{15})))))) = \text{Reactor} \land \\
\text{name}(c(name(lifeline(receiver(\text{MA}_{15})))))) = \text{SServiceHandler} \land \\
\text{precedes}(MA_9, MR_{10}) \land \text{precedes}(MR_{10}, MSS_{11}) \land \\
\text{precedes}(MSS_{11}, MSS_{12}) \land \text{precedes}(MSS_{12}, MSS_{13}) \land \\
\text{precedes}(MSS_{13}, MSS_{14}) \land \text{precedes}(MSS_{14}, MA_{15})) \\
\text{\langle EXIST}\} \\
\langle FORALL M: Message \mid \\
\langle M \in \text{msgs}\rangle \land \\
\text{name}(c(name(lifeline(sender(M))))) = \text{Client} \land \\
\text{name}(c(name(lifeline(receiver(M))))) = \text{Connector} \land \\
\text{sig}(M) = \text{connect}() \rangle \\
\Rightarrow \\
\langle \langle EXIST\} \\
\langle FORALL MR_2: Message \mid \\
\langle MR_2 \in \text{msgs}\rangle \land \\
\text{sig}(MR_2) = \text{registerHandler} \land \\
\text{name}(c(name(lifeline(sender(MR_2))))) = \text{Connector} \land \\
\text{name}(c(name(lifeline(receiver(MR_2))))) = \text{Reactor} \rangle \\
\Rightarrow \\
\text{precedes}(M, MR_2) \\
\langle EXIST\} \\
\langle FORALL MR_3, MR_4, MC_5: Message \mid \\
\langle \{MR_3, MR_4, MC_5\} \subseteq \text{msgs}\rangle \land \\
\text{sig}(MR_3) = \text{handleEvents} \\
\text{\rangle}
\end{align*}
\]
\( \text{sig}(\text{MR}_4) = \text{select()} \) ^
\( \text{sig}(\text{MC}_5) = \text{handleEvent()} \) ^
\( \text{name}(\text{c(name(lifeline(sender(\text{MR}_3)))))} = \text{Client} \) ^
\( \text{name}(\text{c(name(lifeline(receiver(\text{MR}_3)))))} = \text{Reactor} \) ^
\( \text{name}(\text{c(name(lifeline(sender(\text{MR}_4)))))} = \text{Reactor} \) ^
\( \text{name}(\text{c(name(lifeline(receiver(\text{MR}_4)))))} = \text{Reactor} \) ^
\( \text{name}(\text{c(name(lifeline(sender(\text{MC}_5)))))} = \text{Reactor} \) ^
\( \text{name}(\text{c(name(lifeline(receiver(\text{MC}_5)))))} = \text{Connector} \) =>
\( \text{(precedes}(\text{MR}_2, \text{MR}_3) \) ^
\( \text{precedes}(\text{MR}_3, \text{MR}_4) \) ^
\( \text{precedes}(\text{MR}_4, \text{MC}_5)) \) ^
\( \text{(EXISTS} \text{MC}_6, \text{MCS}_7: \text{Message} \) | 
\( \{\text{MC}_6, \text{MCS}_7\} \text{ SUBSETEQ} \text{msgs()} \) ^
\( \text{sig}(\text{MC}_6) = \text{complete()} \) ^
\( \text{sig}(\text{MCS}_7) = \text{open()} \) ^
\( \text{name}(\text{c(name(lifeline(sender(\text{MC}_6)))))} = \text{Connector} \) ^
\( \text{name}(\text{c(name(lifeline(receiver(\text{MC}_6)))))} = \text{Connector} \) ^
\( \text{name}(\text{c(name(lifeline(sender(\text{MCS}_7)))))} = \text{Connector} \) ^
\( \text{name}(\text{c(name(lifeline(receiver(\text{MCS}_7)))))} = \text{CServiceHandler} \) =>
\( \text{(precedes}(\text{MC}_5, \text{MC}_6) \) ^
\( \text{precedes}(\text{MC}_6, \text{MCS}_7)) \) ^
\( \text{(EXISTS} \text{MR}_8, \text{MCS}_9, \text{MCS}_10, \text{MCS}_11: \text{Message} \) | 
\( \{\text{MR}_8, \text{MCS}_9, \text{MCS}_10, \text{MCS}_11\} \text{ SUBSETEQ} \text{msgs()} \) ^
\( \text{sig}(\text{MR}_8) = \text{registerHandler()} \) ^
\( \text{sig}(\text{MCS}_9) = \text{getHandle()} \) ^
\( \text{sig}(\text{MR}_8) = \text{handleEvent()} \) ^
\( \text{sig}(\text{MCS}_9) = \text{service()} \) ^
\( \text{name}(\text{c(name(lifeline(sender(\text{MR}_8)))))} = \text{CServiceHandler} \) ^
\( \text{name}(\text{c(name(lifeline(receiver(\text{MR}_8)))))} = \text{Reactor} \) ^
\( \text{name}(\text{c(name(lifeline(sender(\text{MCS}_9)))))} = \text{Reactor} \) ^
\( \text{name}(\text{c(name(lifeline(receiver(\text{MCS}_9)))))} = \text{CServiceHandler} \) ^
\( \text{name}(\text{c(name(lifeline(sender(\text{MCS}_10)))))} = \text{Reactor} \) ^
\( \text{name}(\text{c(name(lifeline(receiver(\text{MCS}_10)))))} = \text{CServiceHandler} \) ^
\( \text{name}(\text{c(name(lifeline(sender(\text{MCS}_11)))))} = \text{CServiceHandler} \) ^
\( \text{name}(\text{c(name(lifeline(receiver(\text{MCS}_11)))))} = \text{CServiceHandler} \) =>
\( \text{(precedes}(\text{MCS}_7, \text{MR}_8) \) ^
\( \text{precedes}(\text{MR}_8, \text{MCS}_9) \) ^
\( \text{precedes}(\text{MCS}_9, \text{MCS}_10) \) ^
\( \text{precedes}(\text{MCS}_10, \text{MCS}_11)) \)

Listing 4: The Acceptor and Connector pattern specification in PSL

2.5 The Leader/Followers Pattern

Figure 11 and Figure 12 show the class diagram and sequence diagrams representing this pattern, respectively.

Listing 5 demonstrates the Leader/Followers pattern PSL code.

```plaintext
patternName:
 LeaderFollowers

components:
```
Figure 11: The Leader/Followers pattern class diagram (revised from [SOK+00, BHS07])
Figure 12: The Leader/Followers pattern sequence diagram (revised from [SOK+00, BHS07])
\[
\text{(FORALL } M: \text{ Message}; T: \text{ Class } | \\
\text{ (} M \text{ IN } \text{msgs()} \text{ ) } ^{\land} \\
\text{ (} T \text{ IN } \text{classes()} \text{ ) } ^{\land} \\
\text{ name}(T) = \text{LFThread} \implies \emptyset \\
\text{ (} \text{c(name(lifeline(sender(M))))} = T \implies \\
\text{c(name(lifeline(receiver(M))))} = T \implies \\
\text{sig}(M) = \text{joinPool()} \text{) } \rightarrow \\
\text{(EXISTS } \text{ MTP}_2: \text{ Message } | \\
\text{ (} \text{MTP}_2 \text{ IN } \text{msgs()} \text{ ) } ^{\land} \\
\text{ sig}(\text{MTP}_2) = \text{join()} \implies \\
\text{ (} \text{c(name(lifeline(sender(\text{MTP}_2))))} = T \implies \\
\text{c(name(lifeline(receiver(\text{MTP}_2))))} = \text{ThreadPool} \rightarrow \\
\text{(} \text{precedes}(M, \text{MTP}_2) \text{ ) } ^{\land} \text{ calls}(\text{joinPool()}, \text{join()}) \text{) } \rightarrow \\
\text{(FORALL } M_{j1}, M_{j2}, M_{n1}, M_{n2}: \text{ Message}; T_1, T_2, \text{ TP}: \text{ Class } | \\
\{M_{j1}, M_{j2}, M_{n1}, M_{n2}\} \text{ SUBSETEQ } \text{msgs()} \text{ ) } ^{\land} \\
\text{ name}(T_1) = \text{LFThread} \implies \\
\text{ name}(T_2) = \text{LFThread} \implies \\
\text{ name}(\text{TP}) = \text{ThreadPool} \implies \\
\{T_1, T_2, \text{ TP}\} \text{ SUBSETEQ } \text{classes()} \implies \\
\text{ (} \text{c(name(lifeline(sender(M_{j1}))))} = T_1 \implies \\
\text{c(name(lifeline(receiver(M_{j1}))))} = \text{TP} \implies \\
\text{c(name(lifeline(sender(M_{j2}))))} = T_2 \implies \\
\text{c(name(lifeline(receiver(M_{j2}))))} = \text{TP} \implies \\
\text{c(name(lifeline(sender(M_{n1}))))} = \text{TP} \implies \\
\text{c(name(lifeline(receiver(M_{n1}))))} = T_1 \implies \\
\text{c(name(lifeline(sender(M_{n2}))))} = \text{TP} \implies \\
\text{c(name(lifeline(receiver(M_{n2}))))} = T_2 \implies \\
\text{sig}(M_{j1}) = \text{join()} \implies \text{sig}(M_{j2}) = \text{join()} \implies \emptyset \\
\text{ (} \text{sig}(M_{n1}) = \text{notify(NewLeader)} \implies \\
\text{sig}(M_{n2}) = \text{notify(NewFollower)} \text{) } \rightarrow \\
\text{ (precedes}(M_{j1}, M_{j2}) \text{ ) } \rightarrow \\
\text{(FORALL } M: \text{ Message}; T_1, \text{ TP}, \text{ HS}: \text{ Class } | \\
\{T_1, \text{ TP}, \text{ HS}\} \text{ SUBSETEQ } \text{classes()} \implies \\
\text{ name}(T_1) = \text{LFThread} \implies \\
\text{ name}(\text{TP}_1) = \text{ThreadPool} \implies \\
\text{ name}(\text{HS}_1) = \text{HandleSet} \implies \emptyset \\
\text{ (} \text{c(name(lifeline(sender(M))))} = \text{TP} \implies \\
\text{c(name(lifeline(receiver(M))))} = T_1 \implies \\
\text{sig}(M) = \text{notify(NewLeader)} \text{) } \rightarrow \\
\text{ ((EXISTS } \text{ MHS}_2: \text{ Message } | \\
\text{ (} \text{MHS}_2 \text{ IN } \text{msgs()} \text{ ) } ^{\land} \\
\text{ sig}(\text{MHS}_2) = \text{handleEvents()} \implies \\
\text{c(name(lifeline(sender(\text{MHS}_2))))} = T_1 \implies 
\text{ ) \rightarrow 
\end{array}
\]
\begin{verbatim}

\textbf{c(name(lifeline(receiver(MHS_2)))) = HS} \Rightarrow \\
\text{precedes(M, MTHS_2))} \\
(\textbf{EXISTS MHS_3, MHS_4: Message |} \\
(\{MHS_3, MHS_4\}) \textbf{ SUBSETEQ msgs} \\
\text{sig(MHS_3) = select} \\
\text{sig(MHS_4) = eventArrival} \\
\text{c(name(lifeline(sender(MHS_3)))) = HS} \\
\text{c(name(lifeline(receiver(MHS_3)))) = HS} \\
\text{c(name(lifeline(receiver(MHS_4)))) = HS} \Rightarrow \\
(\text{precedes(MHS_2, MHS_3)} \\
\text{precedes(MHS_3, MHS_4)}) \\
(\textbf{EXISTS MT_5, MTP_6: Message |} \\
(\{MT_5, MTP_6\}) \textbf{ SUBSETEQ msgs} \\
\text{sig(MT_5) = notify(EventArrival)} \\
\text{sig(MTP_6) = promoteNewLeader} \\
\text{c(name(lifeline(sender(MT_5)))) = HS} \\
\text{c(name(lifeline(sender(MTP_6)))) = T_1} \\
\text{c(name(lifeline(receiver(MT_5)))) = HS} \\
\text{c(name(lifeline(receiver(MTP_6)))) = TP} \Rightarrow \\
\text{precedes(MHS_4, MT_5)} \\
\text{precedes(MT_5, MT_6))} \\
(\textbf{EXISTS MT_7: Message; T_2: Class |} \\
\text{(MT_7 \textbf{ IN} msgs)} \\
\text{\textbf{T_2 \textbf{ IN} classes}} \\
\text{name(T_2) = LFThread} \\
\text{sig(MT_7) = notify(NewLeader)} \\
\text{c(name(lifeline(sender(MT_7)))) = TP} \\
\text{c(name(lifeline(receiver(MT_7)))) = T_2} \Rightarrow \\
\text{precedes(MT_6, MT_7))} \\
(\textbf{EXISTS MCEH_8: Message |} \\
\text{(MCEH_8 \textbf{ IN}msgs)} \\
\text{\textbf{c(name(lifeline(sender(MCEH_8)))) = T_1}} \\
\text{name(c(name(lifeline(receiver(MCEH_8))))) = ConcreteEventHandler} \Rightarrow \\
\text{precedes(MTP_6, MCEH_8))} \\
(\textbf{EXISTS MT_9, MTP_10, MT_11: Message |} \\
(\{MT_9, MTP_10, MT_11\}) \textbf{ SUBSETEQ msgs} \\
\text{sig(MT_9) = joinPool} \\
\text{sig(MTP_10) = join} \\
\text{sig(MT_11) = notify(NewFollower)} \\
\text{c(name(lifeline(sender(MT_9)))) = T_1} \\
\text{c(name(lifeline(sender(MTP_10)))) = T_1} \\
\text{c(name(lifeline(sender(MTP_11)))) = TP} \\
\text{c(name(lifeline(receiver(MT_9)))) = T_1} \\
\text{c(name(lifeline(receiver(MTP_10)))) = TP} \\
\text{c(name(lifeline(receiver(MT_11)))) = T_1} \Rightarrow \\
\text{precedes(MCEH_8, MT_9)} \\
\end{verbatim}

21
Figure 13: The Half-Sync/Half-Async pattern class diagram (revised from [SC95, BHS07])

Listing 5: The Leader/Followers pattern specification in PSL

2.6 The Half-Sync/Half-Async Pattern

Figure 13 and Figure 14 show the class diagram and sequence diagrams representing this pattern, respectively.

Listing 6 demonstrates the Half-Sync/Half-Async pattern PSL code.
Figure 14: The Half-Sync/Half-Async pattern sequence diagram (revised from [SC95, BHS07])

staticConditions:

associates(ExternalEventSource, AsyncTask) ▼
associates(AsyncTask, MessageQueue) ▼
associates(MessageQueue, SyncTask)

dynamicConditions:

(FORALL M: Message; EES, AT, MQ, ST: Class |
 (M IN msgs() ^ {EES, AT, MQ, ST} SUBSETEQ classes() ^
 name(EES) = ExternalEventSource ▼
 name(AT) = AsyncTask ▼
 name(MQ) = MessageQueue ▼
 name(ST) = SyncTask) @
 (c(name(lifeline(receiver(M)))) = EES ▼
 sig(M) = interrupt()) ▼
=>
((EXISTS MAT_2: Message |
 (MAT_2 IN msgs()) |
 sig(MAT_2) = notify(Event) ▼
 c(name(lifeline(sender(MAT_2)))) = EES ▼
 c(name(lifeline(receiver(MAT_2)))) = AT) =>
 (precedes(M, MAT_2 ▼
 calls(interrupt(), notify(Event)))) ▼
(EXISTS MEES_3: Message |
Figure 15 and Figure 16 show the class diagram and sequence diagrams representing this pattern, respectively.

Listing 7 demonstrates the Monitor Object pattern PSL code.

2.7 The Monitor Object Pattern

Listing 6: The Half-Sync/Half-Async pattern specification in PSL
Figure 15: The Monitor Object pattern class diagram (revised from [Sch00, BHS07])
Figure 16: The Monitor Object pattern sequence diagram (revised from [Sch00, BHS07])
\[
\{\{ML_3, MM_4\} \subseteq \text{msgs()} \}^*
\]
\[
\text{sig}(ML_3) = \text{acquire()} \land \text{sig}(MM_4) = \text{doWork()} \land
\]
\[
c(\text{name(lifeline(sender(ML_3))})) = S \land
\]
\[
c(\text{name(lifeline(receiver(ML_3))))) = S \land
\]
\[
c(\text{name(lifeline(sender(MM_4)))) = L \land
\]
\[
c(\text{name(lifeline(receiver(MM_4)))) = M) \Rightarrow
\]
\[
(\text{precedes}(MS_2, ML_3) \land
\]
\[
\text{precedes}(ML_3, MM_4)) \land
\]
\[
(\forall MC_5: \text{Message} | \text{MC_5} \in \text{msgs()} \land
\]
\[
\text{sig}(MC_5) = \text{wait()} \land
\]
\[
c(\text{name(lifeline(sender(MC_5)))) = M \land
\]
\[
c(\text{name(lifeline(receiver(MC_5)))) = \text{Cond}) \land
\]
\[
(\text{precedes}(MM_4, MC_5) \land
\]
\[
\text{calls}(\text{doWork()}, \text{wait()}))) = \Rightarrow
\]
\[
(\exists ML_6, MT1_7: \text{Message} | \{\{ML_6, MT1_7\} \subseteq \text{msgs()} \}^*
\]
\[
\text{sig}(ML_6) = \text{release()} \land
\]
\[
\text{sig}(MT1_7) = \text{sleep()} \land
\]
\[
c(\text{name(lifeline(sender(ML_6)))) = \text{Cond} \land
\]
\[
c(\text{name(lifeline(receiver(ML_6)))) = L \land
\]
\[
c(\text{name(lifeline(sender(MT1_7)))) = \text{Cond} \land
\]
\[
c(\text{name(lifeline(receiver(MT1_7)))) = T_1) \Rightarrow
\]
\[
(\text{precedes}(MC_5, ML_6) \land
\]
\[
\text{precedes}(ML_6, MT1_7)) \land
\]
\[
(\exists MT2_8, MS_9: \text{Message} | \{\{MT2_8, MS_9\} \subseteq \text{msgs()} \}^*
\]
\[
\text{sig}(MT2_8) = \text{invoke()} \land
\]
\[
\text{sig}(MS_9) = \text{method()} \land
\]
\[
c(\text{name(lifeline(sender(MT2_8)))) = T_2 \land
\]
\[
c(\text{name(lifeline(receiver(MT2_8)))) = T_2 \land
\]
\[
c(\text{name(lifeline(sender(MS_9)))) = T_2 \land
\]
\[
c(\text{name(lifeline(receiver(MS_9)))) = S) \land
\]
\[
(\text{precedes}(ML_6, MT2_8) \land
\]
\[
\text{call}(\text{wait()}, \text{sleep()}))) \land
\]
\[
(\exists ML_10, MM_11, ML_12: \text{Message} | \{\{ML_10, MM_11, ML_12\} \subseteq \text{msgs()} \}^*
\]
\[
\text{sig}(ML_10) = \text{acquire()} \land
\]
\[
\text{sig}(MM_11) = \text{doWork()} \land
\]
\[
\text{sig}(ML_12) = \text{release()} \land
\]
\[
c(\text{name(lifeline(sender(ML_10)))) = S \land
\]
\[
c(\text{name(lifeline(receiver(ML_10)))) = L \land
\]
\[
c(\text{name(lifeline(sender(MM_11)))) = L \land
\]
\[
c(\text{name(lifeline(receiver(MM_11)))) = M \land
\]
\]
\[
27
Figure 17: The Abstract Factory pattern class diagram [GHJV94]

\[
\begin{align*}
    &\mathbf{c}(\text{name(lifeline(sender(ML_{12}))))} = M \land \\
    &\mathbf{c}(\text{name(lifeline(receiver(ML_{12}))))} = L \implies \\
    &\quad (\text{precedes(MS_{9}, ML_{10})}) \land \\
    &\quad \text{precedes(ML_{10}, MM_{11})} \land \\
    &\quad \text{precedes(MM_{11}, ML_{12})} \land \\
    &\quad \text{calls(doWork(), release())})
\end{align*}
\]

Listing 7: The Monitor Object pattern specification in PSL

2.8 The Abstract Factory Pattern

Figure 17 shows the class diagram representing this pattern that provide interfaces for creating two product types, i.e., ProductA and ProductB.
Listing 8 demonstrates the Abstract Factory pattern PSL code.

```plaintext
patternName:
AbstractFactory

components:
{AbstractFactory, Client, ConcreteFactory,
AbstractProduct, Product} SUBSETEQ classes();
createProduct() IN opers(AbstractFactory);
createProduct() IN opers(ConcreteFactory);

staticConditions:

isInterface(AbstractFactory) ~
isInterface(AbstractProduct) ~
realizes(ConcreteFactory, AbstractFactory) ~
realizes(Product, AbstractProduct) ~
depends(Client, AbstractFactory) ~
depends(ConcreteFactory, Product) ~

/*
* For each interface AbstractFactory which references a Client
* there must be two ConcreteFactory, two AbstractProduct, and four Product
* so that ConcreteFactories realizes AbstractFactories;
* Products which must be concrete and unique realizes AbstractProducts and
* there must be dependencies from the ConcreteFactories to the Products
*/
FORALL AF: AbstractFactory; C: Client @ (isInterface(AF) ~
depends(C, AF)) =>

(EXISTS CF1, CF2: ConcreteFactory; AP1, AP2: AbstractProduct;
 PA1, PA2, PB1, PB2: Product | (CF1 <> CF2 ~ AP1 <> AP2 ~
"isAbstract(CF1) ~
"isAbstract(CF2) ~
isAbstract(AP1) ~
isAbstract(AP2)) @(depends(C, AP1) ~ depends(C, AP1) ~
realizes(CF1, AF) ~
realizes(CF2, AF) ~
realizes(PA1, AP1) ~
realizes(PA2, AP1) ~
realizes(PB1, AP2) ~
realizes(PB2, AP2)) => (depends(CF1, PA1) ~ depends(CF1, PB1) ~
depends(CF2, PA2) ~ depends(CF2, PB2) ~
"isAbstract(PA1) ~ "isAbstract(PA2) ~
"isAbstract(PB1) ~ "isAbstract(PB2) ~
```

2.9 The Component Configurator Pattern

Figure 18 and Figure 19 show the class diagram and sequence diagrams representing this pattern, respectively. Here, the dynamic configuration is illustrated by the two configurable components on the diagrams, i.e., ConcreteComponentA and ConcreteComponentB.

Listing 9 demonstrates the Component Configurator pattern PSL code.

```
\textbf{patternName}: ComponentConfigurator
\textbf{components}:
\{ComponentConfigurator, ComponentRepository, Component,
ConcreteComponent\} \text{\textit{SUBSETEQ}} \textit{classes}();
\{insert(), remove()\} \textit{SUBSETEQ} \textit{opers}(ComponentRepository);
\{init(), fini(), suspend(), resume(), info()\} \textit{SUBSETEQ} \textit{opers}(Component);
\textit{configScript()} \textit{IN} \textit{opers}(ComponentConfigurator);
\textit{run()} \textit{IN} \textit{opers}(ConcreteComponent);
\textbf{staticConditions}:
\textit{isAbstract}(Component) \text{"}
\textit{depends}(ComponentConfigurator, ComponentRepository) \text{"}
\textit{aggregates}(ComponentRepository, Component) \text{"}
```
Figure 19: The Component Configurator pattern sequence diagram (revised from [JS97, SSRB00, BHS07])
inherits(ConcreteComponent, Component) ~

// Each abstract component must be inherited by at least two non-abstract ConcreteComponents
FORALL C: Component @ isAbstract(C) =>
  (EXISTS CCA, CCB: ConcreteComponent =>
    (CCA <> CCB ~
      ~isAbstract(CCA) ~
      ~isAbstract(CCB) ~
      inherits(CCA, C) ~
      inherits(CCB, C))

dynamicConditions:

(FORALL M: Message; CC, ConcreteC, CR: Class |
  (M IN msgs) ~
    {CC, ConcreteC, CR} SUBSETEQ classes ~
    name(CC) = ComponentConfigurator ~
    name(ConcreteC) = ConcreteComponent ~
    name(CR) = ComponentRepository) @
    (c(name(lifeline(sender(M)))) = CC ~
      c(name(lifeline(receiver(M)))) = ConcreteC ~
      sig(M) = configScript())
)=>

((EXISTS MCC: Message |
  (MCC IN msgs) ~
    sig(MCC) = init ~
    c(name(lifeline(sender(MCC)))) = CC ~
    c(name(lifeline(receiver(MCC)))) = ConcreteC =>
      (precedes(M, ConcreteC) ~
        calls(configScript(), init())) ~
  (EXISTS MCR: Message |
    (MCR IN msgs) ~
      sig(MCR) = insert(ConcreteComponent) ~
      c(name(lifeline(sender(MCR)))) = CC ~
      c(name(lifeline(receiver(MCR)))) = CR) =>
      precedes(ConcreteC, MCR))) ~

(FORALL M: Message; ConcreteC: Class |
  (M IN msgs) ~
    CC_A IN classes ~
    name(ConcreteC) = ConcreteComponent) @
    (c(name(lifeline(receiver(M)))) = ConcreteC ~
      sig(M) = run())
)=>

(EXISTS MCR: Message; CC, CR: Class |
  (MCR IN msgs) ~
    {CC, CR} SUBSETEQ classes ~
    name(CC) = ComponentConfigurator ~
    name(CR) = ComponentRepository ~}
Listing 9: The Component Configurator pattern specification in PSL

3 The PLSL parser in action

To specify a given PL, we use the PLSL language. Figure 20 displays the PLSL Editor usage in practice.

A The PSL source code in Xtext

This section presents the PSL parser source code in Xtext.

```plaintext
/* The name of produced grammar model with the .psl extension */
generate pSL "http://www.ui.ir/se/mdserg/xtext/PSL"

/* PSL is the name of the generated.ecore model root EPackage. */
import "http://www.eclipse.org/emf/2002/Ecore" as.ecore
```
Figure 20: The PLSL Editor

```
13 Specification:
14   {Specification}
15      (comment += Comment)*
16      'patternName:' name = CID
17      'components:' (components += Declaration ';'?)+
18      'staticConditions:' (staticConditions += StaticCondition '^'?)*
19      'dynamicConditions:' (dynamicConditions += DynamicCondition '^'?)*
20    ;
21    
22 Declaration:
23      comment = Comment? (si = SetInclusion | sm = SetMembership)
24    ;
25    
26 SetInclusion:
27      '{' head = memberName (comma +=',' tail += memberName)* '}'
28      subset = 'SUBSETEQ' set = Type
29    ;
30    
31 SetMembership:
32      member = memberName in = 'IN' set = Type
33    ;
34    
35 // member name can be a Classifier ID (CID),
36 // function ID(param1[], ...), or attribute ID (FUNCID)
37 ```
memberName:

CID | FUNCID ('(' (((CID | FUNCID ('(' ')')?) ',)?* ')')?

Type:

{Type} (basic = BasicType |
  fi = FuncInclusion |
  ps = 'P(' type = Type ')') |
  cid = CID

BasicType:

'Boolean' | 'Integer' | 'String' | 'Value' | 'Class' | 'Interface' |
'Operation' | 'Property' | 'Lifeline' | 'Message' | 'Parameter'

DirectionKind:

'in' | 'inout' | 'out' | 'return'

VisibilityKind:

'public' | 'private' | 'protected' | 'package'

StaticCondition:

{StaticCondition} comment = Comment? scond = Formula

DynamicCondition:

{DynamicCondition} comment = Comment? dcond = Formula

Boolean:

'true' | 'false'

FuncInclusion:

f = FunName '(' (params += Param ,)?* ')
  (op = CompOp value = RangeType)?

FunName:

Reserved | FUNCID

Reserved:

'calls' | 'precedes' | 'subs' | 'name' | 'c' | 'lifeline' |
Param:

cid = (CID | FUNCID)

("::")

(fi = (CID | FUNCID) '(' ')') | aid = (CID | FUNCID))

("#" paramid = (CID | FUNCID))? | FuncInclusion

;

RangeType:

{RangeType} (bool = Boolean |

vk = VisibilityKind |

dk = DirectionKind |

int = INT | // specific to the multiplicity’s lower and upper values

star = '*' | // for specifying the multiplicity’s infinite bound

cid = CID | // represents string type

basic = BasicType | // for the definition of property types

fi = FuncInclusion

)

;

Formula:

neg = UnaryConn?

(sf = SimpleFormula |

qf = QuantFormula | lp = '(' mf = MixedFormula rp = '))'

)

;

QuantFormula:

q = Quantifier types = QTypes

(suchThat = SuchThat st = QFormula)?

(atSign = AtSign at = QFormula)?

conseq = ConseqConn consequent = QFormula

;

QFormula:

f = Formula

;
MixedFormula:
  Formula ({MixedFormula.lhs = current} ((or += 'v' orRHS += Formula)* |
      (and += '∧' andRHS += Formula)*
  ))

Quantifier:
  forall = 'FORALL' | exists = 'EXISTS'

SuchThat:
  '|

AtSign:
  '@'

ConseqConn:
  '=>' | '<=>'

QTypes:
  head = VarType (semicolon += ';' tail += VarType)*

VarType:
  header = CID (colon += ',' trailer += CID)* ':' type = Type

SimpleFormula:
  l = LHSOperand op = CompOp r = RHSOperand |
  si = SetInclusion |
  sm = SetMembership |
  fi = FuncInclusion

RHSOperand:
  cid = CID | fi = FUNCID | bool = Boolean | int = INT | star = '*' |
Listing 10: The PSL’s parser generation code in Xtext

B The PLSL source code in Xtext

This section presents the PLSL parser source code in Xtext.

```
1 // This parser is used to extract a given pattern language
2 // patterns, relationships and consequences of applying patterns
3 // the extension PLSL is an abbreviation for
4 // the Pattern Language Specification Language
5 grammar ir.ui.mdserg.xtext.PLSL
6 with org.eclipse.xtext.common.Terminals
7 hidden (WS) // terminal WS : (' ' | '	' | '' | '
');
8 generate pLSL "http://www.ui.ir/mdserg.xtext/PLSL"
```

38
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

PatternLanguage:
   Comment? 'PatternLanguageName:' name = PID
   Comment? 'Patterns:' '{' head = Pattern (',' tail += Pattern)* '}'
   Comment? 'InitPattern:' initPattern = Pattern
   (Comment? 'Relationships:' relationships += Relationship*)?
   (Comment? 'ConseqsFunction:' consequences += Consequence*)?
   ;

Pattern:
   pid = PID
   ;

Relationship:
   first = Pattern rel = Relation second = Pattern
   ;

Relation:
   relationshipKind = ('uses' | 'refines' | 'conflicts' | 'competes')
   ;

Consequence:
   '(', first = Pattern ',', second = Pattern ')', '->'
   benefits = Benefits ',', liabilities = Liabilities
   ;

Liabilities:
   {Liabilities} '{' (lHead = Liability (',' lTail += Liability)* )? '}'
   ;

Benefits:
   {Benefits} '{' (bHead = Benefit (',' bTail += Benefit)* )? '}'
   ;

Benefit:
   Goal
   ;

Liability:
   Goal
   ;

Goal:
   goal = ('simplicity' | 'modularity' | 'encapsulation' | 'portability' |
   'testability' | 'stability' |
   'flexibility' | 'performance' | 'transparency' |
   'predictability' | 'applicability' | 'scalability' | 'availability' |
\begin{verbatim}
'deadlock' | 'extendibility' | 'lowcoupling' | 'complexity')
;
terminal PID returns ecore::EString:
  ('A'..'Z')('a'..'z' | 'A'..'Z' | '0'..'9' | '_' )*
;
Comment:
  SL_COMMENT | ML_COMMENT
;
terminal SL_COMMENT:
  '//' !('
'|'')* ('' | '
')+
;
terminal ML_COMMENT:
  '/*' -> '*/'
;
Listing 11: The PSL’s parser generating code in Xtext
\end{verbatim}

References


[Bet13] Lorenzo Bettini. Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd, 2013. 2


[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, 1994. 28


[Sch95] Douglas C Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching. 1995. 9, 10

[Sch99] Douglas C Schmidt. Wrapper facade—a structural pattern for encapsulating functions within classes. 1999. 8, 9

