

L4J : a Java API for languages design and execution

Gustavo Cabral
Graduate School of Arts and Sciences
The University of Tokyo, Japan
gustavo@graco.c.u-tokyo.ac.jp

Tetsuo Tamai
Graduate School of Arts and Sciences
The University of Tokyo, Japan
tamai@graco.c.u-tokyo.ac.jp

ABSTRACT

Domain Specific Languages (DSL) play an important role in the Model-driven Engineering (MDE). Their design can be accomplished by defining metamodel and concrete syntax models. Moreover, transformations can be implemented through transformation languages. However, the process of designing the operational semantics for a DSL should be straightforward. The cost of manipulating and transforming models into programming languages code (GPLs, such as Java) should be minimized, or even nil, since the goal is to use DSLs and not only maintain them. Therefore, we propose a framework called L4J (Language for Java) that allows the definition of DSLs and their behavior, enabling them to be executed. The framework is fully based on the Java language making it a *low level of abstraction* solution when compared to existing solutions. Metamodel and syntax concepts are defined in Java making use of object-orientation and code annotation through the L4J API. A transformation component that permits the coding and debugging of transformation is provided. The injection and extraction operations are also implemented. We validate the framework by using it in a test case automation case study.

Categories and Subject Descriptors

PL [Programming Language]: Design—*DSL, GPL*; MDE [Model-driven Engineering]: Metamodel, Concrete Syntax, Transformation; SE [Software Engineering]: System Specification—*requirements, use cases*; FM [Formal Methods]: Formal model generation—*CSP*

Keywords

DSL, GPL, language design, metamodel, concrete syntax, Java, code annotation, software testing

1. INTRODUCTION

The main goal of this research is leveraging the use of models and DSLs when developing software. This brings an interesting level of formalism to the software engineering discipline.

However, current difficulties found in the Model-driven Engineering (MDE) approach still need to be overcome. Therefore, this paper presents solutions for some of the MDE issues we identified during a case study. The main issues are (1) the absence of use of early stage artifacts, which are usually written in Natural Language (e.i. requirements specification), as formal inputs in the MDE process, (2) the cost of implementing the operational semantics (execution) of Domain Specific Languages (DSL), (3) the support to manipulate and transform models into programming languages code, also known as General Purpose Languages (GPL), and (4) the burden of synchronizing models from different phases of the software development.

Since these issues are not directly related to the specification of MDE, but with its implementation by existing tools, we proposed an *integrated* framework that provides support for all the phases of the software development process. The proposed MDE framework is called L4J¹ (Language for Java), since it is entirely based in the Java platform. It targets the deficiencies present in existing frameworks. The issue number 1 is tackled by applying L4J to the Natural Language Processing domain, which is not presented in this paper. Nevertheless, L4J itself solves issues 2, 3 and 4 from last paragraph. L4J enables the execution semantics of a DSL to be implemented in Java. This possibility frees the engineer from the task of translating a DSL to a GPL when the target platform is Java. Next, because models in L4J are classes instances in the memory, the creation of persistent intermediary models from transformations is avoided. When intermediary models are serialized to text (into files) from the memory, they can suffer changes and this leverages problems related to the bidirectional synchronization between source and target models [12]. In our approach, only the set of input models should be editable in order to specify, implement and test the software.

As proof of concept, L4J is used in the design of several DSLs and the definition of transformations that implement an automatic test execution approach [3, 2]. The difficulties we experienced, while using the AMMA² (ATLAS Model Management Architecture) framework, in this testing case study helped us to define a new framework. In the example we present here, L4J is used to design a DSL called CNL

¹Part of the code used in the L4J API belongs to the DSL4J project owned by Eli Konky <eli.konky@gmail.com>. DSL4J is available at code.google.com/p/dsl4j/

²wiki.eclipse.org/AMMA

(Controlled Natural Language) that defines software specification. This specification is later translated into a formal notation called CSP (Communicating Sequential Processes) [10], which is also implemented as a DSL in L4J. The CSP process algebra describes complex aspects of systems in an abstract notation but still very close to implementation. This thin difference in abstraction between CSP and GPLs allowed us to implement the operational semantics of CSP in L4J. Once the CSP model is executed (simulated), it triggers the execution of test cases on the software implementation. This automation approach works by associating the events from the CSP software model to the corresponding actions on the SUT (System Under Test). The implementation of this close combination of DSL (CSP) and GPL (Java) is challenging and could not be easily accomplished using the AMMA framework. Alternatively, L4J overcomes the found difficulties since it is a *low level* API that enables the direct utilization of DSLs in Java programs. The features that are not present in existing frameworks and how L4J implements them are further examined in Section 4. This analysis does not aim to present L4J as an alternative to available DSL frameworks. It tries to understand the advantages of using a rather *lower level* approach to design and use DSLs.

In the next section we present a background overview about metamodel and concrete syntax models. Section 3 shows how L4J implements these concepts; the CNL serves as an example. Section 4 explains how an automatic test execution strategy was implemented using the CSP-to-Java transformation. Moreover, we show the found problems in this approach and present the alternative solution when using L4J. Section 5 lists the advantages and disadvantages of L4J, and shows possible future works.

2. BACKGROUND

The possibility of write, or code, a model is advantageous since it brings a greater sense of control and understanding over the models and its concepts, as presented by the case study in [4]. The connection of syntax to metamodel elements, which describe elements from a specific domain, is essentially the process followed to create a DSL. This task is simplified by using frameworks that assist the creation of the metamodel and the concrete syntax for the DSL.

An example of metamodel definition language is KM3 [7], a language that has similar notations to those found in MOF³ but simpler. It is combined with TCS [8] to define the concrete syntax of DSLs. A TCS model constructs the grammar for a DSL defining primitive templates for datatypes, based on regular expressions, and syntactic templates (grammatical rules) for classes from the metamodel. Concerning model-to-model transformations, the ATL [6], a QVT⁴-like model, uses KM3 metamodels as frames for models input and output. Briefly, it is a hybrid of declarative and imperative programming. Besides model-to-model transformations, model injection (text-to-model) and extraction (model-to-text) are essential operators supported by tools from the AMMA project, which maintains the cited languages. Also part of this framework, the Textual Generic Editor (TGE) is a text editor parameterized by information

³OMG's MetaObject Facility: www.omg.org/mof/

⁴Query/View/Transformation: www.omg.org/spec/QVT/

gathered from the KM3 and the TCS models. Such type of editor assists the engineer when using the designed DSL.

The following subsections show the design of the CNL using the cited AMMA framework. This involves two tasks: the definition of metamodel, which is sometimes called abstract syntax, and the concrete syntax, the actual grammar for the DSL. The metamodel for the CNL is presented in KM3 and an example of a CNL instance is given. In Section 3 L4J is introduced and used to design the CNL.

2.1 CNL as a DSL

The CNL is a simplified version of English. It allows the requirement engineer to write the software specification using a natural language like syntax. Application domain concepts and system usage scenarios (use cases) are specified using the CNL. This language was first designed and used in the context of the Motorola project called Brazil Test Center (BTC) and focused on the software specification of mobile phones [2]. It is now extended to allow the specification of any software that has an interactive behavior, such as an application with GUI (Graphical User Interface) or even network protocols.

The CNL is not only a language that gives structure to the specification document, holding sections for requirements and use cases. It defines the syntax of the sentences used to write these definitions. Its metamodel contains classes that capture the sentence's grammatical structure. Requirement sentences hold definitions of system objects, their relations and possible actions. Figure 1 shows part of the CNL metamodel in KM3; the CNLSPECIFICATION class contains REQUIREMENTS and USE CASES. Each USE CASE has a CODE, NAME, and execution FLOWS, which are sequences of STEPS with ACTIONS and RESPONSES definitions.

```

class CNLSPECIFICATION extends NAMEDELEMENT {
  reference REQUIREMENTS[*];
  container : REQUIREMENT;
  reference USECASES[*];
  container : USECASE;
}
class USECASE extends NAMEDELEMENT {
  attribute CODE : STRING;
  attribute DESCRIPTION : STRING;
  reference FLOWS[1-*];
  container : FLOW;
}
class STEP extends LOCATEDELEMENT {
  attribute CODE : STRING;
  reference ACTIONS[1-*];
  container : SENTENCE;
  reference RESPONSES[0-*];
  container : STATESENTENCE;
}
class NOUNVALUES extends DECLARATION {
  reference VALUES[1-*];
  container : NOUN;
  reference NOUN : NOUN;
}
class VERBSWITHINDIRECTOBJECT extends TRANSITIVEVERBDEF {
  reference INDIRECTOBJECTS[1-*];
  container : INDIRECTOBJECTTEMPLATE;
}
class INDIRECTOBJECTTEMPLATE extends LOCATEDELEMENT {
  attribute PREPOSITION : STRING;
  reference INDIRECTOBJECT : NOUN;
}
class SENTENCE extends LOCATEDELEMENT {
  reference VERB : VERB;
  reference DIRECTOBJECT : DIRECTOBJECT;
  reference INDIRECTOBJECTS[0-*];
  container : INDIRECTOBJECT;
}

```

Figure 1: Part of the CNL metamodel in KM3

The syntax for the CNL is defined in TCS in a previous work [3] but Figure 2 is an example of a CNL specification. It shows its keywords in bold to ease the understanding of the CNL syntax. This example is the Birthday Book system from the Z reference book [11]. This system maintains a list of birthdays and each birthday record contains a name and a date. Since the CNL focuses on the system's objects, interface and the specification of possible user interaction scenarios, this example defines GUI components that allow message passing between the user and the system. NOTICES, BUTTONS, and TEXTFIELDS are some examples from the left column of Figure 2. The ADD BIRTHDAY use case in the

right column explores the scenario when a VALID or an INVALID birthday date is written at the BIRTHDAY textfield. In the VALID date case, after the OK button is CLICKED, the SUCCESS notice is DISPLAYED. In the INVALID date case, the INVALIDBIRTHDAY notice is DISPLAYED. This abstract specification of the system, using INVALID and VALID values, can be refined later by concrete definitions.

```

System: "BIRTHDAY BOOK"

Requirements:
[RQ01] "INTERFACE COMPONENTS"

There are COMPONENTS (COMPONENT),
COMPONENTS can be CLICKED (CLICK),
and DISPLAYED (DISPLAY).

NOTICES (NOTICE) are types of
COMPONENT. NOTICES may be SUCCESS,
INVALIDNAME, BIRTHDAYNOTFOUND, or
INVALIDBIRTHDAY. BUTTONS (BUTTON)
are also types of COMPONENT. OK
is a BUTTON. SEARCH is a BUTTON.

TEXTFIELDS (TEXTFIELD) are types
of COMPONENT. NAME, and BIRTHDAY
are TEXTFIELDS.

[RQ02] "DATA ELEMENTS"

There are VALUES (VALUE). TEXTS
(TEXT) are types of VALUE. TEXTS
may be INVALID, or VALID. TEXTS
are WRITTEN (WRITE) at TEXTFIELD.

Use Cases:
[[UC01]] "ADD BIRTHDAY"
Description: "A NEW BIRTHDAY
is ADDED to APPLICATION
DATABASE".

Flow: "MAIN"
From steps: start
To step: end

[M1] {WRITE a VALID TEXT at
the NAME TEXTFIELD.}
[M2] {WRITE a VALID BIRTHDAY
at the BIRTHDAY TEXTFIELD.}
[M3] {CLICK the OK BUTTON.}
{SUCCESS NOTICE is DISPLAYED.}

Flow: "ALTERNATIVE 1"
From steps: M1 To step: end

[A11] {WRITE an INVALID
BIRTHDAY at the BIRTHDAY
TEXTFIELD.}
[A12] {CLICK the OK BUTTON.}
{INVALIDBIRTHDAY NOTICE is
DISPLAYED.}

```

Figure 2: CNL model for the Birthday Book system

In the next section we design the CNL using L4J. The presented KM3 model and the Birthday Book example are illustrative references to better understand the L4J API elements that hold metamodel and syntax information.

3. METAMODEL AND GRAMMAR IN L4J

The semantics of metamodels, as in KM3 or MOF, and textual concrete syntax, as in TCS, are already well-defined and implemented by tools from MDE and Grammarware fields. However the implementation of these concepts could be made available as an API in order to support a direct combination between DSLs and GPLs. In this section we show how metamodel and syntax are defined in L4J and how injector⁵ and extractor are generated for the designed DSL. Because metamodel and syntax concepts are implemented in Java it is only necessary to know the Java language. Metamodel elements are coded as classes and its attributes, and the concrete syntax is defined as Java Annotations⁶.

Metamodel

The definition of a domain class is as simple as coding a Java PUBLIC CLASS (as in the *Interpreter* design pattern [5]) with its PRIVATE fields, which are accessed through their respective *get* and *set* methods. Next, classes and attributes are decorated with annotations from the L4J API. All annotations that define metamodel constraints are listed on Table 1. @CLASSPROPERTY is a type annotation. Its NEWCONTEXT property determines if a new context (block) should be created to include new definitions from the class (attributes). The context contains a symbol table used to resolve names references. This feature is used at the type-checking and name resolution tasks. Regarding the class fields, if it is a datatype (primitive type) it should have the

⁵In L4J, models stand for classes instances in memory. The serialization to any existing XML-based standard is implemented through transformations (e.i. DSL2Ecore).

⁶JSR 175: A metadata facility for the Java language.

Annotations	Properties
@CLASSPROPERTY	NEWCONTEXT : BOOLEAN
@PRIMITIVEATTRIBUTEPROPERTY	CARDINALITY : CARDINALITY UNIQUE : BOOLEAN
@CLASSATTRIBUTEPROPERTY	CARDINALITY : CARDINALITY
@REFERENCEPROPERTY	CARDINALITY : CARDINALITY OPPOSITEOF : STRING
ENUM CARDINALITY	ZEROORONE, ZEROORMORE, ONE AND ONEORMORE
@PARENT	—

Table 1: Metamodel annotations and its properties

Annotations	Properties
@PRIMITIVETEMPLATE	PATTERN : STRING
@CLASSTEMPLATE	PATTERNS : STRING[]
@PRIMITIVEATTRIBUTETEMPLATE	PATTERNS : STRING[] SEPARATOR : STRING
@CLASSATTRIBUTETEMPLATE	PATTERNS : STRING[] SEPARATOR : STRING
@REFERENCETEMPLATE	PATTERNS : STRING[] SEPARATOR : STRING REFERSTO : STRING
@FLAG	KEYWORDFORTRUE : STRING KEYWORDFORFALSE : STRING

Table 2: Annotations for concrete syntax definition

@PRIMITIVEATTRIBUTEPROPERTY metadata. The CARDINALITY enumeration is a constraint used to determine the number of occurrences of the field. The UNIQUE property is used to mark the field as a *primary key*, so the object can be uniquely identifiable. If the class is a container the @CLASSATTRIBUTEPROPERTY annotation is used at the contained field. Otherwise, if it is reference to an object, the @REFERENCEPROPERTY is employed instead. In the latter annotation, the OPPOSITEOF property is used to set a reference at the opposite container class to the class that has the reference. The @PARENT annotation masks the field that links the class to its container object. These features are also available in the KM3 language.

Grammar

We use the annotations from Table 2 to define the DSL's grammar. The @PRIMITIVETEMPLATE and the @CLASSTEMPLATE annotations are used on types. The PATTERN STRING attribute from @PRIMITIVETEMPLATE defines a regular expression that identifies datatypes (a terminal symbol). The (PATTERNS : STRING[]) property of the @CLASSTEMPLATE annotation holds the syntactic rules for the class (a non-terminal). All other annotations are used on fields and the PATTERNS property has the same denotation as in the @CLASSTEMPLATE annotations. It define choices of syntactical structures available to instantiate the field. The SEPARATOR attribute is used when the attribute has CARDINALITY ZEROORMORE or ONEORMORE. It is the token responsible for separating the objects from a list. In the @REFERENCETEMPLATE annotation, the REFERSTO property determines the field used to identify an object it refers to. Without this property the cross-reference would simply include the name of the variable without any link to its actual instance. Finally, @FLAG defines keywords that are used to give a *true* or *false* value to an annotated field.

Example

As presented, both metamodel constraints and concrete syntax information are annotations on classes and its fields. Figure 3 has some of the domain classes for the CNL from Section 2. The ID class is a `@PRIMITIVE_TEMPLATE`. In KM3 terms, it is a datatype with its regular expression definition. The `CNLSPECIFICATION` class has the `@CLASS_PROPERTY` and the `@CLASS_TEMPLATE` annotations, the latter defines its syntax. In the `PATTERNS` property, the words started with the `$` symbol are references to class fields. Other words are keywords used in the language. Both `REQUIREMENTS` and `USECASES` fields are contained by the `CNLSPECIFICATION` class and therefore the `@CLASS_ATTRIBUTE_PROPERTY` and `@CLASS_ATTRIBUTE_TEMPLATE` are used. The `PATTERNS` property at `@CLASS_ATTRIBUTE_TEMPLATE` specializes the field's syntax. The next class is `USECASE`. It has the `ID_CODE` and the `STRING_DESCRIPTION` fields annotated with `@PRIMITIVE_ATTRIBUTE_PROPERTY` and `@PRIMITIVE_ATTRIBUTE_TEMPLATE`. Their types (`ID` and `STRING`) point to the respective `@PRIMITIVE_TEMPLATE` to be used. The `USECASE` class also includes the `@PARENT` field `SYSTEM` that keeps a reference to the container class. To exemplify the `@REFERENCE_PROPERTY` and the `@REFERENCE_TEMPLATE` annotations we have the `INDIRECTOBJECT_TEMPLATE` class with a reference to a `NOUN` called `INDIRECTOBJECT`. It finds the `NOUN` instance based on the value of its `NAME` field, as determined by the `REFERS_TO` property.

```
@PRIMITIVE_TEMPLATE(PATTERN = (('A'.. 'Z'|'A'.. 'Z'|'-'') +
                              (('A'.. 'z'|'A'.. 'Z'|'0'.. '9')*))
public class ID {
    private STRING VALUE;
}
@CLASS_PROPERTY(newContext = false)
@CLASS_TEMPLATE(patterns = {'SYSTEM : $NAME $REQUIREMENTS $USECASES'})
public class CNLSPECIFICATION extends NAMEDELEMENT {
    @CLASS_ATTRIBUTE_PROPERTY(cardinality = CARDINALITY.ZEROORMORE)
    @CLASS_ATTRIBUTE_TEMPLATE(patterns = {'REQUIREMENTS : $REQUIREMENTS'})
    private List<REQUIREMENT> REQUIREMENTS;

    @CLASS_ATTRIBUTE_PROPERTY(cardinality = CARDINALITY.ZEROORMORE)
    @CLASS_ATTRIBUTE_TEMPLATE(patterns = {'USE CASES : $USECASES'})
    private List<USECASE> USECASES;
}
@CLASS_PROPERTY(newContext = false)
@CLASS_TEMPLATE(patterns = {'[[ $CODE ]] $NAME $DESCRIPTION $FLOWS'})
public class USECASE extends NAMEDELEMENT {
    @PRIMITIVE_ATTRIBUTE_PROPERTY(cardinality = CARDINALITY.ONE)
    @PRIMITIVE_ATTRIBUTE_TEMPLATE()
    private ID CODE;

    @PRIMITIVE_ATTRIBUTE_PROPERTY(cardinality = CARDINALITY.ZEROORONE)
    @PRIMITIVE_ATTRIBUTE_TEMPLATE(patterns = {'DESCRIPTION : $DESCRIPTION'})
    private STRING DESCRIPTION;

    @CLASS_ATTRIBUTE_PROPERTY(cardinality = CARDINALITY.ZEROORMORE)
    @CLASS_ATTRIBUTE_TEMPLATE()
    private List<FLOW> FLOWS;

    @PARENT()
    private CNLSPECIFICATION SYSTEM;
}
@CLASS_PROPERTY(newContext = false)
@CLASS_TEMPLATE(patterns = {'$PREPOSITION $INDIRECTOBJECT'})
public class INDIRECTOBJECT_TEMPLATE {
    @PRIMITIVE_ATTRIBUTE_PROPERTY(cardinality = CARDINALITY.ONE)
    @PRIMITIVE_ATTRIBUTE_TEMPLATE()
    private PREPOSITION PREPOSITION;

    @REFERENCE_PROPERTY(cardinality = CARDINALITY.ONE,
                       oppositeOf = 'INDIRECTOBJECTS')
    @REFERENCE_TEMPLATE(REFERS_TO = 'NAME')
    private NOUN INDIRECTOBJECT; }

```

Figure 3: Some classes from the CNL using L4J

Operations

The two fundamental operators when dealing with DSLs are injection and extraction. As presented on Figure 4 we use the domain classes and its annotations to implement the injection operator. The classes are analyzed using the Java Reflection and the Java Annotation APIs⁷. A grammar model

⁷java.sun.com/javase/6/docs/

is created and used to generate a grammar in the ANTLR⁸ (parser generator) format. This generation is implemented with the Velocity template engine⁹. The grammar file (.g) is then used by the ANTLR to create the actual parser in Java. The parser code contains the code necessary to instantiate the domain classes related to the syntactic rules from the language grammar. Using the `JavaCompiler` interface available in Java (from version 6) it is possible to compile the parser at runtime and instantiate it.

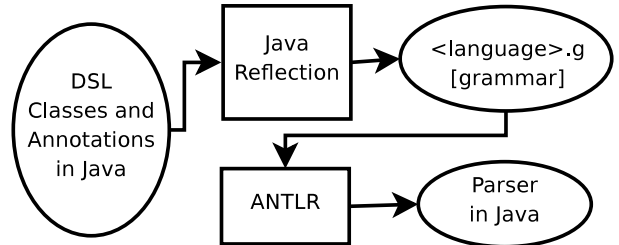


Figure 4: Overview of the L4J injector architecture

The extractor also uses the Java classes to retrieve metamodel and syntax data. It is a generic extractor for any DSL designed with L4J. It is implemented as a *Visitor* [5] that processes object from the whole model instance structure. Figure 5 shows how simple it is to inject and extract models by using the classes from the L4J API. Both `L4JINJECTOR` and `L4JEXTRACTOR` are `GENERIC` classes that require as constructor the main container class from the DSL.

```
// Parser instantiation
L4JINJECTOR<CNLSPECIFICATION> PARSER = L4JFACTORY
    .CREATEINJECTOR(CNLSPECIFICATION.class);

// CNL code injection
CNLSPECIFICATION SYSTEM = PARSER.PARSE(new FILEINPUTSTREAM(
    "BIRTHDAYBOOK.CNL"));

// Check model for errors (reference name resolution)
List<NAMERESOLUTIONREPORT> REPORTS = PARSER.REFERENCECHECK();
for (NAMERESOLUTIONREPORT R : REPORTS) {
    SYSTEM.OUT.PRINTLN(R.TOSTRING());
}

// Create the extractor and print serialized model
L4JEXTRACTOR<CNLSPECIFICATION> E = L4JFACTORY
    .CREATEEXTRACTOR(CNLSPECIFICATION.class);
SYSTEM.OUT.PRINTLN(E.EXTRACT(SYSTEM));

```

Figure 5: CNL code injected, checked and extracted

Similarly to TCS, the domain classes in Java and annotations contain enough information to implement the operators without extra effort. Moreover, as in TCS, the simplicity of the approach makes it hard to create DSLs with large gaps between the metamodel and the syntax. These two concepts are very closely linked here.

4. CASE STUDY

In this section we show how L4J allows us to implement a test execution framework. In a previous work [3], we defined this testing strategy based on transformations that generate a CSP model from the presented CNL specification language. It was implemented in ATL. Thus, the CSP language was also designed using KM3 and TCS. In this approach the yielded CSP model is used as input in the CSP-to-Java transformation, also in ATL, to produce the Java code required to automate the test execution. The

⁸www.antlr.org

⁹velocity.apache.org

CSP channels and datatypes definitions are translated into an interface, its methods, classes and enumerations. The left hand side of Table 3 shows the WRITE channel and the TEXT and TEXTFIELD datatypes definitions in CSP produced from part of a CNL specification. The CSP-to-Java transformation produces the Java code from the right hand side. Using such output code, the next step in this past strategy would be implementing the EVENTS interface with the code that executes the actual events (user actions) on the SUT.

<pre>channel WRITE : TEXT.TEXTFIELD datatype TEXT = INVALID VALID datatype TEXTFIELD = NAME BIRTHDAY</pre>	⇒	<pre>public interface EVENTS { public void WRITE(TEXT TEXT, TEXTFIELD TEXTFIELD); } public enum TEXT { INVALID, VALID; } public enum TEXTFIELD { NAME, BIRTHDAY; }</pre>
----------------------------------------------------------------------------------------------------------------	---	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Table 3: From CSP channel and datatypes to Java

However, the direct transformation from CSP to a GPL (Java) used in this cited strategy has some disadvantages. The list below shows the problems we faced when following this pure transformation-based approach that we cited.

- The implementation of the CSP-to-Java transformation requires the design of at least a subset of Java;
- The CSP-to-Java transformation generates several files, one for each class, interface or enumeration. The output from each transformation rule needs to be managed and consistent with dependent implementations;
- Current transformation frameworks do not support references between multiple models loaded from different files, which is necessary in this approach;
- Changes in the CNL specification could impact on the generated Java code. Consequently, the implementation code may need to be revised.

In order to avoid these difficulties we developed the L4J transformation API, its execution engine and a DSL *execution platform*. Transformations are implemented as unidirectional relations between source and target metamodels. This metamodel mappings is defined in Java in an imperative style by methods that have as parameter classes from the source metamodel and return a class from the target metamodel. It is not quite a sophisticated implementation but simple to code, use and debug, since it is all Java code. This solution allows an effortless combination of the DSLs and Java. This approach includes the following features:

- It does **not** require the implementation of the CSP-to-Java transformation since the output CSP model is an instance of Java classes;
- Consequently the Java language does **not** need to be designed in our approach;
- Not a single or multiple files need to be generated. All necessary information output by the CNL-to-CSP transformation is kept in the memory;
- Intermediary models (files) are neither generated nor needed to be kept. Models loaded in the memory are generated and used at runtime;
- Changes in the input CNL specification affect the produced CSP model but this only imply new instances of the CSP classes are loaded in the memory.

Regarding the execution of the produced CSP model, it is necessary to implement the behavior of the CSP constructors in an execution environment. In the other hand, there is no generated Java code to be compiled. The CSP elements are instances of Java classes in the memory bound to the implemented CSP operational model. Data about these elements, such as channel's and datatype's names, are utilized to locate in the SUT interface the methods that implement the required functionality. This is also done by using the Java Reflection API. Figure 6 shows how we implement the model execution in L4J by using the *Bridge* design pattern [5]. Distinct classes implement the behavior of the DSL (in this case CSP) and the interaction with the SUT through its access API. Another implementation possibility would be using Aspect-oriented Programming toolkits to bind the appropriate testing execution behavior to the associated concerns in the CSP domain classes.

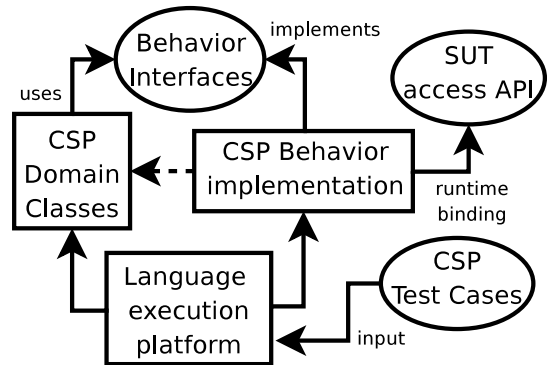


Figure 6: Test execution through CSP simulation

Following the CSP-to-Java approach, the interfaces from the generated code and its implementation are statically bound. This requires cumbersome code maintenance since changes at the input CNL model (requirements and use cases) are very frequent. The adoption of a runtime search for the implementation is more agile in our case since the interface used to interact with the SUT is fixed. We also developed an *event decomposition strategy* [1] to reuse the implementation of user actions. If the implementation is not found at runtime it means there is some problem with the definition of system events in the CNL code.

CSP model and test execution

The L4J execution platform takes injected or generated models and links them to the specified BEHAVIOR IMPLEMENTATION. The operational semantics of CSP and the actual event implementation from the SUT ACCESS API are separated and bound at runtime. In the testing scenario, the test cases (CSP traces) selected through the *test purpose* strategy defined at [9] are used as inputs. These traces guide the model simulation and consequently the test execution, as shown in Figure 6. This strategy is applied to the Java implementation of the Birthday Book system. The Java platform implements classes that provide access to the application's running objects. The JAVA.AWT.ROBOT class intercepts a running application and allows access to variables values and EVENTS DISPATCHING. This mechanism is used to implement the SUT ACCESS API.

5. CONCLUSIONS

We presented L4J, a DSL design framework based on Java. It is basically an API with a set of Java annotations that decorate domain specific classes with metadata. These annotations include metamodel information on types and fields, allowing the injection and extraction of the DSLs. L4J supports the definition of all main concepts available on other metamodel and syntax definition frameworks. Its *low level* approach allows greater control and performance when designing DSLs that have to be executed. Besides the obvious benefit of automatically generating the implementation of the injector and the extractor for the DSL, L4J has all the advantages related to a GPL. Some of these are:

- It is **not** necessary to learn new design languages;
- The DSL-to-Java transformation is **not** necessary;
- Java-related tools support is available (refactoring);
- Implementation of the language's execution is simple;
- Lightweight approach with Java "native" performance.

Because it is a new framework and still under development, its disadvantages are mainly related with the lack of features present in maturer frameworks. Such features are:

- The `OPERATORTEMPLATE` from TCS is under development;
- The extraction does not include *pretty-print* features;
- Extraction to XML is not implemented (e.i. Ecore);
- There is not a DSL editor generation (e.i. TGE).

Regarding the way L4J defines DSL concepts, it is possible to argue that a better design approach would be specifying metamodel and syntax separately, since the metamodel definition is independent of the syntax. Therefore, an alternative solution to separate these definitions is under analysis. It includes the use of Java interfaces to specify metamodel concepts and implementation classes that hold syntax definitions. In this scenario the annotations are applied to methods and not fields.

Using a GPL to design a DSL also raises issues related to technology independence of the solution. Because one of the main goals of MDE is to produce software assets that are resilient to changes in the technologies, L4J may seem as an inappropriate solution. However, L4J should not be seen strictly as a DSL design framework but as an API, since it does not have metamodel and syntax definition languages. L4J can actually be used as the implementation of such languages. It is the infrastructure that allows the implementation of DSL tools and offers an API to support the use of the DSL inside Java programs.

In the L4J transformation definition, the possibility of using Java overcomes the advantages present in existing DSL-based transformation frameworks. The support of tools to debug the execution of transformations had major importance in the success of the definition of our transformations. The direct access to other Java codes and libraries is also an undeniable benefit. There is also the cost associated with learning the syntax and the data manipulation library offered by transformation languages such as ATL. The quality of the debugging assistance is also lower when compared to Java programs debugging tools. Moreover, the L4J transformation performance is also Java "native".

We also used L4J to specify the TXT (Text), the NL (Natural Language) and the KNOW (Knowledge) languages. The first two are used to allow the injection and processing of natural language. The last one is used to accumulate domain knowledge during a natural language processing transformation strategy. These languages are part of a framework that tries to solve problem number 1 mentioned in Section 1. It allows the inclusion of natural language artifacts in the MDE development cycle. Early stage artifacts are processed based on metamodels and the retrieved instances are transformed into more concrete models, such as the KNOW model (an Ontology-like language).

Future works include the implementation of a generic editor with *pretty-print* support based on the L4J meta information. The implementation of features to allow a practical use of L4J are our top priority in order to leverage feedback about the framework. As a future case study we plan to develop languages and transformations that allow rapid development of prototypes for requirements validation with clients. This approach would use languages such as CNL to specify application domain concepts and system behavior.

6. REFERENCES

- [1] G. Cabral and A. Sampaio. Automated formal specification generation and refinement from requirement documents. *JBCS - Journal of the Brazilian Computer Society*, 14, 2008.
- [2] G. Cabral and A. Sampaio. Formal specification generation from requirement documents. *Electronic Notes in Theoretical Computer Science*, 195:171–188, 2008. Proceed. of the Brazilian Symposium on Formal Methods (SBMF 2006).
- [3] G. Cabral and T. Tamai. Requirement-based testing through formal methods. In *TESTCOM/FATES Short Papers*, 2008.
- [4] L. Cao, B. Ramesh, and M. Rossi. Are Domain-Specific Models easier to maintain than UML models? *Software, IEEE*, 26(4):19–21, July-Aug. 2009.
- [5] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Nov 1994.
- [6] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.
- [7] F. Jouault and J. Bézivin. KM3: A DSL for metamodel specification. In *FMOODS'06*, pages 171–185, 2006.
- [8] F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE'06*, pages 249–254. ACM, 2006.
- [9] S. Nogueira, A. Sampaio, and A. Mota. Guided test generation from CSP models. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 258–273. Springer, 2008.
- [10] A. Roscoe, C. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
- [11] J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.
- [12] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *ASE'07*, pages 164–173. ACM, 2007.