

Requirement-based testing through formal methods

Gustavo Cabral and Tetsuo Tamai
{gustavo,tamai}@graco.c.u-tokyo.ac.jp

General System Studies
Graduate School of Arts and Sciences
The University of Tokyo - Japan

Abstract. Testing is not enough to achieve a correct and bug free system; it is necessary to maintain a cohesive set of artifacts, such as use cases, design diagrams, code and test cases. If these elements are loosely coupled, it is very probable that the implementation will not attend the clients expectations. In this context, we provide a model-based DSL editor to assist the engineer in the requirement specification edition and transform it into a formal use model in the CSP notation. This use model is further combined with test directives, or test purposes, to generate test cases using the FDR refinement checker. In addition, a strategy to execute the generated test cases in the Eclipse test platform is proposed.

1 Introduction

Testing may be a laborious and very cost-intensive task even when assisted by tools. The problem lays not only in the testing task. All the previous activities in the development process need to yield coherent outputs and the software configuration must maintain these artifacts under control. The larger the number of redundant or duplicated definitions, the larger is the amount of documentation and the possibility of having an incorrect implementation. Therefore, Model-driven engineering (MDE) brings a viable level of formalism to define artifacts, enabling model transformation and property checking. These two tasks reduce the number of artifacts and make it possible to check for duplicities or inconsistencies of definitions. These preventive measures do not guaranty a bug free system but they will minimize the number of bugs in posterior phases of the software development process. Only with a concise initial set of artifacts it is possible to have quality assured code and test cases.

This paper presents an ongoing research that proposes a tool that assist engineers when specifying requirements to support testing. The informal specification is translated into a use model and using test purposes, test cases are generated. The automatic execution of the generated test cases is also approached. Thus, not only the execution of test cases checks the implementation but considering that the sequence of transformations maintain the coherence between artifacts, this approach aims to keep a set of valid artifacts during system development.

Most of the components that enable this strategy are implemented. The focus of this research is on the specification and testing of Java applications with

Graphical User Interface (GUI). Following some ideas from [16], we propose the use of well formatted requirements to automatically create test oracles that allows the definition of test cases. Moreover, this approach can be extended for testing web applications, web services, and protocols. Enhancements and extensions have been done since the previous research [4], which made use of Natural Language Processing (NLP) techniques instead of a model-based framework.

The current research makes use of model-based frameworks to implement the solution through metamodel, concrete syntax, and model transformation definitions. The solution involves the parsing of a simplified version of English, the Controlled Natural Language (CNL), through an Eclipse plug-in editor. Furthermore, CNL requirements are translated to a process algebraic formal model in the CSP notation [12] and finally an automatic test execution strategy, which involves the implementation of a system interface, is presented. The test generation strategy itself was previously defined in [11] in the context of the Motorola project called Brazil Test Center (BTC), which focused on mobile phone software testing. The previous NLP-based solution [4] was applied in an industrial case and its dependence on the application's domain Ontology showed to be laborious. The current approach has not yet been applied in an industrial project but because it automatically extracts the system's Ontology from requirements definitions it has better cost-benefits.

2 Requirements Specification

The Birthday Book system from the Z reference book [14] is specified in CNL. This system maintains a list of birthdays and each birthday record contains a name string and a date. The specification in Figure 1 is a simplified description and does not cover system constraints such as the existence of a list of birthdays with maximum size; such constructions in the CNL syntax are still under research. This version of the CNL focuses on the system's objects, interface and the specification of user interaction scenarios that are essential for the test case generation approach. It defines interface components to allow message passing between the user and the system, composed by **Buttons** and **Notices**, for instance. It also defines possible values for these objects, such as the **Success** notice, the **Search** button, and the **invalid** value. This last one is associated with the **Text** type, which is used as input by the **write** verb of the **Textfield** object. These components accept events according to verbs definitions from the requirement section. A **Component**, for instance, can be **clicked** and **displayed**. Since **Notice** and **Button** are **Components**, they also expect these events.

The **Add Birthday** scenario explores the possibilities when a **valid** or an **invalid** birthday are 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 **InvalidName** notice is **displayed**. The CNL syntax still does not support objects definitions with more than one word, such as **Invalid Name**.

The CNL editor uses a model-based technology implemented by the Generative Modeling Technologies (GMT) project [2]. In MDE, models are considered

| | |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>System: "Birthday Book"</p> <p>Requirements:</p> <p>[RQ01] "Interface components"</p> <p>There are components (component). Components may be enabled, or disabled. Components can be clicked (click), and displayed (display).</p> <p>Notices (notice) are types of component. Notices may be success, invalidName, birthdayNotFound, or invalidBirthday. Buttons (button) are also types of component. Ok is button. Search is button.</p> <p>Textfields (Textfield) are types of component. Name, birthday, and phone are textfields.</p> <p>[RQ02] "Data elements"</p> <p>There are values (value). Texts (text) are types of value. Texts may be invalid, or valid. Texts are written (write) at textfield. Texts are set (set) at textfield.</p> | <p>Use Cases:</p> <p>[[UC01]] "Add Birthday"</p> <p>Description: "A new birthday is added to application database".</p> <p>Flow : "Main"</p> <p>From steps: start</p> <p>To step: end</p> <p>[M1] {Write an valid text at the name textfield.}</p> <p>[M2] {Write an valid birthday at the birthday textfield.}</p> <p>[M3] {Click the ok button.} {Success notice is displayed.}</p> <p>Flow: "Alternative 1"</p> <p>From steps: M1 To step: end</p> <p>[A11] {Write an invalid birthday at the birthday textfield.}</p> <p>[A12] {Click the ok button.} {InvalidBirthday notice is displayed.}</p> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Fig. 1. Interface components and data elements are specified as requirements. The Add birthday scenario presents the use of different user inputs and their respective system responses.

as first class entities defined according to the semantics of its metamodel. It is a three layers architecture (metametamodel, metamodel, and models) where the metametamodel conforms to itself. In the GMT project, KM3 [6] is the adopted metamodel definition language. In our requirement-based testing approach, the Textual Concrete Syntax (TCS), also component from the GMT project, is used. The TCS [7] language binds metamodel elements to textual concrete syntaxis enabling both text-to-model and model-to-text transformations. The TCS project also deploys a template project to create Domain Specific Languages (DSL) editors based on the Textual Generic Editor (TGE). This editor provides a parameterized edition environment with functionalities such as highlight, hyperlinks of keywords, and traceability between text and metamodel classes.

Model-to-model (M2M) transformation is available through the ATLAS Transformation language (ATL) [5], a hybrid of imperative and declarative programming languages that makes use of OCL expressions. As a complete framework, ATL is compatible with KM3 and TCS allowing the definition of the CNL-to-CSP transformation proposed in Section 3. In the CNL context, a CNL-to-Problem transformation is under development and it will check for inconsistencies in the specified requirements and use cases. The Problem model refers to the elements from the input model and designate comments about the found non conformance. The following sections detail the CNL metamodel and its syntax.

2.1 CNL Metamodel

The CNL is not only a language that gives structure to the specification document, holding sections for requirements and use cases flows. It defines the syntax of sentences used to write requirements and use cases bodies. It has a concrete syntax that captures the sentences structures from the English grammar. These sentences are the actual definitions of system objects, their relations and actions.

Figure 2 shows part of the CNL metamodel in KM3; the **System** class contains requirements and use cases. Each requirement has an id and states the elements of the application domain, their inter-relations, and possible values. The way these elements are manipulated is also expressed as requirements sentences. Similarly, the use case section has a set of use cases, each with an id, name, and execution flows, which are sequences of steps forming system usage scenarios where sentences are used to describe actions and responses.

The requirements statements and the use cases actions and responses sentences are correlated. Requirements specify elements that are manipulated by the use cases events; this resembles the definition of classes in the object-oriented paradigm and the access to methods of classes instances. In the grammatical point of view, nouns and verbs are created in the requirements section when the application's objects are described. Nouns are related to each other forming Ontology classes; this hierarchical structure enables the reference of a class of nouns as a verb's object. Verbs, which are all transitive here, are defined with suitable object complements that contain a preposition and a reference to a noun class. The action field of a step contains sentences with commands delivered to the system interface, GUI for instance. In the CSP point of view, verbs in the sentences represent events and the verb's objects are the messages of these events. Differently, the response field of a step describes the current system state, which refers to instances of classes (datatypes in CSP) and their respective values. System responses differ depending on the current system state, given the same user action. Currently, the proposed CNL does not specifies conditions that could trigger the execution of alternative flows since the system state is not being maintained in the generated CSP model (See Section 3 for details).

2.2 CNL Textual Concrete Syntax

The TCS syntax definition from Figure 3 specifies part of the CNL grammar used in the example from Figure 1. Lexically, the CNL avoids keywords and symbols that would make the specification hard to read; it should not look like a programming code. Thus, pronouns, and prepositions from the English language are used to mark the text and allow a natural language like syntax. Other symbols, such as brackets and colons, also guarantee unambiguous parsing. The verb **to be** is used to declare system elements and to specify the step responses (system state). Since the CNL is not parsed using a NLP technique it is not possible to define some types of natural language structures, which would actually bring ambiguity to the specification. In Figure 3, the TCS syntax for the CNL defines the concrete syntax for each CNL metamodel class. The **Sentence** class,

```

class System 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 NounSubType extends Noun, Declaration {
  reference super : Noun; }

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 container:DirectObject;
  reference indirectObjects[0-*]
  container : IndirectObject; }
}

```

Fig. 2. Requirements define system objects, relations, and possible actions. Use cases document the system behavior through execution flows that show message exchange between objects instances.

for example, simply points to its attributes: `verb`, which is referred through the verb's name, `directObject`, and `indirectObjects`.

3 Formal Model Generation

The CSP process algebra [13] is the target formalism of our strategy because it describes complex aspects of systems, such as concurrency, in an abstract notation but still very close to implementation. In order to produce the CNL-to-CSP transformation it is necessary to define CSP's metamodel in KM3. Moreover, its concrete syntax is also defined in TCS allowing the serialization of the model. These two definitions are not presented here because of lack of space.

After the CNL-to-Problem transformation checks for requirements inconsistencies the CNL-to-CSP transformation is executed to generate the system use model in CSP. Figure 4 shows examples of transformation rules that are used to relate objects from the CNL to the CSP metamodel. The `NounSubType2Datatype` rule relate `CNL!Noun` to `CSP!datatype` definitions; nouns from the same class are gathered in a common CSP datatype. The `NounValues2PossibleValues` rule creates `CSP!TypeComposition` classes that represent the possible values a datatype may assume. Similarly, the `PrimitiveType2PrimitiveType` rule refers to the integer (`Int`) and the boolean (`Bool`) CSP values that a datatype may have. The rule `VerbsWithIndirectObject2Channel` relates verbs to channel definitions, binding the channel to the appropriate datatype based on the verb's direct and indirect objects. Finally, the `Step2Process` rule creates `CSP!Process` objects for each of the steps in a use case flow. These examples of CNL to CSP

```

template System main context
: "System" ":" name {as = string}
(isDefined(requirements)?
  "Requirements" ":" requirements)
(isDefined(useCases)?
  "Use" "Cases" ":" useCases) ;
template UseCase
: "[" code {as = codeId} "]"
(isDefined(name)? name {as = string})
(isDefined(description)? "Description" ":"
  description {as = string} ".") flows ;
template Step addToContext
: "[" code {as = codeId} "]"
{" actions {separator = "."} "." }
(isDefined(responses)?
  "{ responses {separator = "."} "." }")
;
template NounSubType addToContext
: plural "(" name ")" "are" [[ "also" ]]
"types" "of" super {refersTo = name} ;

template NounValues
: [[ noun {refersTo = plural}
  "may" "be" values {separator = ","}
  | values {separator = ","}
  [[ "is" noun {refersTo = name}
  | "are" noun {refersTo = plural}]]
]]
;
template VerbsWithIndirectObject
: directObject {refersTo = plural} "are"
[[ "also" ]] verbs {separator = ","}
indirectObjects {separator = ","}
;
template IndirectObjectTemplate
: preposition {as = preposition}
indirectObject {refersTo = name}
;
template Sentence
: verb {refersTo = name}
directObject indirectObjects ;

```

Fig. 3. CNL syntax definition in TCS: requirements and use cases form the document structure. Requirements declare nouns, values and verbs. Use cases sentences refer to these declarations.

mapping are just part of the defined transformation; it is necessary to define the CSP processes behaviors and relate each other based on the *from steps* and *to step* fields. Analysing the Birthday Book example from Figure 1 and the generated CSP model in Figure 5, it is easy to see the relation between requirements and use cases with channels, datatypes, and processes definitions.

4 Test Case Generation

Using the notion of conformance from [15], the approach in [11] defines an IO (Input-Output) conformance relation in the trace model of CSP and formalise soundness of CSP test cases. A test suite is sound when all correct and possibly some incorrect implementations pass the execution. In other words, a sound test case must not fail when executed against any system under test (SUT) that conforms to the specification. This theory is implemented as CSP processes and combined with test purposes (TP) to select traces via refinement checking using FDR[12]. This allows a uniform and correct implementation of a test case generator in terms of a process algebra. In this section TP are expressed as CSP processes and composed to select test scenarios. Last, the selected test scenario is used to build a sound test case.

4.1 Test Purpose and Test Case Generation

A CSP Test Purpose (TP) is based on the notion introduced in [9]. It is a partial specification describing the behavior the desired tests should have; it is

```

rule NounSubType2Datatype {
  from r : CNL!NounSubType
  to d : CSP!Datatype (
    name <- r.getDtRefName()
  ), ....
}
rule NounValues2PossibleValues {
  from r : CNL!NounValues
  to t : distinct CSP!TypeComposition
  foreach(e in r.values){
    types <- e, datatype <- r.noun
  }
}
rule PrimitiveType2PrimitiveType {
  from r : CNL!PrimitiveType
  to d : CSP!PrimitiveType( type <- r.getType()) }

rule VerbsWithIndirectObject2Channel {
  from r : CNL!VerbsWithIndirectObject
  to d : CSP!Channels (
    channels <- r.verbs,
    possibleValue <- t
  ), ....
}
}
rule Step2Process {
  from r : CNL!Step
  to p : CSP!Process(
    name <- r.code.toUpper(),
    behavior <- thisModule.
      CreateBehavior( r.actions.
        union( r.responses) ) )
}

```

Fig. 4. ATL rules bind CNL to CSP classes. Each rule uses the `from` and the `to` sections to map the metamodel elements. It is also possible to create methods, such as `toUpper`, to help in the mapping.

```

channel click, display : Component
channel write : Text.Textfield
channel set : Text.Textfield
datatype Component = NOTICE.Notice |
  BUTTON.Button | TEXTFIELD.Textfield |
  ENABLED | DISABLED
datatype Notice = SUCCESS | INVALIDNAME |
  BIRTHDAYNOTFOUND | INVALIDBIRTHDAY
datatype Button = ACTIVE | INACTIVE |
  OK | SEARCH
datatype Textfield = NAME | BIRTHDAY | PHONE
datatype Value = TEXT.Text
datatype Text = INVALID | VALID

sysAlph = { | click, display, write, set | }
inAlph = { | click, write | }
outAlph = diff( sysAlph, inAlph)

System = M1; System
M1 = write.VALID.NAME -> (M2 [] A11)
M2 = write.VALID.BIRTHDAY -> M3
M3 = click.BUTTON.OK ->
  display.NOTICE.SUCCESS -> SKIP
A11 = write.INVALID.BIRTHDAY -> A12
A12 = click.BUTTON.OK ->
  display.NOTICE.INVALIDNAME -> SKIP

```

Fig. 5. CSP model generated from the specification in CNL. The channels and datatypes definitions are extracted from requirements and, at the right column, processes are extracted from use cases.

a CSP process specifying a safety property. It uses a set of events, $MARK = \{ | \textit{accept}, \textit{refuse} | \}$, to mark the specification model and allows test scenarios to be selected. In Figure 6 the *ACCEPT* and the *REFUSE* processes mark the specification process. The processes *ANY*, *NOT*, and *UNTIL* are examples of primitive test purposes that are combined to specify complex test purposes. The test purpose *TP1*, for instance, accepts *write.VALID.BIRTHDAY* and wait until *display.NOTICE.SUCCESS* happens to then accept the trace.

In order to retrieve the selected traces we use FDR to check the refinement $System \sqsubseteq_t (System [| \textit{sysAlph} |] TP1)$. Because *TP1* is noted with the *accept* event this refinement will not hold and a counter-example is presented by FDR. In this case the counter-example is *tc1*. Other counter-examples may be found

```

ACCEPT( id) = accept.id -> STOP
REFUSE( id) = refuse.id -> STOP
ANY( evSet, next) =
  [] ev : evSet @ ev -> next
NOT( specAlph, evSet, next) =
  ANY( diff(specAlph, evSet), next)
UNTIL( specAlph , evSet, next) =
  RUN( diff(specAlph,evSet))
  /\ ANY (evSet,next)
RUN(specAlph) = [] ev : specAlph @ ev ->
  RUN(specAlph)
TP1 = ANY( {write.VALID.BIRTHDAY},
  UNTIL(sysAlph,
    {display.NOTICE.SUCCESS},ACCEPT(1)))
assert System [T= ( System [] sysAlph [] TP1)

tc1 = <write.VALID.NAME, write.VALID.BIRTHDAY,
  click.BUTTON.OK, display.NOTICE.SUCCESS,
  accept.1>
TC_BUILDER(<<accept.i, s>>,inAlph,outAlph)
  = PASS

TC_BUILDER(<a>^as,inAlph,outAlph) =
  SUBTC(a,inAlph,outAlph);
  TC_BUILDER(as,inAlph,outAlph)
SUBTC( (cEvent,oEvents), inAlph, outAlph) =
  if member( cEvent, inAlph) then
    currentEvent -> SKIP
  else (
    currentEvent -> SKIP
    [] ANY( diff( oEvents, {cEvent}), INC)
    [] NOT( outAlph,
      union(oEvents,{cEvent}), FAIL) )

tc1_s1 = <(write.VALID.NAME, {}),
  (write.VALID.BIRTHDAY, {}),
  (click.BUTTON.OK, {}),
  (display.NOTICE.SUCCESS, {}),
  (accept.1, {})>
SOUND_TC1_1 = TC_BUILDER(tc1_s1,inAlph,outAlph)
SysExec1 = System [] sysAlph [] SOUND_TC1_1
assert SysExec1 [T=
  SysExec1 [] union(sysAlph,{fail}) []
  UNTIL(sysAlph, {fail}, REFUSE(1))

```

Fig. 6. Functions and processes used to select sound test cases as trace refinement counter-examples

checking if $System \sqsubseteq P(tc1) \sqsubseteq_t (System [] sysAlph [] TP1)$, where P is a function that returns a process given a sequence of events. Next, this strategy is extended to create a sound test case from the system specification.

4.2 Generating Sound Test Cases

A test execution can reach three verdicts: *pass*, *inconclusive*, and *fail*. If a sound TC behaves as $PASS = pass \rightarrow Stop$ the test passes. Similarly if TC behaves like $INC = inc \rightarrow Stop$ the verdict is inconclusive, and as $FAIL = fail \rightarrow Stop$ for fail result. The inconclusive verdict exists because the system may be non deterministic or it may be concurrently interacting with several systems leading to background state changes. In this case, it behaves differently each time; the same test leads to different verdicts when executed several times.

In Figure 5 the *inAlph* and the *outAlph* sets were defined as the input and the output alphabets of the Birthday Book system. In Figure 6, the process *TC_BUILDER* uses these alphabets to construct sound test cases. Initially, TCs retrieved as refinement counter-example, such as *tc1*, are reformatted into a sound test case shape, such as of *tc1_s1*, and used by *TC_BUILDER* (*SOUND_TC1_1*). Interactively, these candidate sound test cases are checked through refinement and if necessary they are updated with information retrieved from a refinement counter-example. In the *tc1_s1* example, it is initialized with the sequence of events from *tc1* and contains the possible output events the system may produce. The process *SUBTC* checks the execution of

each event from $tc1_s1$; if the current event belongs to the input alphabet it is executed. If the current event belongs to the system output alphabet the following scenarios are checked: the output event from the test scenario is the same as the one from the specification (*SKIP*), the specification outputs an event that is different from the test scenario expected output (*inconclusive*), or the specification outputs an event that does not belong to the output alphabet (*fail*). If the refinement $SysExec1 \sqsubseteq_t SysExec1[\text{union}(sysAlph, \{fail\}) |] UNTIL(sysAlph, \{fail\}, REFUSE(1))$ does not hold, FDR yields a counter example meaning that $tc1_s1$ is not sound. Thus, $tc1_s1$ need to be update with the output events from the counter-example.

5 Test Case Execution

Currently, the generated TCs are manually executed; the automatic execution component is not integrated with the test generation component. The automatic TC execution strategy includes the generation of Java code from the CSP use model through a CSP-to-Java transformation in ATL. The CSP channels and datatypes definitions are used to create an interface, its methods, classes and enumerations that correspond to the specified CSP elements. This strategy is applied to Java Desktop applications since 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. The current implementation will be integrated with the Eclipse Test & Performance Tools Platform (TPTP) [1] component to automatically execute the generated test cases and manage execution cycles. The presented test generation and execution approaches will substitute the manual edition and the generation of test cases through the TPTP Automated GUI Recorder. Our main goal is to make use of the execution, monitoring, tracing, profiling, and log analysis capabilities of the TPTP platform.

6 Final Considerations

The direct use of formal methods to verify system properties is well-known by the formal method community. In this research, the mixture of techniques from several computer science areas, such as model-driven engineering, formal methods, and software testing theories, is believed to be the path to solve software validation problems. There is still plenty of open issues related to the gap between requirements, design, implementation, and validation. To bind and maintain artifacts from these different levels of abstraction is not trivial. Here, requirements define details about the system interface and interaction scenarios, such a low level description brings complexity to the elicitation phase. In some cases a more abstract definition of requirements is preferred.

Because this is a requirement-based strategy, documentation needs to be checked and validated so tests can be extracted. The presence of some types

of incomplete definitions or scenarios are currently detected by the CNL editor. More complex properties can be checked through a CNL-to-Problem transformation. In this case, such transformation analyzes requirements and detect missing statements, such as in [8, 3].

The use of use cases is ideal to define simple scenarios; complex use cases tends to be difficult to describe in prose. However, the resulting generated test cases complexity is equivalent to the effort of specifying complicated use cases. The test case generation strategy is totally defined in a single formalism that uses the FDR refinement checker. The execution of test cases using the Eclipse TPTP tools coordinates the execution of test cases test plans and reports. Finally, it is possible to validate the system behavior using the model animator ProBE [10]. This animator can be used to check the specified requirements with the client, just as with a prototype.

References

1. *Eclipse Test & Performance Tools Platform Project*. www.eclipse.org/tptp/.
2. *Generative Modeling Technologies (GMT)*. www.eclipse.org/gmt/.
3. J. Bézivin and F. Jouault. Using ATL for checking models. *Electr. Notes Theor. Comput. Sci.*, 152:69–81, 2006.
4. Gustavo Cabral and Augusto Sampaio. Formal specification generation from requirement documents. *Electr. Notes Theor. Comput. Sci.*, 195:171–188, 2008.
5. F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *OOPSLA'06*, pages 719–720. ACM, 2006.
6. F. Jouault and J. Bézivin. KM3: A DSL for metamodel specification. In *FMOODS'06*, pages 171–185, 2006.
7. 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.
8. Leonid Kof. Scenarios: Identifying missing objects and actions by means of computational linguistics. *RE '07 15th IEEE International*, pages 121–130, 2007.
9. Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M. Potet. Test purposes: Adapting the notion of specification to testing. In *ASE'01*, page 127. IEEE Computer Society, 2001.
10. Formal Systems Ltd. *PROBE Users Manual version 1.25*, 1998.
11. Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Guided test generation from CSP models. Centro de Informática, UFPE, Recife/PE - Brazil, 2008.
12. A.W. Roscoe. Modeling and verifying key-exchange protocols using CSP and FDR. In *CSFW'95*, page 98. IEEE Computer Society, 1995.
13. A.W. Roscoe, C.A.R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
14. J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.
15. Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99*, pages 46–65. Springer, 1999.
16. Qing Xie and Atif Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1):4, 2007.