A Rigorous Methodology for Analyzing and Designing Plug-ins

Marieta V. Fasie
DTU Compute
Technical University of Denmark
DK-2800 Lyngby, Denmark
Email: marietafasie@gmail.com

Anne E. Haxthausen
DTU Compute
Technical University of Denmark
DK-2800 Lyngby, Denmark
Email: ah@imm.dtu.dk

Joseph R. Kiniry
DTU Compute
Technical University of Denmark
DK-2800 Lyngby, Denmark
Email: jkin@imm.dtu.dk

***Abstract*—Today, GUI plug-ins development is typically done in a very ad-hoc way, where developers dive directly into implementation. Without any prior analysis and design, plug-ins are often flaky, unreliable, difficult to maintain and extend with new functionality, and have inconsistent user interfaces. This paper addresses these problems by describing a rigorous methodology for analyzing and designing plug-ins. The methodology is grounded in the Extended Business Object Notation (EBON) and covers informal analysis and design of features, GUI, actions, and scenarios, formal architecture design, including behavioral semantics, and validation. The methodology is illustrated via a case study whose focus is an Eclipse environment for the RAISE formal method's tool suite.**

I. Introduction

Plug-ins, especially in the realm of plug-ins that wrap existing research command-line tools, are notoriously badly designed. Academics simply do not have the resources and expertise to execute on the design and implementation of a quality plug-in. Partly this is due to the fact that there are few examples of best practices in the area, and partly it is because plug-in development is viewed as the dirtiest of the dirty-but-necessary jobs of "selling" systems technology.

Typically, a researcher has developed a novel tool for Java programming, lets call it the `CommandLineWidget`. They want others to use this tool, but few people these days want to mess about with downloading and building source code and, sadly, the barrier to entry for command-line tools is not insignificant these days. Instead, the researcher wants to "sell" their tool by wrapping it in the `CommandLineFeature` for Eclipse, since Eclipse has the mind-share of most Java developers. But the researcher does not know how to think about the UI design of an Eclipse plug-in, design and program the plug-in, nor is she really interested in learning how to do these things.

Eclipse plug-in development is a tricky world. Concepts like features, plug-ins, extension points, windows, views, etc. abound. Enormous, poorly documented APIs are prolific in the Eclipse ecosystem. To implement even the most basic of features sometimes takes hours of digging to find the right three lines of code, and then those lines must change when a new major version of Eclipse comes out. This is a frustrating world for researchers who want to package their demonstrable, useful tools for the Eclipse IDE.

This work is an attempt to help resolve these issues. First, we provide a *rigorous step-wise methodology through which one can do the analysis, architecture design, and user interface (UI) design of a plug-in for an arbitrary integrated development environment (IDE)*. Second, we provide *template example plug-ins* that can be reused by a programmer with a good understanding of Java, but a poor understanding of Eclipse plug-in development, to develop a new Eclipse plug-in or feature.

The methodology used is based upon the Business Object Notation (BON), an analysis and design methodology promoted by Walden and Nerson in the mid-90s within the Eiffel community [1]. Ostroff, Paige, and Kiniry formalized parts of the BON language and reasoned about BON specifications [2], [3], [4], [5]. Fairmichael, Kiniry, and Darulova developed the BONc and Beetz tools for reasoning about BON specifications and their refinement to JML-annotated Java.[1] Finally, Kiniry and Fairmichael have extended BON in a variety of ways to produce Extended BON (EBON), which permits one to add new domain-specific syntax and semantics to the core BON language [6].

For the reader who has never heard of EBON, think of it as the subset of UML that might actually have a clear, unambiguous semantics. EBON's core features are that it is *seamless*, insofar as you use the same specification language for everything from domain analysis to formal architecture specification and its behavior, *reversible* insofar as code generation and reverse engineering to and from code to EBON is straightforward and tool-supported, and *contracted* as formal abstract state-based contracts (invariants, pre, and postconditions) are the fundamental notion used to specify system behavior. EBON has both a textual and a graphical syntax, a formal semantics expressed in higher-order logic, a formal semantics of refinement to and from OO software, and tool support for reasoning about specifications, expressing specifications textually or graphically, generating code from models and models from code, and reasoning about refinement to code.

The methodology is illustrated on a case study that develops an Eclipse environment for the RAISE formal method [7] and specification language (RSL) [8]. The project is called *eRAISE* and it is currently under development at DTU.

Originally RAISE was supported by the *eden* tool suite [9] which was successfully applied in a range of industrial

[1] See http://tinyurl.com/brgcrzc for more information.

projects. However, as this tool suite was only available for SUN workstations, a new tool suite *rsltc* [10], [11], portable for any platform supporting the C language, was developed between 1998–2008.

The *rsltc* tool suite consists of a type checker and some extensions to it supporting activities such as pretty printing, extraction of module dependencies, translation to other languages, generation of proof obligations, formal verification, and generation and execution of test cases. *rsltc* has a command-line interface that exposes different capabilities selected via switches, but is also used from Emacs using menus and key-bindings. *rsltc* was also successfully applied in several industrial projects. However, although it is easy to use for the user comfortable with command-line tools or Emacs, we expect that the creation of a modern Eclipse-based development environment for *rsltc* would broaden its appeal to mainstream software engineers and better enable its use for university-level pedagogy.

A. Related work

There is little published work that focuses on methodologies specific to plug-in development. E.g., Lamprecht et al. reflect over some simplicity principles elicited by many years' experience in plug-in development [12], but do not provide a methodology.

We speculate that there is not much published work because plug-in development was not the focus of scientists until recently. Moreover, it is a fair question whether or not plug-in development is any different from normal systems development where a GUI is involved. We believe that plug-in development is different from normal GUI development as plug-ins must integrate into the larger framework of the IDE, deal with non-GUI events, and work in arbitrary compositions.

II. Analysis and Design Method

The EBON methodology as applied to Eclipse plug-in development has six stages, each described in a separate subsection and presented in the order in which they are applied. These six steps are: *domain modeling*, *user interface*, *events*, *components*, *components communication* and *code generation*.

The methodology is illustrated on the eRAISE case study. Due to space restrictions, the entire project's analysis and design phases are not presented here; instead, only one scenario of the plug-in is shown from beginning to end. A technical report version of this article, cited later, provides the entire case study.

A. Domain Modeling

The first step when analyzing and designing a system is to establish its domain model. This is done in order to create a common vocabulary between those involved in the project and to identify the concepts used in the product development process. This means that the most important entities and high level classifiers related to the system domain must be identified, explained and documented from the very beginning, so they can be unanimously understood and used throughout the entire product life cycle.

```
system_chart eRAISESystem
cluster RSLPerspective
description "The Eclipse RAISE perspective. It contains all
components and functionality relevant for a RAISE project"
end

cluster_chart RSLPerspective
class Editor
description "The RSL text editor"
class Console
description "Displays the output of different components
e.g. TypeChecker, SMLTranslator etc"
class TypeChecker
description "The RSL syntax and type checker"
class SMLCompiler
cluster Translator
description "Contains all translators applicable for RSL
modules"
end

cluster_chart Translator
class SMLTranslator
description "Translates RSL modules to SML code"
class LatexGenerator
description "Integrates RSL specification in Latex"
end

class_chart Console
indexing
in_cluster: "RSLPerspective"
explanation "Displays the output of different components
e.g. TypeChecker, SMLTranslator"
command "Displays informative or error messages",
constraint "Delete content before displaying a new message"
```

Listing 1. System chart describing the eRAISE system.

Looking at the case study name and description, the domain model is constructed by analyzing areas like Eclipse, RAISE and graphical user interface. The result is a list of terminologies along with their explanation, essentially describing entities and elements from a high level point of view. Some examples from the list are notions like *editor, console, typechecker, translator, Standard ML (SML) translator, LaTeX generator, SML compiler, RSL Perspective* and so on.

When performing domain analysis we try to identify concepts that are redundant, which concepts relate to others, etc. Some of these items can be grouped in a more general notion, while others are big enough to covers multiple notions. For example, *SML translator* and *LaTeX generator* can be grouped under the *translator* notion, since both are referring to the process of transforming a RSL specification into another type of specification. Likewise *RSL Perspective* can be seen as a notion that comprises all other items since inside Eclipse all RAISE elements can be grouped under a single perspective.

Such notions are captured using the EBON *system_chart*, *cluster_chart* and *class_chart* elements. The notions that have been identified are documented as classes, which can be grouped under clusters and all these make up a unique system. Listing 1 illustrates a caption of the eRAISE System specified in EBON notation. Please notice how easy it is to capture notions and their description in EBON by just using natural language.

The domain model also describes how concepts behave and how their behavior is constrained. In EBON, behavior is specified by using two concepts called *queries* and *commands*, collectively known as *features*. Behavioral constraints are specified using an EBON concept called *contraints*. A *command*

```
scenario_chart RSLMENU
scenario "MENU1"
description "The user can type check all RSL files in the
workspace. Success or failure messages will be displayed along
with the list of errors in case of a failure"
scenario "MENU2"
description "The user can translate to SML all RSL files in
the workspace. Success or failure messages will be displayed
along with the list of errors in case of a failure"
scenario "MENU3"
description "The user can run all test cases in the workspace.
Success or failure messages will be displayed along with the
list of errors in case of a failure"
scenario "MENU4"
description "The user can generate Latex files for all files
in the workspace. Success or failure messages will be
displayed along with the list of errors in case of a failure"
```

Listing 2. Scenario chart for the RSL menu item.

is a service that a *class* provides that changes the state of the object that implements the class, while a stateless *query* is a request for information from a specific object. Looking at the case study domain model presented in Listing 1, *class_chart Console* offers the service of displaying informative messages or error messages. Within this class there is also a *constraint* stating that the console must be cleared before displaying a new message.

B. User Interface

The purpose of this step is to determine the plug-in functionality from the user's point of view. This means identifying all the things a user can do from the plug-in's UI. This UI feature set consequently derives the requirements for the product (the plug-in) and designs the UI in the same time. Therefore, for each user action that is relevant and important for the plug-in, a mock-up user interface is created. If many user actions are similar, they can be grouped under a single user interface. It is up to the plug-in developer to determine what are the most important features and how, or if, she wants to prioritize them.

The mock-up user interface can be a vague handmade sketch or a precise drawing made with an advanced graphical editing program. The intention here is not presentation and precision, but instead feature completeness and UI consistency.

For the Eclipse case study, it was decided, for example, that a user should have the possibility to typecheck all RSL files (the primary file type of the RAISE tool suite). Also, the user should have the possibility to translate all files to SML, to run all test cases existing in all files, and to generate LaTeX documents for them. Therefore, it was decided that these four actions should be grouped under a menu item which is called *RSL* and presented in the same UI, for consistency and simplicity.

Figure 1 illustrates the graphical user interface for the *RSL* menu item. This illustration was created by taking a screenshot of Eclipse and then hand-editing the resulting image in just a few minutes. The RSL menu item has been added as part of the Eclipse IDE menubar and has four submenu items, each with an associated keyboard shortcut.

While the user interface is being drawn, product requirements are documented using EBON *scenario_chart* elements.

```
event_chart UserActions
incoming
explanation "External events triggering representative system
behaviour"
event "TYPECHECKALL: User clicks RSL menu and then clicks on
Type Check all option or presses Ctrl+Alt+T"
involves TypeChecker, Console
```

Listing 3. Incoming event chart for typechecking features.

The beautiful part about using EBON is that it allows the requirements specification to be captured using natural language. Therefore no intermediate step is required between identifying the requirements and documenting them. For the eRAISE case study, the requirements associated with the user interface in Figure 1 are captured in the *scenario_chart* in Listing 2. For each submenu item there is a *scenario* element defined by a name and a description. The four scenarios are grouped under a *scenario_chart* associated to the Eclipse RSL menu.

C. Events

In this stage of the method the entire system is seen as a black box. The focus is on the external actions that make the system react and on the system's outgoing responses. Not all of the system's outgoing events are of interest, but only the ones that are started by external stimuli. More formally, within EBON, scenarios are composed of events, thus there is a refinement relationship between scenarios and events.

An incoming external event is any action that determines the system to change its state. For example it can be a user clicking a button or another system sending a request. An outgoing internal event is the response the system sends to an incoming external event. The system outgoing event for the action of pressing the button could, e.g., be the display of a new window or writing a message to the standard output.

Looking back at the scenario presented in subsection II-B, the user has the possibility to type check all RSL files. This is illustrated in Figure 1 by the presence of a sub-menu item named *Type check all*. Therefore, the incoming external action in this case is: *the user selects the Type check all sub-menu item*. And this external event has been determined just by looking at the scenarios previously identified. However, there is another user event that triggers the same system reaction and that is using the shortcuts: *The user presses Ctrl+Alt+T*.

Once established, the user actions are captured in EBON using *event_chart* elements. The *event_chart* is either *ingoing* or *outgoing* depending on the type of the events they capture. Since the two user incoming events that have just been identified aims for the same functionality, they are grouped under the same name (*TYPECHECKALL*) and captured in Listing 3. The *involves* part in Listing 3 is explained later in detail, as it denotes component communication patterns.

If an incoming action triggers changes in the system state, the next task is to decide how the system should respond to the action, and about the changes that have taken place. For the eRAISE case study, it was decided that, after the user selects the *Type check all* sub-menu item, a message should be displayed on the standard output. The message informs the user about how the typechecking evolved and since the case study GUI is Eclipse based, the standard output is considered

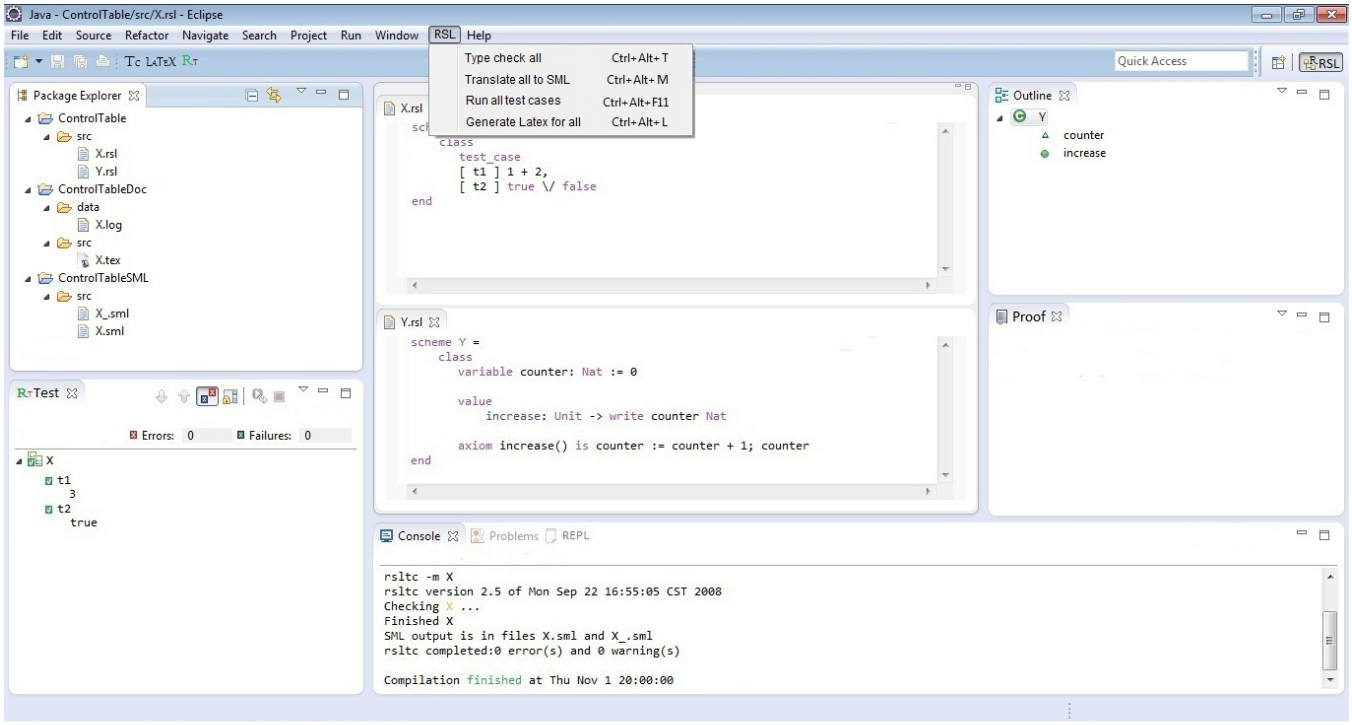

Fig. 1. Eclipse user interface displaying the RSL menu item

```
event_chart UserMessages
outgoing
explanation "Internal events triggering responses meant to
inform the user."
event "CONSOLEUPDATE: Successs or failure messages displayed
in console"
involves Console, TypeChecker
event "PROBLEMSUPDATE: Problems view update"
involves TypeChecker, Console, ConsoleToProblems, Problems
```

Listing 4. Outgoing event chart for typechecking features.

```
scenario_chart RSLMENU
scenario "MENU1"
description "The user can TYPECHECKALL RSL files in the
workspace. This implies PROBLEMSUPDATE, CONSOLEUPDATE and
EDITORUPDATE"
```

Listing 5. Scenario chart comprising the events for the typechecking feature.

by default the Eclipse *Console view*. However, in some cases, the typechecking may not be successful due to errors in the input files. In this case it would be nice to know what caused the problem and where it is found. The system should present the necessary information in the Eclipse *Problem* view. To sum up, after the user selects the *Type check all* sub-menu item, the system updates the *Console* and *Problem* views with appropriate information. Listing 4 presents the two events captured in an outgoing *event_chart* under the names of *CONSOLEUPDATE*, respectively *PROBLEMSUPDATE*.

Once the events have been identified and given a proper name, they can be used to rewrite the scenarios from subsection II-B. The reason for doing this is to emphasize the actions a user takes during a scenario and the responses the system must provide. Only the event name is used inside the scenario description, making it shorter, less open to interpretation and conciser.

Listing 5 illustrates how the *MENU1* scenario description presented in Listing 2 changes once the events names are being added. The ”user can type check all” description was replaced with ”user can TYPECHECKALL”, where Listing 3 describes exactly what the TYPECHECKALL event implies. The ”Success or failure messages will be displayed along with the list of errors in case of a failure” has been replaced by ”This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE”, where PROBLEMSUPDATE and CONSOLEUPDATE events are presented in Listing 4. EDITORUPDATE action has not been captured in any example in this paper, but it is basically an outgoing event that displays a red squiggly line in the editor in the place where an error has been discovered.

D. Components

At this stage in the methodology, in contrast with the previous one, one looks *inside* the system at the components that constitute its architecture. Components refer either to concrete elements, like a *Zoom in button*, or abstract concepts, like *User authentication* which covers everything in the system responsible for authenticating a user.

The place to start identifying the system architecture components is the domain model presented in subsection II-A. The high level classifiers captured there must be transformed into concrete data types in order to bring the system development closer to the implementation phase. The advantage of using EBON is that it simplifies the transit between the domain model and architecture and manages to capture the concrete

```
static_diagram SystemArchitecture
--eRAISE system architecture
component
        cluster RSLPerspective
        component
                class Editor
                class Console
                class TypeChecker
                class Problems

                cluster Translator
                component
                        class SMLTranslator
                        class LatexGenerator
```

Listing 6. eRAISE system architecture

data types in language independent fashion. This is done by taking the entities captured in *system charts*, *cluster charts* and *class charts* and transfer them in *static diagrams*. An EBON *static_diagram* contains multiple *components* which can be *clusters* or *classes* and which have the same meaning as the ones composing the *system_chart* in subsection II-A.

Applying this step on the case study, the system chart presented in Listing 1 becomes the *static_diagram SystemArchitecture* from Listing 6. The system architecture is made of only one component, the RSL perspective cluster, which is further composed of multiple subcomponents. In Listing 6 only five of its subcomponents are captured; the four classes and the Translator cluster which is also including more subcomponents.

E. Components Communication

This section concern is how components interact with each other and what interfaces they present to the other components that want to communicate with them. The starting point is the list of incoming and their corresponding outgoing actions identified in subsection II-C. This helps identifying the components that react first to an external stimuli and the components responsible for the outgoing actions. Once the starting and ending point of the data flow is established, the other interacting components are determined by evaluating the scenarios in subsection II-B.

In EBON notation, the components communication is seen in terms of *client, supplier* relationship. The component providing the interface is a *supplier* and all components using it are *clients*.

In the eRAISE case study, one incoming event is *TYPECHECKALL* presented in Listing 3 and its corespondent outgoing events are *CONSOLEUPDATE* and *PROBLEMSUPDATE* presented in Listing 4. Thus, the first component that reacts to user *TYPECHECKALL* action is the *TypeChecker*. The component responsible for the *CONSOLEUPDATE* is the *Console* and the one for *PROBLEMSUPDATE* is *Problems*. The *TypeChecker* component typechecks the input and directly informs the *Console* about the status. Therefore, the *TypeChecker* is a client of *Console* and this is expressed in EBON as: *TypeChecker client Console*. The client relations must be added in the *static_diagram* after the components declaration.

Since *TypeChecker* is directly stimulated by the *TYPECHECKALL* event and it is a client of *Console* which generates the outgoing response, it can be said that *TYPECHECKALL* event involves the *TypeChecker* and the *Console* components. And this is how the *involves* part in Listing 3 is constructed. The same method is applied to *PROBLEMSUPDATE* event. This event is triggered if there are errors displayed in the console. Therefore *TypeChecker* sends a message to *Console* and if the message is an error, a third component called *ConsoleToProblems*, that monitors the *Console*, notifies the *Problems* component. Therefore, the *PROBLEMSUPDATE* event involves the *TypeChecker, Console, ConsoleToProblems* and *Problems* components. This is captured in Listing 4.

```
class Console
        feature
        update  -> message: LIST[CHAR]
                -> channel: CHANNEL
                require
                        channel = 1 or channel = 2
                        -- 1-stdout 2-stderr
```

Listing 7. Console component interface

Once it was decided what components are interacting, it must be established how to do so. This means establishing the contracts between components by identifying the information a client needs and the messages it sends to its supplier. EBON supports the formal specification of typed interfaces in an programming language-independent fashion. Classes are parameterized and contain formally specified *features*. Each classifier in the domain model maps to exactly one class within the formal model, and each feature of each class within the domain model maps to one formally specified feature in that class's interface.

For example, the *Console* component in the case study has a feature called *update* which allows the client components to sent it messages that will further be displayed to the user. Please refer to the Console command in Listing 1 for better understanding. These messages are just strings like informative messages or errors. The situation is captured in Listing 7, where the typed feature signatures are found. When a client calls *update* on the *Console*, it must send the message to be displayed to the user and the information that specifies if it is an error message or not. The assertion *require* is a precondition that makes sure that *channel* is one of the values 1 or 2, where 1 stands for standard output and 2 for the standard error output. EBON also supports postconditions specified using the *ensure* keyword.

F. Code Generation

Once the analysis and design parts are finished, the next step is to generate the formally specified code skeleton. This step is accomplished using a tool named Beetlz, which automatically generates JML-annotated Java code from an EBON specification [13]. The input of this tool is the EBON *system_chart* and *static_diagram* that were obtained throughout the entire analysis and design. With just one click, Beetlz converts all EBON specifications into JML-annotated, Javadoc documented Java code. Beetlz also performs refinement analysis so that architecture drift is automatically identified as the system evolves, either at the model-level in EBON, or within the implementation in Java.

For example, Listing 8 shows the `Console` class generated by Beetlz for the component with the same name

```
public /*@ nullable_by_default @*/ class Console{
  //@ ensures channel == 1 || channel == 2;
  public void
    update(List<Char> message, Channel channel){}
}
```

Listing 8. Java Console class generated by Beetlz

described in the *static_diagram* in Listing 7.

III. Conclusion

The full specification of our case study is available in a technical report version of this paper. Its appendix contains all of the UI sketches for the various plug-ins in the eRAISE feature, the high-level concept/domain analysis, the specification of events, the architecture specification, the formal component interface design, and the code generated from that design. The example plug-ins and feature are also available in our repository at GitHub.[2]

The student responsible for the design and development of this plug-in (the first author), had to learn and apply this methodology in only a few short weeks. That is evidence for its utility. Only after completing the plug-in will we be able to reflect upon how well the analysis and design match the final implementation, but if past projects using the EBON methodology are any indication, we expect to see no architectural drift and full feature compliance.[3]

There are several opportunities for next steps refining this work in the context of plug-in development.

First, we would like to augment our advanced command-line options tool suite *CLOPS* for plug-in development. CLOPS permits one to declaratively specific the syntax and semantics of the command-line options of a tool. CLOPS reasons about such specifications (it has a formal semantics) and generates a lexer, parser, well-formedness checker, and documentation. Extending CLOPS to generate a set of Eclipse plug-ins and architecture documentation, like that we hand-write in this methodology, would be a valuable exercise and product.

Second, the community should use this method to wrap several more plug-ins following our example so that we can collectively properly evaluate the methodology's utility.

References

[1] K. Waldén and J.-M. Nerson, *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice–Hall, Inc., 1995.

[2] J. Lancaric, J. Ostroff, and R. Paige, "The BON CASE tool," Details available via http://www.cs.yorku.ca/∼eiffel/bon_case_tool/, Mar. 2002.

[3] J. R. Kiniry, "The Extended BON tool suite," 2001, available via http://ebon.sourceforge.net/.

[4] R. Paige, L. Kaminskaya, J. Ostroff, and J. Lancaric, "BON-CASE: An extensible CASE tool for formal specification and reasoning," *Journal of Object Technology*, vol. 1, no. 3, 2002, special issue: TOOLS USA 2002 Proceedings. Available online at http://www.jot.fm/.

[5] R. F. Paige and J. Ostroff, "Metamodelling and conformance checking with PVS," in *Proceedings of Fundamental Aspects of Software Engineering*, ser. Lecture Notes in Computer Science, vol. 2029. Springer-Verlag, Apr. 2001, also available via http://www.cs.yorku.ca/techreports/2000/CS-2000-03.html.

[6] J. R. Kiniry, "Kind theory," Ph.D. dissertation, Department of Computer Science, California Institute of Technology, 2002.

[7] The RAISE Method Group, *The RAISE Development Method*, ser. BCS Practitioner Series. Prentice Hall, 1995, available by ftp from ftp://ftp.iist.unu.edu/pub/RAISE/method_book.

[8] The RAISE Language Group, *The RAISE Specification Language*, ser. BCS Practitioner Series. Prentice Hall, 1992.

[9] P. Bruun and et al., "RAISE Tools Reference Manual," CRI: Computer Resources International, Tech. Rep. LACOS/CRI/DOC/17, 1995.

[10] "RAISE Tool User Guide," 2008. [Online]. Available: http://www.iist.unu.edu/newrh/III/3/1/docs/rsltc/user_guide/html/ug.html

[11] C. George, "The Development of the RAISE Tools," in *Formal Methods at the Crossroads. From Panacea to Foundational Support*, ser. Lecture Notes in Computer Science, B. K. Aichernig and T. Maibaum, Eds. Springer Berlin Heidelberg, 2003, vol. 2757, pp. 49–64. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-40007-3_4

[12] S. Naujokat, A. Lamprecht, B. Steffen, S. Jorges, and T. Margaria, "Simplicity principles for plug-in development: The jabc approach," in *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, june 2012, pp. 7 –12.

[13] E. Darulová, "Beetlz - BON software model consistency checker for Eclipse," Master's thesis, University College Dublin, 2009.

[2] https://github.com/kiniry/eRAISE

[3] Note to the reviewers: we expect that development will be complete by the time of the workshop so that we can demonstrate it in San Francisco.