Refining the *Quick Fix* for the Petri Net Modeling Tool Renew

Jan Hicken, Michael Haustermann, Daniel Moldt

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics, Theoretical Foundations of Computer Science (TGI)
{1hicken,haustermann,moldt}@informatik.uni-hamburg.de

Abstract. "Renew (The Reference Net Workshop) is an extensible Petri net IDE that supports the development and execution of high-level Petri nets and other modeling techniques." [3,10]. Renew allows to add Java inscriptions to elements in Reference nets, which leads to possible syntax errors. One important feature of an IDE is the instant feedback to the developer if syntax errors are detected. This enables the developer do deal with these errors at the moment they are entered and found rather than getting confronted with a large set of syntax errors at compile time. This paper presents the *quick fix* feature, which accomplishes this task of finding and displaying the error, suggesting possible fixes for that error and finally applying the fix automatically.

Keywords: Petri Nets, Quick Fix, Reference Nets, Renew, Parser Generator

1 Introduction

Many modern integrated development environments (IDEs) such as Eclipse [14], NetBeans [12] or IntelliJ IDEA [9] include a *quick fix* feature, which helps the developer deal with syntax errors in the source code. In [7], we introduced a transferred functionality for high-level nets in Renew, where the Java-based inscriptions are treated analogously to source code in IDEs. This allows the developer to be given instant feedback on occurring syntax errors by the Petri net IDE Renew.

In this paper, we define the quick fix in Section 2, outline our architecture of this feature in Section 3 and discuss the evolutionary concept behind it during the past development phases in Section 4. At last, we provide a detailed description of our suggestion mechanism based on examples in Section 5. Moreover, we illustrate, how this mechanism can be used as an auto-complete feature. The practical usage of the quick fix is discussed in Section 6 while also pointing out current limitations. After concluding, possible extensions and pursuing uses of the quick fix feature are presented in the outlook in Section 8.

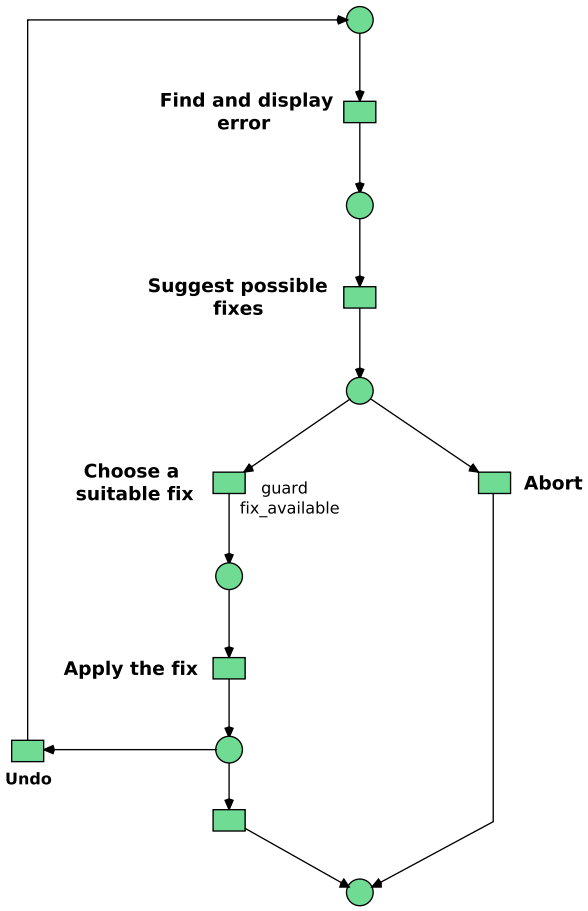

Fig. 1. The quick fix process.

2 Quick Fix

The American Heritage Dictionary describes the etymology of the *quick fix* as follows: "An easy and quick repair or remedy, especially a hastily contrived remedy that alleviates a problem only for the time being" [1]. We actually see the quick fix as a permanent and sufficient repair to a problem, but except for the last part of this definition, it fits our understanding quite well.

In contrast to the *refactoring* [6], which Fowler defines as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [5, p. 53], a quick fix changes the observable behavior of a software. For example, renaming all variable references in a given context is considered a refactoring. In contrast to that, renaming an incorrect reference to the correct one counts as a quick fix. Technically, this

Syntax Error	**Possible Fix**
Constructor not found	Use other constructor, use factory class
Method signature not found	Use other method, change given parameters
Field not found	Use other field from same class or field with that name from other class
Variable not declared	Declare variable in declaration node
Variable and assignment content not compatible	Change variable type or assignment content
Class not found	Create new import statement for class found in classpath

Table 1. Common syntax errors in Renew to be addressed by the quick fix.

seems very similar but the former is considered to be out of scope for this paper and is dealt with in [6].

The typical process for the quick fix is illustrated in Figure 1: First, a syntax error has to be found and presented to the developer, so it can be addressed. Next, possible fixes have to be suggested. The developer can decide, whether there is a fix among the suggestions that alleviates the error and is semantically correct. If so, the fix needs to be applied to the current implementation, otherwise the developer can discard the suggestions and solve the problem on his own. If the developer considers the applied fix not to be appropriate, the fix can be undone, which reverts it to the starting point. In Section 3, we explain the architecture to implement this process in the context of Renew.

Depending on the defined grammar the parser is doing its work on, a variety of different syntax errors can occur. For Renew, we want to address the syntax errors listed in Table 1 using the quick fix feature.

3 Architecture

Renew enables us to use Java code as inscriptions in Reference nets for transitions, arcs and places. Furthermore, an additional figure, the *declaration node*, contains Java import statements and variable declarations. These inscriptions are not pure Java code but Java enriched with additional Reference net-specific syntax (see [11] for details about the inscriptions language). As these inscription are entered as plain text, they obviously can contain syntax errors made by the developer. The quick fix feature addresses these syntax errors in the net, which are found through the net parser; we use *JavaCC* [8] for that purpose. We define the grammar for the different net figure inscriptions using the JavaCC grammar, which is then compiled to a Java parser class. This class is able to find syntax errors, which originate from parsing errors considering the given grammar, and

provide additional information about the error wrapped in a Java exception class, which is called `SyntaxException`.

Whenever a parsing of an inscription is triggered, e. g. after changing it or before a simulation process is started, syntax exceptions may be thrown by the parser or any back-end class and need to be caught and dealt with by the user interface. More precisely, a throw of this `SyntaxException` is the trigger for the quick fix. Each different trigger should comply with a different subclass of that exception and thereby provide the information on what suggestion mechanism should be associated with it. Moreover, the exception should hold detailed information about the error to enable contextualization of the specific suggestion mechanism.

Transferring this to Table 1, each row defines a tuple consisting of a trigger (column 1) and a suggestion mechanism (column 2).

4 Evolutionary Concept

In a very first approach, we developed a frame which would simply display the specific error message given by the parser. This is shown in Figure 2 and Figure 3 in the upper part of the dialog. In this example, this includes the corresponding line and column as well as the specific error message. This functionality implements step 1 of the quick fix process (see Figure 1). In addition to the error message, this information includes the specific figure, whose inscription contains the syntax error. The developer can find the originating inscription by pressing the *Select* button as seen in Figure 3. Especially in very large nets, this information helps to find the culprit quickly and understand, why the net cannot be compiled. Moreover, we added the option to trigger the syntax check manually by clicking a button in the IDE. This allows the developer to constantly check the net for errors while extending it incrementally. However, sometimes the messages generated by the generic JavaCC parser tend to be rather cryptic, especially regarding structuring errors, such as a missing closing bracket or semicolon.

A first improvement considers to add a list of possible expressions for a given syntax error. For example, a list of a class' defined methods is presented to the developer if the parser cannot parse a given method call. This feature required a fundamental change providing the error message, because we need to include contextualized information about the net and make use of the Java Reflection API to determine the suggestions. The developer is now able to compare the implementation with the possible, correct alternatives and rectify it. As a result, we now supply the user interface's error dialog with a detailed error message and an array of feasible fixes. This leads to automation of the second step of the quick fix process. A much more sophisticated suggestion mechanism is explained in Section 5.

The application of suggested expressions is an error-prone manual task, so we wanted to automate it, too. To achieve this, we need information about the precise location of the error's origin in the source code. JavaCC generates token-

based parsers, which are able to supply information about exceptional tokens and their location in the parsed text. This information can be used to add or replace source code in the implementation based on what has been entered already and which suggestion the developer chose to be applied. For example, if the developer tries to add values to a map, the following two expressions have to be treated differently:

```
map.pt(key, value);
map.add(value);
```

In the first line, the developer seems to have made a typing error and forgot the `u` in `put`. In this case, we can leave almost everything as it is and just add the missing letter. In the second example, the developer tries to add a value to the map as if it were a simple collection. In this context, we need to alter the method name as well as the parameter specification, prompting the developer for an additional one. As this simple example demonstrates, the application of quick fix suggestions is not trivial and has to be specified precisely for each suggestion. The specification also has to support a parameterization, so the different contexts can be handled appropriately.

Every action the developer performs in the editor can be undone using Renew's *Undo* operation. This also applies to the application of quick fixes allowing the developer to "preview" and undo it, if not satisfied.

As a result, we have automated nearly all activities of the quick fix process, except the decision for the correct fix among the suggestions. Considering this step, we support the developer by ordering the suggestions by relevance. In some cases, the suggestion mechanism is nearly sure, which suggestion is the right one, in other cases it is only able to provide a broad list. This is obviously dependent on what the developer has already entered and will be explained further in Section 5.

5 Suggestion Mechanism

The first, trivial suggestion mechanism described in Section 3 is a bottleneck for our quick fix process, leading to a high number of *Abort* decisions due to missing suitable fixes. To address this, we introduce a much more sophisticated mechanism for Java class members (Subsection 5.1) and variable declarations (Subsection 5.2).

5.1 Suggesting Class Members

At first, we want to provide a suggestion mechanism for class members. In the Java world, this includes class *fields* and *methods*.

Considering fields, the choices consist of the declared and accessible fields for a denoted class in the source code. If an entered field is unknown, the quick fix can provide a list of fields using the Java Reflect API. However, this can lead to a lot of choices being presented to the developer, when the class defines a great

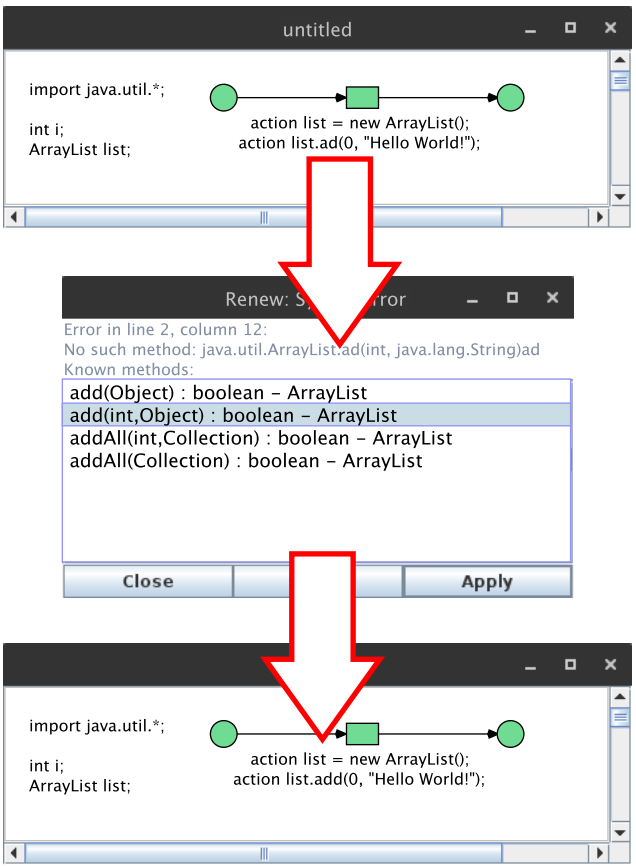

Fig. 2. Quick fix for Java methods with different overloads.

amount of fields. In order to address excessive lists of suggestions, lists may be filtered to fields having the same prefix as entered by the developer.

Similar to the fields, methods can also be searched by using the Java Reflect API. Filtering the list of methods declared by a class may also be done by comparing prefixes. In addition to that, the entered parameters need to be considered when providing suggestions to the user. If the developer entered a method giving the wrong parameter types, the feature can suggest possible method overloads. An example of this mechanism is demonstrated in Figure 2.

This mechanism only considers available class members, which have been declared once in the source code. Generating missing class members the developer references in the net could also be a viable solution. However, we use an external Java IDE to manage and develop our Java source code. In that case, we would need an inter-process communication channel to propagate the required changes to that IDE. Furthermore, the changed source code would need to be compiled

in order to be recognized by Renew, so that the syntax error disappears. We consider this to be too much effort in the current state of development.

5.2 Suggesting Variable Declarations

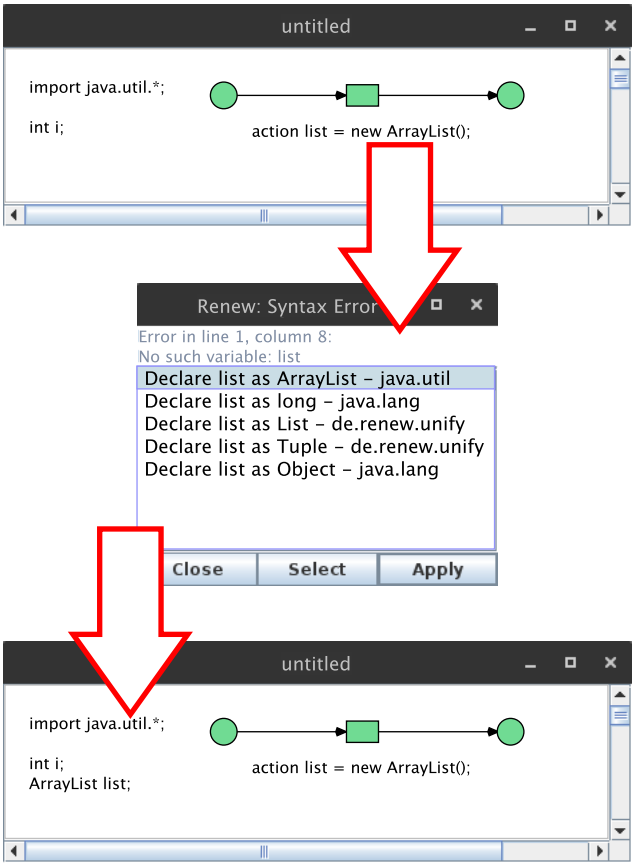

Fig. 3. Quick fix for declaring Java variables.

Variables, that are used in the Petri net can be declared in Renew's declaration node. If the net contains a declaration node with at least one declared variable, all used variables have to be declared. An undeclared variable causes the quick fix to suggest possible types for that variable and adding the corresponding statement in the declaration node.

When assigning a value to a variable, the value can either be a literal, a newly constructed object or the returned value from a method call. Determining

the type in the first case is trivial, because the corresponding literal's primitive type can be evaluated by the compiler immediately. An example of this is given in Figure 3, note that the corresponding package `java.util` has already been imported. To determine the type of a newly constructed object, the constructor's class name has to be well-known, which means the class name is either fully qualified or an already imported class, as seen in Figure 3. Given the case of a well-known class name, the corresponding declaration statement can also be generated easily. Moreover, the information of a fully-qualified class name can be used to construct an import statement for that class or its whole package right away. The last case is also rather trivial, because the return type of a method and its corresponding class object can be determined using the Java Reflect API.

5.3 Using the Quick Fix as an Auto-complete Feature

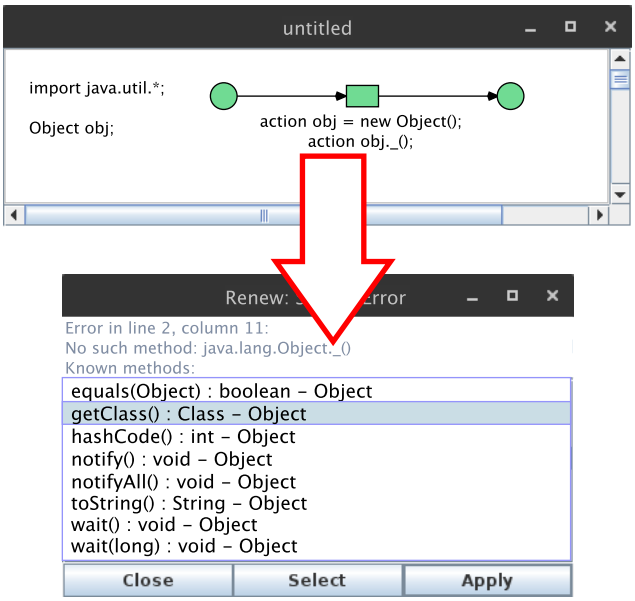

Fig. 4. Using the wildcard character _ to get a list of all methods.

Without suggestions for Java class members, the developer needs to look up every used member in the API documentation. That would require one or more additional applications and their windows to be present in the developer's working environment and tool set. On the one hand, the developer can fix typos or missing parameters etc. using the quick fix suggestions. On the other hand, the quick fix can change the workflow by providing an overview over the class' members, if the developer enters an empty member name on purpose. Pursuing

this approach, the quick fix can be used similar to an auto-complete feature regarding Java class members. For that purpose, we enable the possibility to use an underscore as a wildcard character in order to get a list of all fields or all methods by attaching a pair of round brackets as a suffix. An example of this mechanism is illustrated in Figure 4.

This approach is transferable to variable suggestions analogously considering this example (see also Figure 3): If the developer wanted to construct a new `ArrayList` and assign it to a variable called `list`, the assignment on a transition as well as a corresponding entry in the declaration node would be necessary. Now, the developer can specify the former as needed and omit the latter on purpose and let the declaration be automatically done by the quick fix.

In a nutshell, by providing the quick fix feature, we not only accelerate the development process by supporting the developer finding and fixing errors but also change the developer's workflow while modeling Reference nets. We thereby streamlined the whole development process by enabling a much faster development in an integrated environment.

6 Discussion

Due to Renew's various different configurations and large number of plugins available, the IDE is used in different contexts for many different purposes. For example, one can use it as a process modeling tool such as creating the net seen in for Figure 1, simulate basic P/T nets or even develop agent-oriented software systems using the Mulan/Capa framework [13,4] with the Paose approach [2]. Most likely, the quick fix will neither be used nor triggered considering the use case of the first two examples. In contrast to that, it is very likely, that developers working with the Paose appraoch know the quick fix very well and use it constantly. Hence the quick fix feature will remain hidden for a significant portion of Renew's users, especially because the interaction depends on the defined triggers completely.

Developers, that use Renew in conjunction with Mulan/Capa in order to develop multi-agent software systems, have to deal with a large number of different ontologies, which in turn contain defined concepts (see [2]). These specified ontology concepts are transformed into Java class definitions. As a result, the developers need to develop against a constantly changing API, where the quick fix comes in highly valuable.

As already explained in Subsection 5.2, suggestions for variable declarations can currently be inferred, if the right part of the assignment expression is parseable and the expression's type is known. In particular, this is not true, if the type's class has not been imported yet, even if it is available in the Java classpath. Supporting this depends highly on the ability to automatically import classes found in the classpath and let the developer choose from possible alternatives. Moreover, variables also occur in arc inscriptions. In this case, inferring the type requires an analyzation of other in- and outgoing arcs regarding the particular place or transition and its inscriptions.

The already mentioned IDEs Eclipse (using JDT [15]), NetBeans and IntelliJ (Community Edition) provide similar features within their frameworks, which are available as open-source. However, the fact that the inscriptions in our Reference nets are not fully Java-compliant, because they can contain calls to synchronous channels for example, requires us to adapt these frameworks to our grammar. All three frameworks do not use a parser generator for their Java parsers, which highly increases the required effort to adapt them. Moreover, the existing tools do now allow to throw in single statements to be parsed without a given context. Normally, the context would be given through the enclosing class file containing imports etc., in Renew, we use the declaration node for import and variable declaration statements. These circumstances would also need to be customized. In a nutshell, the frameworks deliver the desired quick fix functionalities but lack customizability for other grammars. The adaption and maintenance for a second Java parser framework to be incorporated with Renew is considered too much effort.

7 Conclusion

Firstly, we provide a definition of the quick fix, that is already part of many modern IDEs such as Eclipse [14], NetBeans [12] or IntelliJ IDEA [9]. We also define a process for the quick fix in Figure 1 and an exemplary list of quick fix scenarios in Table 1.

Considering the architecture, we show that profound changes to the parser and UI were necessary to realize the implementation. We introduce an extensible architecture using Java exceptions with inheritance as triggers and the opportunity to incorporate arbitrary suggestion mechanisms.

The evolutionary concept behind the feature has been explained in detail in Section 4. Each step incrementally increased the usability of our IDE by providing automation of reoccurring tasks while also requiring precise specifications. Thus, we achieved an automation and support of most of the activities in the quick fix process.

In addition to that, using the quick fix as an auto-complete feature changes the development workflow by also providing a quick overview over the used API as discussed in Subsection 5.3. Thus, the quick fix not only accelerates the development process but also streamlines it in a way, that makes use of more integrated features in the Renew environment.

8 Outlook

Currently, the triggers for the quick fix mechanism are based on syntax errors found by the net parser (see Table 1). Many common compilers differentiate between syntax errors and warnings, which for example indicate possible runtime errors. These warnings can also be treated using the quick fix analogously to errors while also indicating a lower severity.

Moreover, the extensible architecture allows us to define arbitrary triggers and corresponding suggestion mechanisms. Hence, the quick fix can support achieving not only a correct syntax but also arbitrary properties of the software system. For example, if the property of a cycle-free net is required, a detection algorithm can serve as a trigger for the quick fix while providing possible changes to break the cycle as suggestions.

Furthermore, the intent of "abusing" the quick fix suggestions as a quick overview over an API can be supported by extending the overview with additional documentation content. More precisely, a method suggestion can be extended by providing the corresponding Javadoc.

The current implementation provides quick fixes only for Java related problems. Since the used inscription language also contains Reference net specific syntax it should also be possible to provide fixes for respective errors. The parser is already capable of detecting erroneous syntax, but many problems can first be discovered during runtime. For example, the unifiability of a synchronous channel may fail as a result of a wrong channel name entered by the developer. Extending the quick fix for these kind of errors requires a different architecture and mechanism of error detection.

References

1. American Heritage® Dictionary of the English Language. Houghton Mifflin Harcourt Publishing Company, fifth edn. (2011), quick fix
2. Cabac, L.: Modeling Petri Net-Based Multi-Agent Applications, Agent Technology – Theory and Applications, vol. 5. Logos Verlag, Berlin (2010)
3. Cabac, L., Haustermann, M., Mosteller, D.: Renew 2.5 – towards a comprehensive integrated development environment for Petri net-based applications. In: Kordon, F., Moldt, D. (eds.) 37th International Conference on Application and Theory of Petri Nets, Toruń, Poland. Lecture Notes in Computer Science, vol. 9698. Springer-Verlag (Jun 2016), (to be published)
4. Duvigneau, M., Moldt, D., Rölke, H.: Concurrent architecture for a multi-agent platform. In: Giunchiglia, F., Odell, J., Weiß, G. (eds.) Agent-Oriented Software Engineering III. Third International Workshop, Agent-oriented Software Engineering (AOSE) 2002, Bologna, Italy, July 2002. Revised Papers and Invited Contributions. Lecture Notes in Computer Science, vol. 2585, pp. 59–72. Springer-Verlag, Berlin Heidelberg New York (2003), http://dx.doi.org/10.1007/3-540-36540-0_5
5. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
6. Friedrich, M., Moldt, D.: Introducing Refactoring for Reference Nets. In: Petri Nets and Software Engineering. International Workshop, PNSE'16, Torun, Poland, June 20-21, 2016. Proceedings (2016)
7. Hicken, J., Cabac, L., Haustermann, M.: Introducing the quick fix for the Petri net modeling tool Renew. In: Moldt, D., Rölke, H., Störrle, H. (eds.) Petri Nets and Software Engineering. International Workshop, PNSE'15, Brussels, Belgium, June 22-23, 2015. Proceedings. CEUR Workshop Proceedings, vol. 1372, pp. 317–318. CEUR-WS.org (2015), http://CEUR-WS.org/Vol-1372

8. JavaCC Project: JavaCC – Java Compiler Compiler – The Java Parser Generator (Apr 2016), https://javacc.java.net/
9. Jetbrains s.r.o.: IntelliJ IDEA – Capable and Ergonomic Java IDE (Apr 2016), https://www.jetbrains.com/idea/
10. Kummer, O., Wienberg, F., Duvigneau, M., Cabac, L., Haustermann, M., Mosteller, D.: Renew – the Reference Net Workshop. Available at: http://www.renew.de/ (Jun 2016), http://www.renew.de/, release 2.5
11. Kummer, O., Wienberg, F., Duvigneau, M., Cabac, L., Haustermann, M., Mosteller, D.: Renew – User Guide (Release 2.5). University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg (Jun 2016), http://www.renew.de/
12. Netbeans Community: Netbeans IDE (Apr 2016), https://netbeans.org/
13. Rölke, H.: Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen, Agent Technology – Theory and Applications, vol. 2. Logos Verlag, Berlin (2004), http://logos-verlag.de/cgi-bin/engbuchmid?isbn=0768&lng=eng&id=
14. The Eclipse Foundation: Eclipse: The Eclipse Foundation open source community website (Apr 2016), http://www.eclipse.org/
15. The Eclipse Foundation: JDT – Java Development Tools (Apr 2016), http://projects.eclipse.org/projects/eclipse.jdt