A Tool Chain for Test-driven Development of Reference Net Software Components in the Context of CAPA Agents

Martin Wincierz

Theoretical Foundations of Computer Science (TGI)
Department of Informatics, University of Hamburg, Germany
http://www.informatik.uni-hamburg.de/TGI/

Abstract Testing is common practice in the realm of software engineering. Especially in agile approaches, where test-driven development can be seen as integral.
Capa agents are developed under the agile Paose approach. Their internal components are implemented using Java reference nets which combine the semantics of P/T nets and Java. The existing testing methods for these kinds of nets are either difficult to learn or ill-suited for test-driven development and regression testing.
In this work a tool chain is presented which allows testing of reference nets using regular Java classes. For this purpose an extension of the well-established *JUnit* framework is provided. All tools are designed to be easily understood by developers of Capa agents. This is achieved by a mixture of automatic code generation, repurposing other tools of the Paose approach, and employing a style of testing that is reminiscent of regular Java tests.
While most of the code generating tools are limited to the context of Capa agents, regression testing is possible for any kind of reference net. The findings may help in the development of testing frameworks for other high-level Petri net formalisms.

Keywords: High-level Petri nets, testing, agile development, agile modeling, regression tests, automated tests

1 Introduction

Capa (Concurrent Agent Platform Architecture) allows for the construction of software agents based on reference nets [6]. These are developed under the agile Paose (Petri net-based Agent-Oriented Software Engineering) approach which is described in [4] and expanded upon in [9]. This approach employs the guiding metaphor of the *multi-agent system of developers*. The communicative nature of agile procedures mirror the developed agent systems. Many agile practices such as Pair-programming and common code ownership are already part of Paose. Until now however, there was no explicit support for automatically executed regression tests, let alone test-driven development, both of which are part of many other agile approaches.

The tools presented in this work allow for the test-driven development of regression tests for Capa agent components. These tests are written using the *JUnit* framework which is already well-supported by many continuous integration softwares. Furthermore the techniques presented are at least partly applicable to general reference nets. As such it is of interest to ask if the ability to develop high-level Petri nets under a test-driven approach is useful in a more general case outside of Capa agents. This will be discussed in the first part of this work, before introducing the tool chain in the second.

2 Background

We briefly introduce (Java) reference nets, the general structure of a Capa agent, some important aspects of Paose and the widely used *JUnit* framework.

2.1 Java Reference Nets

Java reference nets, from here on simply referred to as reference nets, were first introduced by Kummer [12]. They support a hierarchical nets-within-net structure (with nets as tokens), as well as Java inscriptions that are executed when a transition is fired [13]. Reference nets can be executed directly by the Renew Petri net simulator with true concurrency semantics. Reference nets in Renew consist of templates and net instances that are generated from these templates. The net instances can be considered as objects. The semantics allow for most Java commands as well as some additional statements like guards and synchronous channels which are explained under Net Interfaces. Tokens are references to reference nets / Java objects. The content of the Java objects can be changed / manipulated by Java code being executed by firing a transition.

Net Interfaces The interfaces of reference nets are implemented with synchronous channels. These consist of two parts: an *uplink* and a *downlink*, as depicted in Figure 1. Downlinks have the form `<target net instance>:<channel name>(<parameter>*)`. Uplinks have the same form, except they do not have a target net instance. When an up- or downlink is enabled, the simulator tries to find a corresponding counterpart with the same channel name and matching parameters. Previously unbound parameters are bound to the value provided by the other channel if applicable. This allows exchange of objects / values / references between net instances. If a matching channel is found and all parameters can be bound to a value, both transitions fire synchronously. More than two transitions can be synchronized by using more channel inscriptions, with the restriction that only one uplink is allowed per transition.

Stub Classes Reference nets are themselves Java objects. To allow easy access to their interfaces one can use *stub classes* which are generated from stub files with Java-like syntax. They basically function as adapters and make net

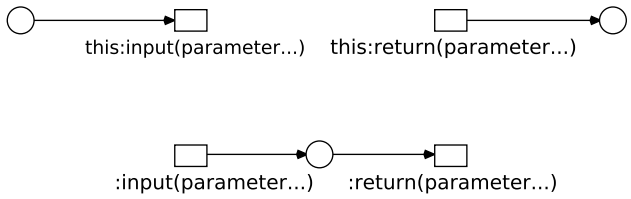

Figure 1. Example of synchronous channels. Top row: Downlinks for the channels named "input" and "return". The target net instance in this case is the same net, represented by the keyword "this". Bottom row: Corresponding uplinks.

instances available for Java classes. Usage of stub classes actually allows to exchange every Java object by a reference net instance and vice versa.

The example shown in Figure 2 shows the stub syntax and resulting Java code for one of the standard interfaces used in PAOSE. The channel name "newExchange" is a standard name that is used multiple times. The specific channel instance is identified with the provided String id. Since this id will never change during runtime, it is declared in the method body. This simplifies the job of testers, as they do not have to know the channel instance of the net. Instead this task is shifted to the person writing the stub file. This is important because the correct implementation of the stub class depends on the use of the interface. Both, uplinks and downlinks, can be used to send and receive information, sometimes both at once. Since Java only allows for a single return value, channels might require multiple corresponding Java methods.

```
1  break void input
2  (Object o,int id) {
3  String s="testChannel";
4  this:newExchange(s,o,id);
5  }
```

```
1  public void input
2  (final Object ppo, final int ppid){
3  ...
4  SimulationThreadPool.getCurrent().
5  execute(new Runnable() {
6    public void run() {
7    de.renew.unify.Tuple inTuple;
8    de.renew.unify.Tuple outTuple;
9    ...
10   outTuple=de.renew.call.
11   SynchronisationRequest.synchronize(
12   _instance,"newExchange",inTuple);
13 }
```

Figure 2. Example for a stub file and generated stub class code. The stub syntax (left) blends elements of Java methods with the syntax of synchronous channels. The resulting Java code can be seen on the right. Synchronization requests are handled by the simulator in the same way as regular transition occurrences are handled. The keyword "break" in the stub file causes the synchronization request to be written in a separate *Runnable*. In this case, it allows us to call the "input"-method several times without having to wait for the simulator to process the request. Later on we use this to emulate a live environment by giving full control to the simulation engine.

2.2 Capa Agents

Figure 3 shows the architecture of a Capa agent. For testing purposes four elements are of interest:

The agent itself is accessible through the "receive" and "send" transitions which are inscribed with synchronous channels of the same name. Messages given as parameters via synchronization are Java objects of a specific type.

Protocols are tied to an act of communication and exist only as long as that communication lasts. They are implemented as instances of reference nets and are held in the place labeled "conversations".

Decision components as well are implemented as instances of reference nets. They are however instantiated when the agent is started and their lifetime is usually the full runtime of the agent itself. They are used to model proactive behavior of the agent, but also to implement services used by multiple protocols.

The exchange transition (located between conversations and decision components) is used to synchronize uplinks of the "dcexchange" channel in protocols with uplinks of the "exchange" channel in decision components. The same channel can be used to allow communication between two decision components (not modeled in the figure). Individual instances of these uplinks are identified via additions to the channel name provided as String parameters. Examples of this can be seen in Figure 4.

Agent-, protocol- and decision component-nets, as well as their respective interfaces, are the main concern when black-box testing Capa agents.

2.3 Some Paose Concepts

Full comprehension of the Paose approach is not required in order to understand this work. Therefore this section will only address two tools which are part of the Paose development process and which are used during testing later on.

Net Components are subnets which serve as templates for recurring functionality within the nets. For example there exist net components for the aforementioned "exchange" channel, as seen in Figure 4. Net Components consist of regular net elements and are only visually distinguishable, i.e. they are easily recognized by humans, but no meta information about them is kept in the net template.

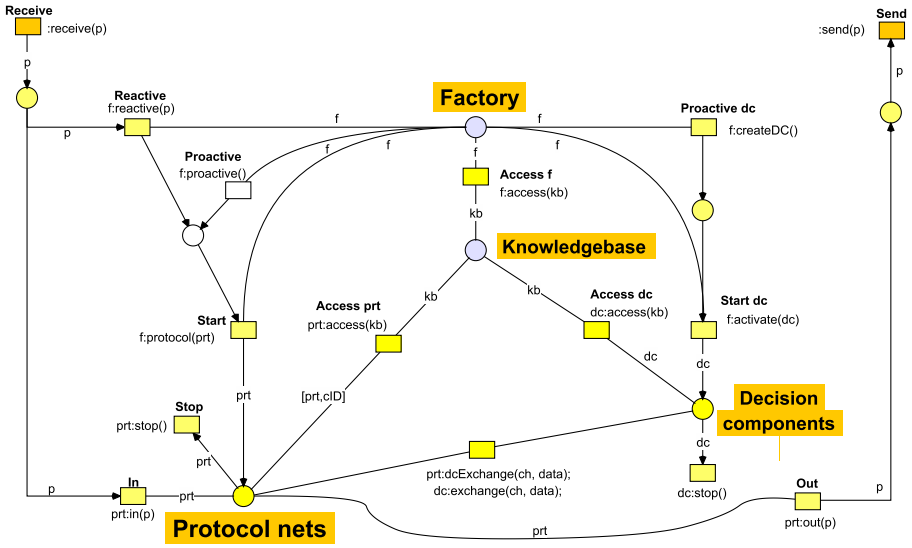

Figure 3. A standard CAPA agent, stripped of some elements for better readability.

Agent Interaction Protocols (AIP) are extended UML sequence diagrams. They are used to model interactions between different agents[1]. An example can be seen later in Figure 7. It is possible to automatically generate net skeletons of protocol nets from the models. This is done by mapping the inscriptions on the diagram elements to Net Components which are then drawn and connected in the same order.

2.4 The JUnit Framework

JUnit is a testing framework for Java applications [1]. It is designed for the creation of regression tests. Automatic execution of *JUnit* tests is supported by many continuous integration programs. The testing process is shown in Figure 5.

Tests can be started, manually or automatically, from the IDE or from the command line using build tools. To execute the tests a controller class called test runner is used. Often test classes specify which runner is supposed to execute them.

The runner creates a `TestResult` object which is usually used to generate a test report. The design of the report depends on the implementation but often includes the number of failed and succeeding tests, the expended time and the stack trace in case of a failure.

JUnit4 heavily relies on reflectively manipulating tests by use of annotations. The `@RunWith` annotation is used to specify the runner class. `@Test` marks the tests themselves. Optionally a time limit may be set, which is useful if deadlocks

[1] More specifically the interactions between agent roles.

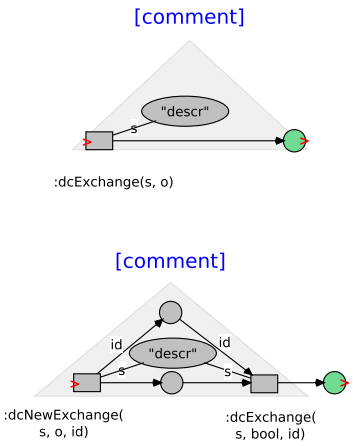

Figure 4. Net Components to be used in decision component nets. The placeholder String "descr" is replaced by the channel name identifying the channel instance and is given as the parameter 's'. The other parameters are 'o', an object reference which is either given or received, and 'id', an integer id created by the agent to map requests to answers.

are a concern. `@Before` and `@After` are called before and, respectively, after each test. They are used for setting up the testing environment and returning it to its prior state after testing.

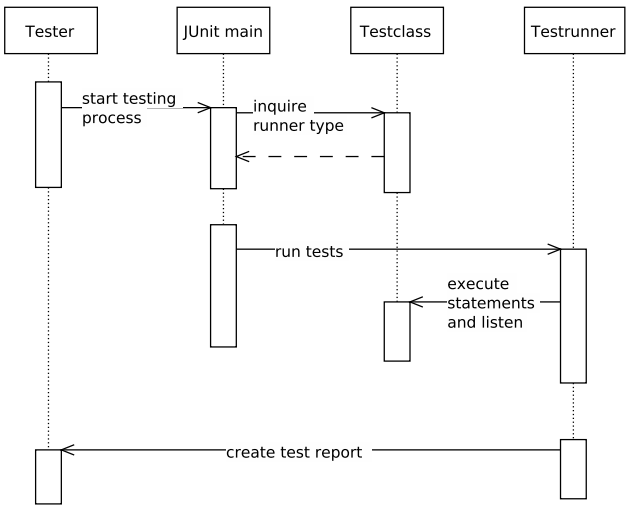

Figure 5. A sample interaction diagram for the *JUnit* testing process.

3 Petri Nets in Agile Development

The core practices of Agile Modeling are introduced and it is shown that Petri nets can be used in conformance with them. It is also argued why Petri nets are especially useful as a modeling language in agile development.

3.1 Agile Modeling

Agile Modeling, as it is presented by Ambler [2], is, much like agile development, not a specified process, but attempts to be a guideline to modelers. "Agile Modeling is not a prescriptive process. [...] it does not define detailed procedures for how to create a given type of model, instead it provides advice for how to be effective as a modeler." [2] Its ideas mirror those of agile development and Ambler explicitly shows its conformity with *eXtreme Programming* [2]. *Agile Modeling* is based on its own set of principles which is put into action via eleven core practices organized into four categories [2]. In this section it is shown that Petri nets can be used according to these practices and are therefore applicable to *Agile Modeling*.

Iterative and Incremental Modeling The first practice is *apply the right artifacts*. Petri nets cannot be considered a universal modeling language but are useful in modeling concurrent behavior. For these kinds of tasks, they are indeed the right artifacts. Similarly the practice *iterate to another artifact* is easy to fulfill if we assume that Petri nets are not our only means of modeling. If one is stuck during the modeling of a net, switching to a different modeling task may bring clarification.

The practices *create several models in parallel* and *model in small increments* require a specific style of nets. More precisely they require nets that are limited in their scope. This can be achieved by hierarchical net structures, as is done in Renew with the nets-within-net approach or CPNTools with hierarchical Coloured Petri Nets [11]. The nets-within-net formalism even allows the exchange of subnets at runtime.

Teamwork The practices of this category are *model with others*, *active stakeholder participation*, *collective ownership* and *display models publicly*. All of this is part of the Paose approach and has been successfully done within the context of a Petri nets-based software development environment. [9]

Simplicity This category encompasses the practices *create simple content*, *depict models simply* and *use the simplest tools*. Again, it is assumed that Petri nets are only used to model concurrent software components. The nets can be refined from simple P/T nets into high-level Petri nets. There exists a number of modeling tools but for early designs they can also be easily drawn by hand.

Validation The first practice of this category, *consider testability*, is the main subject of this work. The second, *prove it with code*, is elaborated on in Section 3.2.

3.2 Combining Model and Implementation

It has been have established that Petri nets can be used as a modeling language in agile development. To take this a step further it is shown that Petri nets are especially useful in agile approaches due to the nets being executable.

Research suggests that there are significant advantages keeping models and code consistent [8]. This is, however, difficult to achieve in agile projects, as the software is continually evolving. Reference nets are Turing equivalent and used as both a modeling and implementation language in our Paose approach. This ensures that the model automatically evolves alongside the code, as they are indeed the same. Petri nets are therefore highly suitable for agile development processes.

The design / testing paradigm favored in *eXtreme-Programming* and other agile approaches is test-driven development [3]. Tests are written before the implementing code and serve as a guideline to programmers. Instead of being seen as additional work after the actual programming job is done, testing is part of the design itself [7]. Executable Petri nets can be seen as both, model and implementation. If they are used in agile development, it is only natural to design them according to agile practices. Therefore the approach to test Petri nets always also incorporates the notion of test-driven modeling.

4 Related Work

The idea of test-driven modeling has been proposed before.

Hawari et al. [10] used the phrase of *test-driven* models. They did not include the technical framework, but rather followed the idea of testing for different parameter simulations. In the following we illustrate how to set up the modeling approach for the use of current software engineering methods.

Zhang and Patel have successfully used similar techniques with executable UML in industry software projects [19,20]. Their process is very close to the one presented later. "First, we create the UML sequence diagrams, then we create both UML model and test cases (for unit, integration, and system testing) according to the sequence diagrams." [19] This work, however, is the first to apply this to Petri nets.

Walkinshaw and Derrick [17] follow the idea of inferring (automata) models from Erlang code and to generate model-based tests according to possible traces of the models to test software. They herewith avoid the involvement of users and gain an automated test generation. In the approach presented the models can be used directly and therefore do not need to be guessed or deduced from some code executions. This is one of the advantage when following a model driven

software engineering approach where the models can directly be used for code execution as in PAOSE .

In the realm of Petri nets, testing is more associated with the techniques of verification or model checking. The presented approach is not competitive, but complementary to these. If the state-space becomes too large or the problem simply becomes undecidable due to added semantics, testing might be a good alternative.

5 Requirements

Decisions that have been made regarding some aspects of the tools are clarified. Requirements and motivations that guided the development are explained.

5.1 Functional and Non-functional Requirements

The testing methods presented are specifically designed to satisfy the needs of agile development. For this purpose three main points that had to be incorporated were identified:

Test-driven Development Adapted to Petri nets, this means tests can be written even before the net structure is known. In the field of testing this is known as black-box testing [15]. Rather than a finished program only the interfaces of the (net-)object are needed. Tests are designed to fail and the code is written iteratively to gradually fulfill the testing requirements.

Small-scale Unit- and System-wide Integration Tests Kent Beck recommended when talking about *eXtreme Programming*:
"If the gap [between writing code and tests] is minutes, the cost of communicating expectations between two people would be prohibitive." [3] The short iterations require the availability of small-scale unit-tests. These are tests of a single net or a small grouping of nets. Beck also acknowledges that usually these tests are not enough:
"A programmer or even a pair bring to their code and tests a singular point of view [...]." He therefore proposes: "One set [of tests] is written from the perspective of the programmers, [...] another set is written from the perspective of customers or users [...]." [3] These tests use the interfaces open to customers. They are system-wide tests, that can be used to avoid unexpected side-effects when integrating smaller software-modules into larger systems. In the context of RENEW and its reference net formalism, which was the main concern of our testing efforts, there is no need to differentiate between the views on a technical level. The nets-within-net property of reference nets allow for hierarchical structures within the nets. System tests are therefore unit-tests of nets that are high in the hierarchy. Trivially, all of these tests have to be regression tests. Once written, they can be called multiple times. During the implementation phase

this is usually done manually by programmers in order to guide them in writing code. Once a test successfully completes, it is called automatically every time new code is integrated into the software. This is done to avoid unexpected side effects and is known as integration testing [15].

Usability and Heterogeneous Skill Sets The importance of this last point is difficult to quantify for a more general case. In academic software-projects however, a significant discrepancy in the abilities of programmers has been observed. As a joint Bachelor's and Master's project, both seasoned programmers with several years of working experience, as well as beginners who have never used a UNIX operating system before are working together [14]. The upfront workload and difficulty of learning PAOSE techniques for the first time often proved to be a hurdle. Therefore one of the goals for this work was to create a testing framework and tool chain that is as easy to use as the rest of the PAOSE techniques, but is as self-explanatory as possible to not further increase the learning time.

5.2 Choosing a Test Language

To write tests the testing-framework *JUnit* is used. Its use was already suggested in earlier works, however favoring a hybrid solution that relied on *JUnit* only for starting and controlling tests. In the first approach the tests themselves were written in the language of the test-objects, i.e. with reference nets [5,16]. In this contribution it was decided on a different approach. The tests are fully written as Java classes. In [18] the approach has been implemented and tested. There are several reasons for this choice.

Familiarity As mentioned earlier, some of the participants of academic CAPA software projects have never seen Petri nets before, let alone used them for programming. However in order to even create something that is in need of testing, a certain degree of Java knowledge can be assumed. By relying solely on Java, tests are not much different from what a Java programmer would usually write, therefore shortening the time needed to learn the testing process. For developers that are completely new to programming with nets, we can also assume that the tests themselves are much less prone to failures and errors compared to the new language of Petri nets. Especially in the context of test-driven development, in which tests are written before the implementation, it is likely that the first attempts using nets are faulty, effectively defeating the purpose of writing these tests.

Support Features RENEW provides some support features to developers, but not nearly as many as comparable IDEs for wide-spread high-level programming languages. The main draw of programming with reference nets, the concurrency, is completely irrelevant for the tests themselves.

JUnit Functionality *JUnit* provides additional functionality, for example methods that are called before each individual test or expecting a test to fail due to an exception.

Separation of Test and Implementation The implementation of the tested functionality is completely separated from its test. The net instance used can be exchanged for a different kind or even a Java class. Apart from technical advantages, this also better conforms to the goals of testdriven development. Theoretically, tests created this way could be written without any knowledge of Petri nets, preventing any chance of the tests being influenced by the implementation.

6 The MulanNetTest Plugin

This Renew plugin was developed as part of a Bachelor's thesis by the author. It was later expanded upon by adding support for the *JUnit* framework. All tools will be explained using the "WebChat" example. A use case can be seen in Figure 6.

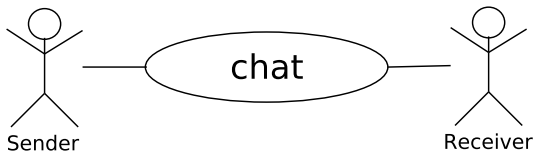

Figure 6. Use case of a simple web chat. A chat message is sent from a 'sender' agent to a 'receiver' agent.

6.1 Extending the AIP

The first task when writing tests before the implementation is done, is to provide interfaces against which can be tested. For agents and protocols this is easy. Agents only provide their standard interfaces. Protocols can be generated from the AIP. Decision components however are entirely created by the developers.

To solve this, the AIP were extended to include both types of internal components. An example for the WebChat application can be seen in Figure 7. Depending on the outgoing and incoming arrows, as well as their inscriptions, net components are generated and connected in the order according to the diagram. From the model in Figure 7 four nets are generated: Two protocol nets and two decision component nets. The net 'Sender_DC' can be seen in Figure 8. The commentary fields employ the new comment tool. It allows developers to append blue text to net elements which is also added to the elements' meta information and can later be retrieved.

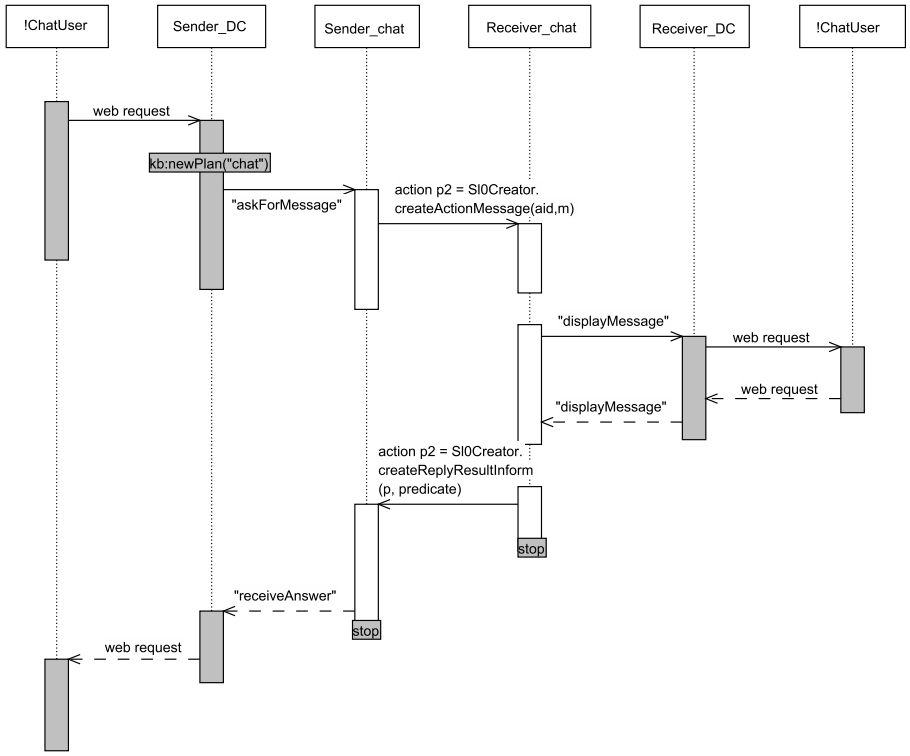

Figure 7. The white figures in the middle are associated with protocols and are part of the original AIP. The gray figures represent decision components. The figures marked with an '!' do not generate net skeletons and represent external access, in this case through a standardized Capa web interface.

6.2 Automatic Net Stub Generation

In order to test the newly generated nets with *JUnit*, an adapter stub class has to be created. This is done automatically by mapping the standardized Net Component interfaces to code in net stub syntax. The Components are identified by naming the place containing the channel name after the interface type. For all places that have non-standardized names getter/setter methods are created. The result can be seen in Figure 9. The commentary is read from the place's meta information and is also written into the final Java class. The stub generation is done within the IDE via a context menu, as seen in Figure 10.

6.3 A JUnit Adapter for Petri Nets

To write the tests themselves the *JUnit* framework is employed and an adapter for this was created. First all newly added elements are explained. An in depth example is given in the next section.

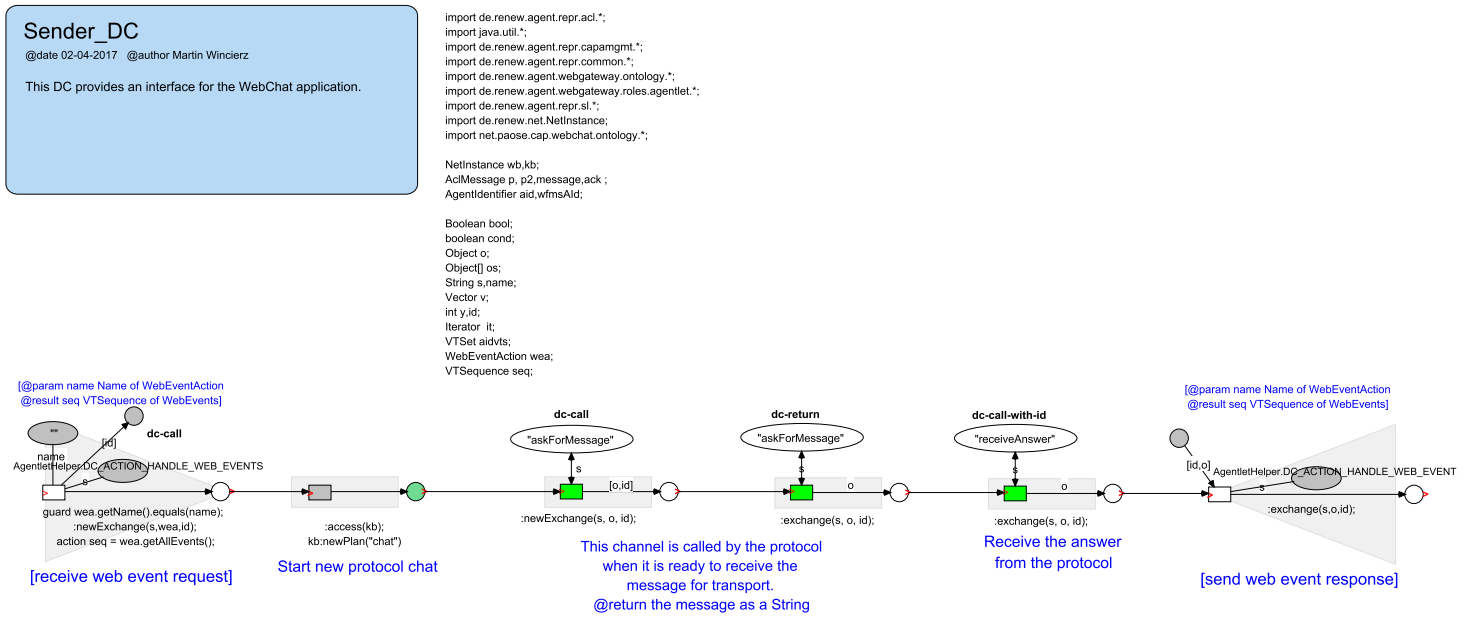

Figure 8. One of the nets generated from the AIP. The elements were rearranged to improve readability and the comment fields and channel names were filled in. Otherwise no new elements were added.

```
1          /**
2          Receive the answer
3          from the protocol
4          */
5          void receiveAnswer(Object o, int id){
6
7          String s="receiveAnswer";
8          this:exchange(s,o,id);
9          }
```

Figure 9. One of the stub methods generated from the net skeleton. The method names are derived from the channel names. If multiple methods share the same name, they are numbered. Illegal characters are automatically removed.

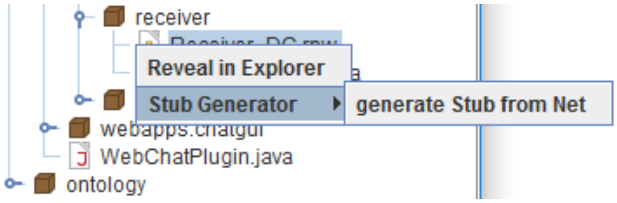

Figure 10. The context menu to generate net stubs. The stub is created in the same folder as the target net and carries the same name supplemented by the suffix "Stub".

Annotations The adapter mirrors *JUnit4*'s use of reflection. New annotations are `@Repeat(int)`, which allows easy repetition of a test class, and `@Concurrent Parameters(Object[][])`, which is modeled after the `@Parameters` class used with the `Parameterized` test runner. It is used to define an array of arrays. For each entry of the super-array, a new test instance will be created and run concurrently. The sub-array values are reflectively injected into fields annotated with `@ConcurrentParameter(int)` according to the given adicity. This is also modeled after the regular *JUnit* functionality of the `Parameterized` runner. Thus users who have used it before, will hopefully understand the principle immediately.

RenewTestRunner The adapter is designed to be very close to regular *JUnit* in its use, as to allow easy adoption of its functionality. The core element is a custom test runner, which automatically starts a new Renew instance. The runner is designed to support the new annotations.

RenewTestClass All tests have to extend this class. This is more akin to the older *JUnit* version 3, where all tests had to extend a `TestClass`. It is necessary because of Renew 's plugin based architecture. New instances of Renew are run in a new class loader to ensure the correct order of loading the plugins. The test runner cannot read the objects annotated with `@ConcurrentParameters`, therefore this task as well as the field injection is done by the test class itself upon

being called reflectively. In addition the `RenewTestClass` provides convenience methods to synchronize testing phases. This is elaborated on in the example.

6.4 Example Test Class

The code example shows a simple test class. The artifact to be tested is the "Sender_DC" net. During setup a new instance of the net is created (line 17/18), which is then wrapped in a stub adapter at the beginning of the test (line 24). The net instance is static because it will be used by the three instances of the test class with the different specified parameters.

The tests are synchronized using the `finishPhase()` method (lines 23,34,38). The test will wait until all instances of the test class have finished the current phase. The phases can be distinguished as executing and evaluation phases. Between each phase the simulation engine is halted / restarted. This ensures that during execution all test instances are active in the same part of the net, thus possibly provoking concurrency failures if existent.

Not all stub methods have been shown, but the methods can be easily matched to their respective channels in the net graphic by their names. In this case only the first half of the net is tested. A message is received from the web interface and given to the protocol net. The test concludes successfully, if both messages are the same. Since no functionality has been implemented, the test will fail after 3000 milliseconds.

```
1  @Repeat(times = 3)
2  @RunWith(value = RenewTestRunner.class)
3  public class WebChatTest extends RenewTestClass {
4
5    @ConcurrentParameters
6    public Object[][] params = {{"house", 1}, {"car", 2}, {"petri", 3}};
7
8    @ConcurrentParameter(value = 0)
9    public String message;
10
11   @ConcurrentParameter(value = 1)
12   public int id;
13
14   static NetInstance instance;
15
16   @Before
17   public void setup() {
18     Net net = Net.forName("Sender_DC");
19     instance = net.buildInstance();
20   }
21
22   @Test(timeout = 3000)
23   public void testMessaging() {
24     this.finishPhase();
25     Sender_DCStub stub = new Sender_DCStub(instance);
26     WebEventAction wea = new WebEventAction();
27     WebEvent we = new WebEvent();
28     VTSequence vts = new VTSequence();
29
30     we.setData(this.message);
31     vts.add(we);
32     wea.setEvents(vts);
33     wea.setName("");
```

```
34
35        stub.AGENTLET_DC_ACTION_HANDLE_WEB_EVENTSIn(wea, this.id);
36        this.finishPhase();
37
38        stub.askForMessageIn(Boolean.TRUE, this.id);
39        String result = stub.askForMessageOut(this.id);
40        this.finishPhase();
41
42        Assert.assertEquals(this.message, result);
43     }
44
45  }
```

7 Discussion

The code generation tools are useful in the context of Capa agents, but difficult to extend to more general cases. The unification mechanism of synchronous channels makes a mapping to Java methods problematic, as the number of required methods increases exponentially with the number of parameters, since any combination of receiving and giving Objects has to be taken into account. Other high-level Petri net formalisms with more specified interfaces might be more suitable.

The *JUnit* extension however allows the testing of all reference nets, provided a net stub is manually created. Writing the tests is about as difficult as testing regular Java classes and should not require special knowledge about Petri nets.

The testing methods have proven to be able to find semantical failures in Capa applications. Other properties, like liveness, are not possible to determine using testing. For this a combined approach with verification techniques might be a solution.

Also the *JUnit* extension in its current form is costly to use. To provide a clean environment, a new instance of Renew is started before each test. This takes about three to five seconds depending on the hardware on which the tests are run.

Despite some flaws the tools presented allow for easier testing and thereby higher quality of code. Automatic test execution will likely improve the integration process during development, although this has to be further observed in practice. The increased degree of testability improves the usefulness of reference nets not only as a programming, but also as a modeling language.

8 Conclusion

High-level Petri nets can combine graphical feedback with early simulation and might therefore be suitable as a modeling language in agile development. Especially of interest is the notion of test-driven modeling, which has been done before with other modeling languages, but has not been tried with Petri nets. A tool chain was presented with which test-driven development of Capa agents becomes possible. The agent's internal components are reference nets which function as both implementation as well as their own model. The testing process was shown using a simple example.

References

1. Junit4. `http://junit.org/junit4/`. Accessed: 2017-01-05.
2. S. Ambler. *Agile Modeling: Effective Practices for EXtreme Programming and the Unified Process.* Programming, software development. Wiley, 2002.
3. Kent Beck. *Extreme Programming Explained.* Addison Wesley, Boston, 2005.
4. Lawrence Cabac. *Modeling Petri Net-Based Multi-Agent Applications*, volume 5 of *Agent Technology – Theory and Applications.* Logos Verlag, Berlin, 2010. http://www.sub.uni-hamburg.de/opus/volltexte/2010/4666/.
5. Lawrence Cabac, Michael Duvigneau, Daniel Moldt, and Matthias Wester-Ebbinghaus. Towards unit testing for Java reference nets. In Robin Bergenthum and Jörg Desel, editors, *Algorithmen und Werkzeuge für Petrinetze. 18. Workshop AWPN 2011, Hagen, September 2011. Tagungsband*, pages 1–6, 2011.
6. Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Concurrent architecture for a multi-agent platform. In Fausto Giunchiglia, James Odell, and Gerhard Weiß, editors, *Agent-Oriented Software Engineering III. Third International Workshop, Agent-oriented Software Engineering (AOSE) 2002, Bologna, Italy, July 2002. Revised Papers and Invited Contributions*, volume 2585 of *Lecture Notes in Computer Science*, pages 59–72, Berlin Heidelberg New York, 2003. Springer-Verlag.
7. Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests.* Addison Wesley, Upper Saddle River, NJ, 2010.
8. Andrew M Gravell, Yvonne Howard, Juan-Carlos Augusto, Carla Ferreira, and Stefan Gruner. Concurrent development of model and implementation. 16th International Conference on Software & Systems Engineering and their Applications, event date: 2/12/2003, 2003.
9. Matthias Güttler. Integration einer agilen Projektmanagementumgebung in ein verteiltes Team. Diploma thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, November 2013.
10. Aliah Hazmah Hawari and Zeti-Azura Mohamed-Hussein. Simulation of a Petri net-based model of the terpenoid biosynthesis pathway. *BMC Bioinformatics*, 11(1):83, 2010.
11. Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems.* Springer Publishing Company, Incorporated, 1st edition, 2009.
12. Olaf Kummer. *Referenznetze.* Logos Verlag, Berlin, 2002.
13. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Lawrence Cabac, Michael Haustermann, and David Mosteller. Renew – the Reference Net Workshop. Available at: `http://www.renew.de/`, June 2016. Release 2.5.
14. Dennis Schmitz. Neugestaltung von Lernmaterialien zur Unterstützung der Lehrenden und Lernenden in der petrinetzbasierten und agentenorientierten Softwareentwicklung. Master thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, February 2016.
15. Uwe Vigenschow. *Testen von Software und Embedded Systems: professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen.* dpunkt, Heidelberg, 2010.
16. Florian von Stosch. Entwicklung eines Testrahmenwerks für Mulan-Protokollnetze. Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, September 2012.
17. Neil Walkinshaw and John Derrick. Incrementally discovering testable specifications from program executions. In Frank S. de Boer, Marcello M. Bonsangue,

Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 272–289. Springer, 2009.

18. Martin Wincierz. Erweiterung des PAOSE Softwareentwicklungsansatzes um ein Testkonzept und Bereitstellung von Plugins zu dessen technischer Umsetzung. Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2016.
19. Y. Zhang and S. Patel. Agile model-driven development in practice. *IEEE Software*, 28(2):84–91, March 2011.
20. Yuefeng Zhang. Test-driven modeling for model-driven development. *IEEE Software*, 21(5):80–86, Sept 2004.