Introducing Refactoring for Reference Nets

Max Friedrich and Daniel Moldt

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

Abstract Modeling of high-level Petri nets requires continuous modifications to adapt to requirements changes. This ultimately leads to structural problems in models. Experiences from software engineering can be used to alter and improve Petri net models, especially if they are used as executables.
In this contribution, we show how to apply refactoring to reference net models. For this purpose, we introduce *bad smells* found in reference nets and refactorings that fix the underlying problems. We present a refactoring tool that supports selected refactorings and is integrated into the Petri net editor and simulator RENEW. We discuss how the results can be applied to other types of high-level Petri nets.

Keywords: Petri Nets, Refactoring, Reference Nets, RENEW

1 Introduction

Refactoring is used in software development with textual programing languages and UML (Unified Modeling Language) modeling to improve software or model quality. It incrementally changes structure while retaining behavior. There is demand for such techniques because both software development and systems modeling require continuous changes that lead to declining software or model quality which is recognizable in shape of *bad smells*. Further reasons to refactor include preparation for future changes and enforcement of good practices.

High-level Petri nets combine Petri net modeling with textual programming. (Java) reference nets are high-level Petri nets that give up parts of the verifiability of Petri nets in exchange for the power of Java inscriptions and the capability to use nets as tokens. It is worth improving reference net models since they are executable and not only used for throw-away prototyping, but also in production software. Therefore, refactoring should be applied to reference nets.

In this paper, we give an overview of refactoring for reference nets. We describe a tool that supports our proposal on how to improve Petri net models without changing behavior. It is integrated into RENEW and supports selected reference net refactorings.

Section 2 introduces reference nets, refactoring and *bad smells*. In Section 3, we discuss exemplary *bad smells* in reference net models. We compile initial classification criteria for reference net refactorings in Section 4. Section 5 is an

excerpt of a first refactorings catalog. We describe a tool that supports a subset of the refactorings in Section 6. Finally, we present related work in Section 7 and discuss a generalization of our experiences in Section 8.

2 Background and Foundations

In this section, reference nets and refactoring are introduced.

2.1 Reference Nets

Reference nets [13] are high-level Petri nets in which tokens can be nets. Java reference nets additionally allow tokens to be Java objects and primitive values. Renew [14] is a graphical development environment for reference nets[1]. In this section, we introduce selected features of reference nets in Renew that are relevant to refactoring. Figure 1 shows a reference net that contains the described features.

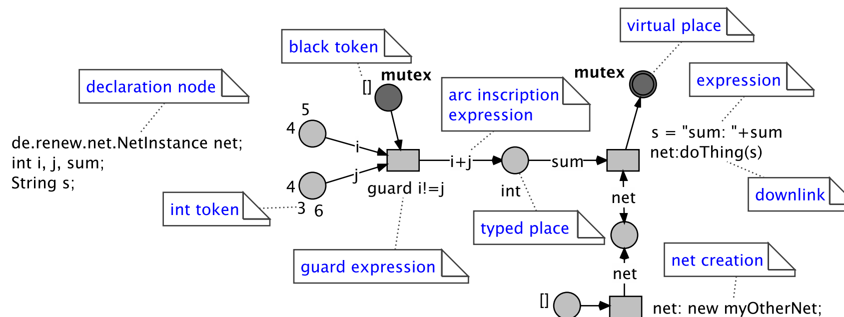


Figure 1. An annotated reference net.

Inscriptions In Renew, places, transitions, and arcs can carry inscriptions. The inscription language is based on Java. Statically typed nets contain a declaration node in which all variables used in the net are declared with a type.

Inscriptions are evaluated during simulation. Unification is used to find a variable assignment that satisfies expressions on transitions and arcs, as well as types on places and in the declaration node. A transition is activated if such an assignment was found. Renew's unification algorithm is described by Kummer [13, Sec. 14.2].

[1] *Reference net* and *Java reference net* are mostly used as synonyms.

Synchronous Channels Channels for synchronous two-way communication in colored Petri nets were introduced by Christensen and Damgaard Hansen [4]. In Renew, synchronous channels allow communication across net instances. Two transitions can fire synchronously if one of them has an uplink and the other has a matching downlink. Uplinks have the form `:ch(x,y)`, in which `ch` is the channel name and `x` and `y` are parameters; Downlinks have the form `n:ch(x,y)`, in which `n` must evaluate to a net reference. An uplink and downlink match if the downlink references the uplink's net, they have the same channel name, and their parameters are unifiable.

Connection to Java Renew's inscription language allows method calls to external Java classes. To call a net from Java, the net needs a stub class that translates method calls to channel calls. Stub classes are compiled to Java classes that provide not only the methods but also a constructor for net instances.

Virtual Places A virtual place is a reference to a place within the same net, marked with a double outline. Virtual places can improve net readability as they can be used instead of long or crossing arcs. However, when using virtual places, a place's preset and postset may not be identifiable at first sight. This can conceal dependencies within nets.

Net Components Net components [3] are reusable subnets that each have a distinct shape and perform one general task. A Renew plugin provides net components that implement control structures. Since the net components follow the Petri net design rules by Jensen [11] that are based on Oberquelle's work [18], modeling with net components results in well readable models *by construction.*

2.2 Refactoring

Fowler defines the noun *refactoring* 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" [7, p. 53]. When used as a verb form (derived from *to refactor*), refactoring means applying refactorings. Thompson's definition in the context of functional programming is similar: "Refactoring is the process of improving the design of existing programs without changing their functionality" [20]. Refactoring counters Lehman's *Law of Increasing Complexity*: "As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it" [15].

Both of the terms *structure* and *behavior* need further examination in this context. Structure is not only dependent on the software itself but also on documentation or features of the development environment that influence programmers' perceptions. It can thus be considered "dynamic" [1].

Software behavior often cannot be entirely expressed in terms of input/output equivalence. When a Petri net of the program's observable behavior at a sensible

granularity is considered, its reachability graph is an additional behavior representation. Beyond that, behavior is heavily context-dependent: e.g. in real-time systems, execution time constraints need to be preserved; in embedded systems, energy and memory consumption play an important role [16]. Petri net behavior must not solely be defined by reachability graphs: if a place with no preset or postset is added, the graph changes because all reachable markings change but behavior can still be considered equivalent.

Fixing programming errors is not refactoring as it purposely changes behavior. Program optimization preserves behavior but often worsens maintainability in exchange for performance improvements.

Refactoring is typically supported by software tools.

2.3 Bad Smells

Bad smells (or *code smells*) are symptoms of structural problems in software artifacts that indicate refactoring opportunities. While the metaphor was popularized by Fowler and Beck [7, Ch. 4], a general effort to create maintainable code and eliminate bad practices predates them, e.g. Dijkstra's "Go to Statement Considered Harmful" [5]. Refactoring does not only neutralize a *smell* but also fix the underlying problem.

Bad smells are no precise criteria. Fowler and Beck find human intuition to trump metrics [7, p. 75] but many development environments today emit warnings when presumed *bad smells* are found.

3 *Bad Smells* in Reference Nets

Our initial approach to collecting *bad smells* was applying Fowler's and Beck's *bad smells* for object-oriented software [7, Ch. 3] to reference nets[2]. Classes in object-oriented software loosely correspond to net patterns; objects correspond to net instances; methods correspond to net sections delimited by uplinks or downlinks to synchronous channels (*subroutines*).

Then, we examined nets developed by students for occurrences of the initial set and searched for additional net-specific *smells*. If a net section raised questions in code reviews and its functionality had to be clarified, it possibly hinted at a *bad smell*.

A key advantage of high-level Petri nets when compared to pure textual programming is their intuitive graphical representation. A common property of many *bad smells* in Petri nets is that they worsen readability and thereby reduce this advantage. We assume that net components that are based on [11,18] are used in modeling. Therefore, simple layout problems that can be solved by rearranging net elements are not covered in the discussions.

In the following subsections, we present three exemplary *bad smells* that occur in reference nets along with refactorings that can be applied to fix the underlying problems.

[2] This section and the following sections describe results of Max Friedrich's bachelor thesis [9].

3.1 Duplicated Net Structure or Inscription

Duplicated Code is the first *bad smell* for object-oriented software described by Fowler and Beck [7, p. 76]. It makes software hard to modify. They recommend unifying duplicated code, e.g. by extracting it into a method.

In high-level Petri nets, net structure and textual inscriptions contribute to functionality. The *smell* can be applied to both of them.

Since net elements can be arbitrarily arranged, duplicated net structure does not have to be immediately recognizable. Net structure (and inscriptions, if the structure consists of a single transition) can be extracted into subroutines with *Extract Subroutine* [9]. If the duplicates occur across nets, *Extract Net* [9] can be applied.

3.2 Large Net

Large Class [7, p. 78] is a *bad smell* in object-oriented software. Classes should only have a single responsibility and therefore be reasonably small. Large classes are harder to navigate and understand. They complicate change because a developer has to consider a whole class when making changes. Besides, large classes often contain hidden duplicate structures.

Large Class can be applied to nets as *Large Net*. Multiple boxes that mark related net sections often indicate that a net is too large. A rule of thumb says that a net should fit onto one screen page. Additionally, large nets have a negative effect on team productivity. Since there is no merge tool for reference nets under version control, only one developer should work on one net at a time.

Extract Net [9] reduces a net's size. If the net contains a complex net structure that could be expressed in a clearer way in Java code, developers can apply *Extract Net Section to Java Method* [9].

3.3 Unclear Control Flow

Murata describes how low-level Petri nets can be used to model data flow computations [17, p. 545]. In high-level Petri nets, it is possible to create data flow models that perform computations, since tokens can contain data. Thus, it seems natural to concentrate on data flow when creating high-level Petri net models, disregarding control flow. A net's marking then implicitly contains control flow. In reference nets, control flow is further divided into all net instances' markings as nets can arbitrarily interact with each other by means of synchronous channels. Unclear control flow can occur even when using net components because many nets need custom structures that are not covered by available components.

Explicating control flow is often worthwhile to improve readability. This especially applies when a large number of data tokens is involved or control flow is further concealed by virtual places and synchronous channels. Besides, explicit control flow tokens are useful in debugging to pin down a failure location.

Introduce Control Flow Place (Section 5.1) explicates control flow.

4 Classification Criteria for Reference Net Refactorings

We propose the criteria featured in Table 1 to classify refactorings for reference nets. The attributes *Number of nets*, *Net features*, and *Associated classes* can be further summarized by the term *Scope*, as they describe the artifacts and structures that are impacted by a refactoring. *Validation* refers to the program life cycle phase in which the success of a refactoring's application can be determined. When structures that depend on unification are modified, validation takes place at run time since unification is only available in this phase. The list of possible values for *Purpose* contains exemplary entries for maintainability improvement refactorings.

All criteria are orthogonal. For *Net features*, *Associated classes*, and *Purpose*, a refactoring may have more than one value.

Attribute	**Possible values**
Number of nets	*single, multiple*
Net features	*inscriptions, net structure, name*
Associated classes	*none, Java classes, stub classes*
Validation	*compile time, run time*
Purpose	*enhance readability, enforce good practices, change interface, ...*

Table 1. Classification criteria for refactorings.

In refactoring catalogs, refactorings are typically divided into chapters of related refactorings but not explicitly classified [7,12]. A reason for this is the homogeneity of most refactorings for textual programming languages regarding the proposed attributes, e.g. typically only one type of artifact is edited.

5 Refactorings for Reference Nets

In this section, we describe three exemplary refactorings for reference nets. Roughly following Fowler's format [7, Ch. 5], each refactoring is presented with a descriptive name, a short introduction, step-by-step instructions for manual application (*mechanics*), and an example. In addition, we classify each refactoring by the criteria compiled in Section 1. The mechanics are purposely explained in small steps. As manual application of refactorings is nevertheless error-prone, a refactoring tool, as described in Section 6, could assume the responsibility of performing the mechanics. Additional refactorings can be found in [9].

5.1 Introduce Control Flow Place

Adding an extra place between sequential transitions to explicate control flow can often significantly enhance readability.

Classification See Table 2.

Attribute	**Value**
Number of nets	single
Net features	net structure
Associated classes	none
Validation	run time
Purpose	enhance readability

Table 2. Classification of *Introduce Control Flow Place*

Mechanics

- Ensure that the transitions are sequential.
- Add a place between the transitions so that its preset contains only the preceding transition and its postset contains only the subsequent transition.
- If the subsequent transition is part of a conflict structure, add alternatives to the postset of the control flow place to make sure that control flow tokens are consumed.
- Optionally add alternatives to the preceding transition to the control flow place's preset.
- Test the system.

Example In Figure 2, let `queryWebService(·)`, `webServiceSuccess(·)`, and `webServiceFailure(·)` be channels with each one uplink and only the depicted downlinks on transitions a, b, and c. The uplinks to `webServiceSuccess(·)` and `webServiceFailure(·)` belong to transitions that will only be activated if `queryWebService(·)` fired before. `webServiceSuccess(·)` and `webService-Failure(·)` are alternative returns, i.e. if one of them fires, the other one is dead. Therefore, the transitions a, b, and c in the example are sequential in such a way that a is before b; a is before c; b and c are mutually exclusive.

In the first step, a place is added between a and b. Since b and c are alternatives, an extra arc is added to ensure that control flow tokens are consumed. Now, a single place explicates control flow between a, b, and c.

5.2 Rename Synchronous Channel

If a channel name does not describe the channel's purpose, it should be changed. This refactoring is similar to *Rename method* for object-oriented software [7, p. 273].

Classification See Table 3.

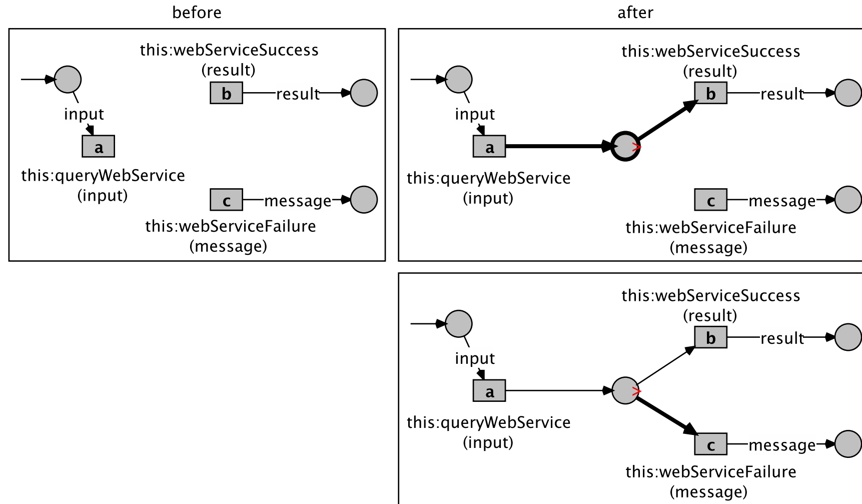


Figure 2. The left box shows the original net section. The right boxes show the process of applying *Introduce Control Flow Place* from top to bottom. In each step, added elements are emphasized with bold outlines. By convention, control flow places are marked with red arrow characters pointing in the direction of control flow.

Attribute	**Value**
Number of nets	multiple
Net features	inscriptions
Associated classes	stub classes
Validation	run time
Purpose	enhance readability, change interface

Table 3. Classification of *Rename Synchronous Channel*

Mechanics

- Check if the new channel name is already used in the system. If it is and the channels could be mixed up (by developers as well as in simulation), choose a new name or perform these steps for the other channel.
- Across all nets in the system, find all transitions with uplinks or downlinks to the channel. (In general, this step is not trivial since uplinks and downlinks are bound to channels at runtime using unification.)
- For each transition with an uplink or downlink to the channel, copy the transition with all arcs and inscriptions and change the channel name[3].

[3] For channels with few uplinks and downlinks, the steps of copying transitions and adding extra places can reasonably be omitted. The channel names are then changed in-place.

– Optionally, add one extra place to each of the original transitions' postsets to check if the transitions fired. (Firings are observable because they produce tokens in the extra place.)
– Change the channel name in all stub classes that refer to the channel.
– Test the system.
– Remove the original transitions and the extra places, then realign the new transitions to restore readability.

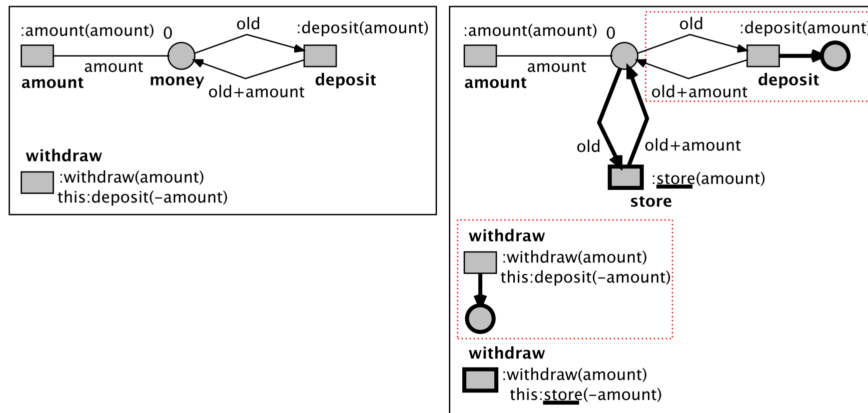


Figure 3. The left box shows the original net section. The right box shows the result of applying *Rename Synchronous Channel*. Added elements have bold outlines; the new channel names are underlined. The elements within dotted boxes can be deleted after ensuring the refactoring's success. Then, the new transitions are moved to the old transitions' positions.

Example Figure 3 shows the net `account` [14, Fig. 3.18] in which the channel `deposit`(·) is to be renamed to `store`(·). Two transitions with references to the channel have been detected in the net. They are copied with their respective preset and postset, then the channel name is changed. Extra places are added after the original transitions. This process is repeated in other nets with references to the channel. After testing and ensuring that the channel with the old name is not used anymore, the original transitions with their incoming arcs and the extra places can be deleted.

5.3 Rename Variable

When the name of a variable does not describe its contents, it should be changed. A specific characteristic of variable names in reference nets is that they should be short yet expressive to keep inscriptions short. A name like `s` for a string can be appropriate if there is only one variable of this type in the net section.

Variable names can be reused throughout a net because the scope of a name is only one transition.

The refactoring can be applied to all occurrences of a variable name or to a part of them, e.g. when a general name is specified in only one net section. We describe only the variant in which all occurrences are replaced.

Classification See Table 4. If the net is statically typed, it can be determined at compile time if all occurrences of the old name were removed and if there were no typing errors. However, the syntax check cannot detect if an occurrence of the old name was accidentally removed or replaced with a different declared name. That is why the refactoring's validation phase is run time.

Attribute	**Value**
Number of nets	single
Net features	inscriptions
Associated classes	none
Validation	run time
Purpose	enhance readability

Table 4. Classification of *Rename Variable*

Mechanics

- If the net is statically typed, add a declaration for the new name to the declaration node.
- Replace all occurrences of the variable name in transition and arc inscriptions with the new name.
- Run a syntax check and test the system.
- If the net is statically typed, remove the old name's declaration in the declaration node. Then run a syntax check and test again.

Example Omitted because of the refactoring's simplicity.

5.4 Summary

Table 5 shows the *bad smells* and refactorings from [9]. It maps *bad smells* to refactorings that can be applied to fix the underlying problems. *Bad smells* and refactorings are both divided into three rough categories: pure reference nets (i.e. without regard to synchronous channels and Java), synchronous channels, Java and inscriptions.

	Rename Net	Extract Net	Introduce Control Flow Place	Rename Synchronous Channel*	Add or Remove Parameter	Extract Subroutine	Rename Variable*	Split Inscription	Extract Net Section to Java Method	Add or Remove Declaration	Introduce Static Typing
Bad Net Name	×	·	·	·	·	·	·	·	·	·	·
Large Reference Net	·	×	·	·	·	×	·	·	×	·	·
Unclear Control Flow	·	·	×	·	·	·	·	·	·	·	·
Duplicated Net Structure	·	×	·	·	·	×	·	·	·	·	·
Complex Net Structure	·	·	×	·	·	·	·	·	×	·	·
Bad Channel Name	·	·	·	×	·	·	·	·	·	·	·
Long Subroutine	·	·	·	·	·	×	·	·	×	·	·
Long Parameter List	·	·	·	·	×	·	·	·	·	·	·
Bad Variable Name	·	·	·	·	·	·	×	·	·	·	·
Long Inscription	·	·	·	·	·	·	·	×	×	·	·
Duplicated Inscription	·	·	·	·	·	×	·	·	×	·	·
Large Declaration Node	·	·	·	·	·	·	·	·	·	×	·
No Declaration Node	·	·	·	·	·	·	·	·	·	·	×

Table 5. Mapping of *bad smells* to refactorings. Refactorings marked with * are supported by the refactoring tool for Renew (Section 6).

6 Refactoring Tool for Renew

In this section, we describe a refactoring tool that is integrated into Renew in form of a plug-in. It provides support for two inscription refactorings: *Rename Synchronous Channel* and *Rename Variable*. The tool is implemented in such a way that adding new refactorings is reasonably simple.

6.1 Rename Synchronous Channel

Application of *Rename Synchronous Channel* (Section 5.2) is guided by a wizard user interface. Since uplinks and downlinks are bound to channels at runtime, the refactoring cannot be executed without additional user input to select uplinks and downlinks that will be replaced. Figure 4 shows the refactoring's application in the net `account` [14, Fig. 3.18]. Again, `deposit(·)` is to be renamed to `store(·)`.

A user selects a transition with an uplink or downlink inscription in the editor and starts the refactoring by clicking a menu item or using a keyboard shortcut. In the first wizard step, the user enters the new channel name and

selects the nets that will be searched for uplinks and downlinks to channels with the original channel name and number of parameters. The second step shows a table in which uplink and downlink matches can be marked for refactoring. The uplink or downlink from which the refactoring was started is always selected. Clicking a "Show" buttons reveals an uplink or downlink in its net pattern. The last step previews the changes in all nets but can still be undone. Net patterns are not saved until the wizard sequence is completed.

In the current implementation, the refactoring skips the steps of copying transitions and extra places since it can be sure that all possible matching uplinks and downlinks were offered to the user. References to channsls in stub classes are not yet included in the search.

6.2 Rename Variable

Rename Variable (in the variant in which all references of a variable name are replaced; see Section 5.3) only spans one net and does not require fine-grained user input like *Rename Synchronous Channel*'s match selection. In testing, we discovered that a wizard based implementation carried too much interaction overhead. Therefore, we experimented with an implementation based on inline text entry with live preview of changes.

An application of *Rename Variable* is shown in Figure 5. In a statically typed variant of `account` [14, Fig. 3.18], the variable `amount` is to be renamed to `dollars`. The refactoring can be started via a menu item or keyboard shortcut when an inscription containing a variable name is selected. In statically typed nets, the refactoring can also be started when no inscription is selected, offering all declared variables for renaming. After the variable selection, the inscription or declaration node changes into a mode where only the variable name can be edited. Other inscriptions containing the name are selected to clarify the refactoring's impacts. As soon as the user starts typing a new name, all references are changed. Invalid names are declined with a red bordered input box. When a valid name is entered, pressing `Ctrl+Return` or clicking outside the text field finishes the refactoring.

If the net is statically typed, the refactoring modifies the variable declaration in-place and skips the steps of adding and removing extra declarations since it can be sure that all references were found.

6.3 Tool evaluation

In this section, the refactoring tool described in Section 6 is evaluated according to Fowler's criteria for refactoring tools [7, pp. 400–406].

Technical Criteria The technical criteria are *Program Database*, *Parse Trees*, and *Accuracy*. The refactoring tool does not have a program database, as searching through reference nets is reasonably fast, even when many nets are involved.

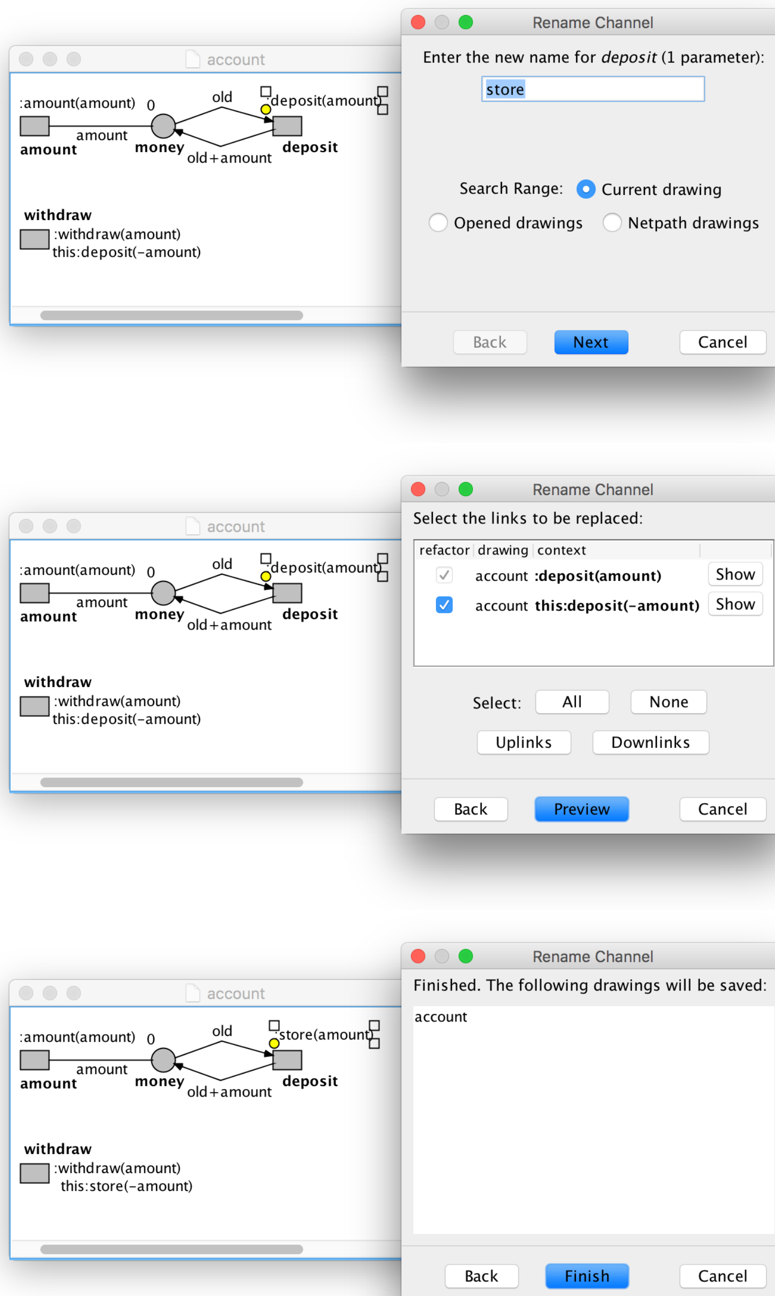
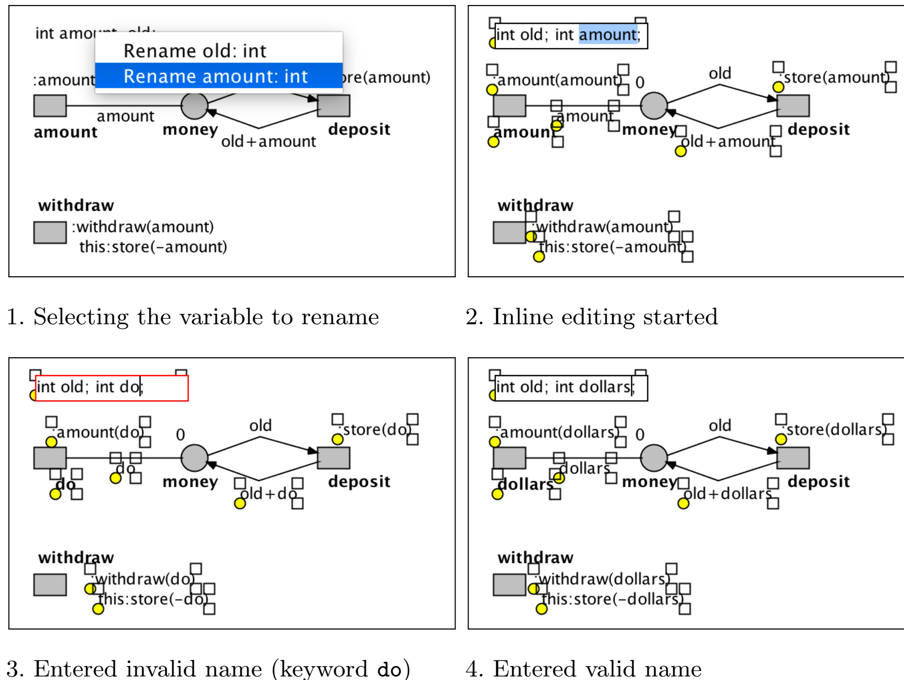


Figure 4. Application of *Rename Synchronous Channel*



1. Selecting the variable to rename
2. Inline editing started
3. Entered invalid name (keyword `do`)
4. Entered valid name

Figure 5. Application of *Rename Variable*

However, much bigger systems are imaginable, in which case a database should be added to the tool.

Renew uses a *JavaCC*[4] generated parser to compile inscription strings into executable objects. It does not emit parse trees and discards token position information after parsing. Only in case of a syntax exception, token position information is returned by the parser in terms of a token object attached to the Java exception. *Quickfix* [10] utilizes this to provide suggestions at the appropriate position. Passing back the token position information by attaching it to the compiled objects posed a challenge in development of the refactoring tool.

Accuracy is guaranteed for refactorings with compile time validation. For other refactorings, additional user input is required, which places the responsibility for correctly selecting matches that will be edited on the developer.

Practical Criteria *Speed*, *Undo*, and *Integrated with Tools* constitute the practical criteria. Both searching through nets and editing nets is fast. By selecting a sensible search range, the developer can speed up search. Inline text editing with live preview of changes proved to be valuable in the *Rename Variable* implementation, as it makes changes instantly visible, which also contributes to speed.

[4] https://javacc.java.net

Undo support is provided in both implemented refactorings. The refactoring tool is well integrated into the Petri net editor Renew.

7 Related Work

Sunyé et al. [19] and Fragemann [8] describe refactorings for UML models such as class diagrams and statecharts.

Models can be considered as graphs, which suggests that graph transformation techniques can be used to refactor models. This has been explored in detail for UML. Ehrig et al. [6] apply graph transformations to Petri nets. The appealing aspect is the formal background that can be used to prove that certain refactorings are correct. This can be of assistance for tools in practical settings of developing executable Petri nets, but as for programming languages, one might want to rely on less tightly regulated modification mechanisms. However, while graph transformations, like refactoring, change structure in small steps, they are mostly applied in order to change behavior, not preserve it.

Berthelot introduces Petri net transformations that preserve classical properties, such as boundedness and liveness [2]. Transformations such as fusing "doubled places" can be considered refactorings. In combination with e.g. Murata's rules for Petri net reductions [17], the boundaries of refactoring with and without changing behavior of models become a relevant issue for tool developers.

8 Discussion

A first step to obtaining a better understanding on how to generalize refactoring could be adding support for more refactorings to the refactoring tool described in Section 6. In addition to reference nets, Renew supports other net types such as Feature Structure nets and Workflow nets, which require specific examinations regarding refactoring.

An analysis tool that detects presumed *bad smells* could be helpful in working with legacy nets. While *smells* like *Large Net* and *Duplicated Inscription* are easily spotted by assigning metrics (e.g. any net with more than 50 places and transitions could be considered a *Large Net*), in-depth analysis is hard because reference nets are Turing complete. At first, a subset of reference nets could be considered for such a tool, before extending it to reference nets and including heuristics to detect *bad smells*. IDEs like *IntelliJ IDEA*[5] tackle this problem by analyzing compiler output and defining properties and changes on higher levels of abstraction.

Other Petri net editors support net classes that implement a different set of concepts with specific constructs. Each construct requires analysis for refactoring, e.g. hierarchical transitions in *CPN Tools*[6]. Beside, different inscription languages like C++, ML, or Lisp require different refactorings, even the latter two functional languages [20].

[5] https://www.jetbrains.com/idea/
[6] http://cpntools.org

9 Conclusion and Outlook

After introducing reference nets and refactoring in Section 2, we describe three *bad smells* that occur in reference nets in Section 3. Two of the smells are applications of *smells* in object-oriented software to reference nets, while *Unclear Control Flow* is specific to nets. A solution for *Unclear Control Flow* is provided in Section 5, along with two renaming refactorings. The refactorings are classified according to criteria presented in Section 4. The renaming refactorings are supported by the refactoring tool for Renew that is described and evaluated in Section 6. We present related contributions in Section 7 and finally discuss the results in Section 8.

Refactoring cannot only be applied to reference nets and Renew, but also to other modeling techniques and tools. Future research should try to formalize refactoring to a sensible degree, putting a stronger focus on semantic aspects.

References

1. Arnold, R.S. (ed.): Tutorial on Software Restructuring. IEEE Computer Society Press, Los Alamitos, CA, USA (1986)
2. Berthelot, G.: Transformations and decompositions of nets. In: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986. pp. 359–376 (1986), http://dx.doi.org/10.1007/BFb0046845
3. Cabac, L.: Net components: Concepts, tool, praxis. In: Moldt, D. (ed.) Petri Nets and Software Engineering, International Workshop, PNSE'09. Proceedings. pp. 17–33. Technical Reports Université Paris 13, Université Paris 13, 99, avenue Jean-Baptiste Clément, 93 430 Villetaneuse (Jun 2009), http://www.informatik.uni-hamburg.de/TGI/events/pnse09/
4. Christensen, S., Damgaard Hansen, N.: Coloured petri nets extended with channels for synchronous communication. In: Valette, R. (ed.) Application and Theory of Petri Nets 1994: 15th International Conference Zaragoza, Spain, June 20–24, 1994 Proceedings. pp. 159–178. Springer Berlin Heidelberg, Berlin, Heidelberg (1994), http://dx.doi.org/10.1007/3-540-58152-9_10
5. Dijkstra, E.W.: Letters to the editor: Go to statement considered harmful. Commun. ACM 11(3), 147–148 (Mar 1968), http://doi.acm.org/10.1145/362929.362947
6. Ehrig, H., Hoffmann, K., Padberg, J.: Transformations of Petri Nets. In: Heckel, R. (ed.) Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004). ENTCS, vol. 148 / 1, pp. 151–172. Elsevier Science, Amsterdam (January 2006), http://tfs.cs.tu-berlin.de/publikationen/Papers06/EHP06.pdf
7. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
8. Fragemann, P.: Refactoring von UML-Modellen. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg (2002)
9. Friedrich, M.: Integration von Refactoring in die Referenznetz- und Paose-Systementwicklung – Diskussion und Implementierung in Renew. Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg (Jun 2016)

10. Hicken, J., Haustermann, M., Moldt, D.: Refining the *Quick Fix* for the Petri Net Modeling Tool Renew. In: Petri Nets and Software Engineering. International Workshop, PNSE'16, Torun, Poland, June 20-21, 2016. Proceedings (2016)
11. Jensen, K.: Coloured Petri Nets: Volume 1; Basic Concepts, Analysis Methods and Practical Use. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin Heidelberg New York (1992)
12. Kerievsky, J.: Refactoring to Patterns. Pearson Higher Education (2004)
13. Kummer, O.: Referenznetze. Logos Verlag, Berlin (2002), http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=deu&id=
14. 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/
15. Lehman, M.: Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68(9), 1060–1076 (Sept 1980)
16. Mens, T., Demeyer, S., Bois, B.D., Stenten, H., Gorp, P.V.: Refactoring: Current research and future trends. Electronic Notes in Theoretical Computer Science 82(3), 483 – 499 (2003), http://www.sciencedirect.com/science/article/pii/S1571066105826246, lDTA'2003 - Language descriptions, Tools and Applications
17. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
18. Oberquelle, H.: Sprachkonzepte für benutzergerechte Systeme. Springer-Verlag, Berlin Heidelberg New York (1987)
19. Sunyé, G., Pollet, D., Traon, Y., Jézéquel, J.M.: UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference Toronto, Canada, October 1–5, 2001 Proceedings, chap. Refactoring UML Models, pp. 134–148. Springer Berlin Heidelberg, Berlin, Heidelberg (2001), http://dx.doi.org/10.1007/3-540-45441-1_11
20. Thompson, S.: Advanced Functional Programming: 5th International School, AFP 2004, Tartu, Estonia, August 14 – 21, 2004, Revised Lectures, chap. Refactoring Functional Programs, pp. 331–357. Springer Berlin Heidelberg, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/11546382_9