An Approach to Formalise Security Patterns

Luis Sérgio da Silva Júnior
Bolsista do CNPq - Brasil
Instituto Federal De Educação,
Ciência e Tecnologia Do Ceará
Fortaleza, Brasil
Email: sergiosilvajr@gmail.com

Yann-Gael Guéhéneuc
École Polytechnique de Montréal
Montréal, Quebéc
Email: yann-gael.gueheneuc@polymtl.ca

John Mullins
École Polytechnique de Montréal
Montréal, Quebéc
Email: john.mullins@polymtl.ca

***Abstract*—The software engineering literature proposes many methods, techniques and tools to ease software development, among which design patterns. The main goal of design patterns is to ease software development through the reuse of good practices in software design and implementation. Design patterns pertain to various domains, including security. In the context of security, security patterns describe design and implementation solutions intended to protect data from a set of possible threats or at least to reduce the risk of their occurrences. Previous works on security patterns defined these patterns and proposed strategies to find occurrences of these patterns using techniques that detect the relationships between software components. However, to the best of our knowledge, these approaches did not describe the behavioural aspects of the components, such as the internal implementation of methods. Behavioural aspects are necessary to investigate and validate the following characteristics: constraints and scope. It is important to guide developers to the correct use of security patterns and preventing wrong implementation and security holes. This article proposes an approach, using Coloured Petri Nets and a set of *API* already available in the Ptidej reverse-engineering tool suite, to formalise and analyse the structural and behavioural aspects of security patterns and identify their occurrences in different kinds of software systems.**

I. Introduction

Design Patterns are “good” solutions to recurring design problems. They have many characteristics and usually are classified with their goal, complexity and behaviour during the system’s life cycle. Gamma *et al.* in their seminal book [1], categorise design patterns as structural, behavioural, and creational. The authors also describe how these classifications have different behaviours that could be or not useful to solve problems during the software development. The design patterns presented in [1] are “generic” because they do not pertain to a particular domain of application. The literature describes also patterns specific to a set of particular domains, including security patterns [2] and antipatterns [3].

In particular, security patterns have been introduced by Joseph Yoder and Jeffrey Barcalo [4] and they have been studied by several authors [5], [6]. Similar to design patterns, each security pattern prescribes a structure and–or a behaviour to avoid security problems. Typical security patterns include:

- Single Access Point: This pattern provides a ”single front door” of a system. It proposes to be the only possible way to access internal system resources from outside. In other words, external entities must not access the system directly. This pattern is the link between the system and its customers.
- Roles: This pattern delegates different levels of access to the system. Each level restricts the access to system’s resources. The users are allowed to access only the resources that they are supposed to get.

Yoder and Barcalow [4] present a classification of these patterns based on their security properties, the threats that they alleviate, their constraints on the structure and behaviour of software components. However, to the best of our knowledge, no previous work investigates if the behaviour of security patters on a running system is correct, as expected from the definitions of the patterns.

Consequently, we propose, through the example of a fictional security pattern, a way to formalise it, verify if its behaviour, previously defined during the example elaboration, is correctly applied in a system and investigate if its implementation is correct. We use *UML* class diagrams, to exemplify the pattern’s structure and formal methods to express its behaviour and the temporal relationships. Formal methods were originally proposed to be used in our approach due their properties to use mathematical concepts to guarantee the formalization during the current state and possible future states of a system. On our scope, formal methods might guarantee the expected behaviour from the security patterns during the life cycle of a system. With these information (i.e., structure from *UML* diagram and behaviour from formal methods), it is possible to verify if these patterns are used correctly in different systems.

Formal methods are a set of tools and techniques to describe formally systems or sub-systems, components or properties of a system. The **OCL**, object constraint language, is a language to formalise constraints on internal entities of a system. Other authors [7], [8] proposed extension of the original **OCL**: **TOCL** or **EOCL**. These approaches expand the original **OCL** and add some characteristics, such that temporal aspects in **TOCL** [8] and help to minimize or eliminate ambiguous modelling aspects. However, other formal languages exist to specify a system and guarantee that it does not enter an inconsistent state during its operation. Among these techniques, we suggest the use of the Petri nets, due their capacity to provide an abstract view of the resources of a system and their mathematical properties. The Petri nets also have a set of tools to simulate how the system would work and prevent inconsistent states.

Petri net is a modelling technique that serves to detail the possible states of a system. It also provides a simulation to observe what happens with tokens and transitions and other petri net entities. Petri nets were proposed by Petri [9]. They are useful in areas where the behaviour must to be strictly formalized (for example, real time systems, distributed systems or concurrent systems).

A Petri net also provides a particular set of entities: tokens, places, arcs and transitions that might provide a visual description to verify the changes on the internal resources of a system.

Jensen [10] introduced Coloured Petri Nets to increase the original abstractive representation of Petri nets using different kind of tokens, guard conditions and other factors that might be helpful to describe interesting problems with more abstraction than its original concepts. We choose Coloured Petri nets for their capacity of simulation and the graphic form to see the behaviour of the modelled systems. An example of Coloured Petri nets is depicted Figure 1. This example shows a Coloured Petri net model of the classic algorithm: the Dining philosophers.

II. State of Art

Security Patterns have been the subject of many works. The first authors to describe security patterns are Yoder and Barcalow in [4], in which they proposed the use of security patterns as recurrent solution to security problems. Authors of [5] and [6] classified security patterns according to the requirements that they fulfill, the kinds of threats against which they must be effective, etc. Based on their classification, in [11], the authors proposed a tool to check whether an example of the Single Access Point pattern is correctly applied in software systems. However, the authors did not describe the method used to investigate the behavioural aspect of the pattern. In the same work, the authors also compared between a previous modelled pattern and a set of structures from an example software and tried to detect occurrence in it. They also analysed how accurate are the detected occurrences found inside the example software based on all entities and associations found on the pattern detection. Authors of [12] presented a study where they investigate if security patterns could be detected by reverse-engineering techniques. The propose of the study was a literature review about the use of reverse engineering to patterns detection. The authors conclude that security patterns are quite different from the design patterns described by Gamma *et al.* and that the techniques used to detect the two families of patterns are also different.

III. Formalising Security Patterns

This work in progress has the main goal to propose an approach and consequently, a tool, to describe formally the security patterns originally proposed by [4] and verify if these patterns are correctly implemented in a software. The expected contributions of this work includes:

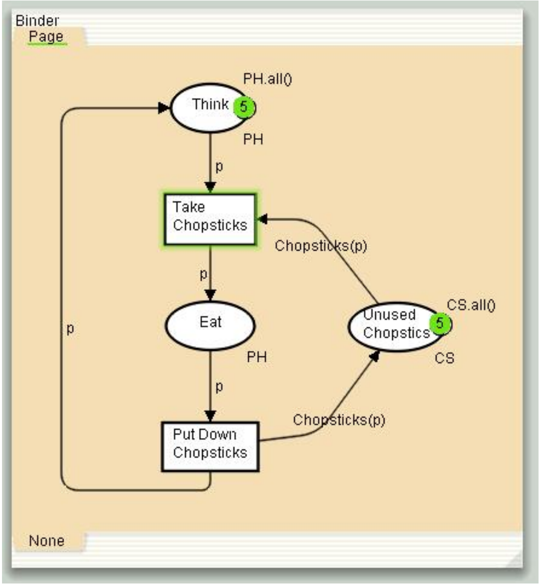

Fig. 1. Illustration of a Coloured Petri Net in CPN-TOOLS [13]

- Formalization of security patterns. It might be useful to promote security requirements during software development.
- Application of Petri net could be used to express the behaviour of a security pattern. Petri nets also might provide an easy way to manage the resources and the states of a system due its graphic representation of entities.

The process we adopt consists of investigating the occurrences of a security pattern in a software. The approach consists in performing two types of analysis on the same system:

- Structural analysis;
- Behavioural analysis.

A. Structural Analysis

The first step consists a generating a model from the chosen security pattern. This model will be “compared” to the model generated from the system. In this paper, we choose the PADL meta-model from the Ptidej reverse-engineering tool suite to model security patterns and the reflection Java API [14] to collect structural information about the statements of the methods declared in the system. Other choices are possible (including other meta-models or dynamic analyses) and will be explored in future work. The information gathered using reflection will be considered as the structural model of the source code. After these steps, the comparison is realised between each model as in Figure 2. We seek a similar structure from the model of the system that can be similar to the pattern model. We plan to explore different matching techniques, in particular the use of error-tolerant graph matching, as in our previous work [15]. Then, the result of the comparison is the most similar structure with the pattern model. The analysis should compare all the structure (classes and their associations) from the PADL

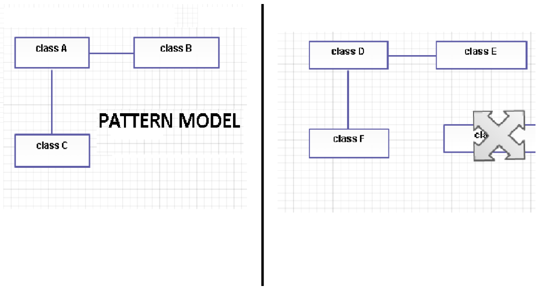

Fig. 2. Illustration showing the comparison on the Structural analysis

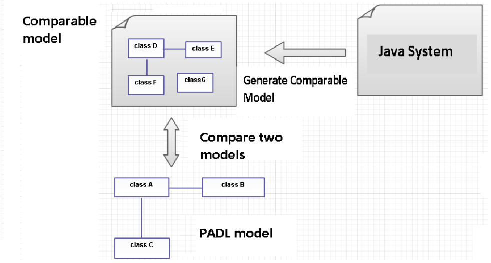

Fig. 3. Illustration showing the Structural analysis

model. After the structural analysis, the structure acquired will be used to provide information about its methods to the next step. The summary of the this step is showed in Figure 3

B. *Behavioural Analysis*

In this step, the approach focuses on the behavioural information about the pattern. This information is formalised by a Petri net. In this Petri net, the entities (tokens, transitions, and so on) describe the information about the behaviour and access of the resources of the pattern. The data from the behaviour of the system is gathered in a model. After the structural analysis, the data from the entities that is not contained in the result of the structural analysis will be discarded. Each entity from the Petri net keeps the information about the behaviour of the system. For example the token (type/color) describes a kind of association. It could be used to express the data type of the entity or resource that is necessary to investigate the behaviour. The transitions and arcs have boolean expressions that could be assumed to express the behaviour from the entities from the pattern before and after the execution of a procedure. The token that goes from an executed Transition to another Place is assumed to represent a resource or variable value inside the system. This information might be used to do a comparison between the internal behaviour of the system with the Petri net model. After the comparison between the models the result is a measure of the closeness of the Petri net from the information found in the Java system.

Thus, the approach will inform how close the pattern models are to the system. The summary of the this step is showed in Figure 4

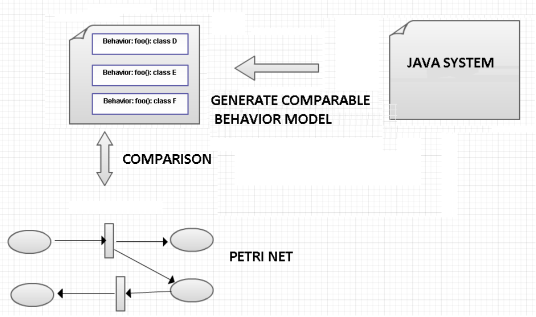

Fig. 4. Illustration showing the comparison on the Behavioural analysis

C. *Proposed tool*

The tool under development is implemented in Java programming language and is divided into two modules to observe the structural and behavioural aspect of some security pattern. The first module uses a model developed using the PADL *API*. PADL is a tool provided by the Ptidej team that reproduces a model of a pattern using its structure to define it using resources with reflection API. The comparison is only to verify the structure (classes and their associations). It is just an approach to seek similarity between the diagram of system and the model proposed and find probable candidates to investigate the internal behaviour. The second one is a module that extracts the behavioural aspects of the software. For example, some constraint of some entity of software and compare it with the constraints previously written by a formal modelling approach. In this module, the tool uses the analysis of coloured Petri nets. These Petri nets are implemented using **CPN-TOOLS** [13]. The result of modelling using this tool is a XML file with all the properties of this Petri net. Basically a file ended by **.cpn**. The data extraction of theses Petri nets was done by the framework **JDOM**.

IV. Case Study

We now present a case study of applying our approach to model structure and behaviour that is at the heart of several security patterns found in [6]. The following patterns: Check Point and Roles have a particular association inside their structure. They have an entity called by Security Policy that works like a connector and it can use a constraint to control the quantity of instances of Roles, or to control the parameters that the Role will receive or another. So the example could be considered a sub–set of a pattern but that is not directly considered a security pattern: the Sender–Receiver design. This example shows the sending of a parameter betweens their entities. The two instances must communicate with each one to transfer the data in a one given path.

We use the the Sender–Receiver to exemplify our approach and validate the idea. Future work includes formalising and validating complete security patterns to assess if they are correctly implemented in real software systems.

In this example, two entities exist called: Sender and Receiver. The Sender is an entity that call the Receiver to send

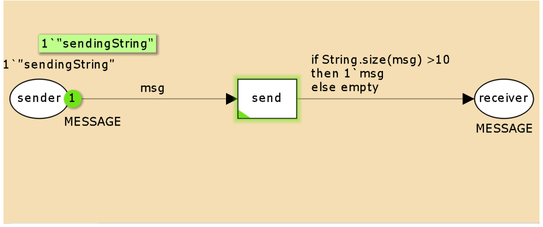

Fig. 5. Coloured Petri Net of model Sender / Receiver

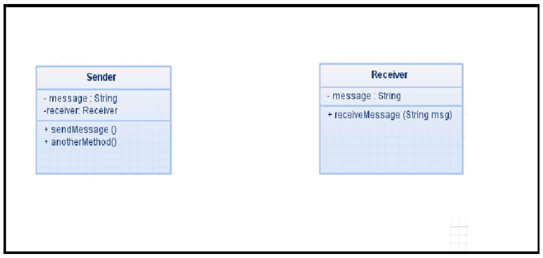

Fig. 6. Class Diagram of Sender / Receiver

a string using the method `toSend(String parameter)`. The Receiver is an entity that receives the message and keep it during all system life-cycle. These entities are represented in the **UML** diagram and Petri net model in Figure 5 and 6. We impose a single constrain on the message between the Sender and Receiver: **the size of the message cannot be longer than 10.**

We now present an example of the use of the tool with the Sender–Receiver. The following text explains all the steps to analyse the information contained on this pattern and compare with the external software.

A. Step 1 – Static Analysis

In this step, the tool compares the objects of the software under analysis with the model of the Sender–Receiver design created using the PADL meta-model. For this case study, we created a single association between the two classes named Sender and Receiver. The result is a PADL class with all implementation describing the components (methods attributes and association between these two entities).

After implementing the model in the first stage. A reference of the object of the previous implemented PADL class is needed to be analysed by the method designAnalysisOfEntity in the implemented tool. This method has 3 parameters, the following PADL model, an reference for the object that is assumed to implement a fictional situation of the Sender–Receiver design and the name of the entity that will be used to do it. We are assuming now at this moment that it is necessary to inform the software each class that is supposed to implement the pattern.

The information from the reference of the external object is captured and extracted from it using the Reflection API provided by Java. In this paper, our use of the Java API provides the relevant information about a target class, its attributes, and its methods parameters, and the method statements. The information provided by this API is used to compare with the information in the PADL model, as said before. The information from inside of the methods of the entities is not obtained in this stage due properties of the Java Reflection API. After the comparison, is generated a measure to show how close is the original object model comparing with the PADL Model. Each class is used and its variables, methods signatures and parameters is compared. After that the difference between the two entities might be viewed and show how close is each one to other.

In summary, in its first step, the tool compares the structural part of the given model of of the pattern implemented by Java language and the PADL. The use of reflection Java API is necessary to extract the data from static class files. After all the comparison, the tool reports the classes in the system potentially playing the roles of Sender and Received. We could then use the typical measure from information retrieval, such as precision and recall, to assess how accurate is the static analysis. After the structural analysis the tool goes to its next step, where the data is extracted from an existing Petri net and try to extract the constraints and verify if these same constraints appear inside of the file of the model structure.

B. Step 2 – Behavioural Analysis

In this step, the tool takes as input the Petri net model of the Sender–Received design to extract information from the Petri net.

To start the analysis, it is necessary to extract the data from the Petri net. Each transition is considered a method call from the previous place that is connected with it. Transitions have guard inscriptions: Boolean conditions that enable or disable the transition before it happens. They also have the Conditional arc inscriptions. Arc inscription is, basically, a group of one or more functions that could be considered close of **OCL** post-condition: in other words, a condition that is checked during the transition with conditional structure.

The method that is called to do the second analysis is called behavioralAnalysis. It has three parameters. The first is a path to the source file of the External entity. It is necessary to get information inside the methods of some class. The Reflection API provided by Java does not give this information. The second parameter is a reference of this same entity. The last one works like a parameter of the first module to send the name of the method to be analysed and compared with the formal model. After that the information from the Petri Net file is extracted and disposed by a Java Structure to represent the components of a Petri Net. Arc, Color, PetriNet, Place, Token, and Transition are a group of representative classes that is possible to obtain all the information provided of the petri net source file. The information contained of the transition `sendMessage` is captured and stored in an Expression list. The same is done with the .java file. The behaviour inside the method `sendMessage` from the Sender class is obtained and keep in a list of Expressions. In this context is necessary to explain two structures that will be used to analyse the behavioural aspect. Expressions: might be looked like condition or attribution of a variable during the execution of a following hypothetical method. Variable: is used to keep all the variables of the pattern class and compare if the petri net model express some constraints or rules using these same entities. The variables will be stored and compared with the colors of the place of the petri net during the analysis. The same procedure is necessary to execute with the petri net model in the .cpn file. The expression condition on this case has 1 element (the condition on the Arc between the transition and the Receiver place); other elements could be added, for example if the token value changes during the transition. It's considered an attribution expression. After this step, the two lists of expressions and variables are compared each one and the result is a simple list of differences between them. During this comparison, the petri net color of each token is comparable with the variables of the software. If they have the same type, the comparison is considered positive. The same procedure is realised with the Expression list.

In summary, in this step, on the one hand, the tool obtains data on the transitions and associates it with the places from the Petri net describing a security pattern. On the other hand, the tool uses a Java extractor to obtain data on the statements in the methods of a system. We cannot use reflection in Java due to its practical limitations in this programming language but conceptually, it is the same. Then, we define a measure $d \in [0, 1]$ that describes how close are the data expected from the Petri nets and that found in the system methods. Our computation of d is currently naive and we will improve it in future work. Essentially if inside the transition of a Petri net, we have only a simple guard condition and inside the analysed method, we have a Boolean expression inside a conditional structure (if or switch) plus an attribution that is located after of if's scope, then we compare each Boolean expression found in the method with the guard condition found in the Petri net's transition. For each Boolean expression matching a guard condition, we set the value of d closer to 1 wrt. the total number of expected guard conditions from the Petri net. After the comparison between each expression, the tool will look for attributions inside the method. For example: if the variable `a` before the method execution changes its value to `a+1` after the method execution, then we expect to see the same change in the corresponding token in the Petri net. If the token changes its value to `a+1`, then we assume that the token has the same attribution. Again, with each matching attributions, we set the value of d closer to 1 wrt. the total number of expected attributions from the Petri net. After gathering this information and computing d, the tool reports whether or not the pattern is correctly implemented or not.

V. Conclusion and Future Work

The proposed tool needs to evolve yet. It expected on the following moment that it should to do a complete scan of a Security Pattern. It will be helpful to expand the case study. Thus, the next step is to implement this same procedure but with the Single Access Point or others most usual patterns, for example Roles or Session, and compare with an external system that try to implement these patterns. It's also necessary to expand it in a second moment of this work to try to validate it on systems that need it. Other approaches should try to include different type of formal methods as **EOCL**, **OCL** or others techniques and methods might be used only to compare the results with the Petri Net model.

A brief consideration: For different criteria, sometimes the implementation of some patterns could be different in a language A from a language B. A clearly example is a pattern implemented using **Lua** [16] programming language or other script language. These languages might has different structures to implement the same patterns. **Lua** [16] uses tables structure instead object oriented programming. So the comparison might produce difference results on a future version of the this tool.

Other topics of interest for future work include:

- Generate a validate version of the tool;
- Compare with more research works;
- Verify patterns using different languages.
- Realise a simulation from the pattern found to see if the analysed behaviour is in the patterns scope.

Acknowledgment

The author would like to thank for their patience, dedication, and help on previous version of this paper Venera Arnaoudova, Foutse Khomh, and Wei Wu.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, Nov. 1994. [Online]. Available: http://www.worldcat.org/isbn/0201633612

[2] C. Steel, R. Nagappan, and R. Lai, *Core security patterns : best practices and strategies for J2EE, Web services, and identity management*. Upper Saddle River, NJ: Prentice Hall PTR,, 2006.

[3] W. J. Brown, R. C. Malveau, H. W. "Skip" McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. Wiley, Apr. 1998. [Online]. Available: http://www.worldcat.org/isbn/0471197130

[4] J. Yoder and J. Barcalow, "Architectural patterns for enabling application security," 1998.

[5] N. Yoshioka, H. Washizaki, and K. Maruyama, "A survey on security patterns," *Progress in Informatics*, vol. 5, pp. 35–47, Mar. 2008.

[6] R. Wassermann and B. H. C. Cheng, "Security patterns."

[7] J. Mullins and R. Oarga, "Model checking of extended ocl constraints on uml models in socle," in *Proceedings of the 9th IFIP WG 6.1 international conference on Formal methods for open object-based distributed systems*, ser. FMOODS'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 59–75. [Online]. Available: http://dl.acm.org/citation.cfm?id=1772150.1772156

[8] P. Ziemann and M. Gogolla, "An extension of ocl with temporal logic," in *Critical Systems Development with UML*, 2002, pp. 53–62.

[9] C. A. Petri, "Kommunikation mit Automaten," Ph.D. dissertation, Institut für instrumentelle Mathematik, Bonn, 1962.

[10] K. Jensen, "A brief introduction to coloured petri nets," in *Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS '97. London, UK, UK: Springer-Verlag, 1997, pp. 203–208. [Online]. Available: http://dl.acm.org/citation.cfm?id=646481.691443

[11] M. Bunke and K. Sohr, "An architecture-centric approach to detecting security patterns in software," in *Proceedings of the Third international conference on Engineering secure software and systems*, ser. ESSoS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 156–166. [Online]. Available: http://dl.acm.org/citation.cfm?id=1946341.1946357

[12] M. VanHilst and E. B. Fernandez, "Reverse engineering to detect security patterns in code."

[13] K. Jensen, L. Kristensen, and L. Wells, “Coloured petri nets and cpn tools for modelling and validation of concurrent systems,” *International Journal on Software Tools for Technology Transfer*, vol. 9, pp. 213–254, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10009-007-0038-x

[14] Oracle, “Java reflection api,” http://docs.oracle.com/javase/tutorial/reflect/index.html.

[15] S. Kpodjedo, F. Ricca, P. Galinier, G. Antoniol, and Y.-G. Gueheneuc, “Madmatch: Many-to-many approximate diagram matching for design comparison,” 2011.

[16] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, “Lua: an extensible extension language,” 1996.