Structural Analysis of the Check Point Pattern

Abdullah A. H. Alzahrani, Amnon H. Eden, Majd Zohri Yafi
School of Computer Science and Electronic Engineering, University of Essex
{aahalz, eden, mzohri}@essex.ac.uk

Abstract—We investigate intuitive claims made in security pattern catalogues using the formal language of Codecharts and the Two-Tier Programming Toolkit. We analyse the Check Point pattern’s structure and explore claims about conformance (of programs to the pattern), about consistency (between different catalogues), and about the relation between (security and design) patterns. Our analysis shows that some of the intuitive claims hold whereas others were found inaccurate or false.

Keywords. Security patterns; Codecharts; design pattern; formal languages; design verification

I. INTRODUCTION

The language of software patterns [1], [2] is traditionally used to catalogue design motifs in common practice. Catalogues of security patterns use a similar language to describe patterns of software security. Each security pattern advocates a certain recipe consisting of structural and behavioural obligations and cites sample implementations. As in other domains of application, patterns of software security are largely described in plain English, commonly organized in the participants and collaborations format, and occasionally using illustrations and UML diagrams. Wasserman & Cheng [3] also formalize some of the specification using temporal logic (LTL) and the Statecharts formalism.

Intent. Check incoming requests. In case of violations the Check Point is responsible for taking appropriate countermeasures.

Structure. A checkpoint is a component that analyses requests and messages. Placed between the Single Access Point and the secured components, it intercepts and monitors all the traffic into the secured part of the system.

Participants

- Check Point: Implements a method to check messages according to the current security policy, triggers actions that might be necessary to protect the system against attackers, grants innocuous messages access
- Countermeasure: Provides actions that can be triggered to react to an attempt to violate access rules.
- Security Policy: Implements the rules which determine whether access is allowed.

Figure 1. The Check Point pattern according to Wassermann & Cheng [3] (abbreviated)
We employ the formal language of Codecharts [4], [5] to formalize the Check Point security pattern (§II) and use it to investigate the following intuitive claims made in catalogues of security patterns: that different catalogues describe the Check Point pattern consistently (§III); that the Java package JAAS conforms to the pattern (§IV); and that Check Point is a special case of the Strategy design pattern (§V). Our analysis shows that the one claim holds whereas two others are inaccurate or false.

II. MODELLING

Consider the Check Point security pattern as described in the catalogue by Wassermann & Cheng [3], summarized in Figure 1. We begin our analysis by formalizing the structural properties of the pattern using the Codechart depicted in Figure 2.

Codecharts are formal specifications in the visual language LePUS3 [4], [6]: an object-oriented design description language. Each Codechart unpacks as a formula in the first-order predicate logic. For example, the Codechart in Figure 2 articulates the structural obligations that Check Point advocates using the following syntactical elements:

- **Class variables** (CheckPoint, Action) represent classes or interfaces
- **Signature variables** (Check, Trigger) represent method signatures
- **Transitive binary relations** (Call+) represent (possibly indirect) relations between elements

Methods are represented by superimposing a signature term over a class term. For example the superimposition `Check@CheckPoint` represents that method in `CheckPoint` whose signature is `Check`.

LePUS3 is limited to modelling fully decidable (recursive) relations. Codecharts can therefore model only structural properties. The advantage in this limitation the ability is to test the conformance of implementations to Codecharts fully automatically, demonstrated in section 0 below.

III. CONSISTENCY BETWEEN CATALOGUES

Software patterns were introduced, among others, to promote shared interpretation of design decisions by packaging their description clearly and unambiguously [1]. Consistency between specifications of a pattern is essential not only in communicating and documenting design ideas but also when checking conformance to design decisions. About three years after Wasserman and Cheng’s paper [3] Schumacher et al published a different catalogue [7] which also listed the Check Point security pattern (Figure 3). Are these two catalogues consistent in their descriptions of the pattern?

Indeed, pattern catalogues do not always offer an unambiguous, obvious evidence for consistency and compatibility with other pattern collections. However, the Codechart in Figure 4, formalizing the description of Check Point in [7], may provide more obvious evidence using the following additional syntactical elements:

- **Hierarchy variables** (CheckPointHrc) represent a set of classes that contains one ‘root’ class such that all other classes in the set inherit (possibly indirectly) therefrom
- **Superposition** is an abstraction mechanism for representing an entire set of correlated methods using only two tokens. Thus for example the set of dynamically-bound methods with the signature Check defined in the CheckPointHrc set of classes is modelled in Figure 4 by superimposing the signature variable Check over the hierarchy variable CheckPointHrc.

Comparing the Codechart in Figure 4 with the Codechart in Figure 2, which formalizes the Check Point in [3], shows that Schumacher et al suggest using only one object to implement both participants CheckPoint and Policy, whereas...
Check Point defines the interface to be supported by concrete implementations to provide the I&A service ... A separate configuration (mechanism) defines which concrete implementation of the Check Point interface to use.

Wassermann et al. (Figure 2) sets them apart. Rather, Schumacher et al. suggest that flexibility in the security policy can be achieved by a hierarchy of Check Point classes, represented as CheckPointHrc in Figure 4.

IV. CONFORMANCE OF IMPLEMENTATIONS

Assumptions about conformance to a given design motif may appear intuitive. However, software systems are often large and complex, and usually undergo frequent evolution steps. Checking conformance to design obligations may therefore be difficult even in early stage of development, and increasingly expensive with the lifetime of the program. It may benefit significantly from automation. Below we demonstrate how structural conformance to Check point is carried out by the Two-Tier Programming Toolkit [8], [9], henceforth the Toolkit.

Schumacher et al. [7] state that Check Point is implemented by the Pluggable Authentication Module (PAM) framework, a standard for providing application-level security [10], [11]. PAM is realized in Java by the Java Authentication and Authorisation Service (JAAS), a package that was integrated into Java SDK 1.4v. Figure 5 depicts a Codechart modelling the part of JAAS that is said to implement Check Point using the following syntactical elements:

1. Class constants (LoginContext, SampleAzn) represent specific, fully defined classes
2. Signature constants (login(), main(java.lang.String[])) represent fully specified method signature
The Toolkit automates the process of design verification which tests conformance, namely whether obligations articulated in a Codechart are satisfied by a given Java program. To check whether JAAS conforms to the Check Point, generic constraints expressed in the Codechart in Figure 2 as relations over variables are made specific by an assignment, namely a mapping which associates each variable to a constant, as demonstrated in Figure 6.

The results of the design verification, depicted in Figure 7, indicate that the specific set of classes and methods depicted in Figure 5 indeed conforms to the Codechart Figure 2, which models the version of the Check Point pattern defined by Wassermann et al.

V. RELATIONS BETWEEN PATTERNS

Similarities and differences between design patterns have been the subject of much discussion [12]. Security patterns are no exception. Schumacher et al.’s description of the Check Point starts with: “Apply the Strategy design pattern [GoF95] to vary the checking behavior”. The claim is repeated and explained e.g., stating that “Check Point interface corresponds to the abstract strategy in Strategy [GoF95].” [7, p. 296]

Again we may use the Codechart of Strategy pattern, depicted in Figure 8. The elements missing from Schumacher et al., highlighted in red, may be minor omissions from the text. For example, the context participant, holds a reference to (which can be implemented as a data member or an attribute of) the hierarchy of strategies, whereas no such relation exists between the corresponding participants in Check Point (SingleAccessPoint, CheckPointHrc). Instead, the UML diagram (Figure 3) specifies that Check Point contains a more general relation between the participants, that of ‘uses’.
VI. CONCLUSIONS

We used the formal language of Codecharts and the Two-Tier Programming Toolkit to investigate some intuitions about the Check Point security pattern. Our analysis shows that some of the claims hold, for example that JAAS implements the Check Point pattern as described in [3]. However our analysis also shows that the Check Point and the Strategy patterns [7] are not as similar as it may seem. Finally, we concluded that different catalogues vary widely when describing patterns with the same name. We conclude that formal languages such as Codecharts can be used effectively in testing intuitions and common assumptions about security patterns.

VII. ACKNOWLEDGMENT

The research in this paper was funded in part by the Silberrad Scholarship; by the Line-Funded Exploratory New Starts (LENS) fund, the Software Engineering Institute, Carnegie Mellon University; and by Umm Al-Qura University.

REFERENCES