SPRing: Iterative Overcoming of Software Misuse

Leah Goldin[1] and Reuven Gallant[2]

[1]Afeka Tel-Aviv Academic College of Engineering, Tel-Aviv, Israel
[2]JCT Lev Academic Center, POB 16031, 91160, Jerusalem, Israel

Abstract. Software misuse may cause very undesirable and expensive outcomes. Our work has proposed and we have been developing *a priori preparation* techniques of an embedded software system for eventual extensions that enable overcoming the consequences of its misuse. The center of gravity of this paper is its *iterative* aspect. In other words, extensions may be added, either continuously or after some time discontinuity. This is attained by means of SPRing, a convenient acronym of *Software Proactive Reengineering*. SPRing is based upon domain knowledge to model the system misuse. Specifically, system behaviors modeled by statechart diagrams, can be reengineered to suitably extend them, in order to correct diverse misuse outcomes. The approach is demonstrated by case studies related to Signal Traffic Lights and their controllers.

Keywords. Software, Misuse, Iterative, Proactive Reengineering, Modeling, Statechart, Problem Domain Knowledge.

1 Introduction

Software upgrade is generally motivated by revealed previously unknown bugs or by intended and unintended system misuse.

Misuse is very challenging. Misuse actually means that the system is not being used according to its originally intended use. We have in previous work [5] dealt with misuse by means of a combination of two complementary techniques: problem domain analysis by PFA [9] and corrections in the solution domain – represented by statechart system modeling.

This work emphasis is on *iterative* solution of misuse problems. By iterative we mean a cyclic process. But in fact it is *loosely cyclic*, i.e. cycles may be continuous or discontinuous – with non-negligible intermissions between cycles.

Moreover, our metaphor the SPRing – standing for *Software Proactive Reengineering* – teaches us that a spring cannot be pulled indefinitely. After some applied force threshold, the spring ceases to behave elastically and in fact may be fractured, without possible remedy. In an analogous way, one expects a limit to the number of misuse correcting cycles. It is an interesting question, whether one can estimate the realistic number of cycles.

After a review of related work in the next sub-section, the remainder of the paper introduces definitions of misuse as opposed to misbehavior (section 2), describe the overall SPRing approach (section 3), deals with an initial iteration of misuse case for normal vision pedestrians (section 4), deals with the next iteration of a misuse case for visually impaired pedestrians (section 5) and concludes with a discussion (section 6).

1.1 Related Work

PFA (Problem Frames Approach) was introduced by M.A. Jackson and is best understood by reading Jackson's own publications [8], [9], [10].

Statecharts were developed by D. Harel and his students in a long series of papers. The semantics of the statecharts presented in this paper is defined in Harel and Kugler [6].

Misuse has been analyzed in a variety of contexts. Steve and Peterson [12] try to actually define misuse. Exman [2] rather looks at system misbehavior, to be contrasted with misues. Alexander [1] deals with hostile misuse. Hope et al. [7] relates misuse to abuse. Sindre and Opdahl [11] look at misuse from the point of view of requirements.

2 Misuse as Opposed to Misbehavior

Misuse is caused by user behavior. Misbehavior is a qualification of system behavior. Their differences are more detailed below.

2.1 Misbehavior

Software system misbehavior means that the system behaves differently from its specified requirements, i.e. the system design and/or implementation do not comply with the system requirements.

Misbehavior examples in the context of traffic lights are: a- they never reach the green light; b- they randomly change colors and/or timing.

2.2 Misuse

Misuse, of a software system by a human user, means that despite the fact that the system functions as it should – i.e. the system behavior complies with the system requirements – some of the following may occur:

- The human user does not comply with the usage instructions;
- The human user uses the system for a different purpose than its intended use;
- Too many human users, each one making normal system usage, cause an overall system performance degradation; the aggregate outcome of multi-user behaviors is misuse.

Examples of traffic lights misuse are: a- a driver does not stop on red light; b- a driver makes a U-turn when a U-turn is not allowed.

Of course not all "surprises" are bad. The system may have a heretofore untapped potential for additional uses not anticipated by the developers. Software developers themselves are empowered users. As they develop the product they may discover such uses. To the extent that the product is open-source it may encourage such discovery, typically occurring during adaptive and preventive maintenance [13].

However, unbridled creativity has a tendency to morph into feature creep, at best, or compromise of mission-critical or safety critical systems. Eternal vigilance is indeed the price of freedom.

3 The SPRing Approach

SPRing is a convenient acronym of *Software Proactive Reengineering*, first introduced in [4]. SPRing is a combination of two proactive modelling techniques, based upon domain knowledge, to overcome software system misuse, seen in Fig. 1:

a. *Problem Analysis via PFA* – this technique that associates new facts in the problem world with needed system requirements' changes;
b. *Statechart Analysis and Upgrade* – modify the system solution, viz. its statechart, by the changed PFA.

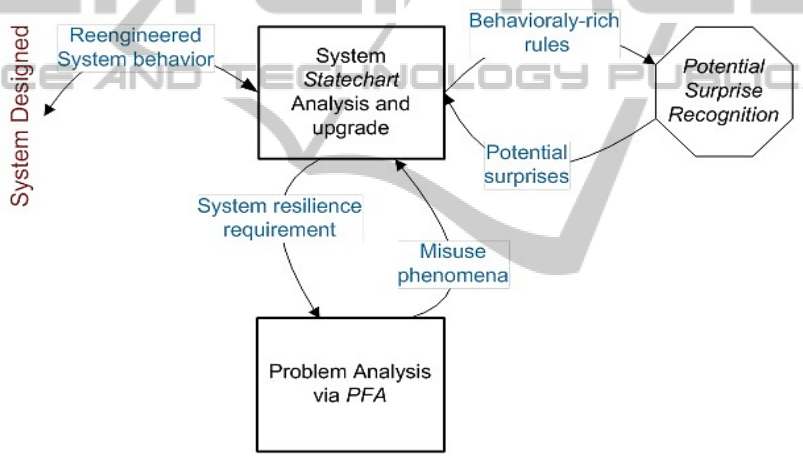
System Designed
Reengineered System behavior
System *Statechart* Analysis and upgrade
Behavioraly-rich rules
Potential Surprise Recognition
Potential surprises
System resilience requirement
Misuse phenomena
Problem Analysis via *PFA*

Fig. 1. *Overall SPRing schematic diagram* – Articulating the interactions between Problem Analysis via PFA and System Statechart Analysis and Upgrade

Next we describe the two techniques in more detail.

3.1 Problem Analysis via PFA

PFA, Problem Frame Approach, describes the problem frame for a software system by means of three concepts:

1) *Requirements* – initially formulated by the stakeholders;
2) *World* – the concrete problem givens, influencing the system behaviour via real phenomena in the outside world;
3) *Machine* – the system control and behavior represented by a statechart.

3.2 System Statechart Analysis and Upgrade

The system statechart is upgraded to overcome the revealed misuse case in each iteration. It is done modularly, to avoid involving the whole software system. It is enough to deal only with sub-systems relevant to the current iteration, as pinpointed by PFA analysis

Modularity implies that the *problem* requires the decomposition of all its aspects (*world, requirements, and machine*) in step. Then, recomposition requires the analysis of how sub-systems must work together.

4 Traffic Lights – 1st Iteration of Misuse Correction: Normal Vision Abuse

In this section we first introduce the original Traffic Lights system, without any misuse, then the 1st misuse correction iteration.

For illustrative purposes, we present a simple road layout and intersection: a single street with a pedestrian crosswalk. Cars cross the intersection according to the traffic lights and pedestrians cross according to pedestrian signals. In principle, there would be cameras that catch drivers that cross on red light. However, as these do not pertain to the misuse case presented herein, we descope them from the model. The IntersectionTrafficController is seen in Fig. 2.

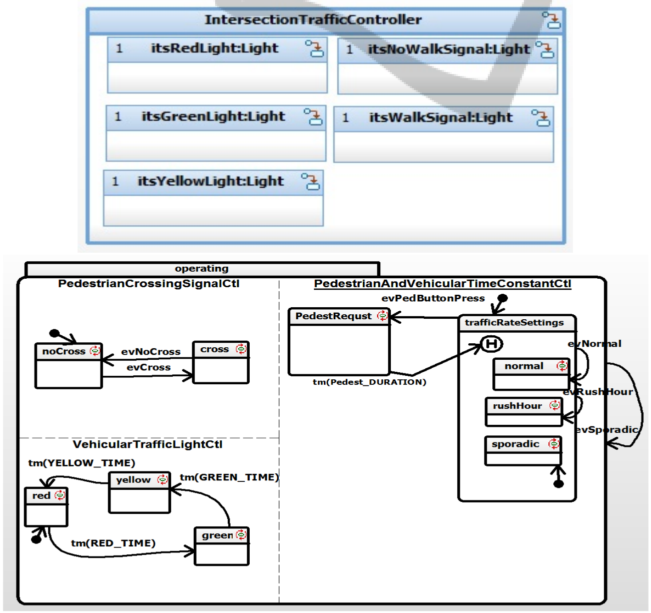

Fig. 2. ***Intersection Traffic Controller.*** Above, the composition of the Controller: on the right hand side, vehicle lights' three objects: *Red*, *Green* and *Yellow*; on the right, the two objects of the pedestrian lights: *Walk* and *NoWalk* signals. Below, the statechart defining the controller logic. There are light time constants for different traffic conditions:normal, rushHour, sporadic; and a button to assist a visually normal pedestrian in crossing the street, by adjusting the light times.

4.1 1st Correction Iteration: Normal Vision Abuse

The first misuse scenario occurs when a normal pedestrian presses the button repeatedly (out of nervousness or playfulness), while waiting for the pedestrian signal to turn to walk.

The correction of this misuse is seen in Fig. 3 (two views: PFA and Statechart).

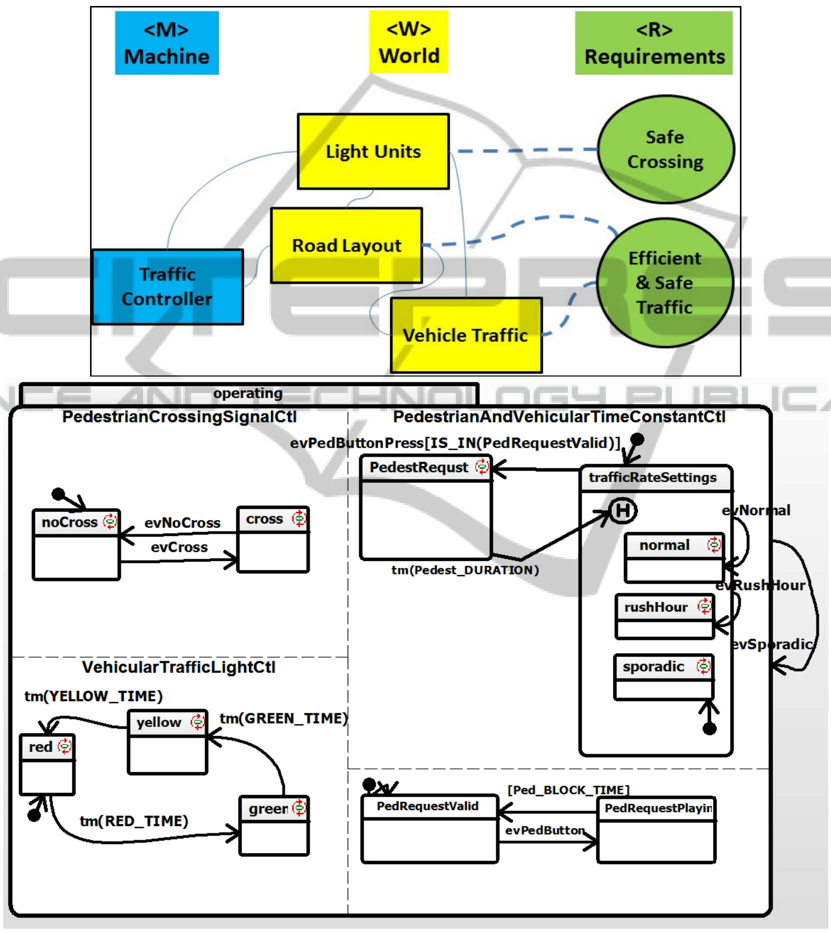

Fig. 3. ***1st Correction Iteration – Normal Vision Abuse Correction***– Above, the PFA diagram; Below, the corresponding statechart, with a BLOCK_TIME during which successive pedestrian button presses are ignored.

One should note that in this iteration the handling of misuse is "hard-coded", as a simple blocking of subsequent request received within Ped_BLOCK_TIME. In the subsequent iterations, this handling will be more flexible via addition of an additional, handling-mapping orthogonal component.

5 Traffic Lights – 2nd Iteration of Misuse Correction: Visually Impaired Pedestrians

Superimposed on the above iteration, one adds to the Traffic Lights system a button for Visually Impaired Pedestrians (VIP). It stops the vehicle traffic for a longer time and also has a voice message telling that the request was accepted. It also emits different sounds when the “walk” signal is on as opposed to the “noWalk” signal.

The VIP added Traffic Lights is seen in Fig. 4 (two views: PFA and Statechart).

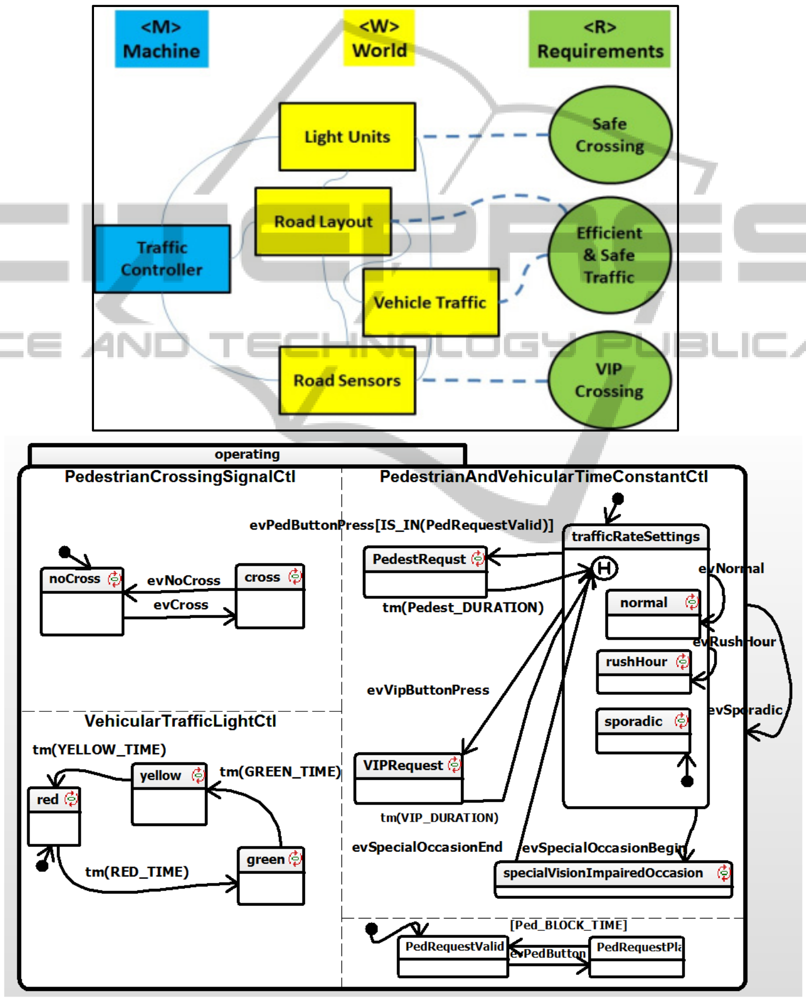

Fig. 4. ***Traffic Lights with added VIP – No Misuse*** – Above, the PFA diagram, below the corresponding statechart.

5.1 2nd Correction Iteration: Impaired Vision Misuse

Misuse here is for instance, a normal vision person pressing the VIP button, or repeated use of this button.

The correction of this misuse is seen in Fig. 5 (Statechart only).

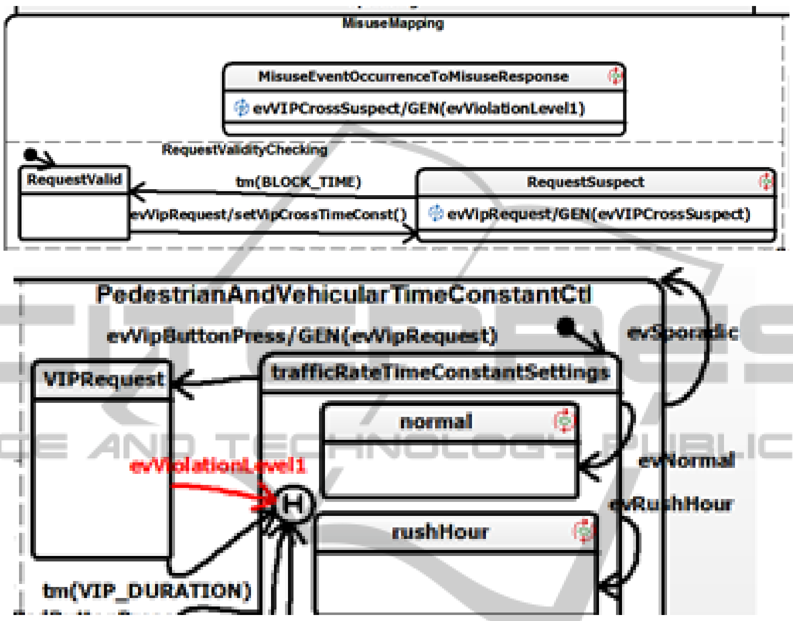
MisuseMapping
MisuseEventOccurrenceToMisuseResponse
evVIPCrossSuspect/GEN(evViolationLevel1)
RequestValidityChecking
RequestValid
tm(BLOCK_TIME)
RequestSuspect
evVipRequest/setVipCrossTimeConst()
evVipRequest/GEN(evVIPCrossSuspect)
PedestrianAndVehicularTimeConstantCtl
evVipButtonPress/GEN(evVipRequest)
evSporadic
VIPRequest
trafficRateTimeConstantSettings
normal
evViolationLevel1
H
evNormal
evRushHour
rushHour
tm(VIP_DURATION)

Fig. 5. ***Traffic Lights with added VIP – Misuse Correction*** – The corresponding statechart additions. Above, the blocking of excessively frequent button presses, and mapping of misuse occurrence to violation level. Below, superposition of violation level handling upon original controller logic.

Note here that flexible mapping of misuse occurrence to misuse handling response has been introduced via the introduction of a mapping orthogonal component. Having deployed this mechanism for the VIP misuse case, it could then be reverse-engineered into the normal pedestrian misuse case.

6 Discussion

Modularity is essential for the misuse corrections, where behavioral modularity is achieved by use of orthogonal components in a single statechart. It is more effective for a software developer to look at two independent behaviors in the same diagram, i.e. orthogonal components of a single statechart.

As shown in the Impaired Vision Pedestrians Case study, orthogonal components establish a natural correspondence between misuse cases and their corrective actions. Thus, decoupling misuse incidents from corrective actions, allows abstraction of both of them. Changing the mapping of misuse incidents to corrective actions is easily performed by changing the association between triggering misuse events and the respective corrective action events.

The important consequence, of the modular reasoning of the previous paragraph for this paper, is that decoupling is also true and useful for consecutive misuse cases, solved iteratively.

On the other hand, still in the Impaired Vision Pedestrians Case study, the orthogonalization of checking for misuse (RequestValidityChecking) and misuse mapping to misuse response event, is indeed modular, but the corrective action responding to a violation level may or may not be tightly coupled to original state-based behavior. An uncoupled response would be a system abort. Slightly more coupled would be a transition to an added external standby state, with a reset event to return system to an initial default operational state.

Regarding the iterative process, the number and duration of iterations is difficult to predict, given the "surprise" emergence of misuse cases. Nevertheless, the heuristic illustrated herein may be typical: A baseline "well-behaved" system is disrupted by "surprise" misuse. The misuse is mitigated by a behavioral correction specified in a statechart. Subsequently, another "surprise" misuse emerges. If there is some degree of commonality between the two misuse scenarios, this may stimulate thought about a more general, flexible mechanism to handle an entire class of misuses. This mechanism is initially applied to the misuse case currently confronted, and then retro-fitted to the prior on.

6.1 Future Work

Future work for evaluation of the actual limitations of the *Software Proactive Reengineering* approach includes:

- Exploration of repeated misuse and corrective actions in a variety of case studies.
- Generic representations of misuse corrections in a statecharts formalism;
- Matching stakeholder competencies and needs to diagrammatic representations. Non-technical stakeholders, may eschew the detailed precise semantics of Harel statecharts in favor of high-level PFA diagrams. But at the end of the day "the devil is in the details" and everyone involved need to understand the subtle interactions that allow and remedy misuse. Elsewhere we have proposed heuristics for utilization of statecharts to elicit behavioral requirements from such users, and adaptation of this heuristic to the present work is a *desideratum*.

6.2 Main Contribution

The main contribution of this work is the recognition that *Software Proactive Reengineering* is a loosely cyclic process, with eventual limitations to system flexibility towards misuse corrections.

References

1. I. Alexander, "Misuse Cases: Use Cases with Hostile Intents", IEEE Software, pp. 58-66, (2003).
2. I. Exman, "Misbehavior Discovery through Unified Software-Knowledge Models", in A. Fred et al. (eds.), Knowledge Discovery, Knowledge Engineering and Knowledge Management, 3rd Int. Joint Conference, IC3K 2011, Paris, France, October 26-29, Revised Selected Papers, CCIS Vol. 348, pp. 350-361, Springer-Verlag, Heidelberg, Germany, (2013).
3. R. Gallant, "Freddy's Dishwasher: Are Statecharts 'Safe' for Stakeholder Evaluation of Safety-critical Reactive Systems?" Proceedings of the 15th International Conference of The Israel Society for Quality, 2004.
4. L. Goldin, R. Gallant, and I. Exman, Software Proactive Reengineering for System Resilience to Misuse, in Proceedings of Software Summit, SWSTE 2014, Bar Ilan University, (2014).
5. L. Goldin and R. Gallant, "Reengineered PFA: An Approach for Reinvention of Behaviorally-Rich Systems", In Proc. SKY'2012 Int. Workshop on Software Knowledge, Barcelona, Spain, October 2012, SciTe Press, Portugal, (2012).
6. D. Harel and H. Kugler, "The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)", Integration of Software Specification Techniques for Applications in Engineering, (H. Ehrig et al., eds.), Lecture Notes in Computer Science, Vol. 3147, Springer-Verlag, Berlin, Germany, pp. 325-354, (2004).
7. P. Hope, G. McGraw and A.I. Anton, "Misuse and Abuse Cases: Getting Past the Positive", IEEE Security and Privacy, pp. 32-34 May/June (2004).
8. M.A. Jackson, Software Requirements & Specifications, Addison-Wesley, Boston, MA, USA, (1996).
9. M.A. Jackson, Problem Frames: Analysing and Structuring Software Development Problems, Addison-Wesley, Boston, MA, USA, (2001).
10. M.A. Jackson, "The Problem Frames Approach to Software Engineering", in Proc. APSEC 2007, 14th Asia-Pacific Software Engineering Conference, (2007).
11. G. Sindre and A.L. Opdahl, "Eliciting security requirements with misuse cases", Requirements Eng. Vol. 10, pp. 34-44, (2005).
12. J. Steven and G. Peterson, "Defining Misuse with The Development Process", IEEE Security & Privacy, pp. 81-84, November/December (2006).
13. ISO/IEC 14764:2006, Software Engineering—Software Life Cycle Processes – Maintenance.