Overcoming Software System Misuse by Domain Knowledge

Reuven Gallant[1] and Leah Goldin[2]

[1] Jerusalem College of Technology, P.O.B. 16031, 91160, Jerusalem, Israel

[2] Afeka Tel-Aviv Academic College of Engineering, Tel-Aviv, Israel

Abstract. Often a perfectly functioning software system is misused causing undesirable and expensive consequences. The quest of this work is to prepare a priori the system for eventual extensions that – while not directly relevant to the system purpose – enable overcoming the consequences of its misuse. This is attained by means of domain knowledge to model the system misuse, beyond the original system model. In particular, if the behaviors of such a system have been modeled by statechart diagrams, these diagrams can be reengineered to suitably extend them, in order to correct the misbehavior consequences.

1 Introduction

Our software development follows M. Jackson's Problem Frame Approach (PFA) (Jackson, [6]). Jackson's idea is that *system development* is a *problem*: the task is to devise a software behavior that produces the required effects in the *physical problem world*. The problem complexity is addressed by decomposition into *sub-problems*, and so on recursively.

In this paper we continue our previous work (Goldin and Gallant, [3]) on *behaviorally-rich* systems. We extend it to deal with misuse of perfectly correct systems. Also here we identify a sort of reengineered PFA, which will be formulated and illustrated by a few case studies.

1.1 Related Work

We present here a short overview of the relevant literature.

The best starting points to understand Jackson's approach are the books and the PFA overview by Jackson himself (see refs. [6],[7],[8]).

Many works deal with misuse of software from a variety of points of view. Surprisingly many of them consider the advantages of misuse in house as a testing procedure, in particular for security issues.

For instance, Alexander [1] refers to misuse cases with hostile intent. Sindre and Opdahl [12] deal with security requirements by means of misuse cases. Hope et al. (Hope et al., [5]) consider misuse as a positive form to learn how to defend software from attackers.

Gallant R. and Goldin L..
Overcoming Software System Misuse by Domain Knowledge.
DOI: 10.5220/0004641700620069
In *Proceedings of the 4th International Workshop on Software Knowledge* (SKY-2013), pages 62-69
ISBN: 978-989-8565-76-1
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

The quantity of work done about misbehavior is also significant. For example, Exman [2] deals with representation of misbehavior by means of statecharts, in particular referring to standard software components, like design patterns.

1.2 Overview of the Paper

Section 2 define basic terms such as misbehavior and misuse. Section 3 describes the reengineering PFA process. Section 4 explains the approach by means of case studies. Section 5 deals with misuse paradigms. Section 6 contains a discussion and conclusions.

2 Misbehavior, Misuse and their Combination

In this section we define some basic terms, such as misbehavior and misuse.

2.1 Misbehavior

Misbehavior of a software system means that the system behaves differently from what is expected from the system requirements. In other words the system design and/or implementation do not fit with the system requirements.

Traffic lights examples are: a- it never reaches the green light; b- it changes randomly both with respect to colors and to timing.

2.2 Misuse

Misuse of a software system means that:

- on the one hand, the system functions perfectly as it should – its behavior fits with the system requirements and so are its design and implementation;
- on the other hand, an end-user of the system makes use of it not according to its purpose and/or does not comply with the usage instructions.

Traffic lights examples are: a- a driver does not stop on red light and crosses a street intersection; b- a driver makes a U-turn when it is not allowed to do that.

A combination of misbehaviour with misuse means that both kinds of problems occur in a certain event. A famous example is the case of the Therac-25 (Leveson, [10]) radiation accidents and their fatal medical consequences.

3 The Reengineering Process

Our approach assumes that the behavior model of a system is given by a statechart.

It was defined above the software development *problem* as the need to obtain a software behavior that will produce the required effect in the physical *problem world*.

Therefore the decomposition of the physical *problem* requires the decomposition of all aspects of the problem (*machine, problem world*) in step.

In other words, only the relevant part of the physical world is considered, and a submachine meeting the sub-requirement in the partial physical world is specified.

The states and events of the machine can then be examined to identify impossible or undesirable events and transitions. Then the software behaviors of the sub-problems can be modified to eliminate them.

3.1 Behavioral Models for Problem Identification

We repeat here the classification of software systems according to their behavioral model – first presented in our preceding paper (Goldin and Gallant, [3]):

- *Behaviorally-rich* – are those systems whose behavioral model hints to a *potential* richness, not found yet in the current model;
- *Behaviorally-poor* – are those systems whose model lacks any hints to potential richness.

This classification is surely dependent on the referred hints. The latter are heuristic rules – to be collected based upon accumulated experience with software system modeling.

We have proposed a few such rules to recognize the potential of behaviorally-rich software systems in their behavioral model:

1- <u>*Number of States*</u> – the number of sub-states in any given state is greater than the number of siblings of the given state;

2- <u>*Transition Chain Linearity*</u> – the transition chain of events linking a set of states is linear, without any bifurcation;

3- <u>*Transition Chain Cycle*</u> – the transition chain of events linking a set of states is a whole cycle, without any internal bifurcation.

We shall illustrate the use of such rules in the Case Studies section.

3.2 Decomposition and Recomposition

The central idea here is that multiplicity of states in simple looking hierarchies hint at possible model improvement. But one should try to confirm that the problem was actually identified, before performing model decomposition and reinvention of critical sub-problem interactions. Our guide is the proposed heuristic rules for recognition of a behaviorally-rich system.

A most important point is the *reuse* of past experience and behavioral design patterns. This concerns two aspects:

1- *Heuristic rules* – as suggested above;

2- *Behavioral Design Patterns* – ready-made behavior structures to be inserted into states identified by the heuristic rules.

Finally, recomposition requires the analysis of how subprograms must work together to accomplish the reengineered system purpose.

4 Case Studies

We shortly describe two case studies to illustrate the process.

4.1 Traffic Lights

Our traffic lights system is as follows. One has an intersection of two perpendicular streets, with cars crossing the intersection according to the lights. There are also cameras that catch drivers that cross on red light.

Suppose now the following event. A driver does not pay attention to the lights changing to red and advances a few meters into the intersection and stops too late. This is a misuse of the system. The standard behavior of the system is to take a photo of the car plate to send a fine to the driver, but otherwise the lights continue to change regularly. But this is dangerous, since the driver stopped quite in the middle of the intersection, and accidents may occur.

We propose the following reengineering of the system: the camera identifies the problematic advance of the car, and modifies the timing of the green light for the perpendicular street to avoid a possible accident. This is based on the traffic domain knowledge. This kind of reengineering can be recognized in the traffic lights' statechart according to the *Transition Chain Cycle* heuristic rule, since the traffic lights – red, yellow, green – is such a cycle.

4.2 Dialysis Equipment

A dialysis system is an artificial replacement for natural kidneys that do not execute anymore their function. Dialysis performs two functions: extraction of small molecules and excess water from blood.

Before dialysis is started, nurses should calculate for each patient the decrease of weight needed and the time duration of the dialysis. From this data the machine executes the necessary program. If the data was miscalculated – an example of misuse – the machine may cause excessive water diffusion, causing high blood viscosity and undesirable medical consequences.

The reengineered system in such a case has a sensor that indirectly measures the blood viscosity and reduces the pressure to decrease the rate of water extraction.

This kind of reengineering can again be recognized in the dialysis system statechart according to the same heuristic rule as above, due to the blood and dialysate cycles.

5 Misuse Paradigms: Dialysis Case Study revisited

This case study follows the example presented by Roux, et al. [11], based on the dialysis operational protocol of Korabik [9], with the following adaptations:

1. We use the UML statechart language as defined by Harel and Kugler [4].

2. We supplement the statecharts in [11] with misuse events and responses according to the severity of the misuse.

Fig. 1, is a statechart representing the overall dialysis treatment process, beginning with patient intake, and ending with post-treatment equipment maintenance.

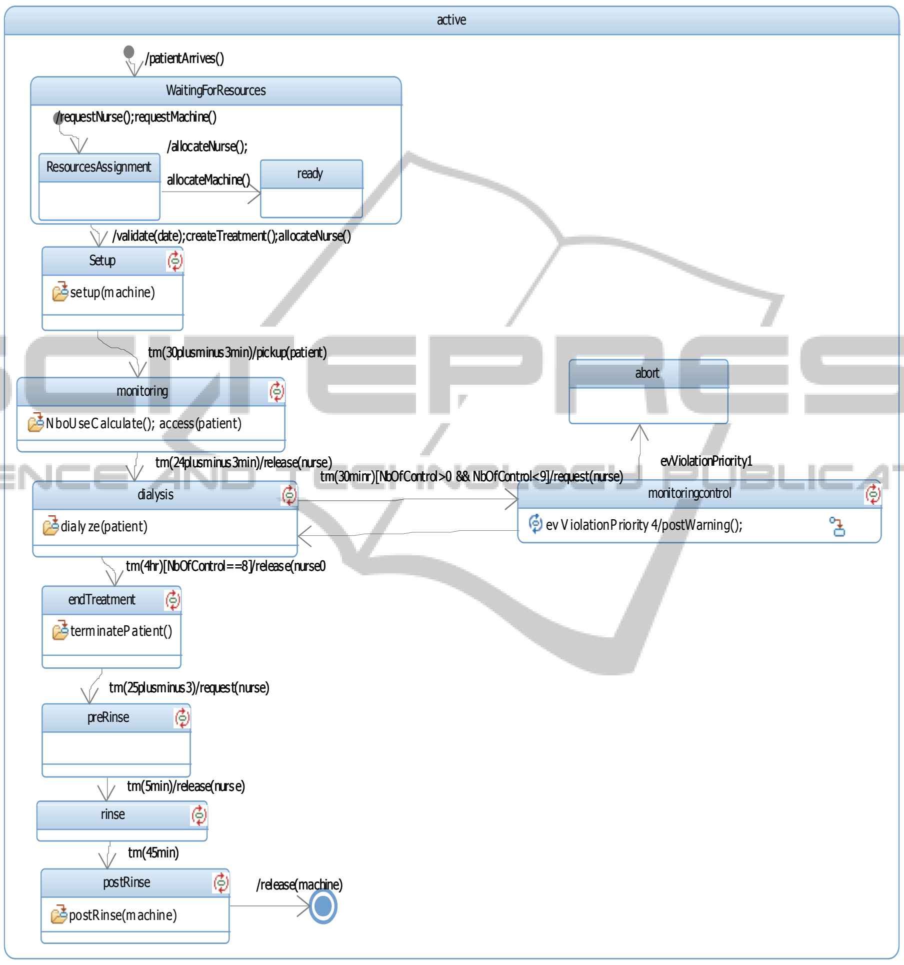
active
/patientArrives()
WaitingForResources
/requestNurse();requestMachine()
/allocateNurse();
allocateMachine()
ResourcesAssignment
ready
/validate(date);createTreatment();allocateNurse()
Setup
setup(machine)
tm(30plusminus3min)/pickup(patient)
monitoring
NboUseCalculate(); access(patient)
abort
tm(24plusminus3min)/release(nurse)
tm(30minr)[NbOfControl>0 && NbOfControl<9]/request(nurse)
evViolationPriority1
dialysis
dialyze(patient)
monitoringcontrol
evViolationPriority4/postWarning();
tm(4hr)[NbOfControl==8]/release(nurse0
endTreatment
terminatePatient()
tm(25plusminus3)/request(nurse)
preRinse
tm(5min)/release(nurse)
rinse
tm(45min)
postRinse
postRinse(machine)
/release(machine)

Fig. 1. Statechart for the dialysis treatment process.

Now we expand the monitoring control state, for which we define misuse incidents and a corrective action for each type of incident. We focus on misuse that may occur within the monitoring control state, which is detailed in Fig. 2.

Herein we define a paradigm for a specific type of misuse, preemptive entrance into a state out of sequence. To simplify the example, assume that the only preemption performed is to advance to the next state, without completing the tasks of the present state. We define 4 levels of misuse violation, and the corrective action for each level.

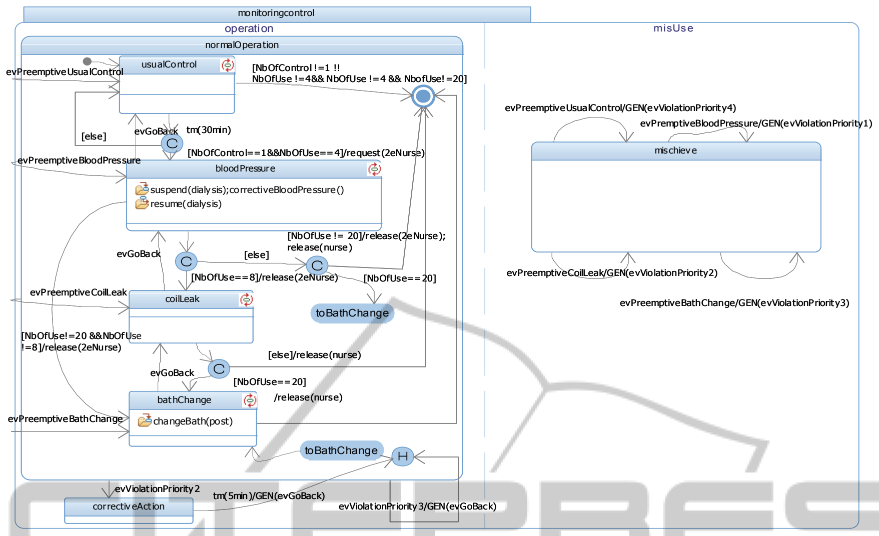

Fig. 2. Monitoring control sub-state.

Representation of Preemptive Misuse

The paradigm we adopted for representing preemptive misuse is as follows:

1. Define a super-state (normalOperation) enclosing the preemptable sub-states (usualControl, bloodPressure, coilLeak, bathChange). (Fig. 2)
2. Associate a preemption event with each preemptable state, and draw a transition from the super-state to each of the preemptable states, triggered by the associated event. The event naming convention is evPreemptive<state name>. (Fig 2).
3. For each Violation Priority associate an event according to the naming convention evViolationPriority<n> where highest priority is n=1. (Fig 2)
4. Divide the monitoring control state into two orthogonal components (operation, misUse). In the misUse orthogonal component, define reentrant transitions (these could have been reactions in state), to map the misuse events (triggers) to violation priorities (actions). (Fig. 2)
5. The corrective action for the various violation priorities are as follows:
   a. Priority 1, aborts the treatment (Figure 1, abort state). (Fig. 1)
   b. Priority 2, allows 5 minutes for corrective action, before reverting to the previous state. (Fig 2)
   c. Priority 3, immediately reverts to the previous state(Fig. 2)
   d. Priority 4, allows the preemption, but posts a warning, as per the the reaction in state defined for monitoringcontrol state (Fig.1).

6 Discussion

Reengineered PFA replaces intuitive speculation with a highly focused purpose – to automatically correct potential misuse. As the stakeholder traverses the model, a reverse engineering /reinvention happens. At each reversal stage, decomposition and afterwards recomposition occur.

Cognitively, it is easier to contemplate interactions between two behaviors if they are captured in the same diagram, i.e. orthogonal components of a single statechart.

In particular, as demonstrated in the Dialysis Case study, orthogonal components effectively partition paradigms for misuse and corrective action. The salient feature of this paradigmatic approach is decoupling of misuse incidents from corrective actions, allowing 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.

6.1 Future Work

The proposals in this paper are still in a preliminary investigation stage. One needs extensive examination of a variety of software systems misuse to validate the approach. Such systems should be of realistic size and complexity.

In order to effectively apply reengineered PFA one would need tools supporting both the recognition of behaviorally-rich systems and their actual improvement.

Future work for enrichment and extension of the paradigmatic approach includes:

- Exploration of other misuse and corrective action paradigms. In particular the preemptive paradigm is most relevant to sequential processes. For event driven behaviors preemption is less relevant.
- Accordingly it is a desideratum, to define various behavioral paradigms, and suggestions for relevant misuse correction paradigms, with attention to their generic representations in statecharts.
- Formulation and simulation of misuse scenarios using tools that support Live Sequence Charts (LSC) such as IBM Rational Rhapsody's Test Conductor.

6.2 Conclusions

This work proposes a reengineered PFA approach to correct software systems misuse, with a state-based behavior model. Statecharts facilitate recognition of behaviorally-rich systems, leading to decompositions supporting paradigmatic abstractions.

References

1. Alexander, I.: Misuse Cases: Use Cases with Hostile Intents, IEEE Software, pp. 58-66, (2003).
2. Exman, I.: Misbehavior Discoverythrough Unified Software-Knowledge Models, in Knowledge Discovery, Knowledge Engineering and Knowledge Management, 3rd Int. Joint Conference, IC3K 2011, Paris, France, October 26-29, Revised Selected Papers, pp. 350-361, Springer-Verlag, Heidelberg, Germany, (2011).
3. Goldin, L. and Gallant, R.: 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).

4. Harel, D. and Kugler, H.: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, pp. 325-354, (2004).
5. Hope, P., McGraw, G. and Anton, A. I.: Misuse and Abuse Cases: Getting Past the Positive, IEEE Security and Privacy, pp. 32-34 May/June (2004).
6. Jackson, M. A.: Software Requirements & Specifications, Addison-Wesley, Boston, MA, USA, (1996).
7. Jackson, M.A.: Problem Frames: Analysing and Structuring Software Development Problems, Addison-Wesley, Boston, MA, USA, (2001).
8. Jackson, M.A.: The Problem Frames Approach to Software Engineering, in Proc. APSEC 2007, 14th Asia-Pacific Software Engineering Conference, (2007)
9. Korabik R. M.: A New Approach to Operational Efficiency for Chronic Renal Dialysis, Winter Simulation Conference I.E.E.E, Highland, pp 659-662, (1978).
10. Leveson, N.: Medical Devices: The Therac-25, appendix in the book Safeware: System Safety and Computers, Addison-Wesley, Boston, MA, USA (1995). Site: http://sunnyday.mit.edu/papers/therac.pdf
11. Roux, O., D. Duvivier, E. Ramat, G. Quesnel, C. Combes: UML-Statechart Modeling Tool For the VLE Simulator: AnApplication To A Chronic Renal Dialysis Unit," 8th International Conference of Modeling and Simulation - MOSIM'10 - May 10-12, 2010 - Hammamet – Tunisia, "Evaluation and optimization of innovative production systems of goods and services.", (2010).
12. Sindre G. and Opdahl, A. L.: Eliciting security requirements with misuse cases, Requirements Eng. Vol. 10, pp. 34-44, (2005).