Reengineered PFA: An Approach for Reinvention of Behaviorally-rich Systems

Reuven Gallant[1] and Leah Goldin[2]

[1]Jerusalem College of Technology, P.O.B. 16031, 91160, Jerusalem, Israel
[2]Shenkar College of Engineering and Design, 52526, Ramat-Gan, Israel

Abstract. In this Position Paper, one considers application of a reengineered PFA as a means of reinvent critical parts of a particular class of *behaviorally-rich* systems. The implementation of such systems, although not necessarily the outgrowth of PFA, could plausibly have been such an outgrowth because of the nature of their behavior and the way they were modeled. The goal of reengineered PFA application for such a system would be to re-examine and re-design critical behaviors. In particular, if the behaviors of such a system have been modeled by statechart diagrams, these diagrams can be leveraged to catalyze examination of whether the as-built behavior is in fact the desired behavior.

1 Introduction

The basic underpinning of M. Jackson's Problem Frame Approach (PFA) (Jackson, 1996) is that *problem* analysis precedes the construction of the solution. In other words, PFA is originally intended for *novel* systems prior to solution, rather than existing *normal* systems, that presumably, in the course of their evolution, have achieved mastery of problem complexity. In the same spirit, PFA eschews consideration of software implementation architecture until a thorough problem analysis has been achieved.

Within PFA for software development, in Jackson's own words "Development of a system is regarded as a *problem*: the task is to devise a software behavior that will satisfy the *requirement*—that is, will produce the required effects in the physical *problem world*. The complexity of the problem is addressed by decomposition into *sub-problems*, and so on recursively." Each such simple sub-problem can be regarded as defining a small system to be developed — with its own software behavior.

In this paper we switch the point of view from novel systems to emphasize *behaviorally-rich* systems. This implies a sort of reengineered PFA, which will be formulated and illustrated by a case study.

1.1 Overview of the Paper

Section 2 presents the key elements of PFA: Decomposition, and Recomposition.

Gallant R. and Goldin L..
Reengineered PFA: An Approach for Reinvention of Behaviorally-rich Systems.
DOI: 10.5220/0004181400710078
In *Proceedings of the 3rd International Workshop on Software Knowledge* (SKY-2012), pages 71-78
ISBN: 978-989-8565-32-7

Section 3 discusses the differences between behaviorally-poor and *behaviorally-rich* systems, and why reengineered PFA is considered more appropriate to the latter. Section 4 details the meaning we impart to reengineered PFA. Section 5 presents a case study that, although modest in scope, exhibits the characteristics that the authors feel makes a system suitable for reengineered PFA. Section 6 contains a discussion and conclusions.

2 PFA Decomposition and Recomposition

According to traditional requirements engineering, requirements are decomposed either by refinement, in which a large requirement is divided into constituent parts that it contains or directly implies, or by derivation, the creation of detailed requirements not contained or obviously implied in the original formulation of the requirement, but nonetheless necessary to satisfy the original requirement.

In model-driven development, these requirements are allocated to model elements. For complex systems, the model will also undergo decomposition, and the model decomposition will typically be influenced by the requirements decomposition, but there is no clear heuristic linking the two.

2.1 PFA Decomposition

In PFA, decomposition is not applied to requirements, but to the software development problem.

Having defined the software development *problem* as the task of devising a software behavior that will produce the required effect (*requirement*) in the physical *problem world*, then decomposition of the physical *problem* requires the corresponding decomposition of all aspects of the problem (*machine, problem world*) in step. That is to say, for each subrequirement, only the relevant part of the physical world is considered, and a submachine meeting the subrequirement in the partial physical world is specified.

PFA distinguishes between two different types of decomposition- requirements decomposition, and instrumental decomposition.

Requirements decomposition decomposes the overall system purpose into a number of subpurposes necessary to achieve it. For each of the subpurposes, the relevant subproblem world is delineated and the behavior of a submachine is defined. Requirements decomposition does not address the intrinsic complexity of the problem's purpose.

In contrast, in instrumental decomposition, for a a not yet decomposed software, an interface is exposed creating decomposed subproblems. For instance, this interface may be a shared data structure, or a set of shared events.

Jackson identified two different types of interfaces to be promoted as a result of instrumental decomposition – one in a designed domain and another in an analogical domain.

2.2 PFA Requirements Recomposition

Decomposition is an intentional oversimplification, allowing the developer to analyse and understand in isolation each subproblem in its relevant portion of the problem world.

This approach defers but does not eliminate the task of recombining the subproblems into an overall coherent requirement. Issues such as read-write contention between instrumented interfaces, timing, and requirement contradictions must all be addressed.

Possibly conflicting subproblems are viewed as defining finite state machines, while the developer can in principle construct their product machine.

The states and events of the product machine can then be examined to identify impossible or undesirable events and transitions, and the software behaviours of the subproblems can be modified to eliminate them.

In traditional requirements engineering, the task of requirements derivation includes requirements decomposition, instrumentation decomposition and requirements recomposition. The approaches of PFA is to do the easy derivations first, and, only after these are mastered, to do the hard work of getting everything to work together coherently

3 Behaviorally-Rich vs. Behaviorally-poor Systems

For systems under development, subsequent to requirements recombination, comes software recombination in which the subproblems are implemented within a consistent whole software architecture.

What about modification of an existing system?

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

We introduce here a novel categorization of existing software systems according to their behavioral model:

- *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 that whose model lacks any hints to potential richness.

This categorization is obviously dependent on the referred hints. These are heuristic rules – to be collected based upon accumulated experience with software system modeling.

We propose as a starting point a few such rules to recognize the potential of behaviorally-rich software systems in their behavioral model:

1- *Number of states* – the number of sub-states in any given state is greater than the number of siblings of the given state;
2- *Transition Chain Linearity* – the transition chain of events linking a set of states is linear, without any bifurcation;
3- *Transition Chain Cycle* – the transition chain of events linking a set of states is a whole cycle, without any internal bifurcation.

The idea is that multiplicity of states in simple looking hierarchies hint at possible model improvement.

4 Reengineered PFA

Reengineered PFA starts from a new point of view regarding problem analysis for an existing software system.

Our new approach is based upon the premise that state-based behavioral models for existing systems facilitate identification and reinvention of critical subproblem interactions. This, without going through the whole PFA process, since the model is already existing, perhaps almost fully developed.

A point of central importance is the *reuse* of past development experience and behavioural design patterns. This concerning two aspects:

1- *Heuristic rules* – to recognize behaviourally-rich software systems, as seen in the preliminary suggestions of the previous section;
2- *Behavioral Design Patterns* – these are ready-made behaviour structures to be inserted into states identified by the heuristic rules as having the potential of model improvement.

Next, we demonstrate the reengineered PFA approach by means a simple case study.

5 Case study

The case study is the dishwasher taken from samples of the IBM Rational Rhapsody tool.

It is a sufficiently simple toy problem to be easily understood, but not too trivial, enabling a clear illustration of the ideas of a behaviourally-rich software system and its potential improvement.

5.1 The Dishwasher

The Dishwasher Class is the central controller, as seen in Fig. 1.

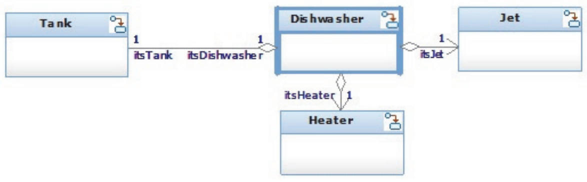

Fig 1. Dishwasher Software Structure Architecture.

The interaction of the dishwasher with its subsystems is governed by the statecharts of each class.

We focus our attention in the statechart of the dishwasher class – which has the role of a controller – seen in Figure. 2.

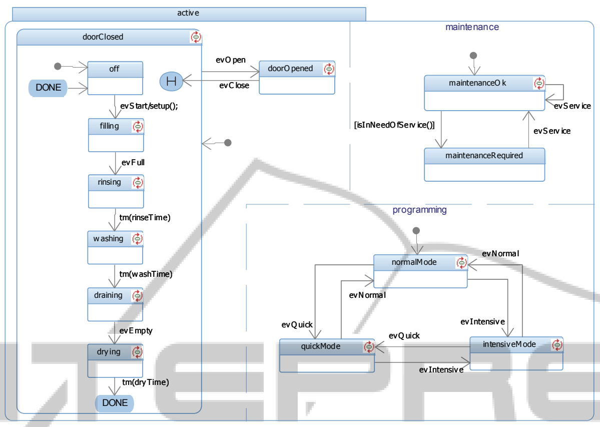

Fig 2. Dishwasher Software Behavior Architecture – this is the statechart of the dishwasher class. The doorClosed state is seen in the left hand side.

5.2 Identifying the Problem

The problem in this case study can be easily identified from domain knowledge as to clean dishes effectively and efficiently.

It may be nonetheless useful to seek confirmation that this is the problem, prior to identifying the decompositions in the model of the product to be improved. We are guided by the proposed heuristic rules for recognition of a behaviourally-rich system, as suggested above in section 3. The appropriate place to contemplate the highest level problem would be in the highest-level controller, whose behavior is defined by the Dishwasher statechart (Fig. 2).

Our attention is immediately drawn to the most populated region of the diagram, the orthogonal state *doorClosed*, whose top-to bottom linear sequence fits the "*Transition Chain Linearity"* rule.

This, to a certain extent, confirms the hypothesis that the purpose of the system is "to clean dishes." The sub-state names, allows refinement of the problem, prior to decomposition: cleaning of dishes requires rinsing, washing and drying of the dishes as well as intake and draining of water.

All this is obvious to anyone with domain knowledge, but the fixed sequence of these activities in the *transition linear chain*, should lead us to ask whether this sequence achieves "effectiveness and efficiency" in all situations, a question we will return to during decomposition.

5.3 Decomposition

The requirements decomposition evident from the Dishwasher statechart has three parts:

1- the cleaning process;
2- dishwasher programming,
3- dishwasher maintenance monitoring,

each of which is encapsulated in a distinct orthogonal component (*doorClosed*, *programming* and *maintenance*, respectively).

Assuming that both the maintenance subprogram and the programming subprogram interact in some way with the cleaning program, instrumental decomposition is necessary to expose the required interface.

As may be discerned from the names of the programming substates, these states are selectors of timing constants for the fixed-sequence cleaning subprogram. This would be an appropriate point to reconsider the cleaning process.

Perhaps, instead of the fixed sequence in the *transition linear chain*, where the "cleaning programs" are merely variations of the timing constants, greater flexibility is required, with different state transition sequences for each program, states that a given "cleaning program" visits more than once, etc. A more complex set of parameters would then be required to support this flexibility, warranting the elevation of the exposed interface to a "designed domain".

This would also be the point in the reengineering process to think about the maintenance states. Perhaps the system health is not merely okay or not okay, but has a number of indicators, appropriately modeled by its own state-machine (or an orthogonal component considerable richer than the one given). In which case the exposed interface models the physical state of the mechanical machine as a complex state machine.

5.4 Recomposition

Recomposition requires the analysis of how the subprograms must work together to accomplish the system purpose.

Here we must consider not only the subprograms depicted in the Dishwasher statechart, but also the next level of requirements decomposition, the statecharts defining behavior of the subsystems controlled by the Dishwasher: Tank, Jet, and Heater.

The economies of statecharts are achieved by hiding crucial information, such as: when does a selection of a new "cleaning program" interact with the "cleaning process." The hiding of this information encourages the stakeholder to consider what should be required before looking what is presently implemented.

The answer to such a question may warrant recursive modification of one or more subproblems. In our example, it is possible to change the cleaning program directly from intensive to quick. If such a change goes into effect even when the "cleaning process" is in progress, it may cause damage to the motor. If so, we may allow direct transition only from a given speed to the next higher or lower one.

Similarly, only in the Dishwasher (Controller) statechart it is evident what happens when the door is opened.

What happens or should happen to the subsystems when the door is opened?

The absence of an answer on the diagrams stimulates the stakeholders to think about what they want. As a corollary of this question, if we decide that some but not all of the subsystems should stop operating when the door opens, what happens when the door is closed and operation of the entire system resumes. Will there be a synchronization problem?

6 Discussion

In a previous paper based on the same case study, a heuristic was proposed to facilitate stakeholder comprehension of overall system behavior and identification of problems regarding interaction of different behaviors.

The heurisitic imposed some degree of structure to the process by recommending top-down traversal of the model and its statecharts. However the process of problem identification proposed, based on a taxonomy of graphical cues was very impressionistic and intuitive.

Reengineered PFA, to a large extent, replaces intuitive meandering with a highly focused teleos. As the stakeholder traverses the model, a model reverse engineering/reinvention occurs. At each stage of the reversal, decomposition (two types) occurs and afterwards recomposition.

Insofar a UML model organized according to behavioral hierarchy will have structure similar to a PFA hierarchy, two separate notations to depict approximately the same hierarchy may sow confusion. Until more experience is gained with complex systems, we would recommend beginning by confining the project to the UML model, supplemented by stereotypes and tagged values to indicate mappings to PFA elements and structural correspondences.

With respect to recombination, this requires consideration of collaborations, these can be captured with UML Sequence diagrams or interaction diagrams. Here we have a conflict between cognitive and workload issues.

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

However, if we wish to capture this interaction in a UML Sequence diagram or interaction diagram, we have two choices: either we can manually draw such a diagram, treating these behaviors as if they were encapsulated in distinct objects, each with its own statechart, or we can actually decompose the object whose statechart contains the two orthogonal components into two objects each of which encapsulates a single behavior in a separate statechart.

The latter approach, imposes a cognitive burden (context switching between two diagrams) but reduces workload: using an executable modeling tool such as Rhapsody, the collaboration can be automatically generated during graphical simulation as an animated statechart.

6.1 Future Work

The proposals in this Position Paper are still in a preliminary stage. One needs extensive examination of a variety of software systems to validate the approach. The future systems to be investigated should be of realistic size and complexity. In order to effectively apply reengineered PFA one would need tools supporting both the recognition of behaviourally-rich systems and their actual improvement.

Acknowledgements. The authors are grateful to Iaakov Exman for a critical reading of early versions of the paper.

References

Jackson, M.A., 1996. *Software Requirements & Specifications,* Addison-Wesley, Boston, MA, USA.

Jackson, M.A., 2001. *Problem Frames: Analysing and Structuring Software Development Problems*, Addison-Wesley, Boston, MA, USA.

Jackson, M.A., 2007. "The Problem Frames Approach to Software Engineering", in Proc. APSEC 2007, 14th Asia-Pacific Software Engineering Conference.