An Analysis-Revision Cycle to Evolve Requirements Specifications

A. S. d'Avila Garcez*, A. Russo*, B. Nuseibeh‡ and J. Kramer*

*Department of Computing, Imperial College
180 Queen's Gate, London, SW7 2BZ, UK
{aag,ar3,jk}@doc.ic.ac.uk

‡Computing Department, The Open University
Walton Hall, Milton Keynes, MK7 6AA, UK
B.A.Nuseibeh@open.ac.uk

Abstract

We argue that the evolution of requirements specifications can be supported by a cycle composed of two phases: *analysis* and *revision*. In this paper, we investigate an instance of such a cycle, which combines two techniques of *logical abduction* and *inductive learning* to analyze and revise specifications respectively.

1 Introduction

This work aims to facilitate the evolution of requirements specifications of state transition systems, by providing the requirements engineer with tools to support the change management process. In particular, "models for reasoning about current alternatives and future plausible changes have received relatively little attention to date, even though such reasoning should be at the heart of requirements engineering"[12]. We argue that the development of requirements specifications can be supported by a cycle composed of two phases: *analysis* and *revision*, as Figure 1 illustrates. The analysis phase is responsible for checking whether a number of desirable properties of a system is satisfied by its partial specification. It also provides appropriate diagnostic information when a certain property is violated by the specification. The revision phase should change the given specification (*Spec*) into a new (partial) specification (*Spec'*) - by making use of the diagnostic information obtained from the analysis phase - in such a way that *Spec'* no longer violates the system's property in question. Throughout, we regard requirements specifications (*Spec*) as composed of a system's description D and domain properties $P_1, P_2, ..., P_n$. In particular, we consider descriptions of deterministic systems.

[0]This work was supported by EPSRC grant GR/M38582 (VOICI project). We are grateful to the RE Group at Imperial College for useful discussions and the anonymous referees for their comments.

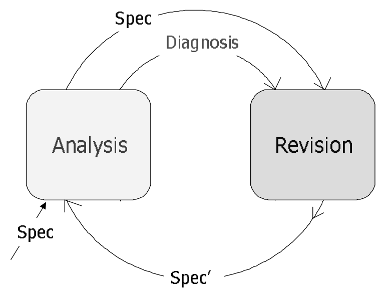

Figure 1: The cycle of requirements specification evolution

Within this framework, we have used *abductive reasoning* [6] during analysis to discover whether a description D satisfies a property P_i ($D \vdash P_i$) and, if not, generate appropriate diagnostic information, and *inductive learning* [9] during revision to change the description D into a new description D', whenever D violates P_i. To bridge the gap between analysis and revision, we have used the *diagnostic information* (Δ), obtained from our abductive procedure, to derive a number of *training examples* (Δ') for inductive learning. Although other kinds of reasoning could be used for analysis and revision (e.g., model checking and belief revision), we have found that the gap between analysis and revision could be easily bridged by abduction and induction, following the ideas put forward in [4], as shown in the sequel.

The paper is organized as follows. Section 2 describes the abductive reasoning technique used for analysis. Section 3 shows how one might generate examples of system behaviors for inductive learning from the counter-examples obtained from abduction. Section 4 describes the inductive learning technique used for revision. Section 5 concludes and discusses directions for future work.

2 Abducing Counter-examples

The tasks of validating system descriptions with respect to system properties and generating appropriate diagnostic information whenever a property is violated are performed here using an abductive reasoning approach [10] that combines both tasks into a single automated decision procedure. The problem of finding whether $D \vdash P_i$ is translated into the equivalent problem of showing that it is not possible to find a set (Δ) of state transitions that is consistent with D and that, together with D, proves the negation of P_i. In logic terms, our abductive procedure shows that $D \vdash P_i$ by failing to find a set Δ of abducibles, which is consistent with D, such that $D \cup \Delta \vdash \neg P_i$. The equivalence of these two tasks is proved in [11]. If, on the other hand, the abductive procedure finds such a Δ (wrong state transitions) then Δ acts as a counter-example to the validity of P_i.

To illustrate, we provide a simple example. Consider an electric circuit consisting of a single light bulb and two switches (A and B), all connected in series. The system's description contains rules such as: if it is not the case that switch A is on at a current state, flicking switch A causes the light to come on at the next state, provided that the light is not already on. In this paper, we represent such information using logic programming [8] and the "prime" notation often used in formal specifications. Unprimed conditions c are used to denote that c is *true* at the current state, and primed conditions c' to denote that c is *true* at the next state. As a result, assume that a (possibly incorrect) description D of our electric circuit includes the following rules: $r_1 = \neg\text{SwitchA\_On} \wedge \neg\text{Light\_On} \wedge \text{Flick\_A} \rightarrow \text{Light\_On}'$, $r_2 = \neg\text{SwitchB\_On} \wedge \neg\text{Light\_On} \wedge \text{Flick\_B} \rightarrow \text{Light\_On}'$, $r_3 = \neg\text{SwitchA\_On} \wedge \text{Flick\_A} \rightarrow \text{SwitchA\_On}'$, and $r_4 = \neg\text{SwitchB\_On} \wedge \text{Flick\_B} \rightarrow \text{SwitchB\_On}'$.[1] A system property that we would like the above description D to satisfy could be: $P = \text{Light\_On} \rightarrow \text{SwitchA\_On} \wedge \text{SwitchB\_On}$.

The abductive procedure starts by negating the property P to get $\neg P = (\neg\text{SwitchA\_On} \vee \neg\text{SwitchB\_On}) \wedge \text{Light\_On})$, which yields two parts: $\neg P_1 = (\neg\text{SwitchA\_On} \wedge \text{Light\_On})$ and $\neg P_2 = (\neg\text{SwitchB\_On} \wedge \text{Light\_On})$. Taking $\neg P_1$, the abductive procedure then tries to find a Δ such that $D \cup \Delta \vdash \neg P_1$. Consider the first condition $(\neg\text{SwitchA\_On})$ of $\neg P_1$. A possible explanation for not having Switch A On at the next state is simply not to have Switch A On at the current state and not to flick Switch A, i.e., a *no change* situation. Consider the second condition (Light_On) of $\neg P_1$. The fact that the light should be On at the next state could be explained by the following: either Switch A and the light are not On, when A is flicked (see rule r_1), or Switch B and the light are not On, when B is flicked (rule r_2). The first case is inconsistent with the *no change* situation that explains ¬SwitchA_On, since it would require A to be flicked. The second case, however, is a plausible explanation for $\neg P_1$. As a result, $\Delta = \{\neg \textit{SwitchA\_On}, \neg \textit{SwitchB\_On}, \neg \textit{Light\_On}, \textit{Flick\_B}, \neg \textit{SwitchA\_On}', \textit{SwitchB\_On}', \textit{Light\_On}'\}$ is such that $D \cup \Delta \vdash \neg P_1$. This proves that property P could be violated by the description D. The repetition of this process for $\neg P_2$ would complete the abductive procedure, possibly producing other explanations for the violation of P.

3 Generating Training Examples

A crucial aspect of the analysis-revision cycle is how to use the diagnostic information provided (Δ) to generate system behaviors (Δ') that should, instead, be covered by the system description (i.e., *training examples*). Since Δ is a counter-example, it informs us that some state transitions are not correct. Considering that an state transition is defined by a *current state*, an *event* and a *new state*,

[1] Rules $r_1, ..., r_4$ could be derived, say, from a state transition diagram. Flick_A and Flick_B are two possible *events* of the system (see [11] for an Event Calculus representation).

Δ' should include information about alternative transitions, in which one or more of these three components has been changed. Therefore, we need to decide (a) which changes to consider, and (b) which of the alternative values of such changes to consider. We address item a by only considering changes in the *new state* of a diagnosed wrong system transition. We address item b by selecting the first new state that makes Δ' consistent with P_i.

In what follows, we use the term *entry configuration* of a system behavior to refer to a current state of the system and an event with associated event conditions (if any), and *exit configuration* to refer to the new state of the system whenever the entry configuration is *true*. The diagnostic information Δ generated by our abductive procedure informs us that a given entry configuration (c_1) should not produce a given exit configuration (c_2). Taking the electric circuit example, Δ informs us that the entry configuration $c_1 = (\neg SwitchA\_On$, $\neg SwitchB\_On$, $\neg Light\_On$, $Flick\_B)$ should not produce the exit configuration $c_2 = (\neg SwitchA\_On'$, $SwitchB\_On'$, $Light\_On')$. A way of solving this problem is to make sure that c_1 produces an exit configuration c_3, different from c_2 (assuming that the system's description must be deterministic). The pair $\{c_1, c_3\}$ would be one of our *training examples*.

In general, $\Delta = \{i_1, ..., i_j, o_1, ..., o_k\}$, where $(i_1, ..., i_j)$ is an entry configuration, and $(o_1, ..., o_k)$ is an exit configuration. We want to find a training example Δ' such that there exists a $o_{l(1 \leq l \leq k)} \notin \Delta'$ and $\Delta' \cup P_i$ is consistent. There are at least $2^k - 1$ training examples to be checked for consistency. In this paper, we restrict the generation of training examples to the cases where there exists a single $o_{l(1 \leq l \leq k)} \notin \Delta'$.

Returning to the electric circuit example, recall that $\Delta = \{c_1, c_2\}$. If we start by changing $\neg SwitchA\_On'$ to $SwitchA\_On'$ in c_2, we may derive an inconsistency from the observation that switch A has changed its position without having been flicked (see c_1). Similarly, if we try to change $SwitchB\_On'$ to $\neg SwitchB\_On'$ in c_2, we may derive an inconsistency from the observation that switch B has not changed its position, despite having been flicked (again, see c_1). The remaining option would be to change $Light\_On'$ to $\neg Light\_On'$ in c_2, obtaining $c_3 = (\neg SwitchA\_On'$, $SwitchB\_On'$, $\neg Light\_On')$ and, therefore, $\Delta' = \{\neg SwitchA\_On$, $\neg SwitchB\_On$, $\neg Light\_On$, $Flick\_B$, $\neg SwitchA\_On'$, $SwitchB\_On'$, $\neg Light\_On'\}$, composed of c_1 and c_3.

4 Inducing New Specifications

We are now in a position to obtain a new system's description D', given Δ', by either defining new rules or appropriately revising existing ones in D. Recall that our ultimate goal is to find a D' such that $D' \vdash P_i$, in which case the set of training examples would be empty (indicating that the analysis-revision cycle could terminate).

In this paper, we have used the *Connectionist Inductive Learning and Logic Programming System (C-IL^{2}P)* [3, 2] to induce a revised description D' from examples Δ' and background knowledge D. *C-IL^{2}P* is a hybrid machine learning system that uses *Backpropagation* [5], the neural learning algorithm most successfully applied in industry, as the underlying learning technique. In what follows, we briefly describe the *C-IL^{2}P* system and present the results of applying it in the above electric circuit example.

C-IL^{2}P is composed of three main modules: *knowledge insertion*, *revision* and *extraction*, as depicted in Figure 2. The first module consists of a *Translation Algorithm* that takes background knowledge, described as a logic program, and generates the initial architecture and set of weights of a single-hidden layer, feedforward neural network (Figure 2(1)). That neural network computes the stable model semantics of the program inserted in it, thus guaranteeing the correctness of the translation (the proof is given in [3]). The second module revises the background knowledge by training the neural network with examples (Figure 2(2)) using standard Backpropagation with momentum. It does so by presenting the network with input and output sequences so that it can adapt (change its weights) to new situations, but taking into consideration the background knowledge that defined its initial set of weights. The third module consists of an *Extraction Algorithm* that takes the trained network and generates symbolic knowledge, described in the form of a revised logic program (Figure 2(3)). The set of extracted rules are generally more comprehensible than the trained network, facilitating the analysis of the knowledge refinement process by a domain expert (the proof of soundness of *C-IL^{2}P's* rule extraction algorithm is given in [2]).

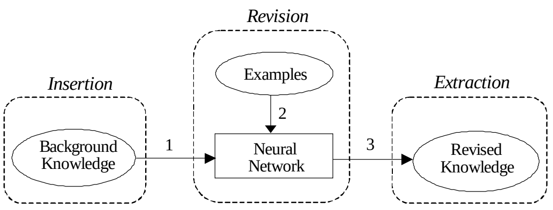

Figure 2: The Connectionist Inductive Learning and Logic Programming System

Rule extraction from trained networks is an extensive topic in its own right (see [1] for a comprehensive survey). Intuitively, the extraction task is to find the relations between input and output concepts in a trained network, in the sense that certain inputs *cause* a particular output. Neglecting many interesting details, *C-IL^{2}P* performs rule extraction by simply presenting the trained network $\mathcal{N}$ with different input sequences, and generating rules according to the output sequence obtained. The core of *C-IL^{2}P's* rule extraction algorithm is concerned with the selection of good candidate input sequences to be presented to $\mathcal{N}$, so that it can be described by a correct and compact set of rules [2].

To illustrate a run of our revision phase using *C-IL^{2}P*, we consider again the electric circuit example. Module 1 of *C-IL^{2}P* is responsible for translating rules $r_1 - r_4$ of the (partial) description D into the initial architecture of a neural network $\mathcal{N}$. It does so by mapping each rule (r_i) from the input layer to the output layer of $\mathcal{N}$, through a hidden neuron N_i. For example, rule r_1 above is mapped into $\mathcal{N}$ by simply: (a) connecting input neurons representing the concepts SwitchA_On, Light_On and Flick_A to a hidden neuron N_1, (b) connecting hidden neuron N_1 to an output neuron representing the concept Light_On$'$, and (c) setting the weights of these connections in such a way that the output neuron representing the concept Light_On$'$ is activated (or *true*) if the input neurons representing SwitchA_On, Light_On and Flick_A are, respectively, deactivated (or $false$), deactivated ($false$) and activated ($true$), thus reflecting the information provided by rule r_1.

Figure 3 shows the neural network obtained from rules $r_1 - r_4$. Note that output neuron $Light'$ must also be activated, now through hidden neuron N_2, if input neurons B_On and $Light$ are deactivated and input neuron $FlickB$ is activated (corresponding to rule r_2 above). In this initial network, positive weights (indicated in Figure 3 by solid lines) are used to represent positive literals (such as Flick_A in r_1) and negative weights (indicated in Figure 3 by dotted lines) are used to represent negative literals (such as ¬SwitchA_On and ¬Light_On in r_1). As a result, output neurons perform an *or* of the concepts represented in the hidden neurons that are connected to them, and hidden neurons perform an *and* of the concepts represented in the input neurons that are connected to them.

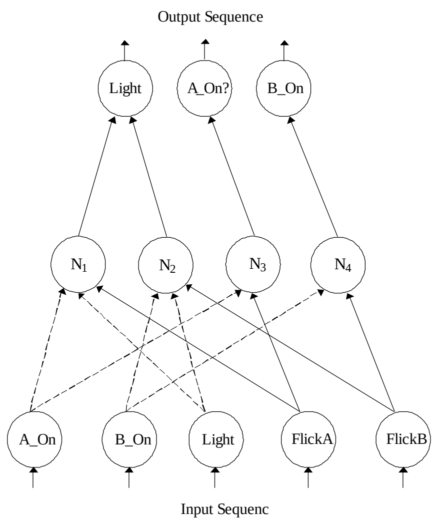

Figure 3: The neural network $\mathcal{N}$ obtained from the system's description D

Recall from Section 3 that one of our training examples is $\Delta' = \{\neg SwitchA\_On, \neg SwitchB\_On, \neg Light\_On, Flick\_B, \neg SwitchA\_On', SwitchB\_On', \neg Light\_On'\}$. As a result, module 2 of *C-IL^{2}P* was used for training the initial network $\mathcal{N}$ with input sequence $i_i = \{-1, -1, -1, -1, 1\}$ and output sequence

$o_i = \{-1,-1,1\}$, where 1 indicates *true* and -1 indicates *false*. Finally, module 3 of *C-IL²P* was applied to extract the new knowledge from the network. The extraction algorithm derived a new rule $r_2' = \{\text{SwitchA\_On} \wedge \neg\text{SwitchB\_On} \wedge \neg\text{Light\_On} \wedge \text{Flick\_B} \rightarrow \text{Light\_On}'\}$, as well as rules r_1, r_3 and r_4. In other words, the learning process has specialized rule r_2 into rule r_2', without having changed the remaining rules. Clearly, rule r_2 was under-specifying the system, and the suggestion of *C-IL²P* to the requirements engineer, as a result of learning Δ', was to add to r_2 the condition that switch A also needs to be *on* for the light to come *on* once switch B is flicked to *on*.

The revision of D into $D' = D - r_2 + r_2'$ guarantees that Δ is no longer an explanation for the violation of the domain property P. It does not guarantee that P will not be violated by the new description D'. This is why we regard the process of revising specifications as cyclic, in which the specification is being refined during each cycle, until the domain properties of the system are provably satisfied, in which case our analysis phase will not produce any new counter-example.

5 Conclusion and Future Work

In this paper, we have seen that the process of systematically changing requirements specifications can be supported by a cycle composed of an analysis phase and a revision phase, in which abductive and inductive reasoning are applied respectively. We have applied the *Analysis-Revision Cycle* in the *Automobile Cruise Control* case study.[7] The results (available upon request) provided some early validation of the cycle's capabilities.

Although the generation of training examples is guided by Δ, the definition of Δ' has been left quite open in Section 3. However, the effectiveness of our analysis-revision cycle depends on the generation of good training examples, in that the better the choice, the faster the convergence of the system to a specification that does not violate any desirable property. This may be domain dependent and, indeed, require the help of an expert. Still, we could apply heuristics to decide between mutually exclusive training examples. Therefore, a first extension of our approach would be to include heuristics to help in the choice of potential training examples.

Although the combination of inductive and analytical learning, via the use of a hybrid machine learning technique, seems to be a good choice for requirements specifications evolution, a second extension of the work would be to investigate the use of other techniques of machine learning. These include *Inductive Logic Programming*, *Knowledge-based Neural Networks*, *Explanation-based Learning* and their hybrids.[9] Finally, note that the abductive derivation of Δ assumes that the system's property (P_i) is correctly defined. However, if Δ is not validated by the stakeholders as a counter-example to P_i, this could indicate that P_i itself is wrong and, therefore, that the cycle of analysis and revision needs to be re-started with a new system property.

References

[1] R. Andrews, J. Diederich, and A. B. Tickle. A survey and critique of techniques for extracting rules from trained neural networks. *Knowledge-based Systems*, 8(6):373–389, 1995.

[2] A. S. d'Avila Garcez, K. Broda, and D. M. Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125:155–207, 2001.

[3] A. S. d'Avila Garcez and G. Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence Journal*, 11(1):59–77, 1999.

[4] P. A. Flach and A. C. Kakas. On the relation between abduction and inductive learning. In D. M. Gabbay and R. Kruse, eds, *Handbook of Defeasible Reasoning, Vol. 4*, 1–33. 2000.

[5] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2nd edition edition, 1999.

[6] A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 5, 235–324, 1994.

[7] J. Kirby. Example NRL/SCR software requirements for an automobile cruise control and monitoring system. Technical Report TR-87-07, Wang Institute of Graduate studies, 1987.

[8] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[9] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[10] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. An abductive approach for handling inconsistencies in SCR specifications. In *Proc. 3rd ICSE WISE*, Limerick, 2000.

[11] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. An abductive approach for analysing event-based requirements specifications. Technical Report TR2001/7, Department of Computing, Imperial College, 2001.

[12] A. van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *ICSE'2000*, Limerick, 2000.