Combining Abductive Reasoning and Inductive Learning 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

The development of requirements specifications inevitably involves modification and evolution. To support modification while preserving the main requirements goals and properties, we propose the use of a cycle composed of two phases: analysis and revision. In the analysis phase, a desirable property of the system is checked against a partial specification. Should the property be violated, diagnostic information is provided. In the revision phase, the diagnostic information is used to help modify the specification in such a way that the new specification no longer violates the original property.

We have investigated a particular instance of such a cycle that combines the techniques of logical abduction and inductive learning to analyse and revise specifications respectively. Given an (event-based) system description and a system property, our abductive reasoning mechanism identifies a set of counter-examples of the property, if any exists. This set is then used to generate a corresponding set of examples of system behaviours that should be covered by the system description. These examples are used as training examples by our inductive learning mechanism, which performs the necessary changes to the system description in order to resolve the property violation. The approach is supported by an abductive decision procedure and a hybrid (neural and symbolic) learning system that we have developed. A case study of an automobile cruise control system illustrates our approach and provides some early validation of its capabilities.

Keywords: Logic-based Analysis, Revision, Abduction, Inductive Learning.

1 Introduction

"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 the requirements engineering process".[38] We aim to facilitate the evolution of requirements specifications by providing the requirements engineer with tools to support the change management process.

We argue that the development of requirements specifications can be supported by a cycle composed of two phases: *analysis* and *revision*, as illustrated in Figure 1. The analysis phase is responsible for checking whether a number of desirable properties of the 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.[1]

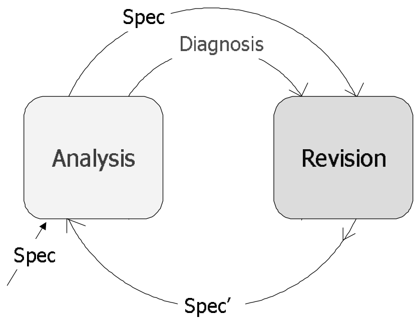

Fig. 1: The cycle of requirements specification evolution

In this paper we have investigated a particular instance of this cycle (Figure 2), which uses techniques of *abductive reasoning* [17] during the analysis phase to (*i*) discover

[1] A preliminary version of the analysis-revision cycle will appear in the Proceedings of the IEEE International Conference on Automated Software Engineering [6].

whether a given system's description satisfies a system property and (*ii*) if not, generate appropriate diagnostic information; and *inductive learning* [24] during the revision phase to change the description whenever it violates a property, utilising a machine learning algorithm. The two techniques are combined together by using the *diagnostic information* (Δ) generated by abduction, to derive a number of *training examples* (Δ') for inductive learning such that Δ' is consistent with system and domain properties. Although other kinds of formal reasoning techniques could be used for analysis and revision (such as model checking and belief revision) the use of abduction and induction facilitates the integration of these two activities, as illustrated throughout the paper and also advocated in [9].

In what follows, we concentrate on requirements specifications composed of deterministic, state transition based *system descriptions*, i.e. system requirements expressed in terms of system reactions to events, and *global system properties,* such as safety properties. An individual run of our cycle would be as follows. An event-based system description D and a system property P can be given as input to our abductive reasoning mechanism. Here, abduction is used in a refutation mode to check whether P is satisfied by the description D. In particular, it attempts to identify a (set of) counter-example(s), if any exists, to the system property P, where each such counter-example is in terms of a "current" system state, a (possibly conditioned) event, and an associated new state. Counter-examples are essentially state transitions that violate the property. Failure to find such a counter-example establishes the validity of the property and no revision is performed on the given description. If, on the other hand, any such counter-example Δ is identified, its particular form makes it simple to evaluate, as it tells us that whenever the system is in a given current state, the occurrence of the detected event should not take the system into the new state identified in Δ. A (possibly singleton) set of training examples Δ' may, therefore, be (automatically) derived from Δ by, for instance, considering transitions that would take the system into different new states that are consistent with the property P. Such training examples can be seen as correct state transitions that should be covered by the system description for the property P not to be violated. The generated Δ' is then given as input, together with the current system description, to our inductive learning mechanism. Finally, the learning process uses the training examples to perform changes in the description of the system, which should (*i*) subsume the correct state transitions Δ' and (*ii*) generate a new system description D' where the property P is expected to be no longer violated.

The paper is organised as follows. Section 2 provides a brief description of the general techniques of abduction and inductive learning. Section 3 describes our analysis-revision cycle. It defines the use of abduction in refutation mode to analyse system's properties and generate diagnostic information (i.e. counter-examples) when properties are violated. It then (*i*) presents a technique for generating correct examples (i.e. training examples) of system behaviours from a given set of counter-examples, (*ii*) describes the inductive learning technique used in the revision phase of the analysis-revision cycle, and (*iii*) shows how this learning technique may be employed to revise existing descriptions with the training examples. Section 4 applies the analysis-revision cycle in a case study, using a specification of an automobile cruise control system. Section 5 reviews related work, and Section 6 concludes and discusses directions for future work.

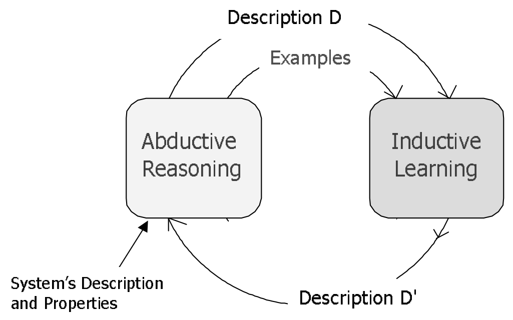

Fig. 2: Combining abductive reasoning and learning

2 Background

Formal reasoning techniques can be of three main forms: *deduction*, *abduction* and *induction.* In general terms, deduction is an *analytic* reasoning process that uses a given set of assumptions Δ (e.g., instances of a system's behaviour), and a rule-based domain-specific description D (e.g., a system's description) to infer consequences α (e.g., a given system's property). In contrast, abduction is a *constructive* reasoning process that identifies the set of assumptions Δ needed in order to infer from the description D the consequence α. Finally, induction is a *synthetic* reasoning process that produces general rules from a collection of specific instances, thus expanding the description D so that it covers such instances. Deductive reasoning is often used to support query-based reasoning on formal specifications, abductive reasoning for diagnosis, planning, theory and database updates, as well as knowledge-based software specification analysis, and inductive reasoning for performing machine learning tasks in different application domains, ranging from bioinformatics to software engineering. The formal framework proposed in this paper combines two of these three reasoning modes – abduction and induction – to support the development process of requirements specifications through iterative phases of analysis and revision. In this section, we briefly introduce these two types of reasoning.

2.1 Abductive Reasoning

Abduction is commonly defined as the problem of finding a set of assumptions (or explanation) that, when added to a given (formal) description, allows a goal (or observation) to be inferred without causing contradictions [17]. In logic terms, given a rule-based domain-specific description D and a goal G, abduction attempts to identify a set of assumptions Δ such that: (1) $D \cup \Delta \vdash G$, and (2) $D \cup \Delta$ is consistent. The set Δ is often required to satisfy two main properties: (i) it must consist only of *abducible* sentences, where the definition of what is abducible is generally related to some domain-specific notion of *causality*, and (ii) it must be minimal. For example, given a simple description composed of just one rule $Measles(X) \rightarrow Red\_Spots(X)$, and the observation $Red\_Spots(John)$, abduction would identify the single assumption $Measles(John)$ as an explanation for such observation. Existing abductive procedures, written as logic programs, work on the assumption that the given goal G is a ground sentence (e.g., an instance). This makes such procedures decidable since they only consider (starting from the goal and reasoning backwards) the ground instances of the rules included in the description D that are necessary to prove the goal.

Abduction could be used to identify assumptions of system bevahiours that would allow the inference of observations of system states from a given system description. In event-driven system descriptions, abduction would, for instance, be used to identify a trace of events and system transitions (starting from the initial state) that would prove a given requirement. This would be a *direct* use of abduction to reason about requirements specifications.

In our approach, abduction is used, instead, in *refutation mode*, to enable the *analysis* of system descriptions with respect to system properties, and the generation of counter-examples (incorrect system transitions) as diagnostic information of properties violation [31, 32]. Section 3.1 illustrates in detail this use of abduction for analysis as part of our analysis-revision cycle.

2.2 Inductive Learning

The task of inductive learning is to find hypotheses, in the form of rules, that are consistent with background knowledge to explain a given set of examples. These hypotheses are definitions of domain concepts, the examples are descriptions of instances and non-instances of such concepts to be learned, and the (possibly empty) background knowledge provides additional information about the domain [23]. In logic terms, given background knowledge $\mathcal{B}$, positive examples e^+, and negative examples e^- of some concept, inductive learning finds a hypothesis h such that: (1) $\mathcal{B} \cup h \vdash e^+$, and (2) $\mathcal{B} \cup h \nvdash e^-$.

Using the same example domain above, assume that the background knowledge is empty, and that the set of examples, also called *training examples*, is composed of the following instances (all positive examples):

$$e_1^+ : Measles(John),\ Red\_Spots(John);$$
$$e_2^+ : Measles(Dan),\ Red\_Spots(Dan);$$
$$e_3^+ : Measles(Susan),\ Red\_Spots(Susan);$$

A hypothesis for the above training examples could be $Measles(X) \rightarrow Red\_Spots(X)$. Of course, an alternative hypothesis would be $Red\_Spots(X) \rightarrow Measles(X)$. Now, assume that the background knowledge, instead of being empty, contained the common-sense knowledge about the domain *diseases trigger symptoms.* This information would clearly eliminate the latter hypothesis $Red\_Spots(X) \rightarrow Measles(X)$. Similarly, the presence of more training examples; such as the observation that even though *Peter* presented red spots, he had not contracted measles; could achieve the same result.[2] This illustrates that the quality of the selection of hypotheses increases as the amount of information about the domain increases, either in the form of more training examples or better background knowledge.

Different inductive learning techniques have been developed, based on one of these three forms of representation: symbolic [30], neural networks [14] or hybrid systems [3]. Symbolic learning has the advantage of using existing background knowledge to reduce the search space during the learning process, making it more efficient. The basic assumption is that the given background knowledge is correct, and that the learned concept should simply be added to such knowledge. An example of symbolic learning technique is any Inductive Logic Programming (ILP) technique [27]. In contrast, neural networks perform inductive learning using statistical, rather than declarative, definitions of data dependency that are encoded as weights on the network. This has enabled the neural learning process to be effective in different application domains, despite the fact that no background knowledge is used. However, neural networks can outperform symbolic learning systems, especially when the set of examples is noisy[3], as indicated in [36]. In response to these results, there is a growing interest in combining symbolic and neural learning systems [7, 5]. Such hybrid models of inductive learning try to exploit some of the advantages of both kinds of learning approaches and overcome their respective limitations. For example, by combining background knowledge and neural networks, the number of training examples that a hybrid system requires to learn a given concept may be reduced. Moreover, when background knowledge is encoded in the set of weights of a neural network, the subsequent neural learning process

[2] For instance, a negative example $e_1^- = Measles(Peter), Red\_Spots(Peter)$ would indicate that new hypotheses must not derive $Measles(Peter)$ and $Red\_Spots(Peter)$ simultaneously.

[3] A set of examples is noisy when some examples are not correct or missing altogether.

(which changes such weights based on the training examples) can not only expand the background knowledge, but also revise it when necessary. The revision phase of our *analysis-revision cycle* makes use of a specific hybrid learning system, called *Connectionist Inductive Learning and Logic Programming System (C-IL^{2}P)* [7, 5], which is explained in more detail in Section 3.3.

3 Evolving Specifications

In this section, we describe how *abductive reasoning* and *inductive learning* can be integrated to, respectively, analyse and revise specifications. We present an automated formal reasoning process that interleaves analysis and revision phases, eventually stopping when no more violations of system properties are detected, thus providing a revised specification that is consistent with such properties. The remainder of this section describes the two reasoning techniques in detail, and illustrates a run of the analysis-revision cycle using the (intentionally corrupted) description of a simple electric circuit. Recall that we are concerned with state transition based descriptions of deterministic systems.

3.1 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 [31, 32] that combines both tasks into a single automated decision procedure. This approach uses abduction in refutation mode. This means that the problem of finding whether a system description D satisfies a system property P (i.e. $D \vdash P$) 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. In logic terms, our abductive procedure shows that $D \vdash P$ by failing to find a set Δ of abducibles that is consistent with D, such that $D \cup \Delta \vdash \neg P$. The equivalence of these two tasks is proved in [32]. If, on the other hand, the abductive procedure finds such a Δ (of incorrect state transitions), then Δ acts as a set of counter-examples to the validity of P. These counter-examples describe particular (system or environmental) events occurring in particular *contexts* (classes of system's current states). These contexts, to be relevant, must themselves satisfy the properties. This is ensured by considering the properties as integrity constraints on a symbolic current state, thus pruning the set of possible counter-examples to be searched.

To illustrate the analysis phase we provide a simple example. Consider an electric circuit consisting of a single light bulb and two switches (SwitchA and SwitchB), all connected in series. Let us assume that a (possibly incorrect) description D of our electric circuit includes the following rules r_1 to r_4, formalised using propositional logic programming [20] and the "prime" notation often used in formal specifications, where unprimed conditions c denote that c is *true* at the current state, and primed conditions c' denote that c is *true* at the next state.[4]

$$\neg\text{SwitchA-On} \wedge \neg\text{Light-On} \wedge \text{FlickA} \rightarrow \text{Light-On}' \qquad (r_1)$$
$$\neg\text{SwitchB-On} \wedge \neg\text{Light-On} \wedge \text{FlickB} \rightarrow \text{Light-On}' \qquad (r_2)$$
$$\neg\text{SwitchA-On} \wedge \text{FlickA} \rightarrow \text{SwitchA-On}' \qquad (r_3)$$
$$\neg\text{SwitchB-On} \wedge \text{FlickB} \rightarrow \text{SwitchB-On}' \qquad (r_4)$$

For example, rule r_1 can be read as "if, in the current state, SwitchA is not on and the Light is not on and the event FlickA happens then the Light will be on in the next state". One of the system properties that we would like the above description D to satisfy is:

$$P = \text{Light-On} \rightarrow \text{SwitchA-On} \wedge \text{SwitchB-On}$$

Let us assume that, at the initial state, both SwitchA and SwitchB are not on. In order to check whether $D \vdash P$, our abductive procedure assumes P to be *true* at (an arbitrary) current state and checks whether its negation ($\neg P$) is true at an arbitrary next state. It applies backward reasoning from the current goal over the rules ($r_1 - r_4$), and makes use of the *default assumption* that each condition preserves its truth-value, unless changed by the occurrence of some event, whenever necessary. We refer to this as the *no change* assumption. The negated property, instantiated at an arbitratry next state, is given by $\neg P' =$ (Light-On$'$ $\wedge$ ($\neg$SwitchA-On$'$ $\vee$ $\neg$SwitchB-On$'$)), which can be re-written as $\neg P' = \neg P'_1 \vee \neg P'_2$, where:

$$\neg P'_1 = \text{Light-On}' \wedge \neg\text{SwitchA-On}'$$
$$\neg P'_2 = \text{Light-On}' \wedge \neg\text{SwitchB-On}'$$

To prove $\neg P$, the abductive procedure checks whether $\neg P'_1$ or $\neg P'_2$ can be proved from the description D. At this point, the procedure makes an arbitrary choice, say $\neg P'_1$, which is taken as an intermediate goal.[5] To prove $\neg P'_1$, the procedure has to prove both Light-On$'$ and $\neg$SwitchA-On$'$. Consider the second condition. Since no rule in the description D defines $\neg$SwitchA-On$'$, no backward reasoning step over the description can be applied. The procedure may use, however, the no change assumption to conclude that a possible explanation for not having SwitchA on at the next state is simply not to have it on at the current state and not to have the event FlickA happening. At this point, the procedure constructs a first temporary set of assumptions $\Delta_0 = \{\neg$SwitchA-On, $\neg$FlickA$\}$, and tries to prove, taking into account the set Δ_0, the second conjunct of $\neg P'_1$, which is Light-On$'$. To prove Light-On$'$, reasoning backwards, we can use either

[4]These rules could be thought of as derived from a state transition diagram, where FlickA and FlickB are the only two possible events of the system. FlickA makes SwitchA-On true at the next state, if it is not true at the current state, and vice-versa. FlickB has the same effect on SwitchB. For a detailed logic representation of this example using the Event Calculus see [32].

[5]Note that $\neg P'_1$ is a ground sentence, as required for the procedure to be decidable.

rule r_1 or r_2. In the first case, the procedure generates, in its abductive phase, the additional temporary assumptions ¬SwitchA-On, ¬Light-On and FlickA, and then checks whether these assumptions are consistency with Δ_0. This consistency check clearly fails, since Δ_0 includes ¬FlickA and the new assumptions include FlickA. Thus, this first attempt to prove Light-On′ is rejected, and the abductive reasoning phase starts again, now considering rule r_2. In its second attempt, similar to that above, the procedure generates the new additional assumptions ¬SwitchB-On, ¬Light-On and FlickB, which, this time round, are all consistent with Δ_0 and, therefore, accepted. As a result, our abductive procedure produces:

Δ={¬SwitchA-On, ¬SwitchB-On, ¬Light-On, FlickB, ¬SwitchA-On′, SwitchB-On′, Light_On′}

which is such that $D \cup \Delta \vdash \neg P_1$. Since $\neg P_1$ is provable from D, whenever Δ is *true*, $\neg P'$ is also provable from D, whenever Δ is *true*. Hence, the property is not validated by the description, and Δ is a counter-example. It says that both SwitchA and SwitchB are not on and Light is also not on, and only the event FlickB happens, taking the system into a new state where the Light is on but SwitchA is not on, thus violating the property P.

3.2 Generating Training Examples

A crucial aspect of the analysis-revision cycle is how to use the diagnostic information Δ, identified by the analysis phase, to generate system behaviours Δ' (i.e. *training examples*) that should, instead, be covered by the system description. 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*, Δ' should include information about alternative transitions, in which one or more of these three components has been changed. For instance, we might want to assume that the current state needs to be changed, or else that the current state and event should take the system into a different new state. Both modifications would be plausible. Therefore, we need to decide (a) which changes to consider, and (b) which of the alternative values of such changes to consider. In this paper, we address item (a) by only considering changes in the *new state* of a diagnosed incorrect state transition. We address item (b) by arbitrarily selecting one of the alternative new states that make Δ' consistent with P.

In what follows, we use the term *entry configuration* to refer to the current state and the event of a given state transition, with associated event conditions (if any), and *exit configuration* to refer to the new state of the transition. 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= (¬SwitchA-On, ¬SwitchB-On, ¬Light-On, FlickB) should not produce the exit configuration c_2= (¬SwitchA-On′, SwitchB-On′, Light-On′). In other words, the set $\{c_1, c_2\}$ is an incorrect state transition. 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 set $\{c_1, c_3\}$ would be one of our *training examples* (or correct state transition).

In formal terms, given a description D and a set P of properties, for each diagnostic information $\Delta = \{i_1, ..., i_j, o_1, ..., o_k\}$, where $\{i_1, ..., i_j\}$ is an entry configuration and $\{o_1, ..., o_k\}$ is an exit configuration, find a training example $\Delta' = \{i_1, ..., i_j, o'_1, ..., o'_k\}$ such that (i) there exists at least one $o'_{l\,(1 \leq l \leq k)} \notin \Delta$, and (ii) Δ' is consistent with system and domain properties.

Returning to the electric circuit example, recall that $\Delta = \{c_1, c_2\}$. If we change, for instance, ¬SwitchA-On′ to SwitchA-On′ in c_2, we may derive an inconsistency from the observation that SwitchA has changed its position without having been flicked (since the event FlickA is not included in c_1). Similarly, if we try to change SwitchB-On′ to ¬SwitchB_On′ in c_2, we may derive an inconsistency from the observation that SwitchB has not changed its position, despite having been flicked (again, see c_1). The remaining option[6] would be to change Light-On′ to ¬Light-On′ in c_2, obtaining c_3= (¬SwitchA-On′, SwitchB-On′, ¬Light-On′) and, therefore, the following valid Δ' composed of c_1 and c_3:

Δ′ = {¬SwitchA-On,¬SwitchB-On,¬Light-On, FlickB, ¬SwitchA-On′,SwitchB-On′,¬Light-On′}

3.3 Inducing a New Specification

So far, we have seen that a desirable property P of a system may be checked against its system description D, *abducing* a number of counter-examples Δ whenever P is violated. This information can then be used to generate a set Δ' of state transitions that should be covered by the system description, if property P is no longer to be violated. In terms of Machine Learning, the set Δ' can be seen as the training examples upon which a learning mechanism can be applied. By either defining new state transition rules or appropriately revising existing ones, a new system description D' is expected to cover all the instances in the set of training examples.

We are now in a position to *induce* a revised description D' from Δ', using D as background knowledge. Recall that our ultimate goal is to find a D' such that $D' \vdash P$, in which case the set of training examples would be empty (indicating that the analysis-revision cycle could terminate). In this paper, we use the *Connectionist Inductive Learning and Logic Programming System (C-IL^{2}P)* [7, 5] to induce D'. *C-IL^{2}P* is a hybrid machine learning system that uses *Backpropagation* [14], the neural learning

[6] In this paper, we restrict the generation of training examples to the cases where there exists a single o'_l such that $o'_{l\,(1 \leq l \leq k)} \notin \Delta$.

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 electric circuit example used above. A discussion about the choice of *C-IL^{2}P*, in comparison with other machine learning techniques, is given in Section 4.2.

C-IL^{2}P is a hybrid system, based on a neural network, that integrates inductive learning from examples and background knowledge with logic programming. The system is composed of three main modules: *knowledge insertion*, *revision* and *extraction*, as shown in Figure 3. The insertion module consists of a *Translation Algorithm* that takes background knowledge, described as a general logic program [20], and generates the initial architecture and set of weights of a (single-hidden layer and feedforward) neural network (Figure 3(1)). That neural network computes the (stable model [12]) semantics of the program inserted in it, thus guaranteeing the correctness of the translation (the proof of the equivalence between the logic program and the neural network is given in [7]). The revision module revises the background knowledge by training the neural network with examples (Figure 3(2)) using standard Backpropagation. 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 extraction module consists of an *Extraction Algorithm* that takes the trained network and generates new symbolic knowledge, described in the form of a logic program (Figure 3(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 the correctness of *C-IL^{2}P's* rule extraction algorithm is given in [5]).

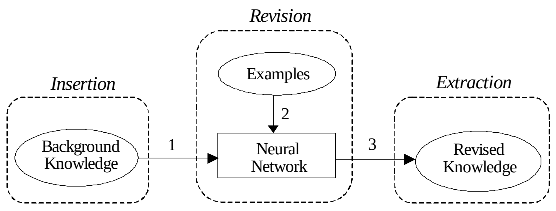

Fig. 3: The C-IL^{2}P System

Rule extraction from trained networks is an extensive research 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 [5].

Example. 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 $r_1 = \neg\text{SwitchA\_On} \wedge \neg\text{Light\_On} \wedge \text{Flick\_A} \rightarrow \text{Light\_On}'$ 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 4 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 $r_2 = \neg\text{SwitchB\_On} \wedge \neg\text{Light\_On} \wedge \text{Flick\_B} \rightarrow \text{Light\_On}'$). In this initial network, positive weights (indicated in Figure 4 by solid lines) are used to represent positive literals (such as Flick_A in r_1) and negative weights (indicated in Figure 4 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. For example, from Figure 4, output neuron $Light'$ will be activated if and only if either N_1 or N_2 is activated. Hidden neuron N_1 will be activated if and only if A_On and $Light$ are deactivated (see dotted lines) and $FlickA$ is activated. Similarly, hidden neuron N_2 will be activated if and only if B_On and $Light$ are deactivated and $FlickB$ is activated. In logic terms, $Light'$ will be $true$ if and only if either A_On and $Light$ are $false$ and $FlickA$ is $true$ (corresponding to rule r_1), or B_On and $Light$ are $false$ and $FlickB$ is $true$ (corresponding to rule r_2).

Recall from Section 3.1 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 $i_i = \{A\_On = -1, B\_On = -1, Light = -1, FlickA = -1, FlickB = 1\}$ and output sequence $o_i = \{Light' = -1, A\_On' = -1, B\_On' = 1\}$, where 1 indicates $true$ and -1 indicates $false$. Finally, after the network was trained with example (i_i, o_i), 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 specialised rule r_2 into rule r'_2, without having changed the remaining rules. Clearly, rule r_2 was underspecifying 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 analogous change to rule r_1 (in which SwitchB_On would be added) would require training another example.

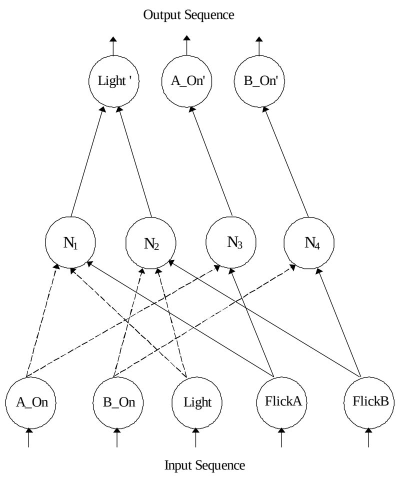

Fig. 4: The network $(\mathcal{N})$ obtained from description D

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 evolving 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 training example.

4 Case Study

We have applied the analysis-revision cycle to evolve a specification of an automobile cruise control system [19]. Different specifications of such a system have been presented in the literature [34]. In this case study, we have considered a deterministic, partial state transition based specification, in which the system must be in one of four possible states at any given time: *off*, *inactive*, *cruise* or *override*. Several environmental and system variables are considered in the specification, such as the ignition switch, the cruise control lever, and the automobile's speed. Events in the environment cause changes in the values of domain and system variables, and may cause the system to change its state, according to the values of the event conditions. For example, when the system is in state *inactive*, if the ignition is on, the engine is running, and the brake is off, the event of moving the cruise control lever to the *activate* position is supposed to take the system into the state *cruise*. Figure 5(a) shows two such state transitions where the event of pressing the brake should take the system from state *cruise* to state *override*, provided that the automobile is not going too fast; and the event of moving the lever to *activate* should take the system from state *override* back to state *cruise*, provided that the brake is not engaged.

One of the cruise control system's safety properties states that "if the system is in state *cruise* then the ignition switch should be on, the engine should be running and the brake should not be engaged". This property has been found to be violated. The partial state transition given in Figure 5(a) implicitly assumes that, whenever the conditions of the event transition *brake* from *cruise* to *override* are $false$, the system will stay in its state *cruise*. As a result, when the system is in state *cruise* and the *brake* is engaged, but the automobile is going $toofast$, the system remains in state *cruise*, therefore violating the above safety property. Such an incorrect transition is depicted in Figure 5(b) as the diagnostic information Δ.

The process of deriving a training example Δ' from Δ is illustrated in Figure 5(c). It can be seen as simply changing the state to which an incorrect transition points to. In this case study, as we will see in the sequel, the incorrect transition Δ becomes transition Δ', so that when the system is in state *cruise* and the *brake* is engaged, but the automobile is going $toofast$, the system moves into state *override*. Finally, we need to accommodate Δ' consistently into the original specification. In this case study, this has been done by changing the conditions of the original state transition from *cruise* to *override*, as illustrated in Figure 5(d) (compare with Figure 5(a)).

In what follows, we will explain in detail how our cycle of analysis and revision has been used to achieve such a change. In order to combine the abductive reasoning process with *C-IL²P*, we have used a logic programming implementation of the original formalisation given in [32]. Each state transition is formalised as follows:

$$s_c \wedge c_1 \wedge \ldots \wedge c_n \wedge e \rightarrow s'_n$$

where s_c and s_n represent, respectively, the current and next state of the transition, e denotes the event transition, and c_1, ..., c_n denote the conditions of the event ($n = 0$ if the event transition does not have any condition). The rule defines the transition of the system from a current state to a (different) next state whenever an event happens and the conditions of the event are *true*. Each rule has a counterpart rule of the form $s_c \wedge c_1 \wedge \ldots \wedge c_n \wedge e \rightarrow \neg s'_c$ to express that, when the transition occurs, its current state is no longer *true*. In addition, each event has rules to express the effect

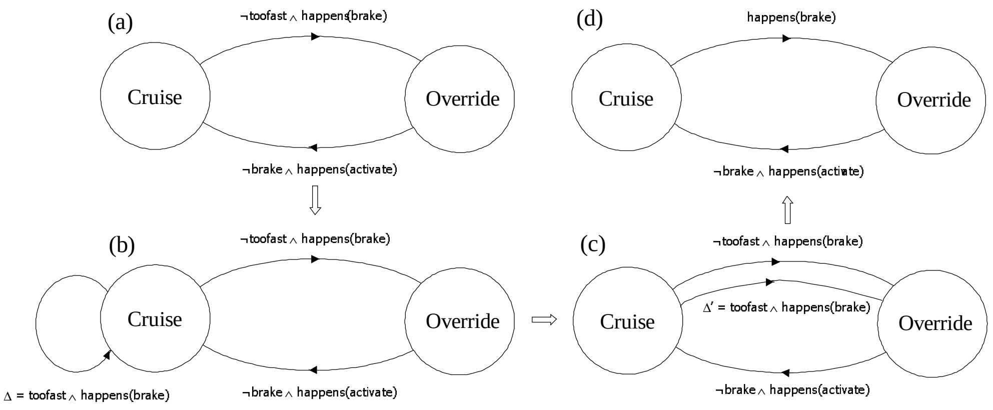
(a)
¬toofast ∧ happens(brake)
Cruise
Override
¬brake ∧ happens(activate)
(d)
happens(brake)
Cruise
Override
¬brake ∧ happens(activate)
(b)
¬toofast ∧ happens(brake)
Cruise
Override
Δ = toofast ∧ happens(brake)
¬brake ∧ happens(activate)
(c)
¬toofast ∧ happens(brake)
Δ' = toofast ∧ happens(brake)
Cruise
Override
¬brake ∧ happens(activate)

Fig. 5: Analysis and revision of state transitions

on its associated variable. For example, for an event *brake*, the rule $\neg brake \wedge happens(brake) \rightarrow brake'$ must be introduced. Domain properties have also been considered, as for example, the property that, at any given time, the cruise control lever has to be in exactly one of the three positions: *activate*, *deactivate* or *resume*. Rules r_1 to r_4 below are part of the logic programming representation of the case study[7]: r_1 = override ∧ ¬brake ∧ ¬activate ∧ happens(activate) → cruise′; r_2 = ¬brake ∧ happens(brake) → brake′; r_3 = cruise ∧ ignited ∧ running ∧ ¬toofast ∧ ¬brake ∧ happens(brake) → override′; and r_4 = ¬activate ∧ happens(activate) → activate′.

The set of properties that the description has to verify at any point in time is given by P_1 to P_5 below, where "|" means "exclusive or" and the logical operator "↔" means "if and only if".

off | inactive | cruise | override	(P_1)
off ↔ ¬ ignited	(P_2)
inactive → ignited ∧ (¬ running ∨ ¬activated)	(P_3)
cruise → ignited ∧ running ∧ ¬brake	(P_4)
override → ignited ∧ running	(P_5)

In the logic programming representation, each of the properties ($P_1 - P_5$) is converted into a set of integrity constraints (*ic*). The rationale behind the conversion is that if, for example, $a \mid b \mid c$ were a desirable property of the system then having any two of a, b and c would not be desirable and should lead to an inconsistency. Of course, having none of a, b and c should also lead to an inconsistency (indicated by $\perp$). Similarly, if $a \rightarrow b$ were a desirable property, having a and $\neg b$ should imply $\perp$, if $a \rightarrow b \vee c$ were a desirable property, having a, $\neg b$ and $\neg c$ should imply $\perp$, and so on. For example, the conversion of property P_4 gives three integrity constraints: $ic_{4.1} = \{cruise \wedge \neg ignited \rightarrow \perp\}$, $ic_{4.2} = \{cruise \wedge \neg running \rightarrow \perp\}$ and $ic_{4.3} = \{cruise \wedge brake \rightarrow \perp\}$. The conversion of P_5 derives two integrity constraints: $ic_{5.1} = \{override \wedge \neg ignited \rightarrow \perp\}$ and $ic_{5.2} = \{override \wedge \neg running \rightarrow \perp\}$.

[7]Rule r_3 defines fully the state transition from *cruise* to *override*, which is only partially illustrated in the diagrams of Figure 5.

4.1 Evolving the Rules with Examples

Let us now illustrate a run of the analysis-revision cycle, when checking whether property P_4 is verified. As mentioned in Section 3.1, the abductive phase tries to prove the negation of P_4 at an arbitrary next state of the system. Taking, for instance, $ic_{4.3}$, it tries to prove $cruise' \wedge brake'$. To prove $brake'$, reasoning backwards, it considers rule r_2 above, and starts constructing a temporary set of assumptions $\Delta_0 = \{\neg brake, happens(brake)\}$. It then tries to prove $cruise'$. One possibility is to use the *no change* (default) assumption, i.e. assume *cruise* to be *true* as a current state and prove that transitions leading to any other state do not happen. In this case, rule r_3 (taking the system into *override*) must not be applied. To prove this, the procedure has to fail proving, consistently with Δ_0, at least one of the conditions of r_3. Both *ignited* and *running* must be *true*; otherwise integrity constraints $ic_{4.1}$ and $ic_{4.2}$, respectively, would be violated. However, $\neg toofast$ can be proved to fail, by simply considering (or abducing) the assumption *toofast*. At this point, the abductive procedure stops, generating a counter-example[8]: Δ = {cruise, ignited, running, toofast, ¬brake, happens(brake), cruise′, brake′, ignited′, running′, toofast′}.

A number of training examples (Δ') can be obtained from Δ. If, for instance, we try and make $cruise'$ *false*, we know from property P_1 that one of off', $inactive'$ or $override'$ has to be made *true*. From property P_2, having off' and $ignited'$ simultaneously would lead to an inconsistency. So, we are left with two options: $inactive'$ or $override'$, and, therefore, two possible training examples: Δ' = {*cruise, ignited, running, toofast, ¬brake, happens(brake), override′, ignited′, running′, toofast′, brake′*}, and Δ'' = {*cruise, ignited, running, toofast, ¬brake, happens(brake), inactive′, ignited′, running′, toofast′, brake′*}. At this point, either we recourse to the knowledge of a domain expert, or we try and apply some heuristics to choose between Δ' and Δ''. A heuristic

[8]Note that other counter-examples could be generated by running the abductive procedure again and applying backward reasoning on other rules of the system description.

that we have used here was as follows: although neither *override'* nor *inactive'* leads to an inconsistency, in the case of Δ', none of the constraints regarding *override'* (see property P_5) could possibly be violated, since Δ' states that both *ignited'* and *running'* must be *true*. On the other hand, one of the constraints regarding *inactive'* (see property P_3) could clearly be violated by Δ''. This is so because, although *running* is *true*, *activate* is undefined in Δ''. If we assumed, thus, that *activate* is *true* then $inactive \rightarrow (\neg running \vee \neg activate)$ would be violated. In this case, we say that Δ' is preferred over Δ'' because on the number of *potential property violations*. The definition of a metric to guide the above choice of mutually exclusive training examples is still work in progress.

Taking Δ' as our training example (Figure 5(c)), we are now in a position to (1) translate the system description into the initial architecture of a neural network N, (2) train N with examples, and (3) extract a revised description from the trained network. Figure 6 shows a small part of the network, in which rules r_2 and r_3 are represented. In addition, the *no change* assumption regarding *cruise* is explicitly represented by connecting input neuron *cruise* to output neuron *cruise'* via hidden neuron r_{def}, and making sure that whenever neuron r_3 is activated, neuron *cruise'* is deactivated (i.e. the outcome of neuron r_{def} is blocked) by means of the dotted line (a negative weight). This corresponds to the situation where the system is supposed to stay in *cruise* unless a transition into *override* happens [4] (Figure 5(d)).

The network N is then fed training example Δ', describd above, which gives the following input and output vectors to N: $i_i = \{cruise = 1, ignited = 1, running = 1, toofast = 1, brake = -1, happens(brake) = 1\}$ and $o_i = \{cruise' = -1, override' = 1, brake' = 1\}$. After the network is trained, the extraction of rules from N indicates that *C-IL²P* has combined the background rule r_3 and the training example Δ' to determine that the truth-value of *toofast* should be irrelevant to the conclusion of *override'*. It has done so by changing the background rule r_3 into the new rule $r_3' = cruise \wedge ignited \wedge running \wedge \neg brake \wedge happens(brake) \rightarrow override'$.

4.2 Tool Support and Discussion

The analysis phase of our analysis-revision cycle uses an abductive logic programming proof procedure. The tool was implemented in Prolog and uses (*i*) a logic program conversion of the given specification, and (*ii*) the abductive logic program module described in [18]. The revision phase of our analysis-revision cycle uses the modules of knowledge insertion, revision and extraction of the *C-IL²P* system, which was implemented in **C**.

The above case study shows how the analysis-revision cycle proposed in this paper can be used to analyse and revise requirements specifications. It indicates that the choice of the training examples is a very important one, in that the better the choice, the faster the convergence of the system to a specification that does not violate any desirable property.

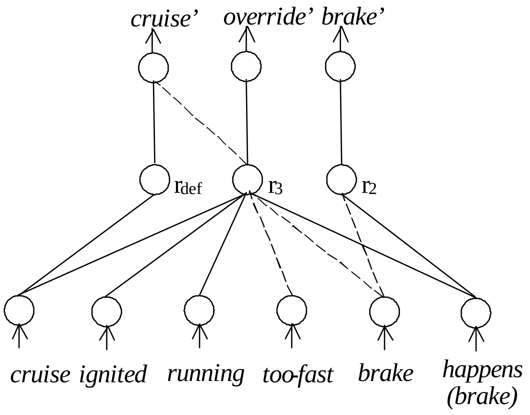

Fig. 6: Part of the network for the cruise control system

Different learning techniques could be applied in the process of revising specifications. As a result, a major problem we had to tackle in this paper was how to find an appropriate inductive learning technique for such a task. In the context of requirements specifications, firstly, it is desirable to use a *push-button* technique, i.e. a technique that does not require the user to have specific knowledge in order to use it. In addition, in comparison with most problems of machine learning, here we are faced with a limited number of training examples, which may compromise the accuracy of the results of the learning process. On the other hand, we have background knowledge, i.e. some domain specific, prior information D, which may be useful to compensate the reduced number of training examples. Moreover, such background knowledge may be incomplete and incorrect, since we work with partial specifications. As a result, traditional techniques of inductive learning that perform training from examples only, do not seem to be adequate. This includes Case Based Reasoning [24] and most models of Artificial Neural Networks [14]. We are left with (*i*) Inductive Logic Programming (ILP) techniques, (*ii*) Hybrid (neural and symbolic) Systems; and (*iii*) Explanation Based Learning (EBL) algorithms (see [24]). Among these, hybrid systems seem to be more appropriate as far as dealing with incorrect background knowledge and theory refinement is concerned [7, 37, 36]. Hybrid systems are not normally push-button techniques though, as they typically use traditional neural learning algorithms (such as *Backpropagation*), which require the adaptation of a learning rate via trial and error. On the other hand, explanation based learning algorithms seem to require less interaction with the user [25], but are not appropriate in the presence of incorrect domain knowledge [24], as they rely heavily on the correctness of the background knowledge in order to generalise rules from fewer training examples. Finally, while the strength of ILP lies in the ability to induce relations due to the use of a more expressive language, most ILP systems present significant limitations in terms of efficiency due to the extremely large space of possible hypotheses that needs to be searched. In the presence of incorrect background knowledge, such a problem may become intractable.

5 Related Work

The approach presented in this paper integrates two formal techniques for analysing and revising requirements specifications. Most of the techniques presented in the literature address either one of these two activities independently, but not both.

Several formal techniques have been developed for analysing requirements specifications, such as those based on theorem proving or model checking, and declarative logic-based approaches. Techniques based on theorem proving [29] might not be decidable, thus not always terminating. On the other hand, techniques based on model checking facilitate automated analysis and generation of counter-examples, when errors are detected [2, 15]. They provide as counter-examples long traces of system executions. In contrast, our abductive procedure generates counter-examples as individual state transitions. This facilitates the mapping of counter-examples into training examples to be handled by inductive learning.

Among declarative logic-based approaches, the work of van Lamsweerde et. al. [39] is particularly relevant. This describes a goal-driven approach to requirements engineering in which "obstacles" are part of a specification that leads to a negated "goal". This approach is similar to our abductive analysis technique in that its notion of goal is similar to our notion of system property, and its notion of obstacles is analogous to our notion of abducibles. However, our abductive decision procedure differs from the goal-regression technique used in [39] in that it uses grounded goals to make the procedure decidable, and integrity constraints to validate the properties efficiently [32]. Other examples of the application of abduction to software engineering tasks can be found in [22].

A variety of (logic-based) formal techniques have also been developed for revising requirements specifications in order to resolve inconsistencies. Our revision process can be seen as one of these techniques, since violation of a property is an example of inconsistency detection between the system description and the system property. Belief revision for default theories has been suggested as a formal approach for resolving inconsistencies arising during the evolution of requirements specifications [41]. The inconsistency detection is implicit in the definition of the belief revision operator, and the process is a single-shot revision. In contrast, our approach provides an explicit analysis of inconsistency via the use of abductive reasoning, and a cyclic, interactive process of revision. Most of the existing techniques for revising requirements specifications (see also [33]) perform revision comprising only the addition and deletion of deducible and existing requirements. In contrast, with the use of inductive learning, the revision process presented in this paper enables the evolution of specifications via the acquisition of genuinely new requirements.

Of the inductive learning-based approaches, the work of van Lamsweerde and Willemet [40] is particularly relevant. It describes the use of an inductive based goal inference procedure to elicit new requirements from sets of operational scenarios. This approach uses symbolic inductive learning and, therefore, differs from our technique in the same way as purely symbolic systems differ from hybrid learning systems. Moreover, while in [40] the use of a learning technique is aimed at the elicitation of new requirements from scenarios provided by the user, in our approach the focus of the learning technique is on the generation of new requirements in order to resolve detected errors in the specification. Training examples are generated by the analysis phases, and the learning process performs a revision of the (possibly incorrect) partial specification. Learning techniques have also been used in other software engineering applications, such as the inference of process models from process traces [11] and the validation of an air trafic control requirements model [21].

6 Conclusion and Future Work

In this work, 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. Our approach provides both theoretical foundations and practical techniques for the development of logic-based methods of requirements specifications. It also contributes to the management of inconsistency in requirements specifications [28, 16, 13]. Following the idea that inconsistency should be seen as a "trigger for actions" [10], this paper shows that one of these actions could be learning [8].

Although the generation of training examples is guided by Δ, the definition of Δ' has been left quite open in Section 3.2. However, the effectiveness of our analysis-revision cycle depends on the generation of good training examples. 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. Thus, an important extension of this work would be to investigate the use of new 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, another extension of this work would be to investigate the use of other techniques of machine learning. These include *Inductive Logic Programming*, *Knowledge-based Artificial Neural Networks* (KBANN) [37] and *Explanation-based Neural Networks*

EBNN [35] (and their hybrids, e.g., [26]). This would allow us to perform more detailed technical evaluation of these techniques, and draw general conclusions on when, why, and for which type of requirements specifications, one technique would be more appropriate than others. The extension of our analysis-revision cycle to elicit new requirements from scenarios would be another interesting direction to pursue.

Finally, note that the abductive derivation of diagnostic information assumes that system properties are correctly defined. However, if a diagnostic information is not validated by the stakeholders as a counter-example to a property, this could indicate that the property 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 artificial neural networks. *Knowledge-based Systems*, 8(6):373–389, 1995.

[2] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. Technical Report NRL-7999, Naval Research Lab, 1997.

[3] I. Cloete and J. M. Zurada, editors. *Knowledge-Based Neurocomputing*. The MIT Press, 2000.

[4] A. S. d'Avila Garcez, K. Broda, and D. M. Gabbay. Metalevel priorities and neural networks. In *ECAI2000, WS Foundations of Connectionist-Symbolic Integration*, Berlin, 2000.

[5] 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.

[6] A. S. d'Avila Garcez, A. Russo, B. Nuseibeh, and J. Kramer. An analysis-revision cycle to evolve requirements specifications. In *Proc. of IEEE Automated Soft. Engineering Conference*, 2001.

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

[8] S. Easterbrook. Learning from inconsistency. In *WS on Software Specification and Design*, Paderborn, 1996.

[9] P. A. Flach and A. C. Kakas. On the relation between abduction and inductive learning. In *Handbook of Defeasible Reasoning and Uncertainty Management Systems, Vol. 4*, pages 1–33. 2000.

[10] D. M. Gabbay and A. Hunter. Making inconsistency respectable: a logical framework for inconsistency in reasoning. part 1: a position paper. In *Fundamentals of AI Research*. Springer, 1991.

[11] P. K. Garg and S. Bhansali. Process programming by hind-sight. In *Proceeding ICSE14*, pages 280–293, Melbourne, 1992.

[12] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Logic Programming Symposium*, 1988.

[13] C. Ghezzi and B. Nuseibeh, eds. *Special Issue of IEEE Transactions on Software Engineering on Managing Inconsistency in Software Development*. November 1999.

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

[15] C. L. Heitmeyer et al. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transaction on Software Engineering*, 24(11):927–947, 1998.

[16] A. Hunter and B. Nuseibeh. Managing inconsistent specifications: Reasoning, analysis and action. *Transactions on Software Engineering and Methodology, ACM Press*, 1998.

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

[18] A. C. Kakas and A. Michael. Integrating abductive and constraint logic programming. In *Proc. International Conference on Logic Programming*, Tokyo, Japan, 1995.

[19] 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.

[20] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1987.

[21] T. L. McCluskey and M. M. West. The automated refinement of a requiremens domain theory. Technical report, School of Computing, University of Huddersfield, 1999.

[22] T. Menzies. Applications of abduction: Knowledge level modeling. *International Journal of Human Computer Studies*, 1996.

[23] R. S. Michalski. Learning strategies and automated knowledge acquisition. In *Computational Models of Learning*, Symbolic Computation. Springer, 1987.

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

[25] T. M. Mitchell and S. B. Thrun. Explanation-based learning: A comparison of symbolic and neural network approaches. In *International Conference on Machine Learning*, Amherst, 1993.

[26] R. J. Mooney and J. M. Zelle. Integrating ILP and EBL. In *SIGART Bulletin*, volume 5, pages 12–21. 1994.

[27] S. Muggleton and L. Raedt. Inductive logic programming: theory and methods. *Journal of Logic Programming*, 19:629–679, 1994.

[28] B. Nuseibeh, S. Easterbrook, and A. Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 56(11), 2001.

[29] S. Owre et al. Formal verification for fault-tolerant architecture: Prolegomena to the design of pvs. *IEEE Transactions on Sofwtare Engineering*, 21(2):107–125, 1995.

[30] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

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

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

[33] K. Satoh. Computing minimal revised logical specification by abduction. In *Proc. WS Principles of Software Evolution*, 1998.

[34] M. Shaw. Comparing architectural design styles. *IEEE Software*, 1995.

[35] S. Thrun. *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Kluwer, 1996.

[36] S. B. Thrun et. al. The MONK's problems: A performance comparison of different learning algorithms. Technical Report CMU-CS-91-197, Carnegie Mellon, 1991.

[37] G. G. Towell and J. W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1):119–165, 1994.

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

[39] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirement engineering. *IEEE Transaction on Software Engineering*, 1998.

[40] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 1998.

[41] D. Zowghi and R. Offen. A logical framework for modeling and reasoning about the evolution of requirements. In *Proc. 3rd IEEE Int. Symposium on Requirements Engineering*, 1997.