Combining abductive reasoning and inductive learning to evolve requirements specifications

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

Abstract: The development of requirements specifications inevitably involves modification and evolution. To support modification while preserving particular requirements goals and properties, the use of a cycle composed of two phases: analysis and revision is proposed. 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. An instance of the above analysis–revision cycle that combines new techniques of logical abduction and inductive learning to analyse and revise specifications, respectively is investigated. More specifically, given an (event-based) system description and a system property, abductive reasoning is applied in refutation mode to verify whether the description satisfies the property and, if it does not, identify diagnostic information in the form of a set of examples of property violation. These (counter) examples are then used to generate a corresponding set of examples of system behaviours that should be covered by the system description. Finally, such examples are used as training examples for inductive learning, changing the system description in order to resolve the property violation. This is accomplished with the use of the connectionist inductive learning and logic programming system—a hybrid system based on neural networks and the backpropagation learning algorithm. A case study of an automobile cruise control system illustrates the approach and provides some early validation of its capabilities.

1 Introduction

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, as illustrated in Fig. 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 (possibly combined with scenarios provided by stakeholders) – in such a way that Spec′ no longer violates the system's property in question [1].

In this paper, we present an instance of the analysis–revision cycle (Fig. 2), which uses techniques of abductive reasoning [2] during the analysis phase to (i) discover whether a given system description satisfies a system property and (ii) if not, generate appropriate diagnostic information in the form of training examples; and inductive learning [3] during the revision phase to change the system description whenever it violates a property, utilising a machine learning algorithm. The two techniques are combined together by using the counter-examples generated by abduction to derive a number of training examples for inductive learning that is consistent with the system property.

More specifically, 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. We use abductive reasoning in refutation mode so that the analysis of a property P consists of an attempt to identify a hypothesis Δ that is consistent with the system description D such that

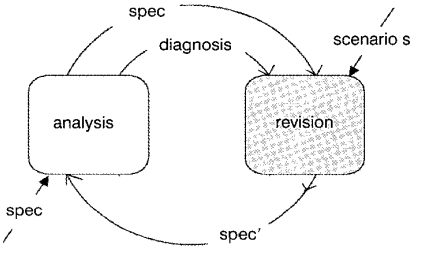

Fig. 1 *Cycle of requirements specification evolution*

IEE Proceedings online no. 20030207

DOI: 10.1049/ip-sen:20030207

Paper first received 24th October and in revised form 11th December 2002

A.S. d'Avila Garcez is with the Department of Computing, City University, Northampton Square, London, EC1V 0HB, UK

A. Russo and J. Kramer are with the Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2BZ, UK

B. Nuseibeh is with the Computing Department, The Open University, Walton Hall, Milton Keynes, MK7 6AA, UK

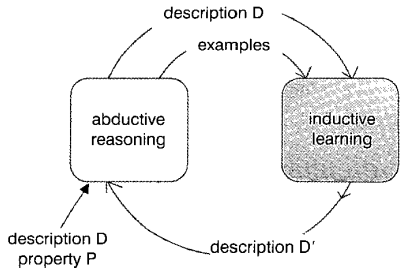

Fig. 2 *Combining abductive reasoning and learning*

D and Δ entail the negation of the property. If the abductive procedure finds a hypothesis Δ then it will act as a counter-example to the validity of the property. The procedure always terminates, and if it fails to find Δ then it guarantees that the system description satisfies the property [4]. In addition, we use a hybrid system—the connectionist inductive learning and logic programming system (C-IL^2 P) [5]—to revise a system description from examples. C-IL^2 P is a neural networks based system that performs inductive learning from examples and background knowledge (in this case, the incorrect description D), using the Back-propagation learning algorithm—the neural learning algorithm most successfully applied in industry, based on efficient gradient descent computation [6].

Of course, other kinds of reasoning techniques could be used for analysis and revision such as Model Checking [7] and Belief Revision [8], respectively. However, when considering the cycle of analysis and revision, the similarities between the techniques of abduction and induction facilitate the integration of these two activities, as also advocated in [9]. There are two integration steps (Fig. 2): one when we move from analysis to revision, and another when we move back, from revision to analysis. In the first step, the analysis mechanism must produce training examples that the revision mechanism can use. In general, there are not many examples available for learning, so that the role of analysis is also to guide the generation of new examples. As we will see in this paper, in particular in Section 4.2, abductive reasoning is an adequate mechanism for doing so. In the second step, the revision mechanism must produce a clear (symbolic) description D′, given an incorrect description D and a reduced number of examples. Not many techniques of inductive learning can cope with such a difficult revision task [3]. As we will also see in the sequel, especially in Section 5.2, hybrid learning systems are an adequate mechanism for revision in this case.

A run of the analysis–revision cycle would be as follows. During analysis, an event-based system description D and a system property P are given to our abductive reasoning mechanism, which checks whether D satisfies P. In particular, it attempts to identify counter-examples to the property P, which are essentially state transitions that violate P in the form of a current system state, an event, and an associated next state. If abduction fails to find any counter-example to the property P then we know that the description D satisfies P, and no revision is performed on D. If, however, a counter-example Δ is identified, it informs us that, whenever the system is in a given current state, the occurrence of the detected event should not take the system into the detected next state. As a result, a (possibly singleton) set of training examples Δ′ may be derived from Δ automatically, for example, by defining alternative transitions, consistent with the property P, that would take the system into different next states. Such training examples can be seen as state transitions that should be incorporated into the system description so that it no longer violates a given property. This is the task of revision. Given background knowledge D and training examples Δ′, our inductive learning mechanism must change D into a new system description D′ that is expected not to violate property P. D′ must incorporate Δ′, which eliminates Δ as a state transition that violates P. The C-IL^2 P hybrid system does so by translating D into the initial architecture of a neural network N, applying the Back-propagation learning algorithm on N in order to incorporate training examples Δ′, thus deriving a trained neural network N′, and translating the trained network N′ back into a symbolic representation D′ [5].

During revision, while the translation of D into N is straightforward, as we will see in Section 4.3, the translation of N′ into D′ is not trivial. It is essential, though, to provide explanation capability to neural networks in the form of symbolic rules. This task is known as rule extraction from trained neural networks [10], and although it is exponential on the number of input neurons in the worst case, it works well in practice for extremely large neural networks [11]. In addition, note that induction alone cannot guarantee that D′ satisfies P, as it normally includes some generalisation of the training examples. As a result, the (sound and complete) abductive mechanism of the analysis phase [4] should be responsible for verifying whether D′ satisfies P. This is one of the reasons why the process of evolving requirements specifications should be seen as an analysis–revision cycle. Of course, due to the way that the training examples Δ′ are generated from violations Δ of properties P, we expect inductive learning to guide us towards a description D′ that satisfies P. Hence, a scalable analysis–revision cycle based on abduction and induction would need to concentrate efforts on the development of efficient rule extraction algorithms and efficient abductive proof procedures, as we will discuss in the sequel. As a result, the nature of the problem of requirements evolution renders the use of abduction and machine learning both appealing and challenging also from a computational intelligence perspective. While requirements engineering can benefit from the use of successful abductive reasoning and machine learning techniques, new and efficient computational intelligence algorithms need to be developed to support the evolution of requirements specifications [12].

2 Abductive reasoning

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 abductive reasoning as an analysis technique. In the next section, we will look at inductive learning as a revision technique.

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 [2]. 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 (a) $D \cup \Delta \vdash G$ and (b) $D \cup \Delta$ 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 behaviours 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, in order to enable both (i) the analysis of system descriptions with respect to system properties and (ii) the generation of counter-examples (incorrect system transitions) as diagnostic information of properties violation [4, 13]. In refutation mode, the analysis task is translated into the equivalent task of showing that it is not possible to consistently extend the description D with assumptions Δ in such a way that the extended description entails the negation of the goal G, i.e. $D \cup \Delta \vdash \neg G$. The equivalence between these two tasks is shown in Theorem 1. Note that, in the context of analysis, our goal G is a global system property P such as safety properties.

Definition 1 [4, 14]: Let $\langle D, Ab, C \rangle$ be an abductive framework, where D is a system description, Ab is a set of abducibles and C is a (possibly empty) set of constraints of the form $c_1 \wedge c_2 \wedge \cdots \wedge c_n \rightarrow \bot$, which indicates that satisfying all conditions c_i $(1 \leq i \leq n)$ simultaneously is undesirable, and should lead to an inconsistency (indicated by $\bot$). Let G be a global system property. A set Δ of abducibles ($\Delta \subseteq Ab$) is called a counter-example of the property G if and only if (i) $D \cup \Delta \vdash \neg G$, (ii) $D \cup \Delta$ is consistent and (iii) $D \cup \Delta \vdash C$ if $C \neq \emptyset$.

Theorem 1 [4, 14]: Let $\langle D, Ab, C \rangle$ be an abductive framework, and let G be a global system property. Then, $D \vdash \neg G$ if and only if there exists no counter-example of property G.

Theorem 1 shows that the analysis of a global system property can be defined as the process of detecting counter-examples to the validity of the property by means of abduction. However, since global system properties can be seen as sentences universally quantified over time, the use of standard theorem proving techniques may become computationally expensive. To overcome this problem, we perform a reduction step, which simplifies the inference task by instantiating the system description and the system property to a time structure consisting of two arbitrary time points t_1 and t_2 such that $t_1 < t_2$. An abductive proof procedure can then be applied more efficiently to such a reduced (ground) system description. Theorem 2 shows that the reduction step is sound and complete. The proof is by induction on t_i and, therefore, only accounts for the analysis of global system properties such as the ones considered in this paper.

Theorem 2 [4, 14]: Let D be a system description and G a global system property. Let T be a time structure consisting of three time points: t_1 and t_2 such that $t_1 < t_2$, and an initial time point t_0. Let D_T be the ground system description, and let Gt_i be the ground property at time t_i $(0 \leq i \leq 2)$. Then $D \vdash G$ if and only if $D \vdash Gt_0$ and $D_T \cup Gt_1 \vdash Gt_2$.

Theorem 2 states that, in order to show that $D \vdash G$, it is sufficient to show that (i) the property is true at an initial time point (an initial system state) and (ii) given a ground system description D_T, if the property is true at an arbitrary time point t_i (a current system state) then it is also true at t_{i+1} (the next system state). This summarises important results on abduction. Section 4.1 will illustrate its use as part of our analysis–revision cycle.

3 Inductive learning

In this section, we introduce inductive learning as a revision technique, and describe the C-IL^{2}P hybrid learning system. Learning can be defined as the change of behaviour motivated by changes in the environment in order to perform better in different knowledge domains [15]. 'A computer program is said to learn from examples E, with respect to some class of tasks T and performance measure M, if its performance at tasks in T, as measured by M, improves with E' [3]. More recently, the importance of adding background knowledge in the form of additional information about the application domain at hand, in order to help the learning process, has been highlighted [16]. In inductive logic programming (ILP), for example, background knowledge is part of the definition of 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 background knowledge provides additional information about the domain' [17]. More formally, given background knowledge $\mathcal{B}$, positive examples e^+ and negative examples e^- of a concept, ILP learning is about searching for the most general hypothesis h such that: (a) $\mathcal{B} \cup h \vdash e^+$ and (b) $\mathcal{B} \cup h \nvdash e^-$. According to Occam's Razor, the most likely hypothesis is the simplest one that is consistent with all observations. As a result, the most general hypothesis h is assumed to be the most simple one (see [18] for a discussion on Occam's Razor in the context of machine learning).

Using the same example domain of Section 2, 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)` → `Red_Spots(X)`. Of course, an alternative hypothesis would be `Red_Spots(X)` → `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)` → `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. For instance, a negative example (e_1^- = `Measles(Peter), Red_Spots(Peter)`) would indicate that new hypotheses must not contain `Measles(Peter)` and `Red_Spots(Peter)` together. 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.

When an incorrect (partial) system description is part of background knowledge, one needs to relax the above restriction of standard ILP, that hypotheses be consistent with background knowledge. Clearly, the use of inductive learning for revision requires the capability to exploit background knowledge (i.e. an evolving specification), even when parts of it are incorrect. The following definition of learning captures this idea.

Definition 2: A computer program is said to learn from positive examples e^+, negative examples e^- and background knowledge $\mathcal{B}$, with respect to some class of tasks T and performance measure M, if it changes $\mathcal{B}$ into $\mathcal{B}'$ such that (a) $\mathcal{B}' \vdash e^+$, (b) $\mathcal{B}' \nvdash e^-$ and (c) the performance of $\mathcal{B}'$ at tasks in T is, on average, greater than the performance of $\mathcal{B}$ at tasks in T, as measured by M.

As before, Occam's Razor should be applied when it comes to deciding how $\mathcal{B}$ ought to be changed into $\mathcal{B}'$. In the case of deterministic systems, the (computationally expensive) task of showing that $\mathcal{B}' \nvdash e_i^-$, for a particular negative example e_i^-, can be reduced to the task of showing that $\mathcal{B}' \vdash e_i^+$, where e_i^+ is obtained from e_i^- by changing its next state (recall that we see training examples e_i^+ and e_i^- as tuples composed of a current system state, an event and a next state). In this case, condition (b) of Definition 2 can be dropped. In the sequel, we will refer to negative examples simply as counter-examples.

Different inductive learning techniques can be applied to evolve $\mathcal{B}$ based on different forms of representation, such as symbolic rules [19], neural networks [20] and hybrid systems [21]. Symbolic learning, such as ILP learning technique [22], has the advantage of using existing background knowledge to reduce the search space during the learning process, making it more efficient. As discussed above, the basic assumption is that the given background knowledge is correct, and that the learned concept should simply be added to such knowledge. In contrast, neural networks perform inductive learning using statistical, rather than declarative, definitions of data dependency, encoded in their weights. This approach has enabled neural networks to outperform symbolic learning systems in different application domains, especially when the set of examples is noisy (a set of examples is noisy when some examples are not 100% correct.), despite the fact that no background knowledge is used [23]. In response to these results, there is a growing interest in combining symbolic and neural learning systems [24]. Such hybrid models of inductive learning try to exploit certain advantages of both the symbolist and connectionist paradigms of artificial intelligence. 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 (which changes such weights based on the training examples) can not only expand the background knowledge but also revise it when necessary. The Connectionist Inductive Learning and Logic Programming System (C-IL^{2}P) [5] is one such hybrid learning system.

C-IL^{2}P neural networks integrate 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 Fig. 3. The insertion module consists of a translation algorithm that takes background knowledge, described as a general logic program [25], and generates the initial architecture and set of weights of a (single-hidden layer) feedforward neural network (Fig. 3(1)). A general logic program is a finite set of rules of the form $L_1 \wedge \cdots \wedge L_n \rightarrow A$, where A is called an atom and L_i ($1 \leq i \leq n$) is called a literal (an atom or the negation of an atom). The revision module revises the background knowledge by training the neural network with examples (Fig. 3(2)) using standard backpropagation [6]. The examples are presented to the network as pairs of input and output sequences, and the revision of background knowledge, which defined the network's initial set of weights, is done by the iterative change of the weights until the network adapts to the examples. In the case of backpropagation, weight changes are done by applying gradient descent on an error surface, which is obtained by comparing the network's output sequences with target output sequences. The objective of gradient descent is to minimise such an error by changing the network's 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 general logic program (Fig. 3(3)). The set of extracted rules are in general more comprehensible than the trained network, thus facilitating the analysis of the knowledge refinement process by a domain expert.

The translation from logic programs $\mathcal{P}$ to neural networks $\mathcal{N}$ is done as follows. Each rule (r_i) of $\mathcal{P}$ is mapped from the input layer to the output layer of $\mathcal{N}$ through one neuron (N_i) in the single hidden layer of $\mathcal{N}$. Intuitively, the translation algorithm has to implement the

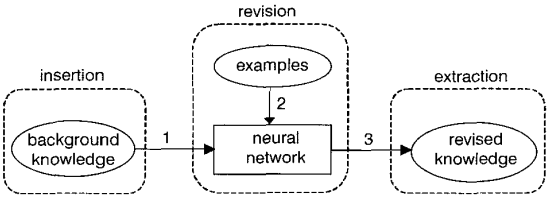

Fig. 3 *The C-IL^{2}P system*

following conditions: (C1) the input potential of a hidden neuron (N_i) can only exceed N_i's threshold (θ_i), activating N_i, when all the positive antecedents of r_i are assigned the truth-value `true` while all the negative antecedents of r_i are assigned `false`; and (C2) the input potential of an output neuron (A) can only exceed A's threshold (θ_A), activating A, when at least one hidden neuron N_i that is connected to A is activated.

Example 1: Consider the logic program $\mathcal{P}=\{r_1 : B \wedge C \wedge \neg D \rightarrow A;\ r_2 : E \wedge F \rightarrow A;\ r_3 : \rightarrow B\}$. The translation algorithm derives the network $\mathcal{N}$ of Fig. 4, in which hidden neurons N_1, N_2 and N_3 encode rules r_1, r_2 and r_3, respectively, and defines the values of the weights (W) and thresholds (θ) in such a way that conditions (C1) and (C2) above are satisfied. Note that, if $\mathcal{N}$ ought to be fully connected, any other link (not shown in Fig. 4) should receive weight zero initially.

Note that, in Example 1, each input and output neuron of $\mathcal{N}$ is associated with an atom of $\mathcal{P}$. As a result, each input and output vector of $\mathcal{N}$ can be associated with an interpretation for $\mathcal{P}$. An interpretation is a function mapping atoms in $\mathcal{P}$ to {`true, false`}. Note also that each hidden neuron N_i corresponds to a rule r_i of $\mathcal{P}$. A model for $\mathcal{P}$ is an interpretation that maps $\mathcal{P}$ `to true`. For example, the models for $\mathcal{P}=\{\rightarrow a;\ a \wedge b \rightarrow c\}$ are M_1: {`a=true, b=false, c=false`}, M_2: {`a=true, b=false, c=true`} and M_3: {`a=true, b=true, c=true`}. To compute the models [26] of $\mathcal{P}$, output neurons should feed their corresponding input neurons (e.g. B in Fig. 4) such that $\mathcal{N}$ is used to iterate $T_\mathcal{P}$, the fixpoint operator of $\mathcal{P}$, defined as follows.

Definition 3: Let $\mathcal{P}$ be a general logic program. Let $\mathcal{B}_\mathcal{P}$ denote the set of atoms occurring in $\mathcal{P}$, called the Herbrand base of $\mathcal{P}$. The mapping $T_\mathcal{P} : 2^{\mathcal{B}_\mathcal{P}} \rightarrow 2^{\mathcal{B}_\mathcal{P}}$ is defined as follows. Let I be an interpretation, then $T_\mathcal{P}(I) = \{A_0 \in \mathcal{B}_\mathcal{P} | A_1 \wedge \cdots \wedge A_n \wedge \neg A_{n+1} \wedge \cdots \wedge \neg A_m \rightarrow A_0$ is a rule in $\mathcal{P}$ and the atom A_i $(1 \leq i \leq n)$ is mapped to true by I, while the atom A_j $(n+1 \leq j \leq m)$ is mapped to false by $I\}$.

For example, if $\mathcal{P}=\{\rightarrow a;\ a \rightarrow b\}$ and we iterate $T_\mathcal{P}$ starting from $\{a=false,\ b=false\}$, we obtain $\{a=true,\ b=false\}$ and then $\{a=true,\ b=true\}$, which is the unique model for $\mathcal{P}$. The following theorem shows that the C-IL^{2}P network $\mathcal{N}$ obtained by the translation of a logic program $\mathcal{P}$ computes the fixpoint operator $T_\mathcal{P}$ of the program. This guarantees the correctness of the translation algorithm.

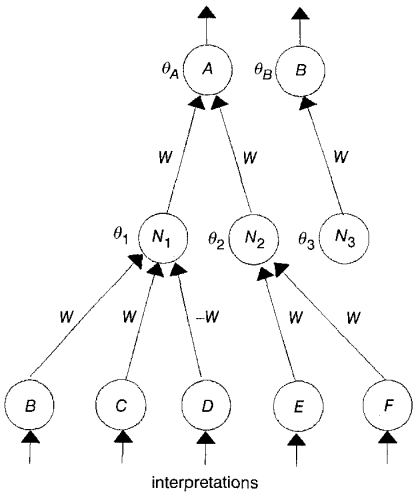

Fig. 4 *Sketch of a neural network for a logic program*

Theorem 3 [5]: For each general logic program $\mathcal{P}$, there exists a feedforward neural network $\mathcal{N}$ with exactly one hidden layer of semi-linear neurons such that $\mathcal{N}$ computes $T_\mathcal{P}$.

C-IL^{2}P networks typically use $f(x)=x$ as the activation function of input neurons (called linear neurons) and $h(x)=1/(1+e^{-x})$ as the activation function of hidden and output neurons, called semi-linear neurons. The activation of each neuron is obtained by applying its activation function on its input potential. The input potential of an input neuron is its associated interpretation (we use 1 to represent `true`, and 0 to represent `false`). The input potential of a hidden or output neuron is the weighted sum of the activation of its predecessor neurons, according to the network's weights, thresholds and input vector. The network's output vector is given by the activation of its output neurons. We say that a neuron is activated if its activation value is greater than a given (pre-defined) minimum activation $A_{\min} \in \Re$, e.g. $A_{\min}=0.5$.

Example 2: Consider the following interpretation for the program $\mathcal{P}$ of Example 1: {`B=true, C=true, D=false, E=false, F=false`}. Since input neurons are linear, the activation of input neurons B, C, D, E and F is 1, 1, 0, 0 and 0, respectively (Fig. 4). Then, the input potential I_{N_1} of hidden neuron N_1 is given by $W \cdot Act_B + W \cdot Act_C - W \cdot Act_D - \theta_1$, where Act_j denotes the activation of neuron j, and θ_1 is called the threshold of N_1. In general, $I_i = \sum_j (W_{ij} \cdot Act_j) - \theta_i$, where W_{ij} denotes the weight from neuron j to neuron i. The activation of hidden neuron N_1 is given by $h(I_{N_1})$. Assume $W=1.0$ and $\theta_1=1.5$. In this case, we obtain $I_{N_1}=0.5$ and $h(I_{N_1})=0.62$ as the activation of N_1. We can repeat this process to obtain the activation of hidden neuron N_2 and N_3 and of output neurons A and B. Taking $\theta_2=0.5$, we obtain $I_{N_2}=-0.5$ and $h(I_{N_2})=0.38$ as the activation of N_2. Finally, the input potential I_A of output neuron A is given by $W \cdot Act_{N_1} + W \cdot Act_{N_2} - \theta_A$. Taking $\theta_A=0.5$, we obtain $I_A=0.5$ and $h(I_A)=0.62$. As a result, if $A_{\min}=0.5$, we say that output neuron A is activated for input vector $\mathbf{i}=[1, 1, 0, 0, 0]$. In other words, A is true if B is true, C is true, D is false, E is false and F is false. Clearly, A must be activated since $B \wedge C \wedge \neg D \rightarrow A$ is a rule in $\mathcal{P}$. Similarly, θ_3 and θ_B must be such that output neuron B is activated for any input vector, since B is a fact in $\mathcal{P}$.

In the sequel, we use $\mathbf{o}=\mathcal{N}(\mathbf{i})$ to denote the output vector $\mathbf{o}$ obtained from network $\mathcal{N}$, given input vector $\mathbf{i}$. In this setting, learning is the process of changing the weights initially defined by $\mathcal{P}$, in order to adapt $\mathcal{N}$ to new examples given in the form of input vectors and their corresponding output vectors. For example, from Example 2, if a new training example stated that, as a matter of fact, A should be false when B is true, C is true, D is false, E is false and F is false, the weights of $\mathcal{N}$ would have to be changed in order to revise the rule $B \wedge C \wedge \neg D \rightarrow A$, originally given in $\mathcal{P}$. The interest on semi-linear (derivable) activation functions is that an efficient gradient descent learning algorithm, such as Backpropagation, can be applied directly onto networks that contain such neurons.

If the application at hand contains too many degrees of freedom and too few training examples, an inductive learning algorithm may end up simply memorising the examples. This behaviour is known as overfitting. The ultimate measure of success, therefore, should not be how well the inductive learner approximates the training examples, but how well it accounts for yet unseen examples, i.e. how well it generalises to new cases. In order to evaluate the generalisation performance of an inductive learning algorithm in a given application, the set of examples is commonly partitioned into a training set and a test set. The training set is used for learning a problem description, while the test set, which is not used for training, provides an estimate of the description's generalisation performance. This process of (randomly) partitioning the set of training examples can be repeated many times to provide an statistically valid estimate.

After learning and evaluation is performed, the final task of C-IL^{2}P is to perform knowledge extraction from the trained neural network $\mathcal{N}'$. Typically, while the original network $\mathcal{N}$ has a neat architecture, the trained network is allowed to be fully connected. For example, after learning, the network of Fig. 4 would have 21 different weights. Since we do not want to impose restrictions on the learning process itself, the task of knowledge extraction becomes challenging, especially in the case of large networks. Rule extraction from trained networks is an extensive research topic in its own right (see [10] 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 to be activated. 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 the network can be described by a correct and compact set of rules [11].

4 Evolving specifications

In this section, we describe how abductive reasoning and inductive learning can be combined 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. In what follows, we describe the combined abductive and inductive reasoning techniques in detail, and illustrate 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.

4.1 Abducing counter-examples

We combine in a single automated decision procedure the tasks of validating system descriptions with respect to system properties and of generating appropriate diagnostic information whenever a property is violated. This procedure uses abductive reasoning 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, entails the negation of P. In logic terms, as discussed in Section 2, our abductive procedure shows that $D \vdash P$ by failing to find a set Δ of abducibles, consistent with D, such that $D \cup \Delta \vdash \neg P$. If, on the other hand, the abductive procedure finds such a set 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). In practice, this is done more efficiently by applying abduction on a ground system description D, and by considering each property P as a number of integrity constraints of the form $c_1 \wedge c_2 \wedge \cdots \wedge c_n \rightarrow \bot$ on an arbitrary current state.

To illustrate the use of abduction for analysis, 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 partial (possibly incorrect) description D of our electric circuit includes the following rules r_1 to r_4, formalised using propositional logic programming [25] 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:

$$\neg\texttt{SwitchA-On} \wedge \neg\texttt{Light-On} \wedge \texttt{FlickA} \rightarrow \texttt{Light-On}' \quad (r_1)$$

$$\neg\texttt{SwitchB-On} \wedge \neg\texttt{Light-On} \wedge \texttt{FlickB} \rightarrow \texttt{Light-On}' \quad (r_2)$$

$$\neg\texttt{SwitchA-On} \wedge \neg\texttt{FlickA} \rightarrow \texttt{SwitchA-On}' \quad (r_3)$$

$$\neg\texttt{SwitchA-On} \wedge \neg\texttt{FlickB} \rightarrow \texttt{SwitchB-On}' \quad (r_4)$$

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'. These rules could be thought of as derived from the state transition diagrams of Fig. 5, where $X = \{A, B\}$ and FlickA and FlickB are the only two possible events of the system.

One of the system properties that we would like the above description D to satisfy (since the two switches are supposed to be connected in series) is

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

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

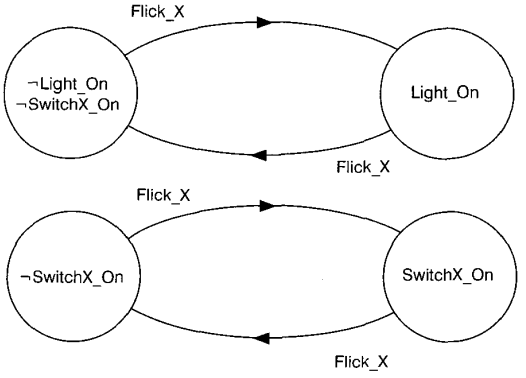

Fig. 5 *State transition diagrams for the light bulb example* $X = \{A, B\}$

current state and checks whether ¬P is true at an arbitrary next state. Starting from ¬P, 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. We refer to this as the no change assumption. For example, the rule r_5: ¬SwitchA-On → SwitchA-On′ would capture a no change assumption about the state of SwitchA. Rule r_5 would need to have lower priority than rule r_3, since when ¬Switch-On and FlickA are both true, we would like to apply r_3 and derive SwitchA-On, instead of r_5, to derive ¬SwitchA-On. Alternatively, rule r_5 could be seen as a domain property to be satisfied by the system description.

The negated property, instantiated at an arbitrary next state, is given by ¬P = ((¬SwitchA-On′ ∨ ¬SwitchB-On′) ∧ Light-On′), which can be re-written in disjunctive normal form as $\neg P = \neg P_1 \vee \neg P_2$, where:

$$\neg P_1 = \neg \text{SwitchA-On}' \wedge \text{Light-On}'$$

$$\neg P_2 = \neg \text{SwitchB-On}' \wedge \text{Light-On}'$$

To prove ¬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. To prove $\neg P_1$, the procedure has to prove both SwitchA-On′ and Light-On′. Consider the first condition. Since no rule in the description D defines ¬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, ¬FlickA$\}$ and tries to prove, taking into account the set Δ_0, the second condition of $\neg P_1$, which is Light-On′. To prove Light-On′, reasoning backwards, we can use either 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 consistent 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 $\Delta_1 = \{\neg$SwitchB-On, ¬Light-On, FlickB$\}$, which, this time round, are all consistent with the assumptions in Δ_0 and, therefore, accepted. As a result, $\Delta = \Delta_0 \cup \Delta_1$ is a possible explanation for $\neg P_1$, since Δ_0 is a possible explanation for ¬SwitchA-On′ and Δ_1 is a possible explanation for Light-On′. Since all conditions in $\neg P_1$ have been considered and $\neg P = \neg P_1 \vee \neg P_2$, the abductive procedure can stop. Otherwise, it would repeat the above process for $\neg P_2$ and, in failing to find any explanation for its violation, it would conclude that P is satisfied by D (according to Theorem 1).

$\Delta = \Delta_0 \cup \Delta_1$ contains a current system state and events that, according to the partial description D, would lead the system into a new state in which ¬SwitchA-On and Light-On are satisfied, according to $\neg P_1 = \{\neg$SwitchA-On′, Light-On′$\}$. This is so because $D \cup \Delta \vdash \neg P_1$. As a consequence, $D \cup \Delta \vdash \neg P$ and $D \nvdash P$. In addition, according to Theorem 2, only two time points t_i and t_{i+1} need to be analysed when it comes to finding counter-examples to the validity of P. We capture all the diagnostic information obtained from abductive analysis in the set $\mathbf{\Delta} = \Delta \cup \neg P_1$, such that

$\mathbf{\Delta} = \{\neg$SwitchA-On, ¬SwitchB-On, ¬Light-On, ¬FlickA, FlickB, ¬SwitchA-On′, Light-On′$\}$

which states 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 Light is on but SwitchA is not on, thus violating the system property. In the general case, $\mathbf{\Delta} = \{\Delta, \neg P_i\}$, where Δ is the first nonempty set of abducibles $\Delta_0 \cup \ldots \cup \Delta_n$ found by the above abductive proof procedure as explanations for atoms $A_0, \ldots, A_m, \neg A_{m+1}, \ldots, \neg A_n$ in ¬P such that $\neg P = \neg P_1 \vee \cdots \vee \neg P_i \vee \cdots \vee \neg P_j$ and $\neg P_i = A_0 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n$.

4.2 Generating training examples

A crucial aspect of the analysis–revision cycle is how to use the diagnostic information $\mathbf{\Delta}$, identified by the analysis phase, to generate system behaviours Δ' that should, instead, be covered by the system description (i.e. training examples). $\mathbf{\Delta}$, as a counter-example, informs us that some state transitions are not correct. Considering that a state transition is defined by a current state, an event and a next 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 next 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 next 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 the property P.

In what follows, we use the term entry configuration to refer to the current state and the event of a given state transition, and exit configuration to refer to the next state of the transition. The diagnostic information $\mathbf{\Delta} = \{\Delta, \neg P_i\}$, generated by our abductive procedure, informs us that a given entry configuration Δ should not produce a given exit configuration $\neg P_i$. In the electric circuit example, $\mathbf{\Delta} = \{\neg$SwitchA_On, ¬SwitchB_On, ¬Light_On, ¬FlickA, FlickB$\}$ should not produce $\neg P_i = \{\neg$SwitchA-On′, Light-On′$\}$. In other words, $\{\Delta, \neg P_i\}$, represents an incorrect state transition, as shown in Fig. 6.

A way of solving this problem is to make sure that entry configuration Δ produces an exit configuration Φ, different from $\neg P_i$ (assuming, as before, that the system description is deterministic), such that Φ is consistent with P. The set $\{\Delta, \Phi\}$ would be one of our training examples (or correct state transition).

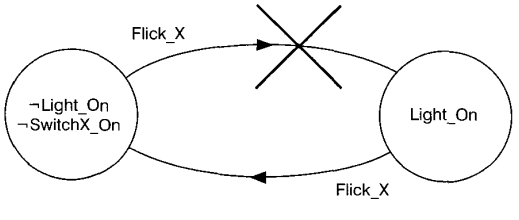

Fig. 6 *An incorrect state transition (counter-example) for the light bulb example*
X = B

Definition 4: Let D be a (partial) system description and P a system property. Let Δ be an entry configuration and $\neg P_i$ and Φ two exit configurations. Let $D \cup \Delta \vdash \neg P$. The set $\Delta' = \{\Delta, \Phi\}$ is called a training example if and only if (i) there exists an atom o_j in Φ such that $o_j \notin \neg P_i$ and (ii) $P \cup \Phi$ is consistent.

Returning to the electric circuit example, given $\neg P_1$, we could flip ¬SwitchA-On′ to SwitchA-On′, obtaining {SwitchA-On′, Light-On′} or we could flip Light-On′ to ¬Light-On′ obtaining {¬SwitchA-On′, ¬Light-On′}. Now, if we assume that SwitchB-On′ is false, we obtain $\Phi_1 = \{$SwitchA-On′, ¬SwitchB-On′, Light-On′$\}$ which is inconsistent with P. $\Phi_2 =$ {SwitchA-On′, SwitchB-On′, Light-On′}, however, is such that $\{\Delta, \Phi_2\}$ is a training example, according to definition 4. Similarly, $\Phi_3 = \{$¬SwitchA-On′, SwitchB-On′, ¬Light-On′$\}$ is such that $\{\Delta, \Phi_3\}$ is a training example. Let us take $\Delta' = \{\Delta, \Phi_3\}$, as shown in Fig. 7:

$$\Delta' = \{\neg\text{SwitchA\_On}, \neg\text{SwitchB\_On}, \neg\text{Light\_On}, \neg\text{FlickA}, \text{FlickB}, \neg\text{SwitchA\_On}', \text{SwitchB\_On}' \neg\text{Light\_On}'\}$$

Although there are $2^k - 1$ potential training examples to be checked for consistency, where k is the number of atoms in the partial description D, in practice, we flip one atom at a time, as done above, and stop when the first training example is derived.

It is interesting to note that the generation of training examples from diagnostic information is based on attempts to find alternative state transitions to an incorrect state transition. As a result, any alternative will be at least as good as the current transition. Instead of concentrating efforts on how to choose the best alternative, which is a highly domain dependent task, in the analysis–revision framework we simply select one and iterate. The analysis–revision cycle is then expected to eliminate all incorrect transitions eventually. This approach could also be appealing from a machine learning perspective, especially in data starved domains.

4.3 Inducing a new specification

So far, we have seen that a desirable property P of a system may be checked against a partial 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 transitions in the set of training examples.

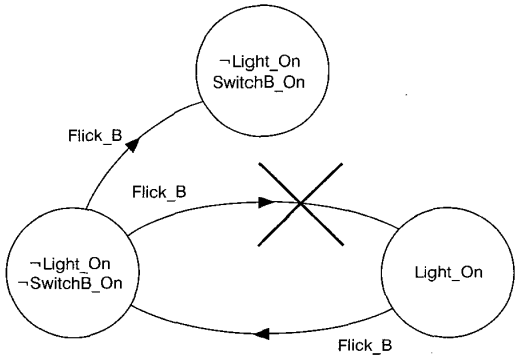

Fig. 7 *An alternative state transition (training example) for the light bulb example*

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 description D′ such that $D' \vdash P$, in which case Δ' 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) [5, 11] to induce D′. C-IL^{2}P is a hybrid machine learning system that uses Backpropagation [20], the neural learning algorithm most successfully applied in industry, as the underlying learning technique. As presented in Section 3, C-IL^{2}P integrates inductive learning from examples and background knowledge with logic programming, using three main modules: knowledge insertion, revision and extraction. In what follows, we illustrate a run of C-IL^{2}P using the electric circuit example above. A discussion about the choice of C-IL^{2}P, in comparison with other machine learning techniques, is given in Section 5.2.

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 $r_1 =$ ¬SwitchA_On ∧ ¬Light_On ∧ Flick_A → 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.

Fig. 8 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 $r_2 =$ ¬SwitchB_On ∧ ¬Light_On ∧ Flick_B → Light_On′). In this initial network, positive weights (indicated in Fig. 8 by solid lines) are used to represent positive concepts (such as Flick_A in r_1) and negative weights (indicated in Fig. 8 by dotted lines) are used to represent negative concepts (such as ¬SwitchA_On and ¬Light_On in r_1). As discussed in Section 3, C-IL^{2}P's Translation Algorithm is responsible for defining these weights such that the network's output neurons perform an or of the concepts represented in the hidden neurons that are connected to them, and the network's hidden neurons perform an and of the concepts represented in the input neurons that are connected to them. For example, from Fig. 8, 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) [5].

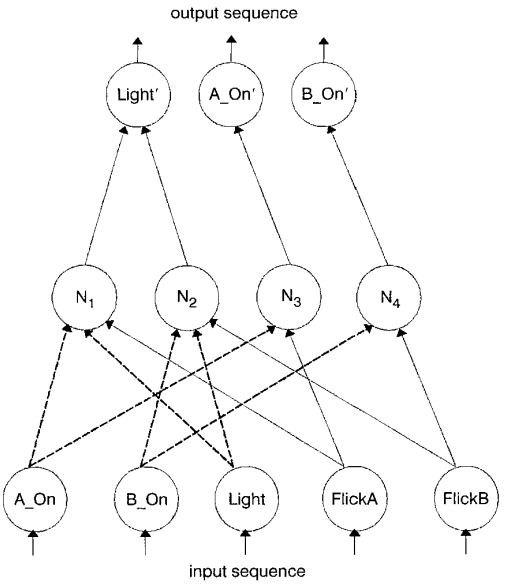

Fig. 8 *The network $\mathcal{N}$ obtained from description D*

Module 2 of C-IL^{2}P is about adapting the weights of $\mathcal{N}$, which encodes a (possibly incorrect) background knowledge, with training examples. Each training example is a pair containing an input sequence **i** and a target output sequence **t** for $\mathcal{N}$ (see Fig. 8). The network is fully connected so that, in addition to changes to the background knowledge, completely new rules can be learned. The input sequence **i** is presented to the network and its corresponding output sequence $\mathbf{o} = \mathcal{N}(\mathbf{i})$ is calculated. If such an output is different from the target **t**, an error $\mathbf{E}_W$ that depends on the current set of weights W of the network is calculated. Typically, $\mathbf{E}_W = 1/2 \sum_i (\mathbf{t}_i - \mathbf{o}_i)^2$. Finally, in the case of backpropagation, a gradient descent method is applied, guiding iterative changes to W so that $\mathbf{E}_W$ is minimised [6]. In this process, the network's output sequence **o** is expected to approach the target sequence **t**. When this happens for all training examples, i.e. when $\mathbf{E}_W$ is close to zero, the learning process is concluded.

From Section 4.2, one of our training examples is $\Delta' = \{\neg$`SwitchA_On`, $\neg$`SwitchB_On`, $\neg$`Light_On`, $\neg$`FlickA`, `Flick_B`, $\neg$`SwitchA_On'`, `SwitchB_On'`, $\neg$`Light_On'`$\}$. As a result, given the network $\mathcal{N}$ of Fig. 8 with input neurons `[A_On, B_On, Light, FlickA, FlickB]` and output neurons `[Light', A_On', B_On']` in this order, the pair `i = [-1, -1, -1, -1, 1]`, `t = [-1, -1, 1]` is a training example for $\mathcal{N}$, where 1 is used to indicate true and -1 is used to indicate false. The choice of -1 instead of 0 to represent false will lead to faster convergence of the learning process in almost all cases. The reason for this is that the update of a weight connected to an input variable will be zero when the corresponding variable is zero in the training pattern [27].

Module 3 of C-IL^{2}P is responsible for extracting the knowledge encoded in the trained network. It uses a rule extraction algorithm to do so [11]. Rule extraction is commonly accepted as the way to provide neural networks with explanation capability, by mapping each network into a set of rules. Some extraction algorithms treat the network as a black box and try to capture the function computed by it, by querying the network with certain input sequences and obtaining their corresponding output sequences. Other algorithms decompose the network into subnetworks, and look at the set of weights to try and capture general rules about the network, sometimes by pruning and clustering weights. Typically, methods that treat the network as a black box produce high quality rules, but are less efficient than methods that decompose the network. On the other hand, methods that decompose the network may produce low quality rules, for example, as a result of pruning the network's weights. In other words, there is a trade-off between the quality of the extracted set of rules, measured in terms of correctness, readability, completeness, etc., and the complexity of the rule extraction algorithm employed. C-IL^{2}P deals with this trade-off by guaranteeing correctness of the extraction algorithm, and by using information from the weights to reduce the number of input sequences queried in average. It does so by defining a partial ordering on the set of input sequences that guides the querying process. For example, in the network $\mathcal{N}$ of Fig. 8, after example (**i**, **t**) has been trained, if one queries input sequence $\mathbf{i} = [1, -1, -1, -1, 1]$, one obtains output sequence $\mathbf{o} = [1, 1, 1]$. This is equivalent to three logic programming rules (e_1, e_2 and e_3, below), all having the same antecedent (given by **i**), and consequents given by the output neurons activated by **i**, in this case Light', A_On' and B_On'. In the general case, given input neurons $[a_1, a_2, \ldots, a_n]$ and output neurons $[b_1, b_2, \ldots, b_m]$, in this order, and an input sequence $\mathbf{i} = [1, -1, \ldots, -1]$, and its corresponding output sequence $\mathbf{o} = [1, -1, \ldots, 1]$, k rules (all with the same antecedent $a_1 \wedge \neg a_2 \wedge \cdots \wedge \neg a_n$) will be derived from (**i**, **o**), where k is the number of output neurons activated by input sequence **i** (shown as 1 in output sequence **o**). For example, $a_1 \wedge \neg a_2 \wedge \cdots \wedge \neg a_n \rightarrow b_1$ is one such derived rule:

$$\text{SwitchA\_On} \wedge \neg\text{SwitchB\_On} \wedge \neg\text{Light\_On} \wedge \neg\text{Flick\_A} \wedge \text{Flick\_B} \rightarrow \text{Light\_On}' \quad (e_1)$$

$$\text{SwitchA\_On} \wedge \neg\text{SwitchB\_On} \wedge \neg\text{Light\_On} \wedge \neg\text{Flick\_A} \wedge \text{Flick\_B} \rightarrow \text{SwitchA\_On}' \quad (e_2)$$

$$\text{SwitchA\_On} \wedge \neg\text{SwitchB\_On} \wedge \neg\text{Light\_On} \wedge \neg\text{Flick\_A} \wedge \text{Flick\_B} \rightarrow \text{SwitchB\_On}' \quad (e_3)$$

Now, as a matter of fact, input sequence $\mathbf{i} = [1, -1, -1, 1, 1]$, in which Flick_A is true, produces output sequence $\mathbf{o} = [1, -1, 1]$ in the trained $\mathcal{N}$. This is the kind of input sequence that normally would not need to be queried in C-IL^{2}P's extraction algorithm, because the same information can be obtained more easily by inspecting the weights of the trained network [11]. In the worst case, 2^n input sequences would need to be queried, where n is the number of the network's input neurons. In practice, less than 10% of them are queried in general, and the extraction algorithm can be halted if a given desired degree of accuracy is achieved in the extracted rule set. Returning to the electric circuit example, C-IL^{2}P extraction has produced the following additional rules:

$$\neg\text{SwitchA\_On} \wedge \neg\text{Light\_On} \wedge \text{FlickA} \rightarrow \text{Light\_On}' \quad (e_4)$$

$$\text{SwitchA\_On} \wedge \neg\text{SwitchB\_On} \wedge \neg\text{Light} - \text{On} \wedge \text{Flick\_B} \rightarrow \text{Light\_On}' \quad (e_5)$$

$$\neg\text{SwitchA\_On} \wedge \text{FlickA} \rightarrow \text{SwitchA\_On}' \quad (e_6)$$

$$\neg\text{SwitchB\_On} \wedge \text{FlickB} \rightarrow \text{SwitchB\_On}' \quad (e_7)$$

Note that e_5 subsumes e_1, and e_7 subsumes e_3. As a result, a simplified final set of extracted rules for the trained $\mathcal{N}$ contains $\{e_2, e_4, e_5, e_6, e_7\}$. It is interesting noting that e_5 is a specialisation of background knowledge rule r_2.

Clearly, rule r_2 was under-specifying the system and, in fact, causing a property violation. The suggestion of C-IL^{2}P 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 e_4 (in which `SwitchB_On` would be added to it) would require another run of the analysis–revision cycle, with the derivation of a new training example to cater for it.

The revision of D into $D' = \{e_2, e_4, e_5, e_6, e_7\}$ 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.

5 Case study

We have applied the analysis–revision cycle to evolve a specification of an automobile cruise control system [28]. Different specifications of such a system have been presented in the literature [29]. 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. Fig. 9*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 Fig. 9*a* implicitly assumes that, whenever the conditions of the events are not satisfied, the system will stay in its current state. As a result, when the system is in state cruise and the brake is engaged, but the automobile is going too fast, the system remains in state cruise, therefore violating the above safety property. Such an incorrect transition is depicted in Fig. 9*b* as the diagnostic information Δ, derived by abduction.

The process of deriving training examples Δ' from Δ can be seen as simply changing the state to which an incorrect transition points. In this case study, as we will see in the sequel, the incorrect transition Δ is turned into transition Δ', as illustrated in Fig. 9*c*, so that when the system is in state cruise and the brake is engaged, but the automobile is going too fast, 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 Fig. 9*d* (compare with Fig. 9*a*).

In what follows, we will explain in detail how the analysis–revision cycle has been used to achieve the above. We use a logic programming implementation of the original cruise control system formalisation given in [4]. Each state transition is formalised as follows:

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

where s_c and s_n represent, respectively, the current and next state of the transition, e denotes the event transition, and $c_1, \ldots, c_n$ 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 $s_c \wedge c_1 \wedge \cdots \wedge c_n \wedge e \rightarrow \neg s'_c$ to express that, when the transition occurs, its current state is no longer true. In addition, each event has rules to express the effect on its associated variable. For example, for an event `brake`, the rule $\neg$`brake` $\wedge$ `happens(brake)` $\rightarrow$ `brake`$'$ must be introduced. Domain

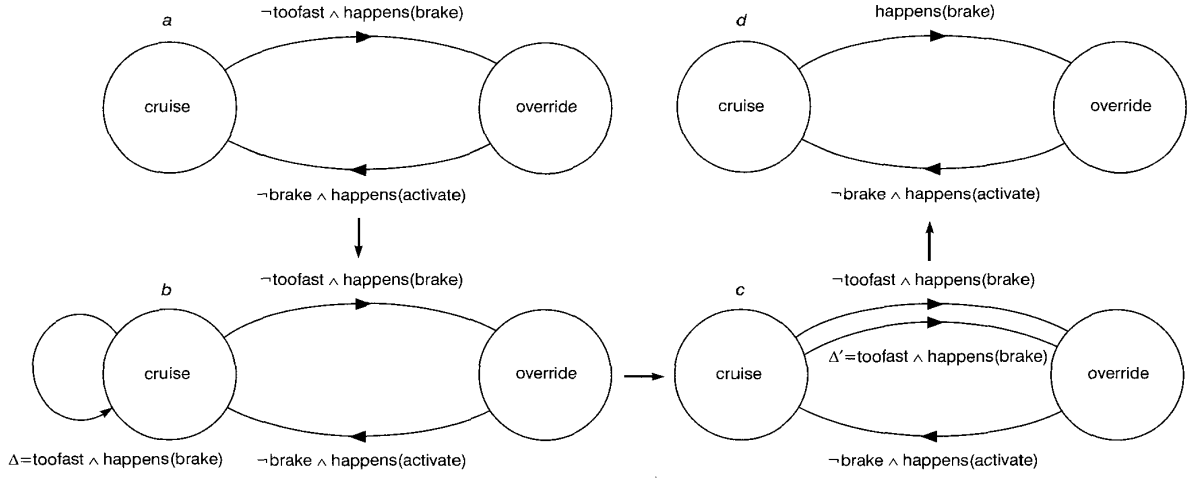

Fig. 9 *Analysis and revision of state transitions*

properties have also been considered, such as 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. Rule r_3 defines fully the state transition from cruise to override, which is only partially illustrated in the diagrams of Fig. 9:

$$\text{override} \wedge \neg\text{brake} \wedge \neg\text{activate} \wedge \text{happens(activate)} \rightarrow \text{cruise}' \quad (r_1)$$

$$\neg\text{brake} \wedge \text{happens(brake)} \rightarrow \text{brake}' \quad (r_2)$$

$$\text{cruise} \wedge \text{ignited} \wedge \text{running} \wedge \neg\text{toofast} \wedge \neg\text{brake} \wedge \text{happens(brake)} \rightarrow \text{override}' \quad (r_3)$$

$$\neg\text{activate} \wedge \text{happens(activate)} \rightarrow \text{activate}' \quad (r_4)$$

The set of properties that the system 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 '$\leftrightarrow$' means 'if and only if':

$$\text{off} \mid \text{inactive} \mid \text{cruise} \mid \text{override} \quad (P_1)$$

$$\text{off} \leftrightarrow \neg\text{ignited} \quad (P_2)$$

$$\text{inactive} \rightarrow \text{ignited} \wedge (\neg\text{running} \vee \neg\text{activated}) \quad (P_3)$$

$$\text{cruise} \rightarrow \text{ignited} \wedge \text{running} \wedge \neg\text{brake} \quad (P_4)$$

$$\text{override} \rightarrow \text{ignited} \wedge \text{running} \quad (P_5)$$

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 $\bot$). Similarly, if $a \rightarrow b$ were a desirable property, having a and $\neg b$ should imply $\bot$, if $a \rightarrow b \vee c$ were a desirable property, having a, $\neg b$ and $\neg c$ should imply $\bot$, and so on. For example, the conversion of property P_4 gives three integrity constraints: $ic_{4.1} = \{\texttt{cruise} \wedge \neg\texttt{ignited} \rightarrow \bot\}$, $ic_{4.2} = \{\texttt{cruise} \wedge \neg\texttt{running} \rightarrow \bot\}$ and $ic_{4.3} = \{\texttt{cruise} \wedge \texttt{brake} \rightarrow \bot\}$. The conversion of P_5 derives two integrity constraints: $ic_{5.1} = \{\texttt{override} \wedge \neg\texttt{ignited} \rightarrow \bot$, and $ic_{5.2} = \{\texttt{override} \wedge \neg\texttt{running} \rightarrow \bot\}$.

5.1 A run of the analysis–revision cycle

Let us now illustrate a run of the analysis–revision cycle, when checking whether property P_4 is verified. As discussed in Section 4.1, 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 $\texttt{cruise}' \wedge \texttt{brake}'$. To prove brake', reasoning backwards, it considers rule r_2 above, and starts constructing a temporary set of assumptions $\Delta_0 = \{\neg\texttt{brake}, \texttt{happens(brake)}\}$. It then tries to prove cruise'. One possibility is to use the no change 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, ¬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. 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:

$$\Delta = \{\text{cruise, ignited, running, toofast, } \neg\text{brake, happens(brake), cruise}', \text{brake}'\}$$

A number of training examples (Δ') can be obtained from Δ. If, for instance, we make cruise' false, we know from property P_1 that one of override', inactive', or off' has to be made true. If we choose override', from property P_5, we must also have ignited and running. If we choose inactive', from property P_3, we must have either ignited and ¬running or ignited and ¬activated. Finally, if we choose off', from property P_2, we must have ¬ignited' as well. Therefore, there are four possible new training examples:

$$\Delta'_1 = \{\text{cruise, ignited, running, toofast, } \neg\text{brake, happens(brake), override}', \text{ignited}', \text{running}', \text{brake}'\}$$

$$\Delta'_2 = \{\text{cruise, ignited, running, toofast, } \neg\text{brake, happens(brake), inactive}', \text{ignited}', \neg\text{running}', \text{brake}'\}$$

$$\Delta'_3 = \{\text{cruise, ignited, running, toofast, } \neg\text{brake, happens(brake), inactive}', \text{ignited}', \neg\text{activated}', \text{brake}'\}$$

$$\Delta'_4 = \{\text{cruise, ignited, running, toofast, } \neg\text{brake, happens(brake), off}', \neg\text{ignited}', \text{brake}'\}$$

At this point, we could recourse to the knowledge of a domain expert to choose between $\Delta'_1, \ldots, \Delta'_4$. Alternatively, we could make use of some sophisticated heuristics. For example, we could try to quantify the potential for future property violation of each training example. For instance, although neither Δ'_1 nor Δ'_2 is inconsistent with $P_1, \ldots, P_5$, in the case of Δ'_1, none of the constraints regarding override' (see property P_5) could possibly be violated, since Δ'_1 states that both ignited' and running' must be true. On the other hand, one of the constraints regarding inactive' (see property P_3) could be violated by Δ'_2 because, although running' is true in Δ'_2, activated' is undefined. If we assumed, thus, that activated' were true in Δ'_2, then inactive $\rightarrow$ (¬running $\vee$ ¬activate) would be violated. In this case, we would say that Δ'_1 is preferred over Δ'_2 according to their potential for property violation. The definition of a metric to guide the above choice of mutually exclusive training examples is work in progress. In this paper, as discussed in Section 4.2, we simply select the first alternative state transition obtained that is consistent with the system properties.

Taking Δ'_1 as our training example Δ' (Fig. 9*c*), we are now in a position to apply C-IL^{2}P in order to (a) translate the system description into the initial architecture of a neural network *N*, (b) train *N* with examples and (c) extract a revised description from the trained network:

$$\Delta' = \{\text{cruise, ignited, running, toofast, } \neg\text{brake, happens(brake), override}', \text{ignited}', \text{running}', \text{brake}'\}$$

Fig. 10 shows a small part of the network, in which rules r_2 and r_3, as well as the no change assumption about cruise, are represented. The no change assumption is encoded in the network with the use of r_{def} rules and a priority relation [30], as follows. In the case of cruise,

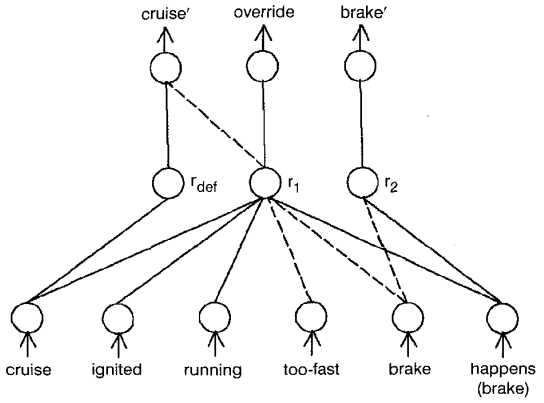

Fig. 10 *Part of the network for the cruise control system*

the no change assumption states that, by definition, r_{def} : cruise $\rightarrow$ cruise', unless any other rule r_i in the system description D supersedes r_{def}. Rule r_3, for example, supersedes r_{def} because, whenever r_3 is applicable (i.e. whenever the antecedent of r_3 holds), the conclusion of r_{def} must not hold. In this case, we say that r_3 has priority over r_{def}, and write $r_3 \succ r_{def}$. Priority relations can be implemented in a neural network with the use of negative weights connecting hidden neurons to output neurons (see [30]). $r_3 \succ r_{def}$, for example, is implemented in the network of Fig. 10 by connecting hidden neuron r_3 to output neuron cruise', and making sure that whenever r_3 is activated, cruise' is deactivated (i.e. the conclusion of rule r_{def} is blocked) by means of a negative weight (represented by a dotted line in the figure). Summarising, a no change assumption can be represented in a neural network by adding a rule of the form $a \rightarrow a'$ with lower priority than that of the rules already in D. In this case, in Δ'_1, the value of toofast' would be true because toofast is true; in Δ'_3, the value of running' would be true because running is true, and so on.

The network N was trained on Δ' using Backpropagation. After that, the C-IL^{2}P rule extraction algorithm produced the following rule:

$$r'_3 = \text{cruise} \wedge \text{ignited} \wedge \text{running} \wedge \neg\text{brake} \wedge \text{happens(brake)} \rightarrow \text{override}'$$

The rule extraction indicates that C-IL^{2}P 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 above new rule r'_3, as anticipated in the state transition of Fig. 9*d*.

5.2 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 [31]. The revision phase of our analysis–revision cycle uses the modules of knowledge insertion, revision and extraction of the C-IL^{2}P system, which was implemented in C.

The choice of the most appropriate learning technique for the task of revision is an important one. Firstly, it is desirable that a push-button technique is used, i.e. a technique that does not require the user to have specific knowledge about inductive learning. 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 incorrect, since we work with partial specifications in an analysis–revision evolving cycle. As a result, traditional techniques of inductive learning, which perform training from examples only, do not seem to be adequate. This includes Case Based Reasoning [3] and most models of artificial neural networks [20]. We are left with (i) inductive logic programming (ILP) techniques, (ii) hybrid (neural and symbolic) systems and (iii) explanation based learning (EBL) algorithms (see [3]). Among these, hybrid systems seem to be more appropriate as far as dealing with incorrect background knowledge and theory refinement is concerned [5, 23, 32]. 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 [33], but are not appropriate in the presence of incorrect domain knowledge [3], 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.

The above case study has indicated that the choice of training examples is important, in that the better the choice, the faster the convergence of the system to a specification that does not violate any system property. Although the generation of training examples is guided by Δ, the efficiency of our analysis–revision cycle could be improved with the choice of good training examples. This may be domain dependent and, indeed, require the help of an expert. Still, we have seen that heuristics could be applied to decide between mutually exclusive training examples. Thus, an important extension of this work, currently being investigated, is the use of different heuristics to help in the generation of good training examples.

6 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 [34] 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 [7, 35]. 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 for analysis, the work of van Lamsweerde *et al.* [36] is particularly relevant. It 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 [36], in that it uses grounded goals to make the procedure decidable, and integrity constraints to validate the properties efficiently [4]. Other examples of the application of abduction to software engineering tasks can be found in [37].

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 a violation of a property is an example of inconsistency 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 [38]. 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 [39]) perform revision by adding or deleting existing and derivable requirements. In contrast, with the use of inductive learning, the revision process presented enables the evolution of specifications through the acquisition of genuinely new requirements from scenario generalisation.

Of the inductive learning-based approaches for revision, 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. The approach uses symbolic inductive learning with no background knowledge. The use of learning is aimed at the elicitation of new requirements from scenarios provided by the user. Differently, in this paper, 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 incorrect partial specification. Learning techniques have also been used in other software engineering applications, such as the inference of process models from process traces [41] and the validation of an air traffic control requirements model [42].

7 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 techniques for the development of logic-based methods of requirements specifications. It also contributes to the management of inconsistency in requirements specifications [43–45]. Following the idea that inconsistency should be seen as a 'trigger for actions' [46], this paper shows that learning could be one of these actions [47].

An extension of this work would be to investigate the use of other techniques of machine learning for revising requirements specifications. These include extensions of Inductive Logic Programming, Knowledge-based Artificial Neural Networks (KBANN) [32] and Explanation-based Neural Networks (EBNN) [48] and their hybrids, e.g. [49]. Experiments on a number of real-world problems 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 operational scenarios would be another interesting direction to pursue, as scenarios can be seen as examples of how system components, the environment and the users should interact to provide system level functionality [50].

Although abduction and induction can be naturally integrated in the analysis–revision cycle, the use of Backpropagation for inductive learning does not necessarily comply with the concept of minimal change (very common in the area of Belief Revision [8]). In other words, when changing a description D into D′ to accommodate a new property P, Backpropagation does not necessarily change as few rules as possible in D. Depending on the initial description D, the use of minimal change could clearly provide a faster convergence of the analysis–revision cycle to a desirable specification D′. A new incremental learning algorithm for the neural networks that satisfies the concept of minimal change is presented in [24, Chapter 7]. An empirical comparison between such an algorithm and Backpropagation in what regards the rate of convergence of the analysis–revision cycle would be highly desirable.

Finally, during analysis, the abductive derivation of diagnostic information assumes that system properties are correct. However, while specifications evolve, system properties themselves could need to be revised. If, for example, the diagnostic information is not validated by stakeholders as a counter-example to a property, this could indicate that the property itself is wrong and, therefore, that the analysis–revision cycle needs to re-start using a new set of system properties. Therefore, another extension of this work would be to investigate the use of the analysis–revision cycle in the presence of potentially incorrect system properties.

8 References

1 D'AVILA GARCEZ, A.S., RUSSO, A., NUSEIBEH, B., and KRAMER, J.: 'An analysis-revision cycle to evolve requirements specifications'. Proceedings of the 16th IEEE International Conference on Automated software engineering ASE01, San Diego, USA, November 2001, pp. 354–358

2 KAKAS, A.C., KOWALSKI, R.A., and TONI, F.: 'The role of abduction in logic programming' in GABBAY, D.M., HOGGER, C.J., and ROBINSON, J.A. (Eds.): 'Handbook of Logic in Artificial Intelligence and Logic Programming' (Oxford Science Publications, 1994) Vol. **5**, pp. 235–324

3 MITCHELL, T.M.: 'Machine learning' (McGraw-Hill, 1997)

4 RUSSO, A., MILLER, R., NUSEIBEH, B., and KRAMER, J.: 'An abductive approach for analysing event-based requirements specifications'. Technical report TR2001/7, Department of Computing, Imperial College, 2001

5 D'AVILA GARCEZ, A.S., and ZAVERUCHA, G.: 'The connectionist inductive learning and logic programming system', *Appl. Intell. J., Special Issue on Neural Networks and Structured Knowledge*, 1999, **11**, (1), pp. 59–77

6 RUMELHART, D.E., HINTON, G.E., and WILLIAMS, R.J.: 'Learning internal representations by error propagation' in RUMELHART, D.E., and MCCLELLAND, J.L. (Eds.): 'Parallel Distributed Processing:

Explorations in the Microstructure of Cognition' (MIT Press, 1986), Vol. **1**, pp. 318–362
7 BHARADWAJ, R., and HEITMEYER, C.: 'Model checking complete requirements specifications using abstraction'. Technical report NRL-7999, Naval Research Lab, 1997
8 ALCHOURRON, C.A., GARDENFORS, P., and MAKINSON, D.: 'On the logic of theory change: partial meet contraction and revision functions', *J. Symbol. Log.*, 1985, **50**, pp. 510–530
9 FLACH, P.A., KAKAS, A.C.: 'On the relation between abduction and inductive learning' in GABBAY, D.M., and KRUSE, R. (Eds.): 'Handbook of Defeasible Reasoning and Uncertainty Management Systems, Volume 4: Abductive Reasoning and Learning' 2000, pp. 1–33
10 ANDREWS, R., DIEDERICH, J., and TICKLE, A.B.: 'A survey and critique of techniques for extracting rules from trained artificial neural networks', *Knowl.-Based Syst.*, 1995, **8**, (6), pp. 373–389
11 D'AVILA GARCEZ, A.S., BRODA, K., and GABBAY, D.M.: 'Symbolic knowledge extraction from trained neural networks: A sound approach', *Artif. Intell.*, 2001, **125**, pp. 155–207
12 McCLUSKEY, T.L., and WEST, M.M.: 'The automated refinement of a requirements domain theory', *J. Autom. Softw. Eng.*, 2001, **8**, (2), pp. 195–218, Special Issue on Inductive Programming
13 RUSSO, A., MILLER, R., NUSEIBEH, B., and KRAMER, J.: 'An abductive approach for handling inconsistencies in SCR specifications'. Proceedings of the 3rd ICSE Workshop on Intelligent software engineering, Limerick, June 2000
14 RUSSO, A., MILLER, R., NUSEIBEH, B., and KRAMER, J.: 'An abductive approach for analysing event-based requirements specifications'. Proceedings of 18th International Conference on Logic programming ICLP02, Copenhagen, Denmark, August 2002, pp. 22–37
15 MICHALSKI, R.S.: 'Learning strategies and automated knowledge acquisition', in 'Computational models of learning' (Springer-Verlag, Symbolic Computation, 1987)
16 HIRSH, H., and NOORDEWIER, M.: 'Using background knowledge to improve inductive learning: a case study in molecular biology', *IEEE Expert*, 1994, **10**, pp. 3–6
17 LAVRAC, N., and DZEROSKI, S.: 'Inductive logic programming: techniques and applications' (Ellis Horwood, Series in Artificial Intelligence, 1994)
18 GAMBERGER, D., and LAVRAC, N.: 'Conditions for Occam's Razor applicability and noise elimination' in SOMEREN, M., and WIDMER, G. (Eds.): 'Proceedings of the European Conference on Machine learning', Prague, Czech Republic, April 1997, pp. 108–123
19 QUINLAN, J.R.: 'Induction of decision trees', *Mach. Learn.*, 1986, **1**, pp. 81–106
20 HAYKIN, S.: 'Neural networks: A comprehensive foundation' (Prentice Hall, 1999)
21 CLOETE, I., and ZURADA, J.M. (Eds.): 'Knowledge-based neurocomputing' (The MIT Press, 2000)
22 MUGGLETON, S., and RAEDT, L.: 'Inductive logic programming: theory and methods', *J. Log. Program.*, 1994, **19**, pp. 629–679
23 THRUN, S.B., BALA, J., BLOEDORN, E., BRATKO, I., CESTNIK, B., and CHENG, J., *et al.*: 'The MONK's problems: A performance comparison of different learning algorithms'. Technical Report CMU-CS-91-197, Carnegie Mellon University, 1991
24 D'AVILA GARCEZ, A.S., BRODA, K., and GABBAY, D.M.: 'Neural-symbolic learning systems: Foundations and applications'. Perspectives in Neural Computing (Springer-Verlag, 2002)
25 LLOYD, J.W.: 'Foundations of logic programming' (Springer-Verlag, 1987)
26 GELFOND, M., and LIFSCHITZ, V.: 'The stable model semantics for logic programming'. Proceedings of the 5th Logic programming Symposium, Seattle, USA, August 1988, pp. 1070–1080
27 BOSE, N.K., and LIANG, P.: 'Neural networks fundamentals with graphs, algorithms, and applications' (McGraw-Hill, 1996)
28 KIRBY, J.: 'Example NRL/SCR software requirements for an automobile cruise control and monitoring system'. Technical Report TR-87-07, Wang Institute of Graduate Studies, 1987
29 SHAW, M.: 'Comparing architectural design styles', *IEEE Softw.*, 1995, **12**, (6), pp. 27–41
30 D'AVILA GARCEZ, A.S., BRODA, K., GABBAY, D.M.: 'Metalevel priorities and neural networks' in FRASCONI, P., GORI, M., KURFESS, F., and SPERDUTI, A. (Eds.): 'Proceedings of ECAI2000, Workshop on the Foundations of Connectionist-Symbolic Integration', Berlin, Germany, August 2000, pp. 38–49
31 KAKAS, A.C., and MICHAEL, A.: 'Integrating abductive and constraint logic programming'. Proceedings of the 12th International Conference on Logic programming, Tokyo, Japan, June 1995, pp. 399–413
32 TOWELL, G.G., and SHAVLIK, J.W.: 'Knowledge-based artificial neural networks', *Artif. Intell.*, 1994, **70**, (1), pp. 119–165
33 MITCHELL, T.M., and THRUN, S.B.: 'Explanation-based learning: a comparison of symbolic and neural network approaches'. 10th International Conference on Machine learning, Amherst, MA, March 1993
34 OWRE, S., RUSHBY, J., SHANKAR, N., and VON HENKE, F.: 'Formal verification for fault-tolerant architecture: prolegomena to the design of pvs', *IEEE Trans. Softw. Eng.*, 1995, **21**, (2), pp. 107–125
35 HEITMEYER, C.L., KIRBY, J., LABAW, B.G., ARCHER, M., and BHARADWAJ, R.: 'Using abstraction and model checking to detect safety violations in requirements specifications', *IEEE Trans. Softw. Eng.*, 1998, **24**, (11), pp. 927–947
36 VAN LAMSWEERDE, A., DARIMONT, R., and LETIER, E.: 'Managing conflicts in goal-driven requirement engineering', *IEEE Trans. Softw. Eng.*, 1998, **24**, (11), pp. 908–926
37 MENZIES, T.: 'Applications of abduction: knowledge level modeling', *Int. J. Hum. Comput. Stud.*, 1996, **45**, pp. 305–355
38 ZOWGHI, D., and OFFEN, R.: 'A logical framework for modeling and reasoning about the evolution of requirements'. Proceedings of the 3rd IEEE International Symposium on Requirements engineering, Annapolis, USA, January 1997, pp. 247–259
39 SATOH, K.: 'Computing minimal revised logical specification by abduction'. Proceedings of International Workshop on the Principles of software evolution, Kyoto, Japan, April 1998, pp. 177–182
40 VAN LAMSWEERDE, A., and WILLEMET, L.: 'Inferring declarative requirements specifications from operational scenarios', *IEEE Trans. Softw. Eng.*, 1998, **24**, (12), pp. 1089–1114
41 GARG, P.K., and BHANSALI, S.: 'Process programming by hindsight'. Proceedings of ICSE14, Melbourne, May 1992, pp. 280–293
42 MCCLUSKEY, T.L., and WEST, M.M.: 'The automated refinement of a requirements domain theory'. Technical Report, School of Computing and Mathematics University of Huddersfield, Huddersfield, UK, 1999
43 NUSEIBEH, B., EASTERBROOK, S., and RUSSO, A.: 'Making inconsistency respectable in software development', *J. Syst. Softw.*, 2001, **56**, (11), pp. 171–180
44 HUNTER, A., and NUSEIBEH, B.: 'Managing inconsistent specifications: Reasoning, analysis and action', *Trans. Softw. Eng. Methodol.*, ACM Press, 1998, pp. 335–367
45 GHEZZI, C., and NUSEIBEH, B. (Eds.): *IEEE Trans. Softw. Eng., Special Issue on Managing Inconsistency in Software Development.* November 1999.
46 GABBAY, D.M., and HUNTER, A.: 'Making inconsistency respectable: a logical framework for inconsistency in reasoning. Part 1: a position paper' in 'Fundamentals of AI Research' (Springer-Verlag, 1991), pp. 19–32
47 EASTERBROOK, S.: 'Learning from inconsistency'. 8th International Workshop on Software specification and design, Paderborn, Germany, March 1996, pp. 136–140
48 THRUN, S.: 'Explanation-based neural network learning: a lifelong learning approach' (Kluwer Academic Publishers, Boston, MA, 1996)
49 MOONEY, R.J., and ZELLE, J.M.: 'Integrating ILP and EBL', *SIGART Bull.*, 1994, **5**, pp. 12–21
50 UCHITEL, S., and KRAMER, J.: 'A workbench for synthesising behaviour models from scenarios'. Proceedings of IEEE International Conference on Software Engineering ICSE01, Toronto, Canada, May 2001