On the Use of Logical Abduction in Software Engineering

ALESSANDRA RUSSO and BASHAR NUSEIBEH

Department of Computing,
Imperial College of Science, Technology and Medicine
London SW7 2BZ, U.K.
E-mail:{ar3, ban}@doc.ic.ac.uk

First Revision September 2000

In this paper we survey recent work on the use of abduction as a knowledge-based reasoning technique for analysing software specifications. We present a general overview of logical abduction and describe two abductive reasoning techniques, developed from the logic and expert system communities. We then focus on two applications of abduction in software engineering, namely, analysis and revision of specifications. Specifically, we discuss and illustrate, with examples, how the above two abductive reasoning techniques can be deployed to reason about specifications, detect errors, such as logical inconsistencies, provide diagnostic information about these errors, and identify (possible) changes to revise incorrect specifications. We then conclude with a discussion of open research issues.

Keywords: Abductive reasoning, analysis, revision, specification, inconsistency, logic programming.

1. Introduction

Specifications are key products of the development process of a software system. Requirements engineering focuses on the development of requirements specifications that model real-world goals and environmental constraints of a system[1] under construction [55]. Design is concerned with the development of specifications of software architectures and their constituent components [33].

Independently of the underlying software development process, specifications are continuosly subject to change and evolution. In the case of requirements specifications, for instance, inconsistencies may arise during the requirements elicitation process, and changes may be necessary in order to handle such inconsistencies [14; 40]. System requirements

might also evolve because of environmental changes or to handle unforseen problems in a system. This leads to changes in the requirements specifications that describe such requirements. Changes in requirements specifications cause changes in design specifications, which themselves lead to changes in the implementation. Analysis and maintenance of specifications are therefore key activities in software engineering.

Various techniques for analysing specifications have been developed [26] and recently an increasing research effort has been dedicated to investigate the use of formal methods at different stages of the software development process, e.g., [1; 4; 27; 52]. The rigour and precision of formal techniques, such as model checking, theorem proving and logic-based reasoning mechanisms like abduction [35; 41; 45], facilitate a better understanding of the system to be developed by uncovering errors, inconsistencies, incompletenesses, which might otherwise go undetected. The formal techniques differ from each other in various ways and have advantages and limitations. Theorem proving techniques have been shown to be successful in supporting verification of software specificiations [42], model checking techniques were initially used in hardware verification and have started to become more widely used to support the analysis and validation of (requirements) specifications [10; 34]. More recently, logic-based reasoning techniques such as goal-regression [53] and abduction [37; 45; 48] have also been shown to be valuable techniques for analysing and managing (requirements) specifications [41; 47; 53]. This chapter is a survey of recent work on the use of one of these formal techniques – *abduction* – in software engineering.

In Artificial Intelligence (A.I.) abduction is one of three common modes of reasoning (the other two being deduction and induction). In general terms, abduction is a useful constructive technique for generating "explanations" or "plans" for given "observations" or "goals". In A.I., abduction has been shown to be suitable for automating tasks such as diagnosis [12], planning [15], and theory and database updates [11; 25; 28]. Recent research results have shown its applicability and utility in software engineering, as a technique for supporting knowledge-based software development [35], and for facilitating analysis and revision of specifications [41; 45; 47].

This chapter will provide an overview of these results, present a critical analysis of why and when abductive reasoning is effective in software engineering, and illustrate how to make use of such techniques. For the purposes of this paper, we assume specifications to comprise of system descriptions (e.g., system requirements or system designs) and system properties (e.g., required system's invariants, safety properties, deadlocks). In this context, analysis of a specification means consistency checking to verify the satisfaction of system properties over a system description. Revision refers to the process of identifying changes (i.e. additions or deletions) to a specification that would re-establish the correctness of the specification.

The chapter is structured as follows. In Section 2, we define the notion of logical abduction. In Section 3 we describe two examples of abductive reasoning techniques. The first is an abductive proof procedure that uses logic programming [23] as the underlying rule-based reasoning engine. The second is an abductive technique that uses a graph-based approach [38] for reasoning about specifications. Sections 4 and 5 describe two main applications of abduction in software engineering, namely, analysis and revision of specifications. Illustrative examples will be given throughout the chapter. We will then conclude with a discussion of directions for future work.

2. Overview of Abduction

Abduction is commonly defined as the problem of finding a set of hypotheses (an "explanation" or a "plan") of a specified form that, when added to a given (formal) specification, allows an "observation" or "goal" sentence to be inferred, without causing contradictions. We consider here an example taken from [30].

Example 1.
Consider a specification D composed of the following rules:

rained-last-night $\rightarrow$ *grass-is-wet*
sprinkler-was-on $\rightarrow$ *grass-is-wet*
grass-is-wet $\rightarrow$ *shoes-are-wet*

Now suppose that we observe that our shoes are wet, and we want to know why this is so. A possible explanation is {rained-last-night} – if we add it to the above explicit domain-specific description D it implies the given observation. Another alternative explanation is {sprinkler-was-on}. Abduction is then the process of computing such explanations for the given observation.

From the point of view of knowledge-based reasoning, the philosopher Pierce first introduced the notion of abduction as one of three fundamental forms of reasoning, the others being induction and deduction [43]. Informally, given a rule-based domain-specific description D (e.g., a system description), some particular cases α (e.g., instances of system behaviors), and a result β (e.g., a system property), deduction is the analytic process of applying the general rules to the particular cases in order to infer the result (i.e. $D \wedge \alpha \Rightarrow \beta$). Within the same context, induction is "learning" some new rules D_i after having seen numerous examples of β and α (i.e. $\alpha \wedge \beta \Rightarrow D_i$), and abduction is using the result and the general rules to infer the particular cases that explain such results (i.e. $\beta \wedge D \Rightarrow \alpha$)[2].

Abduction is therefore a reasoning process that computes explanations for given observations. This reasoning process is in general *non-monotonic*[3], because the explanations generated are strictly dependent on the state of the domain-specific description D. If new information is added to D, new explanations, possibly different from those generated previously can be identified. A formal definition of abduction in logical terms is as follows.

Definition 1
Given a domain description D and a sentence (goal/observation) G, abduction is the process of identifying a set Δ of assertions such that

1. $D \wedge \Delta \models G$
2. $D \wedge \Delta$ is consistent □

The set Δ is required to satisfy two main criteria: (1) it is restricted to belong to a domain-specific set of sentences, called *abducibles*, and (2) it is minimal. The set of abducibles are defined a *priori* to reflect some notion of causality with respect to the given observations. For instance, in Example 1, {*grass-is-wet*} is also an explanation for the observation *shoes-are-wet*. However, such assertion would explain one effect in terms of another effect, since *grass-is-wet* could itself be explained in terms of the cause {*rained-last-night*}. To draw an analog example within the context of requirements specifications, system behaviors can be seen as caused by system states and events in the environment. Observations, such as violation of a system property, should therefore be explained in terms of specific environmental events and states in which the system is when such events occur,

rather then any other intermediate system state. The notion of abducibles helps, therefore, tailor the notion of explanation to the particular application domain, as well as formalise the given domain knowledge. From the knowledge representation viewpoint, in fact, abducible facts should never appear as consequence of rules. For instance, if Example 1 had also included the information *shoes-are-wet*→*rained-last-night*, then *rained-last-night* would not have been considered to be an abducible fact[4]. The minimality property means that the abduced explanations should not be subsumed by other explanations. For instance, in Example 1, the explanation {*rained-last-night*, *sprinkler-was-on*} for the observation {*shoes-are-wet*} is not minimal, since there are two other explanations {*rained-last-night*} and {*sprinkler-was-on*} that subsume it.

The abductive reasoning process can be further refined by means of *integrity constraints* [30]. In general, integrity constraints are used to define the class of legal models of a given specification. For instance, in the case of requirements specifications, integrity constraints could be the natural physics laws of the environment in which the system is supposed to operate. In the presence of integrity constraints, abductive reasoning has to generate legal explanations, namely, explanations that satisfy the given constraints. To take this into account, the definition of abduction needs to be modified sligthly. Given a set I of integrity constraints, condition 2 of Definition 1 needs to be replaced by the following stronger condition:

$$2'.\ D \wedge \Delta \text{ is consistent, and } D \wedge \Delta \models I$$

This alternative condition has the effect of further reducing the collection of alternative explanations generated by the abductive reasoning process. If, for instance, in Example 1 we had the integrity constraint that the sprinkler can never be on, then the abductive process would only have generated the explanation {rained-last-night}.

The applications of abduction illustrated in Section 4 and 5 take into account this notion of integrity constraints.

3. Abductive techniques

The above overview of abduction is independent of any particular computational technique used to implement abductive reasoning. In this section we illustrate two different computational techniques for performing abductive reasoning. The first one is based on logic programming, and the second uses reasoning on dependency-graphs. The choice of these two techniques aims to provide the reader with two different viewpoints of abduction, developed by two different knowledge engineering communities, the logic-based and the expert system communities.

3.1 An abductive proof procedure using logic programming

Abductive logic programming focuses on the development of formal frameworks and techniques for performing abductive reasoning within the (implemetation) context of logic programming [23]. Informally, a logic program is a specific logic-based form of implementation of a given specification, expressed in terms of Horn clauses extended with negation as failure [7]. Horn clauses are rules of the form

$$A \leftarrow L_1,\ldots L_n.$$

where each L_i is either an atomic piece of information B_j or its negation $\sim B_j$. The connective $\sim$ is called *negation as failure*, and should be read as "not provable" from a given logic program. The underlying reasoning engine is based on resolution [22].

An abductive proof procedure for logic programs with negation as failure was first developed by Eshghi and Kowalski [16] and subsequently extended for different types of (extended) logic programs (e.g., logic programs with classical negation). We will describe here the basic structure of such procedure, which still remains at the core of all existing abductive proof procedure for logic programming [30]. This procedure assumes, without loss of generality, that it is convenient to define the set of abducibles among those predicates that are not conclusions of any clause in a given logic program, and that the rules define only positive literals. The first condition automatically ensures that the explanations generated by the procedure are "basic" explanations, i.e. they are not definable in terms of other explanations.

An abductive proof procedure for logic programming consists of two phases, an *abductive phase* and a *consistency phase*, which interleave with each other. The abductive phase is an extension of standard resolution. In standard resolution, selected (sub)goals are unified and resolved with the conclusions of any rules. If this process fails, then the proof fails. In the abductive phase, when a selected (sub)goal fails to resolve with the conclusion of any of the given rules, then it is *abduced*. Each abduced assertion is temporarily added to a set of abducibles that have already been generated. The entire new set of abducibles is then checked for consistency with the specification, using the consistency rules (CC1) and (CC2)[5]:

$$(A \wedge \sim A) \rightarrow \bot \qquad \text{(CC1)}$$
$$(A \vee \sim A) \qquad \text{(CC2)}$$

The consistency checking consists of verifying that (a) it is not the case that for some atomic fact A, both A and ~A can be proved from the specification and a given current set of abducibles, and (b) for each atomic fact A, it is either the case that A is proved or ~A is proved. Because of the close world assumption of logic programs and the type of logic programs considered, where negation does not appear in the consequence of any rule, the two types of consitency checking will only need to be performed on the abduced facts. For more details, the reader is referred to [30]. If the consistency checking succeeds, the temporary assertion added to a current set of abducibles is permanently accepted in the set of abducibles, otherwise it is discharged. The consistency phase may itself invoke the abductive phase, in order to verify the inference of some intermediate sub-goals.

In the presence of domain-specific integrity constraints, the consistency checking has to guarantee that the generated abducibles also satisfy those constraints. We illustrate such a procedure in more detail via an example also taken from [30].

Example 2.
Consider the following (logic program) specification:

$$s \leftarrow \sim p$$
$$p \leftarrow \sim q$$
$$q \leftarrow \sim r$$

We want to discover the possible explanations for the given observation s. The computation is shown in Figure 1, using the box notation adopted in [30]. The proof in single line boxes belong to an abductive phase, whereas the proofs in double line boxes to a consistency phase. It succeeds, generating the set $\Delta = \{\sim p, \sim r\}$. The abductive phase starts as a standard

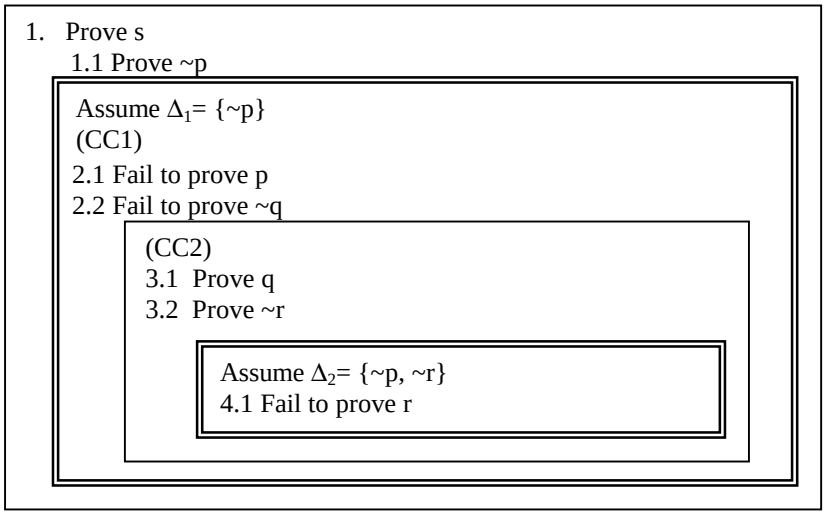

Figure 1. Abductive computation for Example 2.

resolution step; e.g., the goal *s* resolves with the conclusion of the first rule generating the new subgoal $\sim p$. At this point, since there is no rule with conclusion $\sim p$, this subgoal is temporarilly added to the set of generated abducibles, that so far is just given by $\{\sim p\}$. Once an assertion is abduced, the consistency phase begins. In this case, we check that the abduced information satisfies (CC1). This means checking that the assertion *p* is not provable. This checking succeeds if all possible ways (i.e. rules) of proving *p* fail. In this case we have only one rule. To make it fail, we check that $\sim q$ is not provable. At this point, no rule with conclusion $\sim q$ exists, but the integrity constraint (CC2) implies that failure of $\sim q$ is only allowed provided that *q* is provable. The abductive phase is therefore again called to compute the provability of this subgoal. By standard resolution, the subgoal *q* unifies with the third rule and the new subgoal $\sim r$ is generated. Since there is no rule with conclusion $\sim r$, this is added to the set of abducibles and a new consistency phase is called. This point is similar to the abduction step of $\sim p$. In this case however, the consistency checking with (CC1) is immediate. The failure of *r* is guaranteed by the fact that $\sim r$ is (abductively) assumed.

If the (logic program) description includes also some integrity constraints, then the consistency phase has to check their satisfiabilty each time a new assertion is abduced. For instance, if in the Example 2, there was also an integrity constraint of the form $\sim(\sim p \wedge a)$, having assumed $\sim p$, the consistency should also check that *a* fails. This would imply, by (CC2) $\sim a$ succeeds, which means also abducing the $\sim a$. The final set of abducibles would have also included some additional abduced assertions needed in order to verify the integrity constraints.

To summarise, the abductive proof procedure based on logic programming described above has the following characteristics:

1. It can be applied to logic program representations of given (system) descriptions. Therefore, a mapping of such descriptions into a logic program is first needed. Similarly, for any domain-specific constraint.
2. A notion of abducibles needs to be defined. This can be done by simply adding to the logic program clauses of the form abducible-predicate(X), for each propositional letter or predicate X that can be abduced.
3. Any existing implementation of abductive logic programming proof procedure can be applied. Examples are given in [29].

3.2 Graph-based abduction and the HT4 approach

We now describe a second abductive technique, based on dependency-graph representations of specifications. The technique was proposed by Menzies [35], within the context of a knowledge level modelling approach to expert system design [39; 54]. Here, the knowledge base of an underlying system is assumed to be composed of domain-specific knowledge and a model of the underlying problem-solving inference process. For instance, Clancey's knowledge level modelling approach [6] assumes a "qualitative model" (essentially a first-order theory) of the underlying domain knowledge, and a full network of possible proof trees that could be generated from this qualitative model. This network is the model of the inference process. A similar approach is adopted by Menzies in his abductive framework HT4 [35; 36]. This framework uses a graph-theoretic approach, rather then a logic-based approach described in the previous section. A theory, about certain domain knowledge, is given. This is essentially a dependency graph, whose vertices are propositions that can take one of the three values {UP, DOWN, STEADY}, and the edges are labelled with "++" or " --". An example taken from [35] is given in Figure 2. An edge from a vertex X to vertex Y, labelled with "++", means that the proposition Y being UP/DOWN can be explained by the proposition X being UP/DOWN. An edge from a vertex X to a vertex Y, labelled with "--", means that the proposition Y being UP/DOWN can be explained by the proposition X being DOWN/UP. To draw an analogy between this

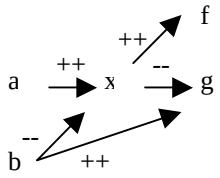

Figure 2. An example of a qualitative theory

representation of domain knowledge and the logic-based approaches, we can think of the value UP being equal to the Boolean value TRUE, the value DOWN being equal to the Boolean value FALSE, and the value STEADY being equal to TRUE∧FALSE[6].

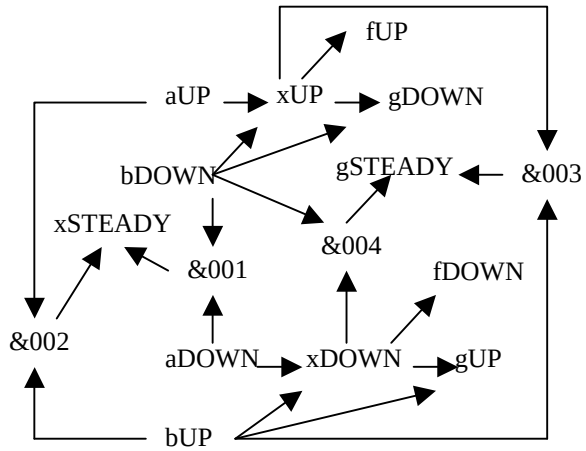

Figure 3. The dependency graph generated from Figure 2.

The HT4 algorithm generates from a given qualitative theory a complete dependency-graph, called a "super-theory", which expresses the above meaning of the "++" and "--" edges in terms of binary dependency relations between the three values that each vertex could have. Figure 3, describes the super-theory generated from the qualitative theory given in Figure 2. The HT4 framework uses this super-theory to generate abductive proofs. The abductive reasoning is performed within a user-supplied task. A task is defined in terms of a collection IN of input vertices (a subset of the vertices in the quality theory), a collection OUT of outputs (data to be proved), and a notion of BEST. The latter is an operator that selects from alternative abductive proofs a preferred one. Of course the notion of preferred one is application-dependent. Different BEST operators are discussed in [35] corresponding to applications such as prediction, qualitative reasoning, planning and monitoring.

Given a task, HT4-style abduction is the search for all possible proofs (i.e. paths in the dependency-graph) from a subset of vertices included in IN to a subset of vertices included in OUT, such that no vertices in these proofs have contradictory values[7]. For instance, given the super-theory described in Figure 3 and the sets IN={aUP, bUP}, and OUT={gDOWN, fDOWN}, an abductive answer would be the separate collections of proofs shown in Figure 4. Note that, for instance, the path bUP→xDOWN→fDOWN is not included in the first collection because it contains the vertex xDOWN which contradicts the vertex xUP in other identified paths, and so vice-versa. The two collections of abductive proofs are called *worlds* [35].

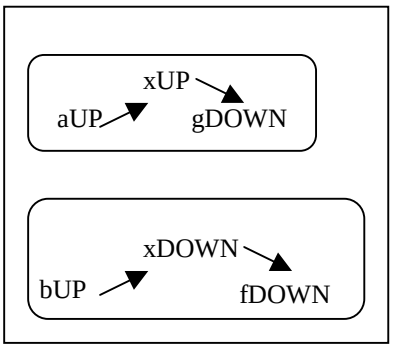

Figure 4. Two possible abductive answers.

To summarise, the HT4 abductive procedure proposed by Menzies and described above has the following characteristics:

1. It can be applied to specifications represented as qualitative theories, since these can be transformed into full, extended dependency-graphs. Therefore, a mapping of any given specification into the dependency-graph syntax is first needed. Note that the vertices in the graph are propositions. The approach is therefore applicable to propositional representation of system specifications. First-order theories could be represented only to the extent that they can be partially evaluated into equivalent ground theories.
2. There is no notion of abducible. The explanation generated in this approach includes both basic information as well as intermediate information inferred during the reasoning process.
3. Different definitions of the BEST operator facilitate the use of the same abductive reasoning engine for supporting different types of reasoning: prediction, qualitative reasoning, planning and validation.
4. The HT4 tool has gone through various phases of improvements and a description of the underlying algorithm can be found in [35].

4. Application of Abduction in Software Engineering

As mentioned in the introduction, two crucial problems in software engineering are the analysis and maintenance of specifications. Recent results have shown how abduction can successfully be applied to software engineering in order to address these problems [35; 45; 47; 48]. The next two sections illustrate two different applications of abduction: abduction for analysing specifications and abduction for revising specifications. Specifically, we will illustrate how abductive reasoning allows: (a) detection of inconsistencies and/or property violations, and identification of related diagnostic information, and (b) reasoning about possible change(s) to perform on a given specification in order to resolve detected errors, or to support consistency management in evolving specifications.

4.1 Analysing specifications

Specifications can be seen as (formal) descriptions that model an underlying system, within a specific context. For instance, requirements specifications model a system in terms of its interaction with the environment and user-goals. Specification analysis is therefore a means of performing analysis of a model of the system rather than of the system itself. Assuming that such a model is a faithful representation of the system, analysing specifications helps identify problems in the system. For the purpose of this chapter, we focus on two particular types of specification analysis, namely, analysis for (a) detecting inconsistencies and violation of system properties, and for (b) identifying diagnostic information about detected errors as a debugging aid for engineers.

4.1.1 Detecting inconsistencies.

Inconsistency detection can be seen as detection of a property violation, where the property is a domain-independent rule of the form “for any sentence A (denoting either any system

variable or any environmental entity), it cannot be the case that both A and ¬A are inferable from a given specification". The validation of this property is computationally equivalent to evaluating the satisfability of the overall specification, taken as a conjunction of formulae. For first-order representations, this is an NP-hard problem. We will therefore restrict our attention to propositional specifications or to first-order representations over finite domains that can be instatiated and grounded down to propositional level.

Abductive reasoning can, in general, be used to detect inconsistencies by considering, as an observation (goal), atomic instantiations of the form $P(X) \wedge \neg P(X)$, for each predicate symbol and ground term used in the specification. If the abductive reasoning process succeeds, i.e. it is able to identify a set of abductibles Δ that satisfies conditions (1) and (2) of Definition 1, then the specification is inconsistent since $P(X) \wedge \neg P(X)$ can be derived from it. Moreover, the set Δ would provide (consistent) diagnostic information for the inconsistency $P(X) \wedge \neg P(X)$ in terms of parts of the specification that lead to this inconsistency. However, if the abductive reasoning technique fails to find such a Δ, it is either the case that $P(X) \wedge \neg P(X)$ is not inferable from the specification or there is no consistent way of constructing such an explanation Δ. In that case we would not be able to conclude that the given specification is consistent with respect to the predicate $P(X)$.

The abductive approach developed by Menzies [35; 36] provides an alternative way of identifying inconsistent information in a given specification. Instead of considering ground individual inconsistencies as properties to validate, the HT4 abductive algorithm [35; 36] identifies inconsistent facts in a given specification as side effects of an abductive reasoning process for a given task. For instance, given a set of INputs and a set of OUTputs of a specification (qualitative theory), the HT4 abductive reasoning process identifies all consistent paths within the network of all possible proof trees that link the INput to the OUTput. Paths that include contradictory propositions are separated in different worlds. Therefore, in order to identify these different worlds, the HT4 abductive process identifies the set of contradictory information. For instance, in the example illustrated in Figure 4 the only contradictory information is the proposition *x*, since it can assume both values UP and DOWN. Of course the set of contradictions identified by the HT4 abductive algorithm depends on the task under consideration. However, considering a task whose set of INputs is the entire set of inputs of a given specification, and whose set of OUTputs covers all possible outputs of a specification, the HT4 abductive reasoning process is able to identify all possible contradictory information that is included in a given specification. This is possible because the HT4 algorithm records as abductive answers all possible proof trees, separating the ones which are inconsistent with each other in different worlds. In the abductive proof procedure for logic programming, on the other hand, abductive information which is inconsistent with the given specification is simply rejected. The final abductive answers are only those Δ (parts of the specification) that are consistent with the specification itself.

An additional advantage of using the HT4 algorithm for analysing specifications is the possibility of reasoning in the presence of inconsistencies. As illustrated in [37], given some (possibly inconsistent) specification and some reasoning task, HT4 is still able to generate abductive answers for the task despite the presence of inconsistencies. These answers are simply collected in different worlds, and the algorithm will generate as many worlds as there are inconsistencies in the specification. This is in particularly useful for

analysing multi-viewpoint specifications, which may be in conflict with each other. For examples and further details on this topic the reader is referred to [37].

4.1.2 Using abduction to validate system properties.

Another type of specification analysis is validation of system properties. The work in [45] provides an example of the application of abduction for this purpose. Specifically, it shows how abductive reasoning mechanisms can be used to detect violation of properties like system invariants in event-based specifications. The basic idea and methodology described in [45] is, in principle, applicable to any type of specifications and system properties. Given a specification composed of a system description and some system invariants, the abductive reasoning mechanism can be used to check if the system invariants are satisfied by the specification. For each system invariant the abductive reasoning mechanism is able to identify a complete set of counterexamples, if any exist, to the invariant. The information included in the counterexample depends on the type of specification under consideration. For instance, in the case of event-driven specifications, such counterexamples are expressed in terms of a symbolic current state of the system and associated event-based transition. If the abductive reasoning mechanism fails to find an answer, this establishes the validity of the invariant with respect to the system description. More specifically, this application of abduction to specification analysis employs the standard notion of abduction, specified in Definition 1, in refutation mode. As illustrated in Figure 5, the goal for the abductive

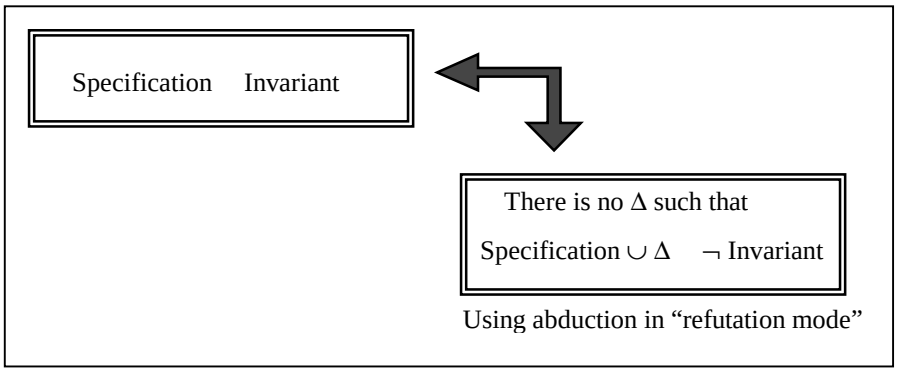

Figure 5. Using abduction to analyse system invariants

reasoning mechanism is the negation of a given system invariant. If the abductive proof procedure succeeds in identifying a set of abducibles Δ, then the invariant is violated, since its negation can be inferred from the specification. The Δ provides the "faulty" behaviours of the system that contradict the given invariant. Therefore, the set Δ can be considered to be a set of counterexamples for the given system invariant.

Whereas the theoretical framework of abduction can in principle be applied to any type of system invariant, existing abductive proof procedures only accept instantiated sentences[8] as observations. This provides some constraint on the practicality of the abductive-based analysis process – only invariants that can be reduced *a priori* to some equivalent ground representation can be analysed using the existing abductive techniques illustrated in Section 3. An example of application of abduction for anlaysing system invariants is given below.

Example 3. Using abduction to analyse invariants in SCR Specifications.
The results shown in [45], illustrate how abductive logic programming techniques can be used to analyse single-state system invariants in Software Cost Reduction (SCR) specifications [21]. These are properties of the form ∀t. I(t), which states that a required system invariant I must always hold. This is typical for safety-critical system properties, where an invariant I states some logical expression of system/environment properties as a function of specific system states (e.g.,∀t.[system_state(t)→ expression(t)]). Examples of such properties in SCR specifications are mode invariants [19]. For instance, in the case of a cruise control system specification [2; 20], a system invariant I would be:

$$system\_cruise \rightarrow system\_ignited \wedge system\_running \wedge \neg brake$$

Within the logical representation of SCR specifications adopted in [45], the above invariant can be formalised as:

$$\forall t.\ Holds(cruise,t) \rightarrow \quad (I)$$
$$Holds(ignited,t) \wedge Holds(running,t) \wedge \neg Holds(brake,t).$$

where *t* denotes the real-life time of the system. The results shown in [45], illustrate that the analysis task for verifying if a given invariant is satisfied in a given SCR specification can be reduced to a ground propositional level in the following way. Given a (first-order) logic representation of an SCR description of a required system behavior (i.e. a mode transition table), denoted with EC(N), where N is an underlying time structure, and given, for instance, the above invariant I, the analysis problem:

$$EC(N) \models I$$

can be reduced to the following two simpler tasks:

(1) $EC(N) \models I(0)$
(2) $EC(S) \wedge I(Sc) \models I(Sn)$

where EC(S) is a ground instantiation of the SCR description with respect to two symbolic time points "Sc"and "Sn", representing a "current time" and a "next time" respectively. The first task is a simple theorem proving problem. The second task is supported by the application of abductive reasoning in refutation mode. So, to show that $EC(S) \wedge I(Sc) \models I(Sn)$ it is equivalent to show that the abductive proof procedure for logic programming fails to find a Δ such that $EC(S) \wedge I(Sc) \wedge \Delta \models \neg I(Sn)$. Note that in this case the observation (goal) given to the abductive technique is ¬I(Sn), which is the ground formula:

$$[Holds(cruise,Sn) \wedge$$
$$[Holds(ignited,Sn) \vee Holds(running,Sn) \vee \neg Holds(brake,Sn)]$$

If, on the other hand, the abductive proof procedure produces a set Δ such that $EC(S) \wedge I(Sc) \wedge \Delta \models \neg I(Sn)$, then this Δ is an explicit indicator of where in the SCR description there is a problem. Detailed results of a case study using abduction to SCR specifications is given in [45].

The advantage of applying such an abductive approach to analyse system invariants is that it provides a formal technique for verifying properties and detecting errors that (a) does not rely on complete descriptions of the system specification and domain knowledge, and (b) provides diagnostic information about the detected errors (e.g., violation of safety properties) as a debugging aid for the engineer. It is the integration of these two features that distinguishes this (logic-based) abductive approach from other existing formal

techniques such as those based on model checking or theorem proving [10]. The current limitation is that results so far have only shown the applicability of abduction to specific types of invariants. The authors have, however, argued in [45] that the same approach can be extended to cover other types of system properties such as liveness, deadlock, etc.

4.2 Revising specifications

One application of abduction is "theory change" [28]. This is particulary useful because it can support automated generation of simple changes of system specifications, in order to either resolve some detected inconsistencies or re-establish some violated system properties. A theory update problem is the problem of identifying changes (e.g., transactions) to be performed on a given theory so that some change request is satisfied (i.e. can consistently be performed). Examples of update requests are, for instance, addition (resp. deletion) of information, which in logic terms means requesting that the information should or should not be inferable from a given specification. Using abduction for performing such tasks means essentially interpreting the update request as an observation to be explained, and the explanation of the observation as changes (transactions) to be performed.

More specifically, using Inoue and Sakama's definition [25], given a specification S and a pre-defined collection of abducibles Ab, if the change request is of the form "add R" (resp. "delete R"), the abductive reasoning generates a pair (Δ^+, Δ^-), where Δ^+ and Δ^- are ground instances of elements from Ab. The abduced facts in Δ^+ are information to be added to the specification , whereas the abduced facts in Δ^- are information to be deleted from the specification. The pair (Δ^+, Δ^-) has therefore to satisfy the following two conditions[9]:

1. $(S \cup \Delta^+) \setminus \Delta^- \models R$ (resp. $(S \cup \Delta^+) \setminus \Delta^-$ R)
2. $(S \cup \Delta^+) \setminus \Delta^-$ is consistent

The change transaction is then given by the addition of each abduced fact in Δ^+ and the deletion of each abduced fact in Δ^-. The new specification is given by $(S \cup \Delta^+) \setminus \Delta^-$. A diagrammatic representation illustrating the use of abduction for revising specifications is shown in Figure 6. As an example, consider the specification S given by {*p(x)*←*¬q(x)*,

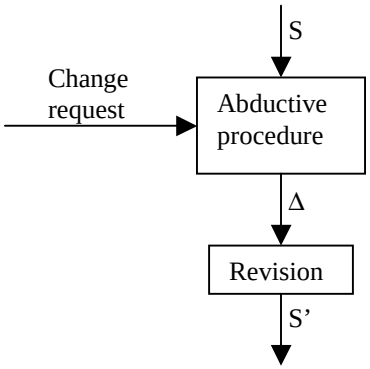

Figure 6. How to use abduction for changing specifications

q(x)←*b(x)*, *b(a)*} and the change request *insert p(a)*. The abductive proof procedure is applied, treating *p(a)* as the observation to explain, and generating a $\Delta = (\Delta^+, \Delta^-)$, with $\Delta^+ = \varnothing$

and Δ^-={~b(a)}. The resulting revision change is then simply to delete *b(a)*, leading to the new specification S'={*p(x)*←~*q(x)*, *q(x)*←*b(x)*}. For more examples the reader is referred to [28].

The use of a logic programming abductive proof procedure for reasoning about change has the following characteristics. As shown in the example, the theory (specification) is composed of some fixed knowledge; e.g., a set of rules that is not subject to change, and explicit knowledge in terms of atomic facts that can be changed. The abductive procedure takes the update request as its observation to explain, and generates an abduced explanation Δ. The negated facts in Δ give the Δ^- used in the above definition and the positive facts in Δ give the Δ^+. This set Δ is then mapped onto a transition operation: negated facts are deleted from the given specification and positive facts are added.

In order to apply such a technique for reasoning about change in software specifications, we need to define what are the fixed and modifiable parts of our given description. The fixed part can either be domain-specific knowlede about change for a given type of specification, so considering the whole specification modifiable, or a part of the specification that we can safely assume not to be subject to change, for instance the set of system's properties or some domain knowledge. Changes can be detected using only the fixed part of the specification. However, the only changes that abduction can help identify are single assertions and not more complex information such as new rules. This is somewhat restrictive, since, in a real-setting, changes to specifications can have different levels of complexity. However, as shown in [41; 47], this kind of abduction does begin to address the difficult problem of specification management. Two example applications of abduction for revising specifications are described in more detail in [41; 47].

4.2.1 Using abduction to evolve inconsistent (requirements) specifications.

The work in [41] shows how abduction can be used for handling inconsistencies in (requirements) specifications. The approach provides an inconsistency handling mechanism that supports incremental evolution of specifications, by identifying changes that address some specification inconsistencies, while leaving others [17]. Specifications are assumed to be composed of many partial specifications (typically developed by different stakeholders), related to each other by means of pre-defined "consistency rules". Each partial specification may or may not contain logical inconsistencies. However, the overall specification is defined to be inconsistent[10] whenever at least one of the pre-defined rules is violated. If a particular consistency rule is violated, the abductive reasoning mechanism identifies (evolutionary) changes to perform on the specification, such that the particular consistency rule is no longer violated. In order to facilitate incremental evolution of specifications, the partial specifications and consistency rules are assumed to be represented in quasi-classical (QC) logic [24] – an adaptation of classical logic that allows reasoning in the presence of inconsistencies without trivilisation[11]. The novelty of this work is in applying existing abductive techniques for logic programming but within the more realistic setting of inconsistent specifications. An example of a library system case study is described in [41]. A simplified version is given here.

Example 4.
The requirements of a library system include the following partial specification:

> "… individuals should be allowed to borrow books from a library if they need these books and if the books are available. A book is available if there is one copy in the library. Once a book is borrowed, it is no longer available for others to borrow. Moreover, a book copy is needed and it is in the library ".

It includes the consistency rule that "if a book is borrowed then it is no longer available". The full specification given in [41] violates such rule. This means that a QC logical representation of the negation of this rule, i.e. *BorrowingBook* $\wedge$ *BookAvailable* is derivable from the QC representation of the specifications. Resolving this inconsistency means eliminating one of the above two literals, *BorrowingBook* or *BookAvailable* from the set of consequences of the specification.

The abductive approach developed in [41], uses an algorithm for mapping QC specifications into logic programs, and then uses existing abductive techniques for logic programming to identify changes that would resolve the detected inconsistency. Part of the logic program generated for this example is given below, where the predicate *HoldsS* provides a reified representation of the QC formulae[12]. The set of abducibles is given by the atomic predicate *HoldsS* whose content are names for single atomic QC formulae, and they are not in the left-and-side of any logic program rule. The set of rules given below correspond to the fixed part of the specification, whereas the single atomic *HoldsS* predicate is that part of the specification that can be modified.

HoldsS(BookAvailable) ←
    HoldsS(BookInLibrary),
    HoldsS(BookInLibrary → BookAvailable).
HoldsS(BookAvailable) ←
HoldsS(BookCopy),
    HoldsS(BookCopy → BookAvailable).
HoldsS(BookInLibrary → BookAvailable) ←
    HoldsS(BookCopy),
    HoldsS((BookCopy ∧ BookInLibrary) → BookAvailable).
HoldsS(BookCopy → BookAvailable) ←
    HoldsS(BookInLibrary),
    HoldsS((BookCopy ∧ BookInLibrary) → BookAvailable).
HoldsS((BookCopy ∧ BookInLibrary) → BookAvailable).
HoldsS(BookCopy).
HoldsS(BookInLibrary).
HoldsS(BookNeeded).

To apply the abductive technique for logic programming to the above logic program to resolve the detected inconsistency, the deletion of *BookAvailable* should be treated as the change request, since this will resolve the violation of the given consistency rule. Part of the abductive reasoning performed in this case is described in Figure 7. The abductive proof identifies two alternative changes Δ^- = {~*HoldsS(BookCopy)*} and Δ^- = {~*HoldsS(BookInLibrary)*}, which lead to two change transitions, *delete BookCopy* or *delete BookInLibrary* from the given specification.

For futher details on this application of abduction for inconsistency handling the reader is referred to [41].

4.2.2 Using abduction to manage consistency in specifications.

The work described in [46; 47; 48] takes a different perspective on the problem of specifications revision. In particular, it follows the more traditional line of research on formal techniques for analysing and managing the impact of changes, where the starting point is a consistent specification, and the revision process is a process of re-stablishing consistency in order to accommodate a given change request. Abduction in this case is used to identify additional changes on the given specification so as to re-establish consistency.

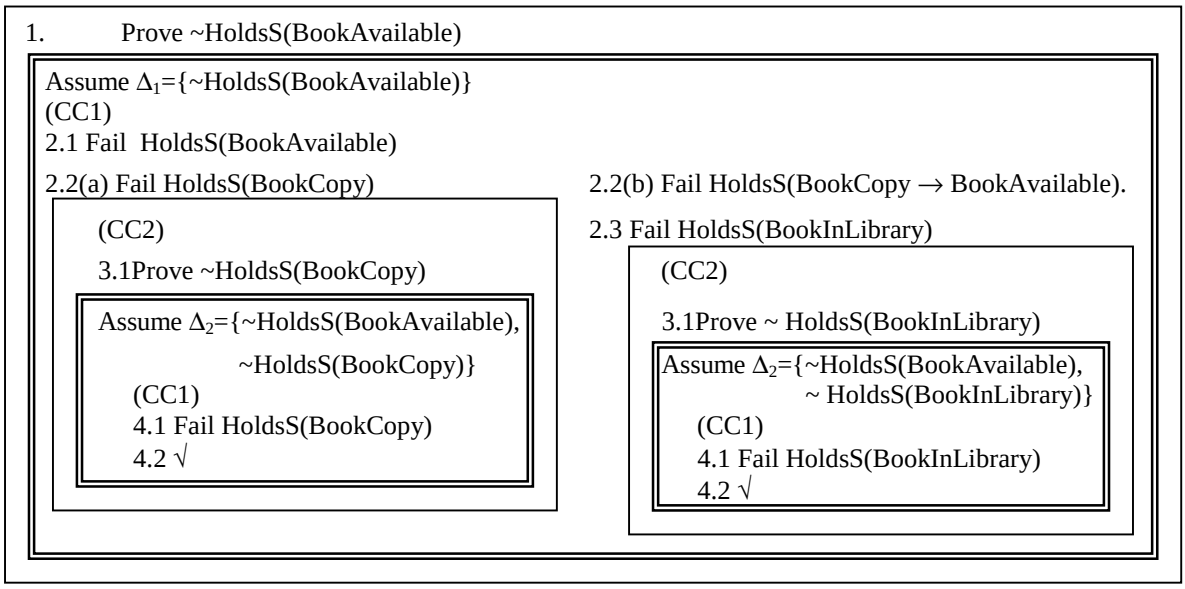

Figure 7. A partial abductive proof for managing the change request "delete((BookAvailable)"

In [46] Satoh describes a logic approach based on abductive reasoning for adding and deleting "pollution markers" from a given specification in order to manage consistency after a change has been performed. The idea of pollution markers for inconsistency handling was first introduced by Balzer [3] as a technique for treating inconsitencies as exceptions and adding or deleting such markers in order to isolate these exceptions (and therefore the inconsistencies) from the rest of the data. Satoh, on the other hand, proposes an abductive technique (a) to help identify these exceptions and the associated pollution markers to add/delete, and (b) to define a revision operator that defines the new specification after an update has been performed. We illustrate the approach via an example taken from [46].

Example 5.

Consider a specification which includes the following constraint:

"Required working hours, R, for a day, for an employee P, must be equal to actual total working hours, T, of all the projects for a day, D".

Now suppose that the specification states that the total working hours is calculated as the sum of the hours for two projects A and B, and that for a particular employee "Bob", his

required hours are 60 and his hours for project B are 30. This specification can be implemented by the following logic program:

$$th(P,D,T) \leftarrow pA(P,D,X), pB(P,D,Y), T = A+B.$$
$$rh(bob,10,60).$$
$$pB(bob,10,30).$$
$$\bot \leftarrow rh(P,D,R), th(P,D,T), R \neq T.$$

where *th(P,D,T)* expresses that the total hours for an emplyee *P* for a day *D* is *T*, *pA(P,D,X)* that the hours for an emplyee *P* for a day *D* on the project *A* is *X*, and similarly for *pB(P,D,Y)* and *rh(P,D,R)* that the required hours for an emplyee *P* for a day *D* is *R*. Note that the last rule is an integrity constraint that defines the constraint given in the specification. Each constraint is then rewritten into a new constraint, following a procedure defined in [46], to include information about possible pollution markers. For instance, the above constraint is rewritten as follows:

$$\bot \leftarrow rh(P,D,R), th(P,D,T), R \neq T, \quad (I1)$$
$$\sim pm(rh(P,D,R),th(P,D,T)), \sim add^*(pm(rh(P,D,R),th(P,D,T))).$$

where *pm* expresses a predicate for the pollution marker and *add** is abducible information.

Suppose now that the leader of project A updates the date with *pA(bob,10,40)*. This update violates the constraints in the specification since it makes the total hours for *Bob* equal to 70 for day 10, while his required hours for day 10 are 60. The abductive proof procedure for logic programming then considers the above integrity constraint as the goal to be satisfied. Therefore, since *rh(bob,10,60), th(bob,10,70), 60≠70* are provable it must be the case that either the predicate *~pm(rh(P,D,R),th(P,D,T))*, or *~add*(pm(rh(P,D,R),th(P,D,T)))* should fail, for the particular instance values *P=Bob*, D=10, R=60 and T=70. Since *add** is a pre-defined abducible predicate, the abductive procedure tries to fail the antecedent *~add*(pm(rh(bob,10,60),th(bob,10,70)))*. This means trying to prove *add*(pm(rh(bob,10,60),th(bob,10,70)))*. This predicate can then consistently be added to the set of abducibles Δ, giving the answer *add*(pm(rh(bob,10,60),th(bob,10,70)))*. The transaction operation on the given specification will then add to the specification the pollution marker:

$$pm(rh(bob,10,60),th(bob,10,70))$$

and the following two constraints:

$$\bot \leftarrow \sim rh(bob,10,60), \sim del^*(pm(rh(bob,10,60),th(bob,10,70))). \quad (I2)$$
$$\bot \leftarrow \sim th(bob,10,70), \sim del^*(pm(rh(bob,10,60),th(bob,10,70))). \quad (I3)$$

This is because the pollution marker states that there is an exception given by the atomic predicates *rh(bob,10,60)* and *th(bob,10,*70), and that if subsequentely there are other changes to the specification for which the constraint in the specification is no longer violated, we will need to delete the pollution marker corresponding to the previous violation. So, for instance, suppose that the leader of the project B deletes *pB(bob,10,30)* and replaces it with *pB(bob,10,20)*. Then the violation of the constraint no longer exists because the total number of hours is now equal to 60. In this case, the abductive proof procedure tries to satisfy each of the above integrity constraints. Constraint (I1) is satisfied because it is not the case that *60≠60*, and constraint (I2) is also satisfied because *~rh(bob,10,60)* fails since *rh(bob,10,60)* is in the specification. The important step is now in the analysis of constraint (I3). The new total number of hours for bob is now 60, so the

predicate *~th(bob,10,70)* succeeds, therefore it must be the case that the predicate *~del*(pm(rh(bob,10,60),th(bob,10,70)))* fails to make the constraint satisfied. Since *del** is also another pre-defined abducible predicate, to fail *~del*(pm(rh(bob,10,60),th(bob,10,70)))* the abductive proof procedure can consistently assume the predicate *del*(pm(rh(bob,10,60),th(bob,10,70)))*. This abducible states that the pollution marker regarding *rh(bob,10,60)* and *th(bob,10,70))* can be deleted. Hence the transaction operation will then delete *pm(rh(bob,10,60),th(bob,10,70))* and the above two integrity constraints (I2) and (I3).

A similar approach has been presented in [48], where the abductive proof procedure for logic programming is also used for consistency management in order to accommodate changes in a specification that can violate required constraints. Specifications are translated into logic programs, as shown above, and constraints rewritten as special integrity constraints, where some auxiliary predicate like *add** and *del** are appropriately added in order to keep trace of the particular instances that violate these integrity constraints. Such instances are then mapped onto assertions to be added or deleted from the specification in order to re-establish consistency, in a similar way to to the addition and deletion of pollution markers to/from specifications. The difference between the last two lines of research work is that the addition and deletion in the latter case is for information already included in the specification, rather then some external artifact like pollution markers. For further details the reader is referred to [48].

5. Conclusion and Future Work

In this paper we have surveyed a number of research results and formal approaches for using abduction in software engineering. We have seen how two different abductive reasoning techniques, based on logic programming and dependency graphs, can facilitate both analysis and revision of specifications. In particular, abductive logic programming has been shown to be suitable for detecting violation of system properties and for identifying diagnostic information as a debugging aid for engineers. This diagnostic capability, typical of abduction, is indeed common to both techniques described in this paper, making them also suitable for supporting automated generation of explanations for a given domain related property. For instance, in the case of graph-based abduction, proof trees can be constructed from some given qualitative theory description of a specifications that provides models or descriptions of how certain OUTput information can be achieved given certain INput data. These descriptions can be seen as explanations for the given OUTput within a pre-defined context (drawn by the given INput). As an additional benefit, because of the ability to construct such explanations consistently the graph-based abductive approach can at the same time identify all or part of the inconsistencies included in the given specification, thereby providing automated support for inconsistency detection.

As a second application of abductive reasoning, we have illustrated the use of abduction for revising specifications, both in the case of evolving inconsistent specifications, and for managing consistency after performing an update. The first type of application is motivated by the fact that, in practice, inconsistency is inevitable in real large-scale specifications [3; 13; 49]. Therefore, living with inconsistency during evolutionary development is a fact of life, and *inconsistency handling* mechanisms need to support

incremental evolution of specifications (i.e. identifying changes that address some specification inconsistencies, while leaving others). The second application on the other hand proposes abduction as a technique for reasoning about how to re-establish consistency in a given specification, once this has been broken by performing some (possibly evolutionary) changes.

A variety of other techniques have been developed for analysing and managing requirement specifications. These range from informal but structured inspections [18], to more formal techniques such as data mining, machine learning, and case-based reasoning. An example of inspection techniques is perspective-based reasoning [50]. This is a scenario-based technique that provides procedural guidance on how to inspect requirements documents written in English. The identification of faulty requirements in this technique is based on answering specifically tailored questions during the development of the specifications. These questions are themselves based on a predefined taxonomy of errors such as ambiguous information. Abductive reasoning is a much more general analysis technique, which can be appropriately tailored to analyse any type of specific domain-dependent system property. Moreover, the rigor of the underlying reasoning process guarantees a complete description of existing errors. Data mining [31] and machine learning [51] are known as techniques for *knowledge discovery*. This means, given a sample of data (in the case of data mining) or positive/negative examples (in the case of machine learning), these techniques are able to abstract such data into new rules. These can subsequently, and if necessary, be added to the given knowledge base. Abduction does not generate new rules. It only defines how to instantiate the truth value of a special set of propositions, so that the resulting model of the specification would satisfy a given observation/goal. Hence, machine learning and data mining are in their nature unsound reasoning processes, whereas abduction is a fully sound and consistent reasoning technique. Case-based reasoning [32] is another example of a (semi-)formal technique used for understanding problems and proposing solutions. This technique differs, however, from abduction in the fact that it assumes the existence of a large library of past-cases; these cases are then used either to generate solutions or as examples to illustrate an existing problem. Abduction, on the other hand, requires only the specification and the invariants of a system to perform its analysis.

Finally, abductive reasoning also differs from other formal techniques like those based on model checking or theorem proving [10] or logic-based approaches (e.g. [35; 47; 52]). The results illustrated in this paper therefore provide some valuable foundations for the use of abduction in software engineering, as well as raising some interesting and challenging research issues for future work. The ability to perform automated analysis and generation of counterexamples when errors are detected makes abduction comparable to other existing formal techniques such as those based on model checking [1; 8; 9; 10], and also highlights some interesting differences. Model checking techniques often require complete description of the initial state(s) of the system in order to compute successor states, and the application of abstraction techniques to reduce the size of the state space. In contrast, abduction doesn't necessarily rely on a complete description of some initial system state, as shown in the case study given in [45]. Moreover, because of the goal- or property-driven characteristic of the abductive techniques it does not require abstraction and it can support reasoning about specifications of systems whose state-spaces may be infinite. However, the practical application of abduction to specification analysis in software engineering is still at an early stage of development. Further research is needed to extend existing results in order to cover a wider spectrum of specification analyses that take into account various forms of system

properties as well as different types of specifications. This will consolidate the existing theoretical foundations for the development of efficient, logic-based methods for automated analysis of specifications using abduction. Their practicality will then need to be measured using a wide range of case studies.

As far as specification handling is concerned, the use of abduction is still at its initial stages of development. A variety of future directions can be taken from the existing results shown in this chapter. In particular, the development of a framework that combines abduction with induction might be useful. The identification of possible changes on a specification by abduction can only be related to single assertion. However, the counterexamples provided by abductive reasoning could be used by some inductive reasoning process in order to identify, by means of a learning process, more elaborate changes to perform on a given specification. Appropriate interleavings between abductive and inductive reasoning could provide a sound, logic-based approach to the specification management process.

Acknowledgements

Many thanks to Tim Menzies and Ken Satoh for providing useful insights about their respective abductive approaches, to Tony Kakas for useful discussions about abduction and abductive tools, and to Jeff Kramer for useful feedback and suggestions about earlier drafts of this paper. This work was partially funded by the UK EPSRC projects MISE (GR/L 55964) and VOICI (GR/M 38582).

References

[1] Anderson, R., et al. (1996). Model Checking Large Software Specifications. *ACM Proceedings of 4th International Symposium on the Foundation of Software Engineering*.

[2] Atlee, J. M., and Gannon, J. (1993). State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, 19(1): 24-40.

[3] Balzer, R. (1991). Tolerating Inconsistency. *Proceedings of ICSE-13*, pp. 158-165.

[4] Bharadwaj, R., and Heitmeyer, C. (1997). Model Checking Complete Requirements Specifications Using Abstraction , *Technical Report* No. NRL-7999, Naval Research Laboratory.

[5] Brewka, G. (1991). *Non-monotonic reasoning: Logical Foundations of Commonsense*. Cambridge, Great Britain: Cambridge University Press.

[6] Clancey, W. J. (1992). Model Construction Operators. *Artificial Intelligence*, 27: 289-350.

[7] Clark, K. (1978). Negation as Failure. In H. Gallaire, and Minker, J. (Eds.), *Logic and Data Bases* (pp. 293-322). New York: Plenum.

[8] Clarke, E. M., Grumberg, O., and Long, D.E. (1994). Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5): 1512-1542.

[9] Clarke, E. M., Grumberg, O., and Long, D.E. (1996). Model Checking. In M. Broy (Eds.), *Deductive Program Design*, *Proc. NATO ASI Series F*, Springer.

[10] Clarke, M., and Wing, M. (1996). Formal Methods, State of the Art and Future Directions. *ACM Computing Surveys*, 28(4): 626-643.

[11] Console, L., Sapino, M.L., and Theseider Dupre, D. (1994). The Role of Abduction in Database view Updates. *Journal of Intelligent Systems*.

[12] Console, L., Portinale, L., and Theseider Dupre, D. (1996). Using Compiled Knowledge to Guide and Focus Abductive Diagnosis. *IEEE Transaction on Knowledge and Data Engineering*, 8(5): 690-706.

[13] Cugola, G., Di Nitto, E., Fuggetta, A., and Ghezzi, C. (1996). A framework for formalising inconsistencies and deviations in human-centred systems. *ACM Transactions on Software Engineering and Methodology*, 5(3): 191-230.

[14] Easterbrook, S., and Nuseibeh, B. (1995). Inconsistency Management in an Evolving Specification. *Proceedings of the 2nd International Symposium on Requirements Engineering*, pp. 48-55.

[15] Esghsi, K. (1988). Abductive Planning with the Event Calculus. *Proceedings of International Conference on Artificial Intelligence*, 1, pp. 3-8.

[16] Eshghi, K., and Kowalski, R.A. (1989). Abduction compared with negation as failure. *Proceedings of the 6th International Conference on Logic Programming*, Lisbon, pp. 234-255.

[17] Finkelstein, A., et al. (1994). Inconsistency Handling in Multi-Perspective Specifications. *IEEE Transaction on Software Engineering*, 20(8): 569-578.

[18] Gilb, T., and Graham, D. (1993). *Software Inspection*. Addison-Wesley.

[19] Heitmeyer, C. L., Labaw, B., and Kiskis, D. (1995). Consistency Checking of SCR-style Requirements Specifications. *Second International Symposium on Requirements Engineering*, York, pp. 27-29.

[20] Heitmeyer, C. L., et al. (1998). Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications. *IEEE Transaction on Software Engineering*, 24(11): 927-947.
[21] Hietmeyer, C. L., et al. (1998). SCR*: A Toolset for Specifying and Analysing Software Requirements. *Computer-Aided Verification*, Canada.
[22] Hogger, C. (1990). *Essentials of Logic Programming*. Clarendon Press, Oxford.
[23] Hogger, C., and Kowalski, R.A. (1991). Logic Programming. In S. C. Shapiro (Eds.), *Encyclopedia of Artificial Intelligence (Second Edition)* New York: John Wiley & Sons.
[24] Hunter, A., and Nuseibeh, B. (October 1998). Managing Inconsistent Specifications: Reasoning, Analysis and Action. *ACM Transactions on Software Engineering and Methodology*.
[25] Inoue, K., and Sakama, C. (1995). Abductive Framework for Non-monotonic Theory Change. *International Joint Conference on Artificial Intelligence*, 1, pp. 204-210.
[26] Jackson, D., and Rinard, M. (2000). Software Analysis: a Roadmap. In A. Finkelstein (Eds.), *The Future of Software Engineering* ACM Press.
[27] Jacky, J. (1995). Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 2: 99-106.
[28] Kakas, A. C., and Mancarella, P. (1990). Database Updates Through Abduction. *Proceedings of 16th International Conference on Very Large Database*, Brisbane, Australia.
[29] Kakas, A. C. (1998). Abductive Constraint Logic Programming <http://www.cs.ucy.ac.cy/aclp/index.html>
[30] Kakas, A. C., Kowalski, R.A., and Toni, F. (1998). The Role of Abduction in Logic Programming. In D. M. Gabbay, Hogger, C.J., and Robinson, J.A. (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming* (pp. 235-324). Oxford University Press.
[31] Korth, S., and Sudarshan *Database System Concepts* (Third Edition). Mc-Grow Hill Companies Inc.
[32] Leake, D. (1996). *Case-Based Reasoning*. AAAI Press/ The MIT Press.
[33] Magee, J., and Kramer, J. (1999). *Concurrency: State Models & Java Programs.* John Wiley & Sons Ltd.
[34] McMillian, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers.
[35] Menzies, T. (1996). Applications of Abduction: Knowledge Level Modeling. *International Journal of Human Computer Studies*, 45: 305-355.
[36] Menzies, T., and Compton, P. (1997). Applications of Abduction: Hypothesis Testing of Neuroendocrinological Qualitative Compartmental Models. *Artificial Intelligence in Medicine*, 10: 145-175.
[37] Menzies, T., Easterbrook, S., Nuseibeh, B., and Waugh, S. (1999). An Empirical Investigation of Multiple Viewpoint Reasoning in Requirements Engineering. *Proceedings of 4th IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, 7-11 June.
[38] Menzies, T. J. (1995) *Principles of Generalised Testing of Knowledge Bases*. PhD thesis, University of New South Wales.
[39] Newell, A. (1982). The Knowledge Level. *Artificial Intelligence*, 18: 87-127.
[40] Nuseibeh, B. (1996). To Be and Not To Be: On Managing Inconsistency in Software Development. *Proceedings of the 8th international Workshop on Software Specifications and Design*, pp. 164-169.

[41] Nuseibeh, B., and Russo, A. (1999). Using Abduction to Evolve Inconsistent Requirements Specifications. *Austrialian Journal of Information Systems, Special Issue on Requirements Engineering. ISSN:1039-7841*: 118-130.

[42] Owre, S., Rushby, J., and Shankar, N. (1992). PVS: A prototype verification system. *Proceedings of the 11th Conference on Automated Deduction*, Vol 607 of Lecture Notes in Artificial Intelligence, pp. 748-752.

[43] Pierce, C. S. *Collected Papers of Charles Sanders Pierce* (Hartshorn *et al.*). Harvard University Press.

[44] Reiter, R. (1978). *On closed world databases*. New York: Plenum Press.

[45] Russo, A., et al. (2000). An Abductive Approach for Handling Inconsistencies in SCR Specifications. *Proceedings of (ICSE2000) International Workshop on Intelligent Software Engineering*, Limerick.

[46] Satoh, K. (1998). Adding and Deleting Pollution Marker by Abductive Logic Programming. *Proceedings of the Asia and Pacific Rim Workshop on Intelligent Software Engineering*, pp. 48-53.

[47] Satoh, K. (1998). Computing Minimal Revised Logical Specification by Abduction. *Proceedings of International Workshop on the Principles of Software Evolution*, pp. 177 - 182.

[48] Satoh, K. (2000). Consistency Management in Software Engineering by Abduction. *Proceedings of (ICSE2000) International Workshop on Intelligent Software Engineering*, Limerick.

[49] Schwanke, R. W., and Kaiser, G.E (1988). Living with Inconsistency in Large Systems. *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, Germany.

[50] Shull, F., Rus, I., and Basili, V. (2000, July 2000). How Perspective-Based reading Can Improve Requirements Inspections. *Computer*, p. 73-79.

[51] van Lamsweerde, A. (1991). Learning Machine Learning. In A. Thayse (Eds.), *Introducing a Logic Based Approach to Artificial Intelligence* (pp. 263-356). Wiley.

[52] van Lamsweerde, A., Darimont, R., and Letier, E. (1998). Managing Conflicts in Goal-Driven Requirement Engineering. *IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development.*

[53] van Lamsweerde, A., and Letier, E. (2000). Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Transactions on Software Engineering, Special Issue on Exception Handling*, 26.

[54] Yost, G. R., and Newell, A. (1989). A Problem Space Approach to Expert Systems. *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 621-627.

[55] Zave, P., and Jackson, M. (1997). Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1): 1-30.

---

[1] Throughout the chapter we will not make any distinction between a system and a software system; these terms are used interchangebly.

[2] The symbol $\Rightarrow$ denotes the particular reasoning process: deduction, induction or abduction.

[3] Recall that a reasoning mechanism is *non-monotonic* if the addition (or deletion) of new information to (from) a specification does not necessarily preserve the set of derivable information (e.g., [5; 41].).

[4] In this specific case, the domain description would also cause the resoning process to loop. Techniques for addressing loop problems in abduction can be found in [30].

[5] These consistency rules reflect, respectively, the standard notion of classical inconsistency and the property of "completion", which is typical of the logic programming semantics. Note that abductive

proof procedures for other types of logic with different underlying semantics may not necessarily need to include such consistency constraints.

[6] Note that in the super-theory given in Figure 3 the STEADY vetices are essentially end points, points of inconsistencies, and therefore cannot be used to explain other information (i.e. they don't have any children in the dependency-graph representation).

[7] This feature corresponds to the notion of a consistent abductive explanation, which is stated in condition 2 of Definition 1.

[8] This is the main reason why existing abductive reasoning mechanisms are proof procedures that always terminate.

[9] The symbol \ denotes the standard operation of subtraction between sets. Note that either Δ^+ or Δ^- can be empty sets, and that $\Delta^+ \cap \Delta^- = \varnothing$.

[10] The word inconsistency means in this case violation of consistency rules. Of course if the consistency rule is of the form $A \wedge \neg A \rightarrow \bot$, then inconsistency means also logical inconsistencies.

[11] Trivialisation in classical logic is the inference of arbitrary information from an inconsistent specification.

[12] The content of the predicate *HoldsS* should be read as a constant name.