Integrating Model Checking and Inductive Logic Programming

Dalal Alrajeh[1], Alessandra Russo[1], Sebastian Uchitel[1,3], and Jeff Kramer[1]

[1] Imperial College London
{da04,ar3,su2,jk}@doc.ic.ac.uk
[2] University of Buenos Aires/CONICET
s.uchitel@dc.uba.ar

Abstract. In this paper we argue that Inductive Logic Programming can provide automated support to correct errors identified by model checking, and that model checking problems provide the relevant context for learning hypotheses that are meaningful within the domain being studied. We present a general framework for such integration, discuss its main characteristics and present an overview of its application.

1 Introduction

Model Checking is an automated technique for verifying formal artefacts. It has been successfully used to verify system designs and properties in a variety of application domains, ranging from hardware and software systems to biological systems. A model checker requires a model provided in some formal description language and a semantic property that such model is expected to meet. The model checker then automatically checks the validity of the specified property in the model semantics. If the property is found to not hold, a counterexample is generated which shows how the property can be falsified.

The automatic generation of counterexamples is one of model checking's powerful features for system fault detection. Counterexamples are meant to help engineers in the tasks of identifying the cause of a property violation and correcting the model. However, these tasks are far from trivial with little automated support. Even in relatively small models such tasks can be very complex since (i) counterexamples are expressed in terms of the model semantics rather than the modelling language, (ii) counterexamples show the symptom and not the cause of the violation and (iii) any manual modification to the model may fail to resolve the problem and even introduce violations to other desirable properties.

Inductive Logic Programming (ILP), on the other hand, is at the intersection of inductive learning and logic programming and is concerned with learning general principles (in the form of logic programs) that explain (positive and negative) observations with respect to an existing background knowledge also expressed as a logic program. As the search space for hypotheses might be very large (sometimes infinite), ILP methods make use of a language bias that constrains the search by defining the syntactic structure of the hypotheses to be computed. However, identifying a priori the precise relationship between background knowledge, observations and language bias that would lead to the most relevant hypotheses within the given domain remains a difficult task.

In this paper, we argue that model checking and ILP can be seen as two complementary approaches with much to gain in their integration – model checking providing ILP with a precise context for learning the most relevant hypotheses in the domain being studied, and ILP supplying model checking with an automatic method for learning corrections to models. We discuss how this integration can be achieved, address some of the relevant issues that need to be taken into consideration and summarise its application to the problem of goal-oriented requirements elaboration.

2 Model Checking

The process of model checking comprises three main tasks: modelling, specification and verification [3]. *Modelling* is the process of describing in some formal modelling language an artefact that is to be reasoned about. In the context of software systems, the artefact is typically related to a system's design or requirements. The formal language is normally the input to a model checker (e.g. a process algebra such as FSP [10]) or one that can be automatically translated into it (e.g. scenario notations such as MSC [7]). The semantics of the formal language is given in terms of a state-based *semantic domain* such as Labelled Transition Systems (LTSs), Kripke structures or Büchi automata. The formal description resulting from the modelling process is referred to as the *model*.

The *specification* task involves formally stating the semantic properties that the model must satisfy. The specification language is usually some logical formalism. In software engineering, temporal logics such as Linear Temporal Logic (LTL) and Computation Tree Logic are commonly used.

As part of the *verification* process, the model checker automatically checks whether the model semantics satisfies the specified property. In other words, it verifies that the system behaviour captured in the model semantics $\sigma(M)$ of the given model M satisfy the specified property S. This is denoted as $\sigma(M) \models S$. When this is not the case, the model checker returns a *counterexample*, i.e. a system behaviour that violates the property. For instance, in the case of an LTS model and an LTL property, a counterexample takes the form of a trace, i.e. sequence of event labels. A model checker may also be used to generate *witnesses* of a property, which are behaviours in $\sigma(M)$ that satisfy the property.

3 MoCIL: An Integration Approach

A general ILP task [11] can be defined as the task of constructing hypotheses H that, together with a given background knowledge B, explain a given set of positive and negative observations $O^+ \cup O^-$, namely:

$$B \wedge H \models O^+ \quad \text{and} \quad B \wedge H \not\models O^- \tag{1}$$

The meaning of $\models$ and the exact relationship between B, H O^+ and O^- rely on several factors which include the representation language used to express them (e.g. definite or non-monotonic logic programs), its semantics and the particular chosen inductive framework [13]. The integration of an ILP task with model checking has to operate at the semantic level of the two reasoning tasks, for the learning process to be able to reason over consequences of the given model

and learn solutions for correcting the model. The logic programming formalism and semantics of the ILP task need, therefore, to capture the model semantics, properties, counterexamples and witnesses of the model checking task, and the learned hypotheses has to suggest solutions in the modelling language on how to change the models. We discuss here the main properties that an integration of ILP and model checking has to address: *soundness* of the integration, *correctness* of the inductive solution with respect to counterexamples and witnesses, and *completeness* of the solutions with respect to the specified property.

Property 1: Soundness of the integration. The first step of the integration is the correspondence of the semantics governing the model checking and the ILP tasks. Models and model semantics of the model checking task are represented in the form of a logic program that preserves the satisfiability relation of the model semantics. Such a correspondence is captured by a translation function τ that has to satisfy the following soundness theorem.

Theorem 1. *Let M be a model with a model semantics $\sigma(M)$. Let τ be a translation function from the modelling and specification languages to a logic program formalism. Then, τ is a sound translation if and only if for any expression P in the specification language if $\sigma(M) \models P$ then $\tau(M) \models \tau(P)$.*

Note that for the above to hold, τ must express in the logic program all semantic considerations of the entailment $\sigma(M) \models P$. For instance, if M is written using FSP as a modelling language, its model semantics $\sigma(M)$ is an LTS and the specification language is LTL, then τ needs to capture, among other aspects, the closed world assumption underlying the LTS/LTL entailment.

Property 2. Correctness of an inductive solution. In ILP, a solution to an inductive task is a hypothesis H that satisfies condition (1), where the meaning of $\models$ is given by the underlying logical programming semantics and the chosen inductive framework. Sakama and Inoue [14] have identified several inductive frameworks under which a hypothesis can constitute a solution to an inductive problem. These include explanatory, brave, cautious, and learning from satisfiability. The choice of the inductive framework affects the set of acceptable solutions an ILP system may compute and hence the correctness of these solutions with respect to the detected counterexamples.

The aim of the integration of ILP and model checking is to reduce the set of counterexamples that a model M covers for a given property S to the empty set. In other words, it is concerned with computing a refined model $M \cup \tilde{M}$ that no longer includes the counterexamples, but preserves the witnesses of S in $\sigma(M)$.[3] Given a model M, a property S consistent with M, a counterexample ψ^- and a witness ψ^+ of S, a *correct extension* of M with respect to ψ^+ and ψ^- is a model $\tilde{M}$ so that $\psi^- \notin \sigma(M \cup \tilde{M})$ and $\psi^+ \in \sigma(M \cup \tilde{M})$.

The task of finding $\tilde{M}$ can be expressed as an ILP task. The model M constitutes the core of the background knowledge B, the property S is an integrity constraint I that has to be satisfied by the learned hypotheses, the counterexample and witness(es) represent the negative and positive observations of the learning

[3] We consider the case where errors in a model can be corrected by extensions only.

respectively. In this context, a solution H is a logic program that does not entail the negative observations under cautious induction, written $B \wedge H \not\models_c O^-$, whilst it entails the positive observations under brave induction, written $B \wedge H \models_b O^+$, and it is consistent with the integrity constraints I. We refer to such solutions as *correct inductive solutions* of O^+ and O^- with respect to B and I. The relationship between correct extensions and correct inductive solutions is captured by the following theorem.

Theorem 2. *Let M be a model, S a property, ψ^- a counterexample and ψ^+ a witness such that $\psi^- \in \sigma(M)$ and $\psi^+ \in \sigma(M)$. Let $B = \tau(M)$, $I = \tau(S)$, $O^- = \tau(\psi^-)$ and $O^+ = \tau(\psi^+)$ and L a language bias. An hypothesis H is a correct inductive solution of O^+ and O^- with respect to B and I if and only if $\tau^{-1}(H)$ is a correct extension of M with respect to ψ^+ and ψ^-.*

Note that in the above theorem we assume that the ψ^- and ψ^+ can be characterised in the specification language and hence can be translated using τ. For example a trace e_1, e_2, ..., e_n can be described in LTL as $\bigwedge_{i=1}^{n} \bigcirc^i e_i$.

Property 3. Completeness of Solutions. Although the above notion of correctness of an ILP solution H may guarantee that the detected counterexample is no longer covered by $\sigma(M \cup \tau^{-1}(H))$, it does not guarantee that this extended model satisfies the specified property S. To ensure the satisfaction of the property, all counterexamples to S have to be removed while witnesses must be preserved. This is achieved through iterative steps of correct extensions. Assuming that there exists such an extension $\tilde{M}$ for which $\sigma(M \cup \tilde{M}) \models S$, several iterations may be required, where at each iteration i, (for $i \geq 0$) the newly extended model $M \cup \tau^{-1}(H_1) \cup .... \cup \tau^{-1}(H_i)$ is verified against the specified property S and if a counterexample is detected, a new extension $\tau^{-1}(H_{i+1})$ is learned that eliminates it.

Although the number of counterexamples maybe infinite, it may be sufficient to learn solutions that are correct inductive solutions with respect to a finite set of finite counterexamples. Eliminating such set would have to guarantee elimination of all counterexamples in the model semantics. We refer to this set as the *set of counterexample characterisations*. For instance, such a set exists when using safety-LTL as the specification language and LTS as the model semantics. The following theorem captures the completeness property of a set of correct extensions $(\tau^{-1}(H_1) \cup .... \cup \tau^{-1}(H_m))$ for the verification problem $\sigma(M) \models S$.

Theorem 3. *Let M be a model and S a property consistent with M. Given a set $\{\psi_i^-\}$ of counterexample characterisations and a set $\{\psi_j^+\}$ of witnesses of S, then there exists a set $\{H_i\}$ of correct inductive solutions of $\tau(\{\psi_i^+\})$ and $\tau(\{\psi_i^-\})$ with respect to $\tau(M)$ and $\tau(S)$ such that $\sigma(M \cup \tau^{-1}(H_1) \cup .... \cup \tau^{-1}(H_m)) \models S$.*

4 Problem solving using MoCil

We have successfully applied the MoCIL approach to solve various problems within the software engineering domain including goal-oriented requirements elaboration [1] and model transition system refinement [2]. Though these differ in the input languages, semantics and the specific class of correct extensions, they

share a number of characteristics. Their models describe event-based systems, with a model semantics expressed in terms of finite-state transition systems, which can be represented as normal logic programs. Due to lack of space, we briefly summarise here the application to the first.

Goal-oriented Requirements Elaboration refers to the process of identifying explicit constraints, called operational requirements, on the operations to be performed by a software so that the system only behaves in a manner that satisfies the given goals. In [1] we have showed how model checking can be used to detect incompleteness of operational requirements (i.e. models) with respect to goals (i.e. properties), and how ILP can be used to learn operational requirements (i.e correct extensions) that are complete with respect to the given goals. Table 1 instantiates the concepts discussed above in this problem domain.

In brief, the verification problem is concerned with checking whether an LTS, i.e. the model semantics of a given operational requirements model R with domain knowledge D, expressed in Fluent Linear Temporal Logic (FLTL), satisfies given goal properties G, also expressed in FLTL, i.e. $\sigma(D \cup R) \models G$, where $\models$ is interpreted as the FLTL satisfaction relation [6]. When this is not the case, the LTSA model checker [10] produces the shortest trace ψ^- where the goal G is violated. The LTSA is also used to generate a witness ψ^+ of G. The task is then to compute a correct extension $\tilde{R}$ of operational requirements such that $\sigma(D \cup R \cup \tilde{R}) \models G$.

The model $D \cup R$, counterexample ψ^- and witness ψ^+ are encoded into an EC program. The choice of EC is influenced by its similarity to FLTL, where it allows explicit representation of and reasoning about event occurrences and fluent values at different (time) points over a linear time structure. Furthermore, as requirements are expressed as formulae with negated literals and the valuations of fluents in traces of the LTS consider a close world assumption, the soundness of the integration is with respect to EC normal logic programs with negation as failure and stable model semantics. The generated program is locally stratified, and therefore has a single stable model. The soundness property can be proven by showing that for any fluent f in the FLTL language and position i in a trace σ in the LTS, if $\sigma, i \models f$, then the atom *holdsAt(f,i,σ)* is also true in the stable model of $\tau(R \cup D) \ \wedge \ \tau(\psi)$ (Theorem 5.1 in [1]).

An ILP system based on the XHAIL algorithm [12] has been used to compute correct extensions with respect to ψ^- and ψ^+. The language bias is defined to capture clauses that correspond to the syntax of operational requirements expressions. The system computes multiple correct inductive solutions. These are translated back into FLTL; one is then selected and added to the initial model $R \cup D$; the next iteration of the process is then started. At each iteration it is guaranteed that the inverse translation of the chosen H (i.e. $\tau^{-1}(H)$) is a correct extension of the correct model with respect to the counterexample ψ^- and witness ψ^+ (Theorem 6.1 in [1]). In [1] it is shown that this process terminates and a complete set of correct extensions has been learned for which $D \ \cup \ R \ \cup \ \tau^{-1}(H_1) ... \cup \tau^{-1}(H_n) \models G$.

Concept	**Instantiation**
Model	Operational requirements and domain properties [9] in FLTL [6]
Model Semantics	Maximal (w.r.t. traces) deterministic Labelled Transition System [6]
Property	Time-bounded achieve goals [4] expressed in FLTL
Counterexample	Shortest trace violating a goal
Witness	Finite trace that satisfies the goals
Logic Program	Event Calculus (EC) [8] normal logic programs
LP models	Stable model semantics [5]

Table 1. Concepts in MoCil applied to the requirements elaboration problem.

5 Conclusion and Future Work

In this paper we have briefly presented a framework for integrating model checking and ILP. We have summarised an instantiation of the framework to the problem domain of goal-oriented requirements elaboration. We believe that this integrated framework can be used to a variety of problems in different application domains including bioinformatics, business process modelling, network and security managements and normative systems.

References

1. D. Alrajeh. *Requirements Elaboration using Model Checking and Inductive Learning*. PhD thesis, Imperial College London, London, United Kingdom, 2010.
2. D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. An inductive approach for modal transition system refinement. In *Technical Communications of the 27th Intl. Conf. on Logic Programming*, pages 106–116, 2011.
3. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
4. A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1):3–50, 1993.
5. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R.A. Kowalski and K. Bowen, editors, *Proc. of 5th Intl. Conf. on Logic Programming*, pages 1070–1080, 1988.
6. D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proc. of the 9th European softw. Eng. conf.*, pages 257–266, 2003.
7. ITU. *Message Sequence Charts*. Intl. Telecommunications Union, Telecommunication Standardisation Sector, 1996.
8. R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New generation Comp.* , 4(1):67–95, 1986.
9. E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In *Proc. of 10th ACM SIGSOFT Symp. on Foundations of Softw. Eng.*, pages 119–128, 2002.
10. J. Magee and J. Kramer. *Concurrency : State Models and Java Programs*. John Wiley and Sons, 1999.
11. S. Muggleton and L. de Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20(Supplement 1):629 – 679, 1994.
12. O. Ray. Nonmonotonic abductive inductive learning. *J. of Applied Logic*, 7(3):329–340, 2009.
13. C. Sakama and K. Inoue. Brave induction. In *Proc. of the 18th Intl. Conf. on Inductive Logic Programming*, pages 261–278, 2008.
14. C. Sakama and K. Inoue. Brave induction: a logical framework for learning from incomplete information. *Machine Learning*, 76:3–35, 2009.