

# A Light-Weight Integration of Automated and Interactive Theorem Proving

KARIM KANSO<sup>1†</sup> and ANTON SETZER<sup>2‡</sup>

`{cskarim1, A.G.Setzer2}@swansea.ac.uk`

*Department of Computer Science  
Swansea University  
UK*

*Received January 2011*

In this paper we present a novel connection between automated and interactive theorem proving paradigms. This technique allows for a powerful interactive proof framework that facilitates efficient verification of finite theorems and guided construction of the proof of infinite theorems. Such situations typically occur with industrial verification. As a case study, an embedding of SAT and CTL model checking is presented, both of which have been implemented for the dependently typed proof assistant Agda.

## 1. Introduction

Martin-Löf dependent type theory offers a powerful mechanism to construct mathematical formulæ and write functional programs (Nordström, Petersson & Smith 1990); it is essentially typed  $\lambda$ -calculus with the dependent product and algebraic data-types. By the Curry-Howard correspondence (Curry 1934, Curry, Feys, Craig & Craig 1958, Howard 1980), propositions can be represented as types, where an element of the type is a proof of the proposition. Another perspective in type theory is that a type is a specification of a problem such that its elements are programs that satisfy the specification.

In this paper, we investigate and actualise an embedding of automated theorem proving (ATP) decision procedures into Martin-Löf dependent type theory. The motivation is twofold. Firstly to integrate Agda (Bove, Dybjer & Norell 2009) with fast external tools that facilitate feasible verification of large finite problem sets, archetypal of industrial verification. Secondly to explore a functional proof framework where finite (or finitisable) components of a theorem are proved automatically and the infinite components are proven with human guidance.

Mature proof assistants such as Isabelle and COQ have supported external tools for many years (Böhme & Nipkow 2010, Müller & Nipkow 1995, Boutin 1997); whereas

<sup>†</sup> The first author is partly supported by Invensys Rail Systems, UK.

<sup>‡</sup> The second author is supported by EPSRC grant EP/G033374/1, theory and applications of induction-recursion.

Agda, a less mature tool does not yet. Agda’s maturity has provided an opportunity for the community to experiment with new approaches (some official and some not) regarding many theoretical aspects of Interactive Theorem Proving (ITP) tools. The new approaches taken in Agda have resulted in it being an intuitive proof assistant which is also a programming language; programs and proofs are defined using the same functional constructs that can be compiled and executed. We believe that Agda can be extended into a platform for the development of verified software.

Our wider ambition is to have a substantial program (such as a control system) which executes in Agda and is proven to be correct. In this scenario, concrete finite theories need to be proven to show the system satisfies some property, and abstract theories need to be proven that show correctness of the verification techniques.

To clarify the situation, consider the development of a critical system that must satisfy a number of *safety principles* (lemmata) from the target domain. It could be possible to prove in general using ITP that these *safety principles* imply the actual “safety” of the system, where the concept of “safety” still remains to be validated by domain experts and can be modelled mathematically. For instance, within the train domain where *safety principles* are *signalling principles* it might be possible to prove that the *signalling principles* imply “trains do not collide or derail”. So instead of validating all the *safety principles*, some of them have been verified. It might be the case that more *safety principles* are required, or that some are redundant. Thus, part of the validation procedure has become a verification procedure, reducing the total amount of validation required. Then, as a concrete step, develop in Agda and verify with ATP that the system satisfies the *safety principles*. Thus the actual “safety” of the system has been verified in Agda, not only the *safety principles*.

### 1.1. Related Work

Within the ITP community the issue of automatically solving problem sets has been studied many times (Bierman, Gordon, Hrițcu & Langworthy 2010, Böhme & Nipkow 2010, Boutin 1997, Fontaine, Marion, Merz, Nieto & Tiu 2006) and many more. Three existing approaches have been identified during this research, none of which categorises our approach.

**Oracle** The use of an oracle, which is an operation that provides a proof of a theorem, and which when invoked calls an external tool. See for instance approaches by Tverdyshev, Müller and Nipkow in Isabelle; and Bierman, Gordan and Langworthy in M (Müller & Nipkow 1995, Bierman et al. 2010). The problem with this approach is that the result of the oracle does not reduce to head normal form, and therefore destroys the ability to execute programs. Another problem is that it is not clear whether the external tool is correct and whether the result of the external tool was interpreted correctly.

**Reflection** One verifies the correctness of a decision procedure and extracts from this proof a verified ATP tool (Boutin 1997). The extracted program is called by a tactic to build a proof. This approach has for instance been taken by Verma; Hendriks; and Lescuyer and Conchon (Verma 2000, Hendriks 2002, Lescuyer & Conchon 2008), and has been widely applied within the COQ community. Reflection requires a correctness

proof for an ATP tool in the ITP tool; proving the correctness for state-of-the-art theorem provers would be cumbersome and one would expect the state-of-the-art tools to have significantly improved by the time the proof is complete.

**Oracle with Justifications** One uses an ATP tool which provides in case of a positive answer a proof object, which can then be translated automatically into an ITP proof of the corresponding ITP formula. Since the ITP proof is now checked, the correctness relies entirely on the correctness of the ITP checker. Furthermore, a proof object is kept and therefore the ATP has to be executed only once. This approach has been used for instance by (Paulson & Susanto 2007, Böhme & Nipkow 2010, Dong, Ramakrishnan & Smolka 2003, Weber 2006, Fontaine et al. 2006). Notably Simon Foster has used this technique to connect Agda with Waldmeister (Foster & Struth 2011). Problems are that creating the proof object slows down the ATP tool, and that many ATP tools do not provide a proof object. Furthermore the proofs provided by the ATP tool might be very big (proof sizes of several hundreds of Megabytes have been reported for Satisfiability Modulo Theories (SMT) solving (Stump 2009)). Therefore type-checking translated ITP proofs might be infeasible.

A good introduction to the various different flavours of theorem proving can be found in (Harrison 2008), and a more technical discussion in (Boutin 1997). A review of formal methods relating to industrial projects can be found in (Woodcock, Larsen, Bicarregui & Fitzgerald 2009).

## 1.2. Our Approach

**Oracles and Reflection** Assume a logic (such as propositional), define the set of formulae and a satisfaction relation with respect to the logic. Then the decision procedure is implemented naïvely within Agda’s logic, and proved to be correct with respect to the satisfaction relation. In Agda, a function’s implementation can be overwritten by a native Haskell implementation (after checking that the function fulfils a number of axioms); using this fact, the inefficient decision procedure is replaced by a call to an external ATP tool. Evaluation of the decision procedure is as follows: if applied to a closed term, the efficient ATP tool is executed; if applied to an open term, the naïve (inefficient) Agda implementation is evaluated. Since the correctness proof refers to open terms, it refers to the Agda implementation and not the ATP tool. If the resulting proof object is inspected, it is lazily evaluated using the naïve implementation; otherwise it behaves as if it has been postulated. Any transformations required into the external tool’s input language are defined within Agda’s logic, thus in case of non-trivial transformations it is possible to prove that the input to the tool is correct<sup>†</sup>. To increase trustworthiness, the input and outputs of the ATP tool are placed in the log window, allowing a user to manually verify the translation and the tool’s result are correct. Our approach yields a high level of soundness and efficiency when using certified ATP tools. See Section 2.2 for technical details on the embedding.

<sup>†</sup> For instance, in the case of CTL model checking (see Section 2.4) the transition relation is made total before executing the external tool, and proven to preserve correctness.

### 1.3. Comparison with Existing Approaches

Obviously, compared with only using an oracle the approach has the advantage that proofs normalise and programs can be extracted.

Pure reflection has the highest level of soundness of all approaches as no external tools are used. In comparison, our approach, in the case of open terms is equivalent; but in the case of closed terms weakens soundness and significantly increases in efficiency.

The third approach where the external ATP tool provides a justification is motivated by the fact that in many cases, non-trivial translations into the ATP tool's input language are defined outside the logic of the ITP tool making it hard to prove that they preserve correctness (Fontaine et al. 2006). Instead with our approach these translations are defined inside Agda, mitigating this requirement. It follows from this, that our approach is more efficient and offers a greater flexibility with respect to the choice of the ATP tool. The ATP tool can range from unverified state-of-the-art tools to certified but technologically less advanced tools.

In all four approaches there will always be a degree of trust (meta-mathematics, Gödel's Incompleteness Theorem), also the hardware, operating system and the correctness of theorem prover compilation are important issues.

1.3.1. *Soundness.* We believe that the approach is sound up to executing the correct ATP tool in-situ of the decision procedure and the correctness of the tool its self. This is because the decision procedure has been verified with respect to correctness of the chosen ATP theory and translations into the ATP tool's input language are verified. A (Maybe Boolean) valued result is obtained from the tool meaning that it is not known if the correct tool was executed; to mitigate this the tools output is placed in the log (window) and it is possible to view the input given to the tool. Of course, in an ideal solution it would be possible to query the tool about what ATP theory it realises.

Even when the ATP tool has not been certified, but is widely used we believe does not weaken soundness significantly as these tools are as trustworthy (if not more) as Agda in its current state.

### 1.4. Overview

Section 2.1 provides a comparison of automated and interactive theorem proving techniques and an overview of finite and infinite theories, Section 2.2 introduces the technique used for embedding ATP theories into type theory. Section 2 then concludes with two examples of the embedding, the first example being Boolean tautology checking, and the second example being Computation Tree Logic (CTL) model checking in Section 2.3 and Section 2.4, respectively.

Section 3 discusses how Agda was extended to allow for the technique described in this paper. This entailed extending the type-checker to check the decision procedures against axioms and low-level interfacing of the external ATP.

Section 4 provides an example of the composition with respect to specifying, developing and verifying a control system (Pelicon crossing) for specific domain. It demonstrates

“safety”, *safety principles* and *safety conditions*; and how these concepts relate to ITP and ATP theorem proving.

Finally, Section 5 provides concluding remarks and future work.

## 2. Methodology

In this section we discuss the advantages and disadvantages of automated and interactive theorem proving, and present a general technique of embedding automated theorem proving theories. This is concluded with two examples of the embedding, namely Boolean tautology checking and CTL model checking.

### 2.1. Comparison of Automated and Interactive Theorem Proving Techniques

Generally, theorem proving tools can be placed into one of two categories, interactive or automatic (Boutin 1997). The first category, ATP tools attempt to prove a theorem by automatically deducing the proof from already proven lemmata. In some cases, intermediate lemmata are introduced and proven automatically. The user has no direct influence over the derivation and proving process. Conversely, the second category, ITP tools or proof assistants work by allowing the user to guide the derivations and proofs of lemmata, culminating in a proof of the desired theorem.

2.1.1. *ATP Tools* are very powerful when dealing with finite, concrete theorems as in SAT and finitisable theories as with temporal logics. In some cases ATP tools can be applied to infinite theorems as in SMT (Barrett, Sebastiani, Seshia & Tinelli 2009) and first order provers, but this class of tools typically become semi-decidable decision procedures (De Moura & Bjørner 2009). Industrial hardware and software verification is archetypal of finite concrete theorems; large but not inherently complex problems. ATP tools often allow the system to be modelled using an intuitive language, and the desired properties of the system to be specified in the tool’s logic. The tool will attempt to prove these properties, when this is not possible the tool will either provide a counter-example of the property or declare an unknown result. These unknown results typically occur as a result of attempting to prove a theorem that has an infinite component and not knowing which lemma to apply or when a resource is not sufficient to complete the proof.

2.1.2. *ITP tools*. Consider a theorem of the form  $\forall n.\varphi$  in which  $\varphi$  has an infinite component. It could be possible to prove it using standard induction, but often the theorem needs to be strengthened such that the new theorem  $\forall n.\varphi \wedge \psi$  implies the desired theorem. The choice of this strengthened theorem, in general requires input from a human being. The reason is that when proving the inductive step,  $\varphi(n)$  might not be sufficient to imply  $\varphi(n+1)$ ; whereas a stronger theorem  $\varphi(n) \wedge \psi(n)$  might be sufficient. Choosing  $\psi$  such that it is strong enough to allow the theorem to be proved without being too strong is a complex task and cannot be mechanised in general. ITP tools have the advantage that unknown results do not occur as the user guides the proof.

As an example of the limitation of ATP, consider a simple reactive system realised using

Boolean valued equations which compute the next state from the current state and input variables. The most natural way to verify such a system is using SAT based verification. Assume a safety<sup>‡</sup> property  $P$ . One would have to construct the propositional formula which expresses that  $P$  holds in all reachable states, i.e.  $\forall \text{reachable state}.P$ . Induction can be applied here that would yield two proof obligations which take into account reachable states, namely the base case and inductive step. After the user has entered these into a SAT solver and determined validity of both cases, there is still a meta step to be performed by the user. The meta step here is to prove validity of induction outside the SAT solver. The user can then assemble the three proofs to determine that  $P$  always holds in all reachable states. SAT solving alone is not sufficient to prove this theorem. This example, perhaps contrived, shows the limitation of ATP alone, and in general this final task of assembling proofs is more complicated.

When using ITP tools on large finite concrete theorems the work delegated to the user is exorbitant (Jones, Grov & Bundy 2010). Industrial verification being a special case where large numbers of mechanisable, and a small number of un-mechanisable proof obligations arise (Woodcock et al. 2009). It would be nice to use ATP tools for the mechanisable proof obligations.

The next section discusses our embedding of ATP theories into type theory to mitigate the user's work-load.

## 2.2. General Technique

The embedding of a chosen ATP theory into type theory is as follows. Firstly, assume a logic  $\mathcal{L}$  such as propositional logic, CTL or modal- $\mu$  calculus that the chosen ATP theory is defined over.

Formulae in  $\mathcal{L}$  are inductively defined types, whose elements are finite. These formulae can usually only hold with respect to a model  $\mathcal{M}$  and an environment  $\xi$ . In this work, the model is what is fixed and does not change and the environment is what varies. For example, in CTL model checking (see Section 2.4) the model is a transition system and a state; the environment is an infinite run of the transition system from the state identified in the model. In SAT (see Section 2.3) there is no model, but the environment assigns Boolean values to variables in the formula. In the case of first order theorems the model consists of the semantics of the signature and the environment is an assignment to variables in the formula. For technical reasons, the model and environment are defined first as the formulae can depend upon these structures.

It is now possible to give semantics to these formulae by defining a satisfaction relation that assigns types to formulae with respect to  $\mathcal{M}$  and  $\xi$ .

$$\llbracket \_ \vDash \_ \rrbracket \_ : \text{Model} \rightarrow \text{Formula} \rightarrow \text{Environment} \rightarrow \text{Set}$$

The decision procedure  $\text{Dec}_{\mathcal{M}} : \text{Formula} \rightarrow \text{Bool}$  for the ATP theory is formalised as a function. In our experience an inefficient simplistic definition that recurses over the

<sup>‡</sup> A safety property is a property that must hold in all reachable states.

structure of the formula, model and environment is preferable as this helps with proving correctness of  $\text{Dec}_{\mathcal{M}}$ . Correctness of  $\text{Dec}_{\mathcal{M}}$  is then a proof of

$$\text{Dec}_{\mathcal{M}}(\varphi) \Leftrightarrow \forall \xi. \exists \llbracket \mathcal{M} \models \varphi \rrbracket_{\xi}$$

where the choice of quantification depends upon the ATP theory. E.g. in the case of the Boolean satisfiability problem, we have “there exists a satisfying assignment” (satisfiability testing) or “all assignments are satisfying” (tautology checking). It is possible to define more complicated quantification schemes, whereby the environment is split into sub-environments; but this is not considered in this work.

The proof of correctness is then used to prove theories of the form  $\Box \xi. \llbracket \mathcal{M} \models \varphi \rrbracket_{\xi}$  where  $\Box \in \{\forall, \exists\}$  by transferring the proof of the Boolean valued result of  $\text{Dec}_{\mathcal{M}}(\varphi)$  generated by the ATP theory.

The type theoretic implementation of  $\text{Dec}_{\mathcal{M}}$  is typically inefficient compared to purpose written tools, because  $\text{Dec}_{\mathcal{M}}$  is defined naïvely to simplify proof of correctness. This inefficiency is exasperated by many implementations of proof systems; specifically relating to this work, type systems make heavy use of rewriting and normalisation resulting in large terms which would consume vast resources in attempting to evaluate  $\text{Dec}_{\mathcal{M}}$  on all but the simplest examples. For this reason, we replace  $\text{Dec}_{\mathcal{M}}$  with an actual ATP tool for an efficient implementation, see Section 3 for more information regarding the implementation.

The technique presented here is generic and abstract, it does not only apply to type theoretic applications but generally within mechanised mathematics. To solidify the above technique two examples are presented, Boolean tautology checking and CTL model checking in Section 2.3 and Section 2.4 respectively.

### 2.3. SAT

In this work, standard Boolean satisfiability is not applied, instead tautology testing of a Boolean valued formula with variables is explored, which is an equivalent problem. Tautology checking was chosen over satisfiability testing because SAT verification typically relates to checking safety conditions, that is that something bad will never happen.

The model is trivial and contains no information, for completeness it is the canonical element from the singleton set. The environment assigns Boolean values to the variables in the formula. Before introducing the definition of Boolean formulæ with variables, the notion of finite sets are introduced as they index the variables. Finite, or enumeration sets,  $\text{Fin} : (n : \mathbb{N}) \rightarrow \text{Set}$  are finite sets with  $n$  distinct elements, namely  $\text{Fin } n := \{0, \dots, n-1\}$ ; notably  $\text{Fin } 0 = \emptyset$ .

Boolean formulæ are defined as follows:

```

data BooleanFormula (n : N) : Set where
  const : Bool → BooleanFormula n
  var    : Fin n → BooleanFormula n
  ¬      : BooleanFormula n → BooleanFormula n
  _ ∧ _  : BooleanFormula n → BooleanFormula n → BooleanFormula n
  _ ∨ _  : BooleanFormula n → BooleanFormula n → BooleanFormula n
  _ ⇒ _  : BooleanFormula n → BooleanFormula n → BooleanFormula n

```

where the underscores ( $\_$ ) denote syntactic positions of required arguments. In the following we write  $x_n$  for  $(\text{var } n)$ .

The semantics of BooleanFormula with respect to an environment is

$$\begin{aligned}
\llbracket \_ \rrbracket : \forall \{n\} \rightarrow \text{BooleanFormula } n \rightarrow (\text{Fin } n \rightarrow \text{Bool}) \rightarrow \text{Set} \\
\llbracket \text{const } b \rrbracket_\xi &= \text{T } b \\
\llbracket x_i \rrbracket_\xi &= \xi i \\
\llbracket \neg \varphi \rrbracket_\xi &= (\llbracket \varphi \rrbracket_\xi) \rightarrow \emptyset \\
\llbracket \varphi \wedge \psi \rrbracket_\xi &= \llbracket \varphi \rrbracket_\xi \times \llbracket \psi \rrbracket_\xi \\
\llbracket \varphi \vee \psi \rrbracket_\xi &= \llbracket \varphi \rrbracket_\xi + \llbracket \psi \rrbracket_\xi \\
\llbracket \varphi \Rightarrow \psi \rrbracket_\xi &= \llbracket \varphi \rrbracket_\xi \rightarrow \llbracket \psi \rrbracket_\xi
\end{aligned}$$

To define the decision procedure, assume a function

$$\text{instantiate} : \text{BooleanFormula } (\text{suc } n) \rightarrow \text{Bool} \rightarrow \text{BooleanFormula } n$$

that instantiates all occurrences of  $x_0$  with the second (Boolean valued) argument; all other variables are shifted down by one, i.e.  $x_{n+1} \mapsto x_n$ .

The decision procedure for tautology checking a BooleanFormula  $n$  is defined naïvely to simplify the correctness proof; it is defined by  $2^n$  applications of `instantiate`, then canonically with respect to the Boolean connectives.

$$\begin{aligned}
\text{tautology} &: \forall n \rightarrow \text{BooleanFormula } n \rightarrow \text{Bool} \\
\text{tautology zero } \text{const } b &= b \\
\text{tautology zero } \neg \varphi &= \neg(\text{tautology zero } \varphi) \\
\text{tautology zero } \varphi \wedge \psi &= (\text{tautology zero } \varphi) \wedge (\text{tautology zero } \psi) \\
\text{tautology zero } \varphi \vee \psi &= (\text{tautology zero } \varphi) \vee (\text{tautology zero } \psi) \\
\text{tautology zero } \varphi \Rightarrow \psi &= (\text{tautology zero } \varphi) \Rightarrow (\text{tautology zero } \psi) \\
\text{tautology (suc } n) \varphi &= \text{tautology } n (\text{instantiate } \varphi \text{ true}) \wedge \\
&\quad \text{tautology } n (\text{instantiate } \varphi \text{ false})
\end{aligned}$$

Proof of correctness is then an element of:

$$\forall n \rightarrow (\varphi : \text{BooleanFormula } n) \rightarrow (\text{T}(\text{tautology } n \varphi) \leftrightarrow ((\xi : \text{Fin } n \rightarrow \text{Bool}) \rightarrow \llbracket \varphi \rrbracket_\xi))$$

which is proven by simple induction over  $n$ .

The above embedding of Boolean tautology checking has been implemented in Agda requiring 39 lines of code for the decision procedure `tautology`, associated definitions (including natural numbers and Booleans). The proof of correctness requires an additional

≈ 100 lines of code which includes many basic lemmata about products and sums (Curry-Howard isomorphism). The code can be downloaded from the project home page.

This concludes the first example.

#### 2.4. CTL Model Checking

One aim of this work is to use Agda to develop and verify control systems for safety and liveness properties, to this end we have outlined a process of embedding CTL model checking that will verify whether some property holds in a finite transition system.

When dealing with theorems which have substantial structure, such as model checking that is defined over a transition system, it is important to choose definitions that simplify the embedding process. The transition systems used here are finite and can deadlock, i.e. the transition relation is not total.

Finite state machines (FSM) are the transition systems used in this work. They are defined by the number of states, the number of atomic propositions, an initial state, a transition relation between states and a labelling of the states. The transition relation is given by two functions: *arrow* which determines for each state the number of transitions from it, and *transition* which determines for each state and arrow from this state the successor state.

```

data FSM : Set where
  fsm : (state atom : ℕ)
        → (arrow : Fin state → ℕ)
        → (initial : Fin state)
        → (transition : (s : Fin state) → Fin (arrow s) → Fin state)
        → (label : Fin state → Fin atom → Bool)
        → FSM

```

The model of the CTL model checking is a pair, the transition system  $\mathcal{M}$  and current state  $s_0$ . The environment (under combined operators) is an infinite run  $\langle s_0, s_1, \dots \rangle$  rooted at the current state  $s_0$ ; defined by means of a co-algebraic data-type.

```

data Run $\mathcal{M}$  (s : Fin state $\mathcal{M}$ ) : Set where
  next : (a : Fin (arrow $\mathcal{M}$  s)) → ∞ Run $\mathcal{M}$  (transition $\mathcal{M}$  s a) → Run $\mathcal{M}$  s

```

where  $\infty$  prefixes a term that can potentially be unfolded infinitely many times.

In the following, we write  $run_i$  for the  $i^{\text{th}}$  state in  $run = \langle s_0, s_1, \dots, s_i, \dots \rangle$ .  $\pi$  is used for finite paths and  $\pi_i$  is the  $i^{\text{th}}$  state in  $\pi$ .

CTL formulæ can be defined over the model using a minimal set of combined CTL operators (Huth & Ryan 2004). EX - exists next, EG - exists globally, E[\_U\_] - exists

until and P - state proposition.

```

data CTL (M : FSM) : Set where
  false      : CTLM
  ¬          : CTLM → CTLM
  _ ∨ _     : CTLM → CTLM → CTLM
  P         : Fin atomM → CTLM
  EX        : CTLM → CTLM
  EG        : CTLM → CTLM
  E[_ U _] : CTLM → CTLM → CTLM

```

where  $\text{atom}_{\mathcal{M}}$  is *atom* projected from  $\mathcal{M}$ .

The semantics of a CTL formula is as follows:  $\text{false}$ ,  $\neg$  and  $\_ \vee \_$  are the same as in propositional logic.  $P\ a$  has the meaning that atomic proposition  $a$  holds in the current state. The remaining cases specify properties about infinite runs of the transition system rooted at some state  $s$ . Exists next ( $\text{EX } \varphi$ ) holds when there exists a run  $\text{run}$  from  $s$  such that  $\varphi$  holds at  $\text{run}_1$ . Exists globally ( $\text{EG } \varphi$ ) holds when there exists a run from  $s$  such that at each point  $i$  on the run,  $\varphi$  holds. Exists until ( $\text{E}[\varphi\ \text{U}\ \psi]$ ) holds when there exists a run from  $s$  such that there exists a point  $k$  where  $\psi$  holds, and for all points  $j < k$ ,  $\varphi$  holds.

The semantics of a CTL formula with respect to a model is

$$\begin{aligned}
\llbracket \_ , \_ \models \_ \rrbracket &: (\mathcal{M} : \text{FSM}) \\
&\rightarrow (\text{Fin state}_{\mathcal{M}}) \\
&\rightarrow \text{CTL}_{\mathcal{M}} \\
&\rightarrow \text{Set} \\
\llbracket \mathcal{M} , s \models \text{false} \rrbracket &= \emptyset \\
\llbracket \mathcal{M} , s \models \neg\varphi \rrbracket &= \llbracket \mathcal{M} , s \models \varphi \rrbracket \rightarrow \emptyset \\
\llbracket \mathcal{M} , s \models \varphi \vee \psi \rrbracket &= \llbracket \mathcal{M} , s \models \varphi \rrbracket + \llbracket \mathcal{M} , s \models \psi \rrbracket \\
\llbracket \mathcal{M} , s \models P\ a \rrbracket &= \text{T}(\text{label}_{\mathcal{M}}\ s\ a) \\
\llbracket \mathcal{M} , s \models \text{EX } \varphi \rrbracket &= \exists (\text{Run}_{\mathcal{M}}\ s) (\lambda\ \text{run} \rightarrow \llbracket \mathcal{M} , \text{run}_1 \models \varphi \rrbracket) \\
\llbracket \mathcal{M} , s \models \text{EG } \varphi \rrbracket &= \exists (\text{Run}_{\mathcal{M}}\ s) (\lambda\ \text{run} \rightarrow (i : \mathbb{N}) \rightarrow \llbracket \mathcal{M} , \text{run}_i \models \varphi \rrbracket) \\
\llbracket \mathcal{M} , s \models \text{E}[\varphi\ \text{U}\ \psi] \rrbracket &= \exists (\text{Run}_{\mathcal{M}}\ s) \\
&\quad (\lambda\ \text{run} \rightarrow \exists\ \mathbb{N}\ (\lambda\ k \rightarrow \\
&\quad\quad ((j : \mathbb{N}) \rightarrow j < k \rightarrow \llbracket \mathcal{M} , \text{run}_j \models \varphi \rrbracket) \times \llbracket \mathcal{M} , \text{run}_k \models \psi \rrbracket))
\end{aligned}$$

where  $\text{label}_{\mathcal{M}}$  is *label* projected from  $\mathcal{M}$ . Here the environment  $(\text{Run}_{\mathcal{M}}\ s)$  is existentially quantified.

Determining whether a CTL formula holds in the simple Boolean operator cases is canonical with respect to the operators and not discussed further. The decision procedure for the first substantial operator,  $\text{EX } \varphi$  (exists next) does this by searching for a path of length  $\text{state}_{\mathcal{M}} + 1$  and verifying that  $\varphi$  holds at the second point, i.e. there exists a successor state where  $\varphi$  holds. The argument for correctness of this procedure is a simpler case of correctness for exists globally, and follows by Lemma 2.1.

In the case of  $\text{EG } \varphi$  (exists globally), an infinite run is required such that at each

point on this run,  $\varphi$  holds. Naïvely checking each point on this run would take an infinite amount of time, thus we finitise the problem.

The pigeon hole principle<sup>§</sup> (Dedekind 1863) (which is the principle underlying the proof of the pumping lemma (Bar-Hillel, Perles & Shamir 1964)), and the finiteness of the transition system allow the decision procedure for EG to check for a finite path of fixed length from the state  $s$  such that  $\varphi$  always holds. If a path  $\pi$  of length  $\text{state}_{\mathcal{M}} + 1$  exists from  $s$  such that  $\varphi$  holds at each point, then it can be extended infinitely many times into a run. This follows by Lemma 2.1.

**Lemma 2.1** ( $\mathcal{M}, s \models \text{EG } \varphi$ ). Assume a finite transition system  $\mathcal{M}$  with  $n$  states. There exists an infinite run from state  $s$  such that  $\varphi$  holds at each point *iff* there exists a path  $\pi$  of length  $n + 1$  from state  $s$  such that  $\varphi$  holds at each point on  $\pi$ .

*Proof.*

- $\Rightarrow$  An infinite run where  $\varphi$  holds at each point can be truncated to a path of length  $n + 1$ .  
 $\Leftarrow$  By the pigeon hole principle, at least one state has been repeated in  $\pi$ , i.e.  $\exists(0 \leq i < j \leq n) . \pi_i = \pi_j$ . Therefore a (possibly trivial) loop in the transition system exists containing  $\pi_i$ , this loop can be repeated infinitely many times, and we obtain an infinite run.

□

In the case of exists until, things are a little more complicated.  $E[\varphi U \psi]$  means there exists an infinite run *run* such that at some point  $k$  in the future  $\psi$  must hold, but up to and not including that point,  $\varphi$  must hold. Intuitively, the decision procedure checks for a path  $\pi^\varphi$  with length  $\leq \text{state}_{\mathcal{M}}$  such that  $\varphi$  holds at each point, and then checks for a path  $\pi^\psi$  of length  $\text{state}_{\mathcal{M}} + 1$  starting at the end of  $\pi^\varphi$  such that  $\psi$  holds at  $\pi_1^\psi$ . This follows by Lemma 2.2.

**Lemma 2.2.** Assume a finite transition system  $\mathcal{M}$  with  $n$  states.  $\mathcal{M}, s \models E[\varphi U \psi]$  holds *iff* there exists a path  $\pi^\varphi$  with length  $\leq n$  from the state  $s$  such that  $\varphi$  holds at each point of  $\pi^\varphi$ , and there exists a path  $\pi^\psi$  of length  $n + 1$  such that the end of  $\pi^\varphi$  equals the beginning of  $\pi^\psi$  and  $\psi$  holds at  $\pi_1^\psi$ .

*Proof.*

- $\Rightarrow$  There exists a point  $k$  on the infinite run *run*, such that for all points  $j < k$ ,  $\varphi$  holds and at point  $k$ ,  $\psi$  holds. We show that  $\pi^\varphi$  and  $\pi^\psi$  exist by course-of-value induction on  $k$ :  
**case**  $k \leq n$ : We are done,  $\pi^\varphi$  is a prefix of the run, and  $\pi^\psi$  equals the succeeding  $n + 1$  states from  $k$ .  
**case**  $k > n$ : By the pigeon hole principle there exists two points  $0 \leq l < m < n + 1$  such that  $\text{run}_l = \text{run}_m$ . Therefore a loop exists before point  $k$ , this loop can be removed such that  $\varphi$  holds up to point  $k - (m - l)$  and  $\psi$  holds at point  $k - (m - l)$ . Let *run'* be the resulting run. By the induction hypothesis and *run'*, the assertion follows.

<sup>§</sup> The pigeon hole principle states: *if you put  $n$  things into  $m$  boxes where  $n > m$ , then there exists at least one box that contains more than one item.*

⇐By Lemma. 1, path  $\pi^\psi$  can be extended infinitely many times, thus an infinite run can be constructed consisting of  $\pi^\psi$  extended infinitely many times concatenated to  $\pi^\varphi$ . As  $\varphi$  holds along  $\pi^\varphi$ , and  $\psi$  holds at  $\pi_1^\psi$ , the infinite run satisfies  $\varphi$  until  $\psi$ .

□

The decision procedures for EX, EG and E[-U\_] can be implemented by bounded traversals of the transition system and taking disjunctions between choice points in the traversals. Our implementation requires  $\approx 75$  lines of Agda code, this includes the definitions of Bool,  $\mathbb{N}$ , Fin, transition system, CTL formulæ and the decision procedure. Proof of correctness requires  $> 1000$  lines of code because this includes the proof of the pigeon hole principle ( $\approx 450$  lines of code) and many lemmata reasoning about finite sets and the transition system.

### 3. Implementation

So far, everything presented has been fully contained in the ITP tool’s logic, but in practise the decision procedures here are inefficient in comparison with purpose written automated theorem proving tools. This is the case because mechanisation of a type system loses the low-level procedural access to the computer needed for efficient implementations.

For this reason, we customised the ITP and programming language Agda to allow for the type-checker to call external ATP tools in-situ of evaluating the decision procedures. This technique has allowed for an efficient integration of automated theorem proving and type theory.

Both of the examples presented in this paper have been implemented in Agda. This entailed not only the steps identified in Section 2.2, but also providing translations between Agda’s terms and the tool’s input language. The Boolean valued result of the tools also have to be transferred back to Agda.

#### 3.1. Plug-in Interface

We have modified Agda by adding six tags to the type-checker, definitions in the plug-in (Agda source file) can then be tagged. The first tag **ATPProblem** tags something in Set which corresponds to the problem set that the decision procedure is defined over. The second tag **ATPInput** depends upon the first, and tags a function of type  $\text{ATPProblem} \rightarrow \text{String}$  that translates the problem into a string that is passed to the external tool. The third tag **ATPTool** tags a string which is a path to the external tool; and the fourth tag **ATPDecProc** tags the actual decision procedure which must be of the type  $\text{ATPProblem} \rightarrow \text{Bool}$ .

The plug-in mechanism will not be activated until a proof of correctness has been provided; the final two tags tag the semantic relation and a proof of correctness. The intuition behind these final two tags force the user to prove that the provided decision procedure implements their chosen ATP theory, reducing the risk of providing the wrong external tool.

When the function tagged by **ATPDecProc** is reduced on  $\gamma : \mathbf{ATPProblem}$ , the type-checker will execute the external tool pointed to by **ATPTool**. But, first applies the function tagged by **ATPInput** to  $\gamma$  that yields a string in the tool's input language.

The Boolean valued result is computed by examining the return value of the tool, should the tool return 0 a true value is used, should the tool return 1 then false is used; any other value results in an exception being raised and the tools output dumped into the log. Typically, it is required to write a simple wrapper for the external tool that sets the return values.

Currently we have developed three versions of the plug-in mechanism. The generic version for arbitrary ATP theories outlined above does not perform an axiom check on **ATPDecProc**; although we have developed Agda formulæ which type-check iff the Agda implementation is used correctly and plan to implement this soon. The other two versions are specifically for SAT and CTL; they perform axiom checks, avoiding the issue of binding the ATP tool to a wrongly defined decision procedure.

### 3.2. Results

Without connecting Agda to an efficient SAT solver implementation, we struggled to show validity of formulæ with  $\geq 10$  variables using the naïve decision procedure due to the exorbitant resources (time and space) required by the type-checker. The connection between Agda and Boolean tautology checking has proven to be useful. It is often the case while proving properties about industrial systems that proof obligations arise which can be proven by showing the validity of a Boolean valued formula, which has now been automated. The solver currently used by the system is the award-winning **iProver**, translation of the formulæ into clauses is done by **eProver**.

We have used this system to successfully verify a real world interlocking system provided by our sponsor: Invensys Rail, UK. More information about the verification technique can be found in (Kanso, Moller & Setzer 2009, Kanso 2008). These problems had approximately 600 variables, well within the feasible range. Currently we are facing the problem that the type-checker takes ( $< 5$  minutes) on initial checking of the interlocking system files and computing proof obligations for the SAT solver; once compiled to native code (via `ghc`) and executed, this problem is mitigated (in tests, problems with  $\approx 5000$  variables were solved, and proofs fully explored in  $\approx 1$  second). Work is ongoing to identify resource leaks in Agda.

Work is ongoing to lift the CTL model checking presented to symbolic CTL model checking, a more useful variant.

## 4. Example – Pelicon Crossing

To exemplify the wider vision of composing ATP and ITP with respect to verifying and validating critical systems, consider the following scenario.

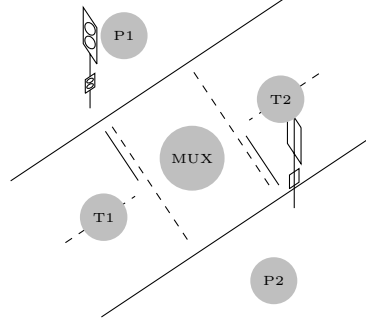


Fig. 1. Layout of a Pelicon crossing. They consist of two sets of lights, the smaller set for the pedestrians and the larger set for road traffic. In this diagram only two aspects are shown for road traffic, but in practise a third aspect for warning the lights are about to change would also be present. Also, a button for the pedestrians is present but not depicted. The areas T1 and T2 are for road traffic, P1 and P2 are for pedestrians, and MUX (mutual-exclusion) represents the area of the crossing used by both road traffic (travelling between T1 and T2 through MUX) and pedestrians (travelling between P1 and P2 through MUX).

On roads in the UK (any many other countries) there are **Pedestrian Light Controlled** (Pelicon) crossings, see Figure 1; they consist of two sets of lights, one for road traffic and one for pedestrians. A pedestrian indicates to the Pelicon crossing that they wish to cross the road by pressing a button; after a small delay, road traffic will be shown a red (transitioning from green) light and pedestrians will be shown a green (transitioning from red) light that indicates it is now safe to cross the road. After a further delay the lights transition back. Formally for a given time  $t$  the Pelicon crossing can be modelled abstractly as

$$\begin{aligned}
 \text{numbercars}_t & : \text{Area} \rightarrow \mathbb{N} \\
 \text{numberped}_t & : \text{Area} \rightarrow \mathbb{N} \\
 \text{movingcars}_t & : \text{Area} \rightarrow \text{Area} \rightarrow \mathbb{N} \\
 \text{movingped}_t & : \text{Area} \rightarrow \text{Area} \rightarrow \mathbb{N} \\
 \text{traffic}_t & : \{\text{green}, \text{red}\} \\
 \text{pedestrian}_t & : \{\text{green}, \text{red}\}
 \end{aligned}$$

where  $\text{Area} := \{\text{T1}, \text{T2}, \text{P1}, \text{P2}, \text{MUX}\}$ , see Figure 1 for the location of these areas.

Initially at time 0 it is assumed that there is no road traffic in, or moving into MUX; similarly for pedestrians. The initial axioms for road traffic are

$$\begin{aligned}
 \text{numbercars}_0 \text{ MUX} & \equiv 0 \\
 \text{movingcars}_0 \text{ T1 MUX} & \equiv 0 \\
 \text{movingcars}_0 \text{ T2 MUX} & \equiv 0
 \end{aligned}$$

The axioms for cars travelling between areas T1 and T2 (via MUX) are

$$\begin{aligned}
& (\text{traffic}_t \equiv \text{red}) \rightarrow \text{movingcars}_{(t+1)} \text{ T1 MUX} \equiv 0 \\
& \quad \wedge \text{movingcars}_{(t+1)} \text{ T2 MUX} \equiv 0 \\
& \text{movingcars}_{(t+1)} \text{ MUX T2} \equiv \text{movingcars}_t \text{ T1 MUX} \\
& \text{movingcars}_{(t+1)} \text{ MUX T1} \equiv \text{movingcars}_t \text{ T2 MUX} \\
& \text{movingcars}_{(t+1)} \text{ T1 MUX} \leq \text{numbercars}_t \text{ T1} \\
& \text{movingcars}_{(t+1)} \text{ T2 MUX} \leq \text{numbercars}_t \text{ T2} \\
& \text{numbercars}_{(t+1)} \text{ MUX} \equiv (\text{numbercars}_t \text{ MUX}) \\
& \quad + (\text{movingcars}_{(t+1)} \text{ T1 MUX}) \\
& \quad - (\text{movingcars}_{(t+1)} \text{ MUX T2}) \\
& \quad + (\text{movingcars}_{(t+1)} \text{ T2 MUX}) \\
& \quad - (\text{movingcars}_{(t+1)} \text{ MUX T1})
\end{aligned}$$

The axioms for the pedestrians travelling between areas P1 and P2 (via MUX) are similar but not presented.

In this setting the notion of actual “safety” which should be validated by domain experts could be “at any point in time exclusive use of the crossing is given to pedestrians or to road traffic”,

$$\forall t \quad \text{numbercars}_t \text{ MUX} \equiv 0 \vee \text{numberped}_t \text{ MUX} \equiv 0$$

this is the desired property that we wish to prove.

In the following, *safety principles* and *conditions* are lemmata which imply actual “safety”. A *safety principle* can be viewed as an intermediate lemma, typically deduced from *principles* in the target domain. For example, in the train domain they have a large volume of literature detailing various signalling principles which in our setting correspond to *safety principles*. A *safety condition* is a concrete formula that can be proven by ATP, which implies a/some safety principle/s.

A *safety principle* which implies this notion of “safety” is “at any point in time only road traffic or pedestrians are allowed to enter the crossing”,

$$\begin{aligned}
\forall t \quad & (\text{movingcars}_t \text{ T1 MUX} \equiv 0 \wedge \text{movingcars}_t \text{ T2 MUX} \equiv 0) \\
& \vee (\text{movingped}_t \text{ P1 MUX} \equiv 0 \wedge \text{movingped}_t \text{ P2 MUX} \equiv 0)
\end{aligned}$$

For a given Pelicon control system, giving a direct proof of the *safety principle* or actual “safety” would be a cumbersome activity as ATP tools do not typically yield abstract solutions, but concrete solutions for concrete problems. Our solution would be to use the ATP tool to give a concrete proof of the *safety condition* “pedestrians have a green light or road traffic has green light, but not both”,

$$\forall t \quad \text{traffic}_t \equiv \text{red} \vee \text{pedestrian}_t \equiv \text{red}$$

and prove that this implies the *safety principle* (and in turn “safety”). It should be noted that the *safety condition* can be concretely represented in terms of a specific control system’s output (and possibly input/internal) variables.

Thus, part of the validation procedure has been reduced to a verification procedure via ITP; and the control system is verified by ATP.

From our experience with verifying train control systems (Kanso et al. 2009, Kanso 2008), proving that the *safety principles* imply “safety” is more complicated than with this scenario. In the train domain there is a larger volume of domain knowledge which must be considered and many more *safety (signalling) principles* (that change between train lines). In these non-trivial situations, it could be the case that the *safety principles* are shown not to be sufficient or that some are not necessary.

## 5. Concluding Remarks

This paper explored the differences between interactive and automated theorem proving techniques with a discussion about their respective uses. Namely, using ATP to solve finite concrete theorems and ITP to solve infinite abstract theorems. The discussion was concluded with our motivation for composing these two paradigms, i.e. a proof framework with the advantages of ATP and ITP.

We have presented a novel approach to embed an arbitrary ATP theory into type theory, along with two examples of this, namely Boolean tautology checking and CTL model checking. In its native form, the technique would be very inefficient so we have mitigated this by extending Agda to allow an efficient ATP tool to be called in situ. This has facilitated in an efficient framework for proving finite and infinite theorems, specifically with respect to software verification.

The technique described can be used to efficiently define a type, such that elements of this type are correct programs. Such a system would be of use for the development of critical systems as only correct programs can be written, greatly reducing work of testing. Our vision is to use Agda to model domain knowledge; then to develop, verify, compile and execute a substantial program. The motivation here is that a large portion of the software development cycle is contained within a single language and tool; this reduces erroneous translations between languages that occur within typical verification procedures.

We have developed a plug-in mechanism that allows for anybody to embed their chosen ATP theory into Agda by providing the items required in Section 2.2 and Section 3. This plug-in mechanism, and plugins for SAT and CTL can be downloaded from the project website at:

<http://cs.swansea.ac.uk/~cskarim/agda/>

## References

- Bar-Hillel, Y., M. Perles & E. Shamir (1964), ‘On formal properties of simple phrase structure grammars’, *Language and Information: Selected Essays on their Theory and Application* pp. 116–150.
- Barrett, Clark, Roberto Sebastiani, Sanjit A. Seshia & Cesare Tinelli (2009), *Frontiers in Artificial Intelligence and Applications*, Vol. 185 of *Handbook of Satisfiability*, IOS Press, chapter 12 Satisfiability Modulo Theories, pp. 737–797.
- Bierman, G.M., A.D. Gordon, C. Hrițcu & D. Langworthy (2010), Semantic subtyping with an SMT solver, in ‘Proceedings of the 15th ACM SIGPLAN international conference on Functional programming’, ACM, pp. 105–116.

- Böhme, S. & T. Nipkow (2010), Sledgehammer: Judgement day, in J.Giesl & R.Hähnle, eds, ‘Automated Reasoning’, Vol. 6173, LNCS, pp. 107–121.  
**URL:** <http://www4.informatik.tu-muenchen.de/~nipkow/pubs/judgement.pdf>
- Boutin, Samuel (1997), Using reflection to build efficient and certified decision procedures, in ‘Theoretical Aspects of Computer Software’, Vol. 1281 of *Lecture Notes in Computer Science*, Springer, pp. 515–529.  
**URL:** <http://www.springerlink.com/content/51v27169p8420xt4/>
- Bove, Ana, Peter Dybjer & Ulf Norell (2009), A Brief Overview of Agda – A Functional Language with Dependent Types, in S.Berghofer, T.Nipkow, C.Urban & M.Wenzel, eds, ‘Theorem Proving in Higher Order Logics’, Vol. 5674 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 73–78. 10.1007/978-3-642-03359-9\_6.  
**URL:** [http://dx.doi.org/10.1007/978-3-642-03359-9\\_6](http://dx.doi.org/10.1007/978-3-642-03359-9_6)
- Curry, H.B. (1934), ‘Functionality in Combinatory Logic’, *Proceedings of the National Academy of Sciences of the United States of America* **20**(11), 584.
- Curry, H.B., R. Feys, W. Craig & W. Craig (1958), *Combinatory Logic, Vol. 1*, North-Holland.
- De Moura, L. & N. Bjørner (2009), ‘Satisfiability Modulo Theories: An Appetizer’, *Formal Methods: Foundations and Applications* pp. 23–36.
- Dedekind, R. (1863), *Vorlesungen über Zahlentheorie*, F. Vieweg und Sohn.
- Dong, Yifei, C. R. Ramakrishnan & Scott A. Smolka (2003), ‘Evidence exploration for model checking’.
- Fontaine, P., J.Y. Marion, S. Merz, L. Nieto & A. Tiu (2006), ‘Expressiveness+ automation+ soundness: Towards combining SMT solvers and interactive proof assistants’, *Tools and Algorithms for the Construction and Analysis of Systems* pp. 167–181.
- Foster, Simon & Georg Struth (2011), Integrating an Automated Theorem Prover into Agda. To appear in: NASA Formal Methods Symposium 2011.
- Harrison, J. (2008), Automated and interactive theorem proving, in O.Grumberg, T.Nipkow & C.Pfaller, eds, ‘Formal Logical Methods for System Security and Correctness’, Vol. 14 of *NATO Science for Peace and Security Series*, IOS Press, pp. 111–147.
- Hendriks, D. (2002), ‘Proof reflection in Coq’, *Journal of Automated Reasoning* **29**(3), 277–307.
- Howard, W.A. (1980), ‘The formulae-as-types notion of construction, To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism (J.R. Hindley and J.P. Seldin, eds.)’.
- Huth, M. & M. Ryan (2004), *Logic in computer science*, Cambridge University Press Cambridge.
- Jones, C.B., G. Grov & A. Bundy (2010), Ideas for a High-Level Proof Strategy Language, Technical Report CS-TR-1210, Newcastle University.
- Kanso, K. (2008), Formal Verification of Ladder Logic, Master’s thesis, Dept. Computer Science, Swansea University.
- Kanso, K., F. Moller & A. Setzer (2009), ‘Automated Verification of Signalling Principles in Railway Interlocking Systems’, *Electronic Notes in Theoretical Computer Science* **250**(2), 19–31.
- Lescuyer, Stéphane & Sylvain Conchon (2008), A Reflexive Formalization of a SAT Solver in Coq, in ‘TPHOLS 2008: In Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLS)’.  
**URL:** <http://www.lri.fr/~conchon/publis/lescuyer-conchon-tphols08.pdf>
- Müller, Olaf & Tobias Nipkow (1995), Combining model checking and deduction for I/O-automata, Vol. 1019, LNCS, pp. 1–16.
- Nordström, B., K. Petersson & J.M. Smith (1990), *Programming in Martin-Löf’s Type Theory*, Vol. 7 of *International Series of Monographs on Computer Science*, Oxford University Press.

- Paulson, L. & K. Susanto (2007), Source-level proof reconstruction for interactive theorem proving, in K.Schneider & J.Brandt, eds, ‘Theorem Proving in Higher Order Logics’, Vol. 4732, LNCS, pp. 232–245.
- Stump, Aaron (2009), ‘Proof checking technology for satisfiability modulo theories’, *Electronic Notes in Theoretical Computer Science* **228**, 121 – 133.
- Verma, Kumar Neeraj (2000), Reflecting symbolic model checking in coq, Master’s thesis, Mémoire de DEA, DEA Programmation, Paris.
- Weber, Tjark (2006), ‘Integrating a sat solver with an lcf-style theorem prover’, *Electronic Notes in Theoretical Computer Science* **144**(2), 67 – 78. Proceedings of the Third Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2005).  
**URL:** <http://dx.doi.org/10.1016/j.entcs.2005.12.007>
- Woodcock, J., P.G. Larsen, J. Bicarregui & J. Fitzgerald (2009), ‘Formal methods: Practice and experience’, *ACM Computing Surveys (CSUR)* **41**(4), 1–36.