A Verified Model Checker for the Modal μ-calculus in Coq

Christoph Sprenger

Computer Networking Laboratory,
Swiss Federal Institute of Technology, Lausanne, Switzerland
sprenger@di.epfl.ch

Abstract. We report on the formalisation and correctness proof of a model checker for the modal μ-calculus in Coq's constructive type theory. Using Coq's extraction mechanism we obtain an executable Caml program, which is added as a safe decision procedure to the system. We thereby avoid the generation of large proof objects while preserving the high reliability of the proof environment. An example illustrates the combination of model checking with deductive techniques.

1 Introduction

There is an obvious advantage in combining theorem proving and model checking techniques for the verification of reactive systems. The expressiveness of the theorem prover's (often higher-order) logic can be used to accommodate a variety of program modelling and verification paradigms, so infinite state and parametrised designs can be verified. However, using a theorem prover is not transparent and may require a fair amount of expertise. On the other hand, model checking is transparent, but exponential in the number of concurrent components. Its application is thus limited to systems with small state spaces. A combination of the two techniques can therefore alleviate the problems inherent to each of them when used in isolation. Such an integration pays off even more, when used in combination with *reduction techniques* which transform infinite state or parametrised systems into finite state ones, while preserving the properties of interest. These are often small enough to be amenable to model checking. Examples of such techniques are abstract interpretation [7, 15, 11] and inductive reasoning at the process level [27, 14].

Various model checkers have already been integrated in theorem proving environments [24, 18, 12]. Common to all these cases is that the model checker is an external program that is invoked as needed and, most importantly, whose results are trusted. The question of the correctness of the model checker itself is rarely posed. In this paper, we take the position that this is an important question, whenever the proof environment we use should be highly *reliable*. This question gains even more importance in the context of provers based on intuitionistic type theory such as Coq [5], Alf [2] and Lego [16], where explicit *proof objects* (i.e. λ-terms) are constructed during the proof. These proof objects are then verified by an inference engine implementing the basic proof rules. Since there are only a few rules and the correctness of any proof depends only on the correct implementation of these rules, these systems can be regarded as very reliable.

We see two possibilities for the integration of a model checker into such a framework: (1) we implement it as an external program that generates the necessary proof object and add it as a tactic to the system or (2) we prove the model checker itself formally correct and then consider it as a trusted decision procedure. In both approaches the proof system for the temporal or modal logic is implemented in the prover and is therefore available for deductive proofs.

The first approach has been followed by Yu and Luo [28], the work which is closest to ours. They have implemented a model checker for the modal μ-calculus for Lego in this way. While integrating very smoothly into the prover, this approach has the problem of being inefficient. The size of the generated proof objects grows linearly with the number of applications of proof rules. This generates large proof objects even for quite small examples. The second approach is more efficient, but integrates somewhat less smoothly into the proof environment, as the results produced by the model checker have to be introduced as (safe) axioms into the prover.

Our approach is a compromise between the two. We have formalised the modal μ-calculus, a specification of the model checker in [26] and proved it correct in Coq. Using Coq's program extraction mechanism our proof is then translated into an executable Caml program. Moreover, we also have the possibility to directly run the (proof of the) model checker in Coq itself and generate a proof object. We see our contribution as two-fold. Firstly, the specification and correctness proof of the model checker provides a case study in developing provably correct sequential (functional) programs. To the best of our knowledge, this is the first formally verified model checker. Secondly, the formalisation of the μ-calculus can be used to prove properties of (possibly infinite) transition systems. For finite state systems, the model checker provides a useful decision procedure which relieves the user from tedious details of a proof. Reduction techniques can be used to reduce infinite state systems to finite state, which can then be proved automatically with the model checker. We illustrate this use with an example.

The outline of the rest of the paper is as follows. The next section gives an overview of the Coq system. Section 3 recalls the syntax and semantics of the modal μ-calculus. In section 4 we describe our formalisation of the modal μ-calculus, the proof system underlying the model checker and the correctness proof of the algorithm. Section 5 reports on an example illustrating the combination of deductive proof and automatic proof using the model checker. Finally, section 6 concludes the paper.

2 Overview of Coq

Coq [5] is an interactive proof development system implementing the Calculus of Inductive Constructions (CIC) [22, 25]. The underlying pure Calculus of Constructions [10] is the most powerful system in Barendregt's λ-cube [3]. It combines polymorphic, higher order and dependent types. The additional inductive types provide a powerful and natural mechanism for the definition of data types, specifications and predicates as well as for proofs by structural induction. Formally, CIC is a typed lambda calculus. Its natural deduction style proof rules are used to derive judgements of the form $\Gamma \vdash t \colon T$ meaning that in context Γ, term t has type T. Since proving T in context Γ involves the explicit construction of a λ-term t inhabiting T, the Curry-Howard correspondence allows us to identify proofs with programs and types with specifications.

2.1 The pure calculus

In Coq the following notation for the basic term and type constructions is used: $[x\colon A]M$ is the abstraction of $x\colon A$ from M (usually noted $\lambda x\colon A. M$), $(M\ N)$ denotes application of M to N and $(x : A)B$ the dependent product of A and B (often noted $\forall x : A. M$ or $\Pi x : A. M$). The function space $A \rightarrow B$ is the special case of the product when x does not occur free in B. Function application associates to the left and products to the right. In this paper, we write the dependent product as $\forall x\colon A. M$ in order to improve readability.

Moreover, there are the three constants Prop, Set and Type, called *sorts*. The pure calculus can be specified as the pure type system [3] with sorts $\mathcal{S} = \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}\}$, axioms $\mathcal{A} = \{\mathsf{Prop}\colon \mathsf{Type}, \mathsf{Set}\colon \mathsf{Type}\}$ and rules $\mathcal{R} = \mathcal{S} \times \mathcal{S}$.

2.2 Inductive types and recursion

A positive inductive type is specified by an arity, and a set of constructors. An *arity* is a type of the form $\forall x_1 \colon A_1 \ldots \forall x_n \colon A_n. s$, where s is a sort. We say the arity is of sort s. Along with each inductive type a *structural induction principle* is automatically generated. For our purpose, the definition of inductive types is best explained with a couple of examples.

Example 1. (Natural numbers) The (data) type of natural numbers is specified by the following inductive definition:

$$\mathsf{Inductive}\ \mathtt{nat}\colon \mathsf{Set} := O\colon \mathtt{nat} \mid S\colon \mathtt{nat} \rightarrow \mathtt{nat}.$$

This type has arity Set and two constructors $O\colon$ nat and $S\colon$ nat$\rightarrow$nat. In this case, the induction principle is a term nat_ind of the familiar type:

$$\forall P\colon \mathtt{nat} \rightarrow \mathsf{Prop}. (P\ O) \rightarrow (\forall n\colon \mathtt{nat}. (P\ n) \rightarrow (P\ (S\ n))) \rightarrow \forall n\colon \mathtt{nat}. (P\ n)$$

The construct Cases...of...end defines a function by case analysis; it may be combined with the Fixpoint construct to define primitive recursive functions. For instance, addition on natural numbers can be defined by primitive recursion:

```
Fixpoint add [n: nat]: nat→nat :=
    [m: nat]Cases n of O ⇒ m | (S p) ⇒ (S (add p m)) end.
```

Note that by emphasising the first argument (named n), the system is able to verify that it becomes structurally smaller in each recursive call, thus guaranteeing its termination.

Example 2. (Predicates) The predicate $\leq$ on natural numbers is defined by:

$$\begin{array}{l} \textsf{Inductive}\ \texttt{le}\ [n\colon \texttt{nat}]\colon \texttt{nat} \to \textsf{Prop} := \\ \quad \texttt{le\_n}\colon\ (\texttt{le}\ n\ n) \\ \mid \texttt{le\_S}\colon\ \forall m\colon \texttt{nat}.\,(\texttt{le}\ n\ m) \to (\texttt{le}\ n\ (S\ m)). \end{array}$$

In fact, this defines the family of inductive predicates "$n \leq .$", indexed by $n\colon \texttt{nat}$, to be greater or equal to n.

Example 3. Logical connectives can be defined as non-recursive inductive types. The types of the constructors take the role of introduction rules, while the induction principle provides the elimination rule. As an example, we take existential quantification:

$$\begin{array}{l} \textsf{Inductive ex}[A\colon \textsf{Set}; P\colon A \to \textsf{Prop}]\colon \textsf{Prop} := \\ \quad \texttt{ex\_intro}\colon \forall x\colon A.\,(P\ x) \to (\textsf{ex}\ A\ P). \end{array}$$

The associated induction principle reminds of the $\exists$-elimination rule known from natural deduction:

$$\begin{array}{l} \texttt{ex\_ind} : \forall A\colon \textsf{Set}.\,\forall P\colon A \to \textsf{Prop}.\,\forall Q\colon \textsf{Prop}. \\ \qquad\qquad (\forall x\colon A.\,(P\ x) \to Q) \to (\textsf{ex}\ A\ P) \to Q \end{array}$$

2.3 Program development and extraction

According to Heyting's constructive interpretation of propositions [13], a proof of the formula $\forall x\colon A.\,(P\ x) \to \exists y\colon B.\,(Q\ x\ y)$ is a function taking a value i and a proof of $(P\ i)$ and constructs value o along with a proof that $(Q\ i\ o)$. So, this formula can be understood as the specification of a program with precondition P and input-output relation Q.

Any proof of this specification is a valid implementation. However, from a computational point of view, we are only interested in the input and output values and not in the proofs of P and Q, which are of purely logical content. The two sorts Set and Prop are used to mark terms of computational and of logical content, respectively. The *extraction* function, which we denote by ε, strips off (sub-)terms whose types are of sort Prop, while keeping those with types of sort Set. It also forgets about dependencies of types on terms. Its codomain is the subsystem of CIC without dependent types, called F_ω^{ind}. CIC is used as specification language for F_ω^{ind} programs. These may then be translated into executable Caml programs[1].

In Coq, there is a type `sig` isomorphic to ex but whose arity is of sort Set. It replaces ex in specifications. $(\texttt{sig}\ A\ P)$ is written as $\{x\colon A \mid (P\ x)\}$. Extraction

[1] provided they are typable in Caml, which is the case for most practical applications

yields the inductive type sig′ of arity Set$\rightarrow$Set with its only constructor of type $A \rightarrow (\texttt{sig}'\ A)$. This type can be simplified to the isomorphic type $[A\colon \mathsf{Set}]A$. So, a proof of the specification $\forall x\colon A.\,(P\ x) \rightarrow \{y\colon B \mid (Q\ x\ y)\}$ extracts to a function $f : A \rightarrow B$. The correctness of the extractum is justified by the *realisability* interpretation [20, 21], ensuring in this case that f satisfies $\forall x : A.\,(I\ x) \rightarrow (Q\ x\ (f\ x))$.

Decision procedures are specified by a variant of logical disjunction (with arity of sort Set) given by:

```
Inductive sumbool [A: Prop; B : Prop]: Set :=
  left: A→(sumbool A B) | right: B→(sumbool A B)
```

The notation for (sumbool A B) is $\{A\} + \{B\}$. Its extraction is isomorphic to the type of booleans. For example, $\forall x, y\colon \mathbf{nat}.\ \{x = y\} + \{\neg x = y\}$ specifies a decision procedure for equality on the natural numbers.

Proof methods. There are two possibilities to prove a program specification. The first one is to use the usual tactics and tacticals provided by Coq. Primitive recursive functions are constructed by structural induction on one of their arguments. More sophisticated pattern matching requires stating and proving specialised induction principles, which are then applied to obtain the desired control structure [23].

The idea of the second method is roughly to give the desired program to the system right from the beginning and then apply a special Program tactic which tries to synthesise the computational parts of the proof and generates the logical lemmas necessary to complete the proof. This is the inverse to the extraction process. However, as extraction is not invertible, the raw F_ω^{ind} program is not sufficient and the tactic needs some hints which are given by annotating the program with specifications [19]. Such annotated programs are called *realizers* and the language of realizers is called Real.

3 The propositional modal μ-calculus

The modal μ-calculus subsumes in expressive power many modal and temporal logics such as LTL and CTL. It is interpreted over labelled transition systems (LTS), which are structures of the form $T = (St, Act, \rightarrow)$, where St is a set of states, Act is a set of actions and $\rightarrow \subseteq St \times Act \times St$ is the transition relation. We write $s \xrightarrow{a} t$ for $(s, a, t) \in \rightarrow$. Assume a countable sets Var of variables and AP of atomic propositions. A *model* is a pair (T, ρ) consisting of a LTS T and an environment ρ which assigns to each variable and atomic proposition a set of states. The abstract syntax of the modal μ-calculus is now defined by

$$\phi ::= X \mid A \mid \neg A \mid \phi \vee \phi \mid \phi \wedge \phi \mid \langle\alpha\rangle\phi \mid [\alpha]\phi \mid \mu X\{U\}.\phi \mid \nu X\{U\}.\phi$$

where $X \in Var$ is a variable, $A \in AP$ is an atomic proposition and $\alpha \in Act$ is an action. The fixed point operators μ and ν are tagged with a finite set U of states. We write σ whenever we mean either of μ or ν. The semantics is then inductively defined as follows:

$$\begin{aligned}
\|X\|\,\rho &= \rho(X)\\
\|A\|\,\rho &= \rho(A)\\
\|\neg A\|\,\rho &= St \setminus \|A\|\,\rho\\
\|\phi_0 \vee \phi_1\|\,\rho &= \|\phi_0\|\,\rho \cup \|\phi_1\|\,\rho\\
\|\phi_0 \wedge \phi_1\|\,\rho &= \|\phi_0\|\,\rho \cap \|\phi_1\|\,\rho\\
\|\langle\alpha\rangle\phi\|\,\rho &= \{s \in S \mid \exists s' \in S. s \xrightarrow{\alpha} s' \wedge s' \in \|\phi\|\,\rho\}\\
\|[\alpha]\phi\|\,\rho &= \{s \in S \mid \forall s' \in S. s \xrightarrow{\alpha} s' \Rightarrow s' \in \|\phi\|\,\rho\}\\
\|\mu X\{U\}.\phi\|\,\rho &= \mu S. (\Psi(S) \setminus U)\\
\|\nu X\{U\}.\phi\|\,\rho &= \nu S. (U \cup \Psi(S))
\end{aligned}$$

where $\Psi(S) = \|\phi\|\,\rho[S/X]$. The usual $\sigma X.\phi$ is defined as $\sigma X\{\varnothing\}.\phi$. Note that the false ($F$) and true ($T$) propositions are definable as $\mu X.X$ and $\nu X.X$, respectively. This presentation of the calculus, where negation occurs only in front of atomic proposition is called positive normal form.

4 Implementation of the model checker

This section describes the formalisation of the μ-calculus in Coq and the implementation and correctness proof of the model checker described in [26].

4.1 Fixed points

Assume an arbitrary type U. Then (`Ensemble` U) is the type of sets over U (which are implemented as predicates $U \to$ Prop). We abbreviate this type to `EnsU`. Suppose further that F: `EnsU` $\to$ `EnsU` is a monotone function w.r.t. the inclusion ordering. We define the following two operators `mu` and `nu`:

Definition `mu`: (`EnsU`$\to$`EnsU`) $\to$ `EnsU` :=
  [F: `EnsU`$\to$`EnsU`][s: U] $\forall X$: `EnsU`. (`Included` (F X) X) $\to$ (`In` X s).

Definition `nu`: (`EnsU`$\to$`EnsU`) $\to$ `EnsU` :=
  [F: `EnsU`$\to$`EnsU`][s: U] $\exists X$: `EnsU`. (`Included` X (F X)) $\wedge$ (`In` X s).

According to Tarski's theorem, these two operators define the least and greatest fixed points of F, respectively, as is easily proved in Coq. The next ingredient is Winskel's reduction lemma, which forms the basis for the model checker:

Theorem `Reduction_lemma`:
  (`Included` P (`nu` F)) $\leftrightarrow$
  (`Included` P (F (`nu` [S: `EnsU`]`Union` P (F S)))).

It states that a set P is contained in the greatest fixed point of a monotone function exactly if it is contained in a certain kind of unfolding of that fixed point, where P is added to F under the fixed point operator.

4.2 μ-calculus syntax and semantics

Our development of the model checker will be parametrised by a labelled transition system. We assume that the set of states is finite and that we have a function which, for any state s and action a, computes a list of a-successors of s. This is expressed in the following lines:

```
Parameter Act, St : Set.
Parameter Trans: St → Act → St → Prop.
Axiom finite_state : (Finite (Full_set St)).
Axiom post_spec :
  ∀s: St.∀a: Act. {l: (list St) | ∀t: St.(Elem t l) ↔ (Trans s a t)}.
```

The inductive type defining the syntax is then defined by:

```
Inductive MuForm : Set :=
    Var:  nat → MuForm
  | Lit:  (St → bool) → MuForm
  | And:  MuForm → MuForm → MuForm
  | Or:   MuForm → MuForm → MuForm
  | Box:  Act → MuForm → MuForm
  | Dia:  Act → MuForm → MuForm
  | Mu:   (list St) → MuForm → MuForm
  | Nu:   (list St) → MuForm → MuForm.
```

Variables are encoded in the standard way using de Bruijn indices. The valuation of atomic propositions is directly coded into the syntax in the form of computable predicates of type St → bool. Since this type is closed under negation we can drop negation altogether from the syntax. The fixed point operators are tagged with a list of states.

The type Env of environments is defined as nat → EnsSt, which can be seen as an infinite lists of sets of states. We introduce an operation env_cons : EnsSt → Env → Env with (env_cons R ρ) returning R for O and (ρ j) for $j + 1$. The function recursively computing the semantics of a formula ϕ with respect to environment ρ is defined by:

```
Fixpoint Sem [φ: MuForm] : Env → EnsSt :=
  [ρ : Env]Cases φ of
      (Var i)      ⇒ (ρ i)
    | (Lit p)      ⇒ (cf2ens St p)
    | (And φ1 φ2)  ⇒ (Intersection St (Sem φ1 ρ) (Sem φ2 ρ))
    | (Or φ1 φ2)   ⇒ (Union St (Sem φ1 ρ) (Sem φ2 ρ))
    | (Box a φ)    ⇒ (BoxSem a (Sem φ ρ))
    | (Dia a φ)    ⇒ (DiaSem a (Sem φ ρ))
    | (Mu l φ)     ⇒ (MuSem l [R: EnsSt](Sem φ (env_cons R ρ)))
    | (Nu l φ)     ⇒ (NuSem l [R: EnsSt](Sem φ (env_cons R ρ)))
  end.
```

The function (`cf2ens St`) transforms a predicate of type `St` $\rightarrow$ `bool` into the set of states (of type `EnsSt`) verifying the predicate. `BoxSem`, `DiaSem` are the predicate transformers defining the semantics of the modalities. In the cases of the fixed point operators, the second argument to `MuSem` and `NuSem` is the de Bruijn version of $\lambda S.\, ||\phi||\, \rho[S/X]$ when X is the variable bound to the fixed point operator. Here, `env_cons` has the effect of shifting the interpretation of free variables by one, accounting for the increased abstraction depth under these operators. For illustration, we give the definitions of `DiaSem` and `NuSem`.

```
Inductive DiaSem [a: Act; R: EnsSt]: EnsSt :=
  dia_intro : ∀s,t: St. (Trans s a t) → (In St R t) → (In St (DiaSem a R) s).

Definition NuSem: (list St) → (EnsSt → EnsSt) → EnsSt :=
  [P: (list St)][Φ: EnsSt → EnsSt]
    (nu St [R: EnsSt](Union St (list2ens St P) (Φ R))).
```

Substitution. We define the type of substitutions `Subst` to be the functions of type `nat` $\rightarrow$ `MuForm` assigning each variable a μ-calculus formula. Substitution is thus a function `subst` : `MuForm` $\rightarrow$ `Subst` $\rightarrow$ `MuForm`. The following table introduces some notation which is useful in the context of de Bruijn-coded variables:

notation	**definition**	**name**
id	$[i\colon \mathtt{nat}](\mathtt{Var}\ i)$	"identity"
$\uparrow$	$[i\colon \mathtt{nat}](\mathtt{Var}\ (S\ i))$	"shift"
$\phi \cdot \theta$	$[i\colon \mathtt{nat}]\mathsf{Cases}\ i\ \mathsf{of} O \Rightarrow \phi \mid (S\ k) \Rightarrow (\theta\ k)\ \mathsf{end}$	"cons"
$\theta \circ \theta'$	$[i\colon \mathtt{nat}](\mathtt{subst}\ (\theta\ i)\ \theta')$	"composition"
$\Uparrow(\theta)$	$O \cdot (\theta \circ \uparrow)$	"lift"

In order to improve readability, we will use the usual notation $\phi[\theta]$ instead of (`subst` $\phi\ \theta$). In `subst`, the cases of the fixed point operators use 'lift' to push substitution inside, i.e. we have $(\sigma\ l\ \psi)[\theta] =_\beta (\sigma\ l\ (\psi[\Uparrow(\theta)]))$. The 'cons' operator is useful in unfolding fixed point formulas: $\psi[(\mathtt{Nu}\ l\ \psi) \cdot id]$ corresponds to the unfolding of $(\mathtt{Nu}\ l\ \psi)$. With these definitions, we can prove:

Lemma 4. $\phi \cdot (\theta \circ \theta') = \Uparrow(\theta) \circ (\phi \cdot \theta')$.

The next lemma establishes a standard semantical correspondence between substitution and environment. It is proved is by structural induction on ϕ.

```
Lemma Substitution_lemma :
  ∀φ: MuForm. ∀ρ: Env. ∀θ: Subst.
    (Sem φ[θ] ρ) = (Sem φ [i: nat](Sem (θ i) ρ)).
```

4.3 Correctness assertions

The satisfaction relation `sat` on states and formulas is defined as:

```
Inductive sat [s: St; φ: MuForm]: Prop :=
  sat_intro: (∀ρ: Env. (In St (Sem φ ρ) s)) → (sat s φ).
```

We call the proposition (sat s ϕ) a *correctness assertion* and write it as $s \models \phi$. In Coq, we can prove the following lemma:

Lemma 5. *For $\varphi, \varphi_0, \varphi_1$ and $(\sigma\ l\ \psi)$ closed formulas, we have*

1. $s \models (\mathtt{And}\ \phi_0\ \phi_1) \leftrightarrow s \models \phi_0 \wedge s \models \phi_1$
2. $s \models (\mathtt{Or}\ \phi_0\ \phi_1) \leftrightarrow s \models \phi_0 \vee s \models \phi_1$
3. $s \models (\mathtt{Dia}\ a\ \phi) \leftrightarrow \exists s' : \mathtt{St}. (\mathtt{Trans}\ s\ a\ s') \wedge s' \models \phi$
4. $s \models (\mathtt{Box}\ a\ \phi) \leftrightarrow \forall s' : \mathtt{St}. (\mathtt{Trans}\ s\ a\ s') \rightarrow s' \models \phi$
5. *if* $(\mathtt{Elem}\ s\ l)$ *then (a)* $\neg(s \models (\mathtt{Mu}\ l\ \psi))$, *and (b)* $s \models (\mathtt{Nu}\ l\ \psi)$
6. *if* $\neg(\mathtt{Elem}\ s\ l)$ *then for* $\sigma \in \{\mathtt{Mu}, \mathtt{Nu}\}$*:*
   $s \models (\sigma\ l\ \psi) \leftrightarrow s \models \phi[(\sigma\ (\mathtt{cons}\ s\ l)\ \psi) \cdot id]$

Proof. Items (1)-(5) follow directly from the semantic definition. For (6), we need the Reduction and Substitution Lemmas. In the case of the least fixed point, a dual version of the Reduction Lemma is used. □

These equivalences, when cast into proof rules, can be used to establish properties of arbitrary (possibly infinite state) transition systems deductively.

4.4 The algorithm

In this section, we describe the specification and correctness proof of Winskel's local model checking algorithm [26] in Coq. It decides the truth or falsity of correctness assertions by exploring the neighbourhood of the state of interest. The idea is to exploit the equivalences of the previous Lemma 5 by considering them as simplification rules (in going from left to right). As our transition system is assumed to be finite, the quantifiers appearing in cases (3) and (4) of the modal operators can be replaced by finite disjunction and conjunction, respectively.

Specification. Given a closed formula ϕ of the μ-calculus and a state s of the transition system, the model checker is supposed to decide whether s satisfies ϕ or it not. This leads us to the following Coq specification:

$$\mathtt{MuChk}: \forall\phi: \mathtt{MuForm}. (\mathtt{Closed}\ \phi) \rightarrow \forall s: \mathtt{St}. \{s \models \phi\} + \{\neg(s \models \phi)\}.$$

We apply Lemma 5 in order to gradually transform the decision problem into (boolean combinations of) simpler ones. The fixed point operators are dealt with by unfolding them while adding the current state to the tag, whenever it is not already there. In cases 1-4 there is a structural reduction in going from left to right. Case 5 provides the base. In case 6 the reduction is less obvious. This means that the correctness proof will proceed by well-founded induction. However, the proof also requires that we extend our specification to arbitrary formulas, be they open or closed. This leads to the following generalised specification MuChk_plus, using the auxiliary predicates Q and Q^+.

Definition Q: $\texttt{MuForm} \to \mathsf{Set} :=$
$[\phi\colon \texttt{MuForm}]\,\forall s\colon \texttt{St}.\,\{s \models \phi\} + \{\neg(s \models \phi)\}$.

Definition Q^+: $\texttt{MuForm} \to \mathsf{Set} :=$
$[\phi\colon \texttt{MuForm}]\,\forall \theta\colon \texttt{Subst}.$
$\big(\forall i\colon \texttt{nat}.\,(\texttt{Elem}\ i\ (\texttt{fv}\ \phi)) \to (\texttt{Closed}\ (\theta\ i))\big) \to$
$\big(\forall i\colon \texttt{nat}.\,(\texttt{Elem}\ i\ (\texttt{fv}\ \phi)) \to (Q\ (\theta\ i))\big) \to (Q\ \phi[\theta])$.

$\texttt{MuChk\_plus}\colon\ \forall \phi\colon \texttt{MuForm}.\,(Q^+\ \phi)$

The first condition in the definition of Q^+ means that the substitute $(\theta\ i)$ for each free variable i of ϕ is a closed formula. The second condition expresses the assumption that we know how to decide the satisfaction problem for these substitutes. Since a closed formula trivially satisfies both of these conditions, Q^+ is equivalent to Q in this case. With these definitions the original specification MuChk reads $\forall \phi\colon \texttt{MuForm}.\,(\texttt{Closed}\ \phi) \to (Q\ \phi)$.

As we will make extensive use of the proof synthesis method provided by the tactic `Program`, it is interesting to look at the types extracted from Q and Q^+:

$$\varepsilon(Q) = \texttt{St} \to \texttt{sumbool}$$
$$\varepsilon(Q^+) = \texttt{Subst} \to (\texttt{nat} \to \varepsilon(Q)) \to \varepsilon(Q)$$

The term $(\forall i\colon \texttt{nat}.\,(\texttt{Elem}\ i\ (\texttt{fv}\ \phi)) \to (\texttt{Closed}\ (\theta\ i)))$ in Q^+ is of sort Prop and is therefore eliminated by extraction. Also, in $(\forall i\colon \texttt{nat}.\,(\texttt{Elem}\ i\ (\texttt{fv}\ \phi)) \to (Q\ (\theta\ i)))$ the subterm $(\texttt{Elem}\ i\ (\texttt{fv}\ \phi))$ is of sort Prop and thus removed. Extraction also forgets about the dependencies of Q and Q^+ on MuForm. Therefore, both extracted types are of the informative sort Set.

Correctness proof. We prove the generalised specification MuChk_plus by well-founded induction. The well-founded induction principle (WFI) is a theorem part of the Coq library. It is stated in the following.

$\texttt{well\_founded\_induction}$:
$\forall A\colon \mathsf{Set}.\,\forall R\colon A \to A \to \mathsf{Prop}.\,(\texttt{well\_founded}\ A\ R) \to$
$\forall P\colon A \to \mathsf{Set}.\,(\forall x\colon A.\,(\forall y\colon A.\,(R\ y\ x) \to (P\ y)) \to (Px)) \to \forall a\colon A.\,(P\ a)$

The computational content of the proof of the well-founded induction principle obtained by extraction is a general recursor. Its type is $\forall A, P\colon \mathsf{Set}.\,(A \to (A \to P) \to P) \to A \to P$. Note, however, that by the recursive realisability interpretation [23] any program extracted from a proof by well-founded induction is guaranteed to terminate on arguments satisfying the specified preconditions.

Proof of main theorem MuChk_plus. As we follow basically the proof in [26], we try here to point out the application of the proof method provided by realizers and the `Program` tactic.

Definition 6. Let $\prec$ be the proper one-step[2] subformula relation on μ-calculus formulas. Then relation R: $\texttt{MuForm}\rightarrow\texttt{MuForm}\rightarrow\textsf{Prop}$ is defined by:

$$[\phi,\phi'\colon \texttt{MuForm}]\big(\phi \prec \phi' \ \vee\ \exists s\colon \texttt{St}.\,\exists l\colon (\texttt{list St}).\,\exists\psi\colon \texttt{MuForm}.\\ (\neg(\texttt{Elem}\ s\ l)\ \wedge\ \phi \equiv (\sigma\ (\texttt{cons}\ s\ l)\ \psi)\ \wedge\ \phi' \equiv (\sigma\ l\ \psi))\big)$$

Well-foundedness of R follows from the assumption that the set of states St is finite. By the well-founded induction principle, MuChk_plus follows from:

$$\forall\phi\colon \texttt{MuForm}.\,(\forall\psi\colon \texttt{MuForm}.\,(R\ \psi\ \phi)\rightarrow(Q^+\ \psi))\rightarrow(Q^+\ \phi). \tag{1}$$

The proof proceeds by case analysis on the form of ϕ, which generates eight subgoals, one for each constructor of MuForm. We pick out the case of the greatest fixed point which we state as the lemma:

Lemma chk_Nu_plus:
$\quad\forall l\colon (\texttt{list St}).\,\forall\phi\colon \texttt{MuForm}.$
$\quad\quad(\forall\psi\colon \texttt{MuForm}.\,(R\ \psi\ (\texttt{Nu}\ l\ \phi))\rightarrow(Q^+\ \psi))\rightarrow(Q^+\ (\texttt{Nu}\ l\ \phi)).$

After unfolding the definitions of Q^+ and Q, introducing the hypothesis into the context and pushing substitution inside Nu, we obtain the sequent:

h: $\forall\psi\colon \texttt{MuForm}.\,(R\ \psi\ (\texttt{Nu}\ l\ \phi))\rightarrow$
$\quad\quad\forall\theta'\colon \texttt{Subst}.$
$\quad\quad\quad(\forall j\colon \texttt{nat}.\,(\texttt{Elem}\ j\ (\texttt{fv}\ (\texttt{Nu}\ l\ \psi)))\rightarrow(\texttt{Closed}\ (\theta'\ j)))\rightarrow$
$\quad\quad\quad\quad(\forall j\colon \texttt{nat}.\,(\texttt{Elem}\ j\ (\texttt{fv}\ (\texttt{Nu}\ l\ \psi))\rightarrow(Q\ (\theta'\ j)))\rightarrow(Q\ \psi[\theta'])$
θ: Subst
h_0: $\forall i_0\colon \texttt{nat}.\,(\texttt{Elem}\ i_0\ (\texttt{fv}\ (\texttt{Nu}\ l\ \phi)))\rightarrow(\texttt{Closed}\ (\theta\ i_0))$
h_1: $\forall i_0\colon \texttt{nat}.\,(\texttt{Elem}\ i_0\ (\texttt{fv}\ (\texttt{Nu}\ l\ \phi))\rightarrow(Q\ (\theta\ i_0))$
s: St
============================
$\quad\{s \models (\texttt{Nu}\ l\ (\phi[\Uparrow(\theta)]))\}+\{\neg(s \models (\texttt{Nu}\ l\ (\phi[\Uparrow(\theta)])))\}$

Note that the extracted type of the induction hypothesis h is $\texttt{MuForm}\rightarrow\varepsilon(Q^+)$. The realizer for this goal depends on two lemmas which are proved in the context above with the additional hypothesis $nel\colon \neg(\texttt{Elem}\ s\ l)$. The first one is:

Lemma Q_Nu_cons: $(Q\ (\texttt{Nu}\ (\texttt{cons}\ s\ l)\ \phi)[\theta])$
Realizer $(h\ (\texttt{Nu}\ (\texttt{cons}\ s\ l)\ \phi)\ \theta\ h_1)$.

This is automatically proved by Program_all using Definition 6. The second one corresponds to the right hand side of Lemma 5(6):

Lemma Q_Nu_unfold: $\big(Q\ (\phi[\Uparrow(\theta)])\big[(\texttt{Nu}\ (\texttt{cons}\ s\ l)\ (\phi[\Uparrow(\theta)]))\cdot id\big]\big)$

[2] i.e. if $\phi \prec \phi'$ then there is no ϕ'' s.t. $\phi \prec \phi'' \prec \phi'$

Using Lemma 4, we first rewrite this to $(Q\ \phi[(\texttt{Nu}\ (\texttt{cons}\ s\ l)\ (\phi[\Uparrow(\theta)]))\cdot\theta])$. Now, since by Lemma Q_Nu_cons we know how to decide $(\texttt{Nu}\ (\texttt{cons}\ s\ l)\ (\phi[\Uparrow(\theta)]))$ (which is convertible with $(\texttt{Nu}\ (\texttt{cons}\ s\ l)\ \phi)[\theta]$) and by hypothesis h_1 we know how to do so for each $(\theta\ i)$, we can use the induction hypothesis h to construct the following realizer

```
Realizer (h φ (Nu (cons s l) (φ[⇑(θ)])) · θ
         [i: nat]Cases i of O ⇒ Q_Nu_cons | (S j) ⇒ (h1 j) end)
```

Applying the tactic Program_all leaves us with two subgoals which are easily solved. Now, with Lemma 5(5b) and (6) in mind, we are ready to give the realizer for the goal of our original sequent:

```
Realizer if (is_elem_spec s l) then true else (Q_Nu_unfold s).
```

where is_elem_spec: $\forall s$: St.$\forall l$: (list St).$\{$(Elem s l)$\}+\{\neg$(Elem s l)$\}$. The subgoals generated by Program_all are all easily proved using Lemmas 5(5b) and 5(6).

A realizer for the control structure. The steps taken in the beginning of the proof (application of the WFI and case analysis) can be replaced by the following realizer for MuChk_plus:

```
Realizer <Q+>rec muchk_plus :: :: {R}
  [φ: MuForm]Cases φ of
     (Var i)          ⇒ (chk_Var_plus i)
   | (Lit p)          ⇒ (chk_Lit_plus p)
   | (constr args)    ⇒ (chk_constr_plus args muchk_plus)
   | ...                ...
  end.
```

The notation <P>rec h :: :: $\{R\}$ $[a{:}\ A]M$, where h is the name of the induction hypothesis and $M : P$, is syntactic sugar for (well_founded_induction A P $[a{:}\ A][h{:}\ A \to P]M$). The identifiers chk_*constr*_plus, where *constr* is the name of a recursive constructor of MuForm, denote lemmas proving the different cases for ϕ in subgoal (1).

5 Application

In this section we illustrate a possible combination of theorem proving and model checking with an example. We want to show that a simple synchronous protocol P_n is observationally equivalent to a single buffer cell independently of the capacity n of the channel it uses. The proof uses the specification preorder [17, 9] and a network invariant [27] representing the whole family of (fixed-size) channels in order to reduce the problem to the verification of three preorder relations. These relations can then be established with the model checker by using a μ-calculus formula characterising the specification preorder.

All the notions in this section have been formalised in Coq. We use usual mathematical notation for brevity.

CCS and the specification preorder. We recall the basic definitions. For more detail, we refer the reader to [17,9]. Let $\mathcal{A}$ be a set of *names*, their complements $\overline{\mathcal{A}} = \{\bar{l} \mid l \in \mathcal{A}\}$ and the set of *labels* $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$. We set $\bar{\bar{l}} = l$. f Define the set of *actions* by $Act = \mathcal{L} \cup \{\tau\}$, where with τ the *invisible/silent* action. f is a relabelling function if $f(\bar{l}) = \overline{f(l)}$ and $f(\tau) = \tau$. Suppose a set $\mathcal{K}$ of *process constants.* The set $\mathcal{P}$ of processes is defined by the abstract syntax:

$$p ::= nil \mid \perp \mid a.p \mid p_0 + p_1 \mid p_0 \mid p_1 \mid p[f] \mid p \backslash L \mid A$$

where $a \in Act$, f a relabelling function, $L \subseteq \mathcal{L}$ and $A \in \mathcal{K}$. $\perp$ represents the undefined process. Let T be the transition system $(\mathcal{P}, Act, \rightarrow)$, whose transition relation $\rightarrow$ is inductively defined by the rules:

$$\begin{array}{rcl}
a.p \xrightarrow{a} p & & \\
p \xrightarrow{a} p' & \Rightarrow & p+q \xrightarrow{a} p',\ \ q+p \xrightarrow{a} p', \\
& & p \mid q \xrightarrow{a} p',\ \ q \mid p \xrightarrow{a} p',\ \ p\{f\} \xrightarrow{f(a)} p'\{f\} \\
p \xrightarrow{a} p',\ a, \bar{a} \notin L & \Rightarrow & p \backslash L \xrightarrow{a} p' \backslash L \\
p \xrightarrow{a} p',\ A \overset{\text{def}}{=} p & \Rightarrow & A \xrightarrow{a} p'
\end{array}$$

The *partiality* predicate $\uparrow$ is the complement of $\downarrow$ which is defined by: (i) $nil \downarrow$, $a.p \downarrow$, (ii) $p \downarrow,\ q \downarrow \Rightarrow\ \ p+q \downarrow,\ p \mid q \downarrow$, (iii) $p \downarrow \Rightarrow\ \ p \backslash L \downarrow,\ p[f] \downarrow$, (iv) $A \overset{\text{def}}{=} p,\ p \downarrow \Rightarrow\ A \downarrow$. Intuitively, $\uparrow$ denotes the underdefined processes.

Definition 7. Let $l \in \mathcal{L}$ and $a \in Act$. Define

1. $\overset{l}{\Rightarrow} = \overset{\tau^*}{\rightarrow} \overset{l}{\rightarrow} \overset{\tau^*}{\rightarrow}$ and $\overset{\tau}{\Rightarrow} = \overset{\tau^*}{\rightarrow}$
2. $p \Uparrow$ iff $\exists p'. p \overset{\varepsilon}{\Rightarrow} p' \wedge p \uparrow$
3. $p \Uparrow a$ iff $p \Uparrow \vee \exists p'. (p \overset{\varepsilon}{\Rightarrow} p' \wedge p' \Uparrow)$

$p \Downarrow$ ($p \Downarrow a$) is the complement of $p \Uparrow$ ($p \Uparrow a$). We say that a process p is *totally defined* if for all p' reachable from p: $p \Downarrow$. Otherwise, it is *partially defined.*

Definition 8. Define the *specification preorder* $\trianglelefteq$ as the greatest fixed point of the function F on relations over $\mathcal{P}$ defined by $(p, q) \in F(R)$ iff for all $a \in Act$ s.t. $p \Downarrow a$ we have:

1. $q \Downarrow a$,
2. if $p \xrightarrow{a} p'$ then $\exists q'. q \overset{a}{\Rightarrow} q' \wedge (p', q') \in R$,
3. if $q \xrightarrow{a} q'$ then $\exists p'. p \overset{a}{\Rightarrow} p' \wedge (p', q') \in R$.

Let $\approx$ denote weak bisimulation equivalence [17].

Lemma 9. *If $p \trianglelefteq q$ and p is totally defined, then q is totally defined and $p \approx q$.*

Theorem 10. *([9]) The preorder $\trianglelefteq$ is a precongruence w.r.t. parallel composition, restriction and relabelling, i.e. if $p \trianglelefteq q$ then $p \mid r \trianglelefteq q \mid r$, $p\{f\} \trianglelefteq q\{f\}$ and $p \backslash L \trianglelefteq q \backslash L$.*

Verification of $\trianglelefteq$ using the model checker. It is possible to define a μ-calculus formula characterising the specification preorder. This formula has to be interpreted over pairs of processes. Therefore, we introduce the product transition system $T^{\times} = (\mathcal{P} \times \mathcal{P}, Act \uplus Act, \longrightarrow_{\times})$, where $\longrightarrow_{\times}$ is defined by the two rules:

$$\frac{p \xrightarrow{a} p'}{(p,q) \xrightarrow{\iota_0(a)}_{\times} (p',q)} \qquad \frac{q \xrightarrow{a} q'}{(p,q) \xrightarrow{\iota_1(a)}_{\times} (p,q')}$$

$\uplus$ denotes disjoint sum with injections ι_0 and ι_1. Next, we define some left and right modalities for the μ-calculus interpreted over the transition system $T^{\times}$:

$$\begin{array}{ll} \langle a\rangle_l\, \phi = \langle \iota_0(a)\rangle \phi & \langle \tau^*\rangle_l\, \phi = \mu X.\, \phi \vee \langle \tau\rangle_l X \quad (X \notin FV(\phi)) \\ \langle\langle \ell\rangle\rangle_l\, \phi = \langle \tau^*\rangle_l \langle \ell\rangle_l \langle \tau^*\rangle_l\, \phi \quad (\ell \in \mathcal{L}) & \langle\langle \tau\rangle\rangle_l\, \phi = \langle \tau^*\rangle_l\, \phi \end{array}$$

Of all these we define "right" versions, but with $\langle a\rangle_r \phi = \langle \iota_1(a)\rangle \phi$. Left and right box modalities, $[a]_\ell$ and $[a]_r$, are similarly defined. We also introduce left/right versions of the partiality predicates:

$$\uparrow_l = \uparrow \times \mathcal{P} \qquad \Uparrow_l = \langle\langle \tau\rangle\rangle_l \uparrow_l \qquad \Uparrow_l(a) = \Uparrow_l \vee \langle\langle a\rangle\rangle_l \Uparrow_l$$

Similarly, "right" versions are defined using $\uparrow_r = \mathcal{P} \times \uparrow$. Now, supposing the set Act is finite, the function F from definition 8 can be expressed as the μ-calculus formula:

$$\tilde{F}(X) = \bigwedge_{a \in Act} \Big(\neg \Uparrow_l(a) \vee (\Uparrow_r(a) \ \wedge\ [a]_l \langle\langle a\rangle\rangle_r X \ \wedge\ [a]_r \langle\langle a\rangle\rangle_l X)\Big)$$

We define $\chi_{\trianglelefteq} = \nu X.\tilde{F}(X)$. This formula characterises the specification preorder as is stated in the following result:

Lemma 11. *For Act finite:* $p \trianglelefteq q \ \Leftrightarrow\ (p,q) \models \chi_{\trianglelefteq}$.

A simple protocol. A simple protocol P_n is composed of a sender S synchronously transmitting signals over a buffer B^n of size n to a receiver R. With $X \parallel Y \stackrel{\text{def}}{=} (X[out/z] \mid Y[in/z])\backslash\{z\}$, the definition is:

$$\begin{array}{ll} B \stackrel{\text{def}}{=} in.\overline{out}.B & B^n \stackrel{\text{def}}{=} \|_{i=1}^{n}\, B \\ S \stackrel{\text{def}}{=} send.\overline{in}.ack.S & R \stackrel{\text{def}}{=} out.\overline{recv}.\overline{ack}.R \\ E \stackrel{\text{def}}{=} (S \mid R)\backslash\{ack\} & P_n \stackrel{\text{def}}{=} (E \mid B^n)\backslash\{in, out\} \end{array}$$

We define a specification of the protocol by $Spec \stackrel{\text{def}}{=} send.\overline{recv}.Spec$. We want to show that the behaviour of the protocol is independent of the size of the buffer.

Theorem 12. *For all* $n \geq 1$: $Spec \approx P_n$.

Proof. The proof is decomposed into the following two steps:

1. find a network invariant J such that for all $n \geq 1$: $J \trianglelefteq B^n$
2. verify that $Spec \trianglelefteq (E \mid J)\backslash\{in, out\}$

The result then follows from Theorem 10 and Lemma 9, a fact which is proved by deduction in Coq. We define the network invariant J as follows:

$$J \stackrel{\text{def}}{=} in.J' \qquad\qquad J' \stackrel{\text{def}}{=} \overline{out}.J + in.\perp$$

Step (1) is proved by an implicit induction on n: (a) $J \trianglelefteq B$ (base case) (b) $J \trianglelefteq B \parallel J$ (inductive step). Both these steps can be proved with the model checker, by using the characteristic formula $\chi_{\trianglelefteq}$. That (a) and (b) imply (1) is proved "by hand" in Coq. Step (2) can be delegated to the model checker as well. □

As any property, expressed in a version of the modal μ-calculus with weak modalities only, is preserved by weak bisimulation equivalence, we can verify it on the specification *Spec* and conclude that it also holds for each of the P_n.

6 Conclusions

We have formalised a model checker for the modal μ-calculus and its correctness proof in Coq. From this proof we can extract an executable Caml program, which can be considered as a trusted decision procedure. In this way, we are able to avoid the inefficiency due to the generation of large proof objects, while preserving the high reliability of the proof environment. Further, we have illustrated the combination of theorem proving and model checking with an example.

Our development is parametrised by an arbitrary labelled transition system and by a function computing successor states. This makes it possible to use any formalism for specification whose operational semantics can be given by a LTS with computable successor function. This is in contrast to [28], where (part of) the implementation depends on the particular formalism used, because they use an external program to generate Lego proof terms.

On the other hand, their model checker is more tightly integrated into the proof environment. In our approach, although direct normalisation of the proof inside Coq is possible, it is very inefficient, since a lot of logical information (which is stripped off by extraction) has to be carried around in the process. When using the extracted Caml program, we currently have to introduce the results of model checking by hand as axioms into Coq. A tighter integration would be desirable. Generally speaking, some way of reintroducing the results of extracted programs (together with the guarantees given by the realisability interpretation) into Coq would be needed for a tighter integration of efficient decision procedures. Some techniques currently being developed by B. Barras [4] and aiming at bootstrapping Coq in Coq might provide a solution to this problem in the future.

The efficiency of the algorithm could be improved. No optimisations have been made yet. One obvious optimisation would be to replace the tag lists in the fixed point formulas by a more efficient data structure such as balanced trees. Other optimisations such as the ones proposed in [8] are possible.

As part of future work we plan to investigate the application of abstract interpretation techniques to value-passing process calculi. Another track concerns model checkers for infinite state systems such as the ones proposed by Bradfield and Stirling [6] and Andersen [1]. It would be interesting to see to what extent these techniques could be automated in Coq.

Acknowledgements. I would like to thank the Coq development team at INRIA Rocquencourt for helpful discussions and in particular Hugo Herbelin for inviting me there.

References

1. H. R. Andersen. *Verification of Temporal Properties of Concurrent Systems.* PhD thesis, Computer Science Department, Aarhus University, June 1993.
2. L. Augustsson, T. Coquand, and B. Nordström. A short description of another logical framework. In G. Huet and P. G., editors, *Preliminary Proceedings of Logical Frameworks*, 1990.
3. H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2: Background: Computational Structures, pages 118–309. Oxford University Press, 1992.
4. B. Barras. Coq en Coq. Technical Report 3026, INRIA, Oct. 1996.
5. B. Barras, S. Boutin, C. Cornes, J. Courant, j.-C. Filiâtre, E. Giménez, H. Herbelin, G. Huet, and al . *The Coq Proof Assistant Reference Manual, Version 6.1.* Projet Coq, INRIA Rocquencourt, CNRS - ENS Lyon, Dec. 1996.
6. J. Bradfield and C. Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96:157–174, 1992.
7. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, Sept. 1994.
8. R. Cleaveland. Tableau-based model checking in the propositional μ-calculus. *Acta Informatica*, 27:725–747, 1990.
9. R. Cleaveland and B. Steffen. A preorder for partial process specifications. In *CONCUR' 90*, volume 458 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.
10. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
11. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.
12. J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *CAV '95*, volume 939 of *Lecture Notes in Computer Science.* Springer-Verlag, 1995.
13. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambrdge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.
14. R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *8th ACM Symposium on Principles of Distributed Computing*, pages 239–248, 1989.

15. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and B. S. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.
16. Z. Luo and R. Pollack. Lego proof development system: User's manual. Technical Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.
17. R. Milner. *Communication and Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall, 1989.
18. O. Müller and T. Nipkow. Combining model checking and deduction for I/O-automata. In *TACAS 95*, volume 1019 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1995.
19. C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives.* PhD thesis, Ecole Normale Supérieure de Lyon, Jan. 1995.
20. C. Paulin-Mohring. Extracting F_ω programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on the Priciples of Programming Languages*, Austin, Texas, Jan. 1989.
21. C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* PhD thesis, Université de Paris VII, Jan. 1989.
22. C. Paulin-Mohring. Inductive definitions in the system Coq – rules and properties. Technical Report 92-49, Laboratoire de l'Informatique du Parallélisme, ENS Lyon, France, Dec. 1992.
23. C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system coq. *Journal of Symbolic Computation*, 11:1–34, 1993.
24. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97. Springer-Verlag, 1995.
25. B. Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université de Paris 7, France, 1994.
26. G. Winskel. A note on model checking the modal μ-calculus. *Theoretical Computer Science*, 83:157–167, 1991.
27. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, 1989.
28. S. Yu and Z. Luo. Implementing a model checker for LEGO. In *Formal Methods Europe*, 1997.