Modelisation of Timed Automata in Coq

Christine Paulin-Mohring

Université Paris-Sud, Laboratoire de Recherche en Informatique,
UMR 8623 CNRS, Bâtiment 490, F-91405 Orsay Cedex, France,
paulin@lri.fr, http://www.lri.fr/~paulin

Abstract. This paper presents the modelisation of a special class of timed automata, named p-automata in the proof assistant Coq. This work was performed in the framework of the CALIFE project[1] which aims to build a general platform for specification, validation and test of critical algorithms involved in telecommunications. This paper does not contain new theoretical results but explains how to combine and adapt known techniques in order to build an environment dedicated to a class of problems. It emphasizes the specific features of Coq which have been used, in particular dependent types and tactics based on computational reflection.

1 Introduction

An important research area is the design of methods and tools for simplifying the design of zero-default programs. Formal methods are mainly used in the context of specific algorithms involved in critical systems. The behavior of the program and its expected specification are represented in a mathematical model and then a proof of correctness has to be produced. This task represents often a lot of work but it cannot be avoided in order to gain a high-level of confidence in the appropriate behavior of the program. Formal models are indeed required for getting the highest level of certification for systems manipulating critical informations like smart-cards. The generalization of formal methods requires to provide appropriate computer tools and specialized libraries for simplifying the task of development of new algorithms.

1.1 Theorem Proving vs Model-Checking

Theorem proving and model-checking are two different formal methods for the validation of critical systems. Model-checking tries to automatically prove properties of programs represented as labelled transition systems. A specification on transition systems is often expressed as a temporal logic formula that is checked using exploration techniques on the graph representing the transition system. There are well-known problems when using model-checking: the model of the program has to be simplified in order to be treated automatically, then it becomes more difficult or impossible to insure that what is proven corresponds

[1] http://www.loria.fr/Calife

N. Kobayashi and B.C. Pierce (Eds.): TACS 2001, LNCS 2215, pp. 298–315, 2001.

to the real world. Because they are complex programs, model-checker are not bug-free. Model-checkers are appropriate tools for the automatic detection of errors. On the other hand, if the program correctness property is expressed as a mathematical statement, it is possible to handle it using general tools for theorem proving. The interest of this method is to be able to represent complicated systems involving non-finite or non specified data or continuous time. However, for such theories, the complete automation of proofs is hopeless and then the work has to performed by hand.

During the last years, there has been several attempts to combine these two methods. One possibility is to use a theorem prover to reduce an abstract problem into one that can be handle by a model-checker used as an external tool. It is possible to trust the model-checker like in the system PVS [19] or to be more suspicious and keep a trace of theorems which depend on properties justified by a non-certified procedure like in HOL [13]. Another attempt is to integrate model-checkers to theorem provers in a certified way [24,26,22] but it raises the problem of efficiency. Theorem proving can also be used in order to prove that a property is an invariant of the labelled transition system which models the system [12]. Interactive theorem prover based on higher-order logic are also appropriate tools for mechanizing meta-reasoning on the sophisticated formalisms involved in the modelisation of concurrent systems like the π-calculus [15,16].

1.2 Studying a Conformance Protocol

In this paper, we are considering the interaction between theorem proving and automatic proof-procedures in the context of the study of a conformance protocol (named ABR) normalized by France Telecom R& D to be used in ATM networks. This example has been widely studied since it was introduced as an industrial challenge for formal methods. Proofs of this protocol have been both done using theorem provers and automatic tools. A recent synthetic presentation of these experiments can be found in [4].

The purpose of the conformance algorithm is to control the bit rate available to the users in an ATM network. The system receives information from the network on the maximum rate available at time t, it also receives the data from the user and checks that the actually used rate does not exceed the available rate. But there are delays in the transmission, such that the information sent by the network at time t will only be known to the user after time $t+\tau_3$ and no later than $t+\tau_2$. The general principle is that the user is always allowed to choose the best rate available to him or her. Assume a maximum rate R_1 arrives at time t_1, the user should wait until $t_1 + \tau_3$ to take this rate into account. Assume R_1 is less than the rate available before arrival of R_1, the user may wait until $t_1+\tau_2$ to conform to the R_1 rate. If a new better rate $R_2 > R_1$ arrive at time t_2 such that $t_2 + \tau_3 < t_1 + \tau_2$, the user will be allowed to ignore R_1 and adopt the rate R_2 already at time $t_2 + \tau_3$. The algorithm has to program a schedule of maximum available rates. But because it has only a small amount of memory, the algorithm only computes an approximation of the schedule: it stores the currently available rate and at most two other rates with the corresponding time. The verification

consists in proving that the algorithm favors the user, namely the computed rate is greater than the theoretical available rate.

This protocol involves generic time parameters τ_2 and τ_3, it was consequently first considered as more suited for interactive tools of theorem proving rather than automatic tools of model-checking. The first proofs were done by J.-F. Monin using Coq [17,18], a modelisation and proof was also developed by D. Rouillard using the CClair library [8] developed in HOL/Isabelle to reason on transitions systems. However, because the operations involved are of a simple shape, it appeared that it was possible to prove the algorithm using automatic tools like the model-checker for timed automata Hytech [14,3], the system of constraint logic programming Datalog [20,11] or the rewriting system ELAN [2].

When trying to modelise the problem in the theorem prover Spike, F. Klay introduced a generalized version of the protocol which uses a memory of arbitrary size. This general protocol has been studied in PVS [21] and in Coq [10].

A byproduct of the study of this example was the definition of a special class of timed automata called p-automata (for parameterized automata) introduced in [3]. We shall now describe how to represent this class of automata in Coq.

1.3 Outline of the Paper

The rest of the paper is organized as follows. In section 2, we briefly present the main features of the system Coq used in this paper. In section 3, we overview the model of timed automata which has been chosen. In section 4, we show the general principles for encoding automata in Coq. Finally in section 5, we discuss a method to automate the proofs of correctness of properties of automata.

2 Description of Coq

In this section, we give a quick overview of the system Coq and introduce the notations used in the rest of the paper.

2.1 What Is Coq?

Coq is a proof assistant based on Type Theory. It is an environment for the edition and verification of mathematical theories expressed in higher-order logic. Coq provides a language, called the Calculus of Inductive Constructions, to represent objects and properties. The main ingredient of this calculus is an higher-order typed lambda-calculus: functions, types or propositions are first-class objects and it is consequently allowed to quantify over functions, types or propositions variables.

It is well-known that the higher-order universal quantification over all propositions is sufficient together with implication to represent usual logical connectives such as conjunction, disjunction.

Example. We illustrate this fact on the definition of the conjunction $A \wedge B$ of two propositions A and B.

The three following properties that can be introduced as axioms for $A \wedge B$ in a first-order framework are easily derivable when $A \wedge B$ is defined as an abbreviation for $\forall X.(A \Rightarrow B \Rightarrow X) \Rightarrow X$.

$$A \Rightarrow B \Rightarrow A \wedge B \qquad A \wedge B \Rightarrow A \qquad A \wedge B \Rightarrow B$$

2.2 Particularities of Coq

Coq shares many similarities with other proof assistants based on higher-order logic such as systems in the HOL family (including Isabelle/HOL) and PVS. However Coq differs from these systems on some specific points that we shall explain.

Intuitionistic logic. Coq is based on an intuitionistic logic, which means that the principle of excluded middle which states that for any proposition A, we have $A \vee \neg A$ is not taken for granted. At first sight, this may seem to be a limitation. However, the classically equivalent property $\neg(\neg A \wedge \neg\neg A)$ is indeed intuitionistically provable. The intuitionistic aspect of the logic can be understood as a strong interpretation of the disjunction operator. In order to be able to prove $A \vee \neg A$, we should be able to provide a way to decide which one from A or $\neg A$ actually holds. On the same spirit, in order to prove the property $\exists x.P(x)$ we should be able to provide a witness a such that $P(a)$ holds.

Intensional functions. A function can be seen intensionally as an algorithm to transform an input into an output. Another point of view is to consider a function extensionally as an input-output relation. The type of functions from τ to σ in Coq represents the type of algorithm (described by λ-terms) taking inputs in the type τ and computing outputs in the type σ. This intensional point of view gives the possibility to the system to automatically compute with functions. An extensional function can also be represented by a binary relation in Coq but in that case, some deduction as to be performed in order to justify that some value is actually the output value of a function for some input.

To illustrate this point, we take the example of the `div2` function on natural numbers. In an extensional framework, it is enough to specify relation $\texttt{div2}(x) = y$ as the property $x = 2 \times y \vee x = 2 \times y + 1$ while in an intensional framework, a method of computation has to be given in some kind of algorithmic language, for instance:

$$\texttt{div2}(x) \equiv \mathsf{if}\ x = 0\ \texttt{or}\ x = 1\ \mathsf{then}\ 0\ \mathsf{else}\ \texttt{div2}(x-2) + 1$$

Obviously different algorithms can be given for the same extensional function. If a function h is given as an algorithm, then it can easily be translated into a binary relation $H(x, y) \equiv h(x) = y$, the opposite is obviously not true: some functional relations do not correspond to recursive functions and consequently

cannot be represented as algorithms. This is the case for instance for a function ϕ that takes a function f from natural numbers to boolean values as an argument and try to answer whether we have $f(x) = \texttt{true}$ for all integers x. ϕ cannot be represented in Coq as a function of type $(\texttt{nat} \to \texttt{bool}) \to \texttt{bool}$ and should instead be introduced as a binary relation:

$$\phi(f, b) \equiv \forall x. f(x) = \texttt{true}$$

The class of functions that can be represented as an algorithm in Coq is a subset of the class of recursive functions. For functions from natural numbers to natural numbers, it contains at least all recursive functions provably total in Higher-Order Logic.

The main interest to use intuitionistic logic and intensional functions is to reflect inside the logical system, informations related to the computational aspects of objects.

Dependent types In Coq, any object is given a type. The fact that an object inhabits a certain type can be verified automatically. A type is associated to an arbitrary expression representing a computation but it reflects actually a property of the value which is the result of the computation. It is important for the type system to be general enough to accept a large class of programs and allow their use in many different contexts. The type of inputs of a program should be as general as possible, and the type system should accept polymorphic programs. On the other side, the type of the output of the program should be as precise as possible depending on the inputs.

The Coq type system allows arbitrary polymorphism in the type of objects. For instance, assume that we have defined the type $\texttt{list}_\alpha$ of lists of elements of type α one may define a function of type:

$$\forall\alpha.\forall\beta.(\alpha \to \beta) \to \texttt{list}_\alpha \to \texttt{list}_\beta$$

This kind of polymorphism is usual in functional programming languages of the ML family. But it is also possible to build more complicated types, like for instance the type LIST of lists on an arbitrary type. An element in this type will be a pair built from a type τ and a list of type $\texttt{list}_\tau$. This kind of structure plays sometimes the role of module specification in programming languages and consequently is useful for building generic libraries. For instance, the type of automata will be, as in mathematics, a structure containing the type of labels and locations.

A type may depend not only on another type parameter like in the case of $\texttt{list}_\alpha$ but also on other objects. We may define for instance the type of lists of length n or of sorted lists or of lists which are a permutation of a given list or lists with elements satisfying a predicate P ...

We shall use this possibility in section 2 in order to define the synchronization of an arbitrary family of automata and to get a compact representation of state variables.

Explicit proof objects In Type Theory, terms are not only used for representing objects of the logic but also for representing proofs. A property P of the logic is represented as a type of the language, P is true when it is possible to exhibit a term p with type P which represents a proof of P. The correctness of the proof system consequently relies on the correctness of the type-checking algorithm. A proof-term can be built using arbitrary programs (called tactics) but it is eventually typechecked by the kernel of the Coq system.

In order to use a complicated decision procedure in a safe way in Coq it is necessary for this decision procedure not only to answer "yes" or "no" but in the case of a positive answer, it should compute a term with the appropriate type. However, because of the expressiveness of the language, there are many possible choices for this term. We shall come back to this matter in section 5.

2.3 Notations

We shall not enter into the details of the Coq syntax which are precisely described in the Reference Manual [23]. In the rest of the paper, we shall use a few constructions that we explain now.

Sets. Data-types are represented by objects with type a special constant named `Set`. For instance, the type `nat` of natural numbers is itself an object of type `Set`.

If T and U are objects of type `Set`, then $T \to U$ is a new object of type `Set` which represents the type of functions from T to U. New types may be introduced by inductive declarations:

Inductive *name* : Set := $c_1 : T_1 \mid \ldots \mid c_n : T_n$

This command adds a new concrete type *name* in the environment, as well as the constructors of this type named $c_1, \ldots, c_n$. Each c_i has type T_i. A constant constructor will be declared as c : *name*, a unary constructor expecting an argument of type α will be declared as $c : \alpha \to$ *name*. The constructor can accept recursive arguments of type *name*. For instance the definition of lists of type α is defined as:

Inductive $\texttt{list}_\alpha$: Set := $\texttt{nil} : \texttt{list}_\alpha \mid \texttt{cons} : \alpha \to \texttt{list}_\alpha \to \texttt{list}_\alpha$.

If t is an object in an inductive type T, it is possible to build an object by pattern-matching over the structure of the values in the type T. When T is inductively defined, it is possible to define a recursive function depending on x of type T as soon as the recursive calls in the definition of x are done on objects structurally smaller than x.

Propositions. Logical propositions are represented by objects with type a special constant named `Prop`. If T and U are objects of type `Prop`, then $T \to U$ (also written $T \Rightarrow U$ in this paper) is a new object of type `Prop` which represents the property "T implies U". The prelude of Coq contains the definition of usual

logical connectives such as conjunction, disjunction or equality. The universal quantification of a property P depending on a variable x of type A is written $(x : A)P$.

Abstraction and application. A type or object t can be freely abstracted with respect to a type or object variable x of type τ. The abstracted term is written $[x : \tau]t$ or simply $[x]t$ when τ can be deduced from the context. A term t representing a functional construction can be applied to a term u of the appropriate type, the application is written $(t\ u)$.

Predicates. A predicate P over a type α is a term of type $\alpha \to$ `Prop`, ie a function which given an element a of type α produces a property $(P\ a)$ of type `Prop`.

It is possible to define a proposition either as a function defined by case and recursion over an inductively defined parameter or inductively. An inductively defined predicate looks like the definition of a Prolog program with clauses.

For instance, given a binary relation R of type $\alpha \to \alpha \to$ `Prop` and a predicate P of type $\alpha \to$ `Prop`, it is possible to define inductively the predicate `Pstar` which is true for any y of type α such that there exists a path from x which satisfied P to y with two consecutive elements satisfying R. The definition is as follows:

Inductive Pstar : $\alpha \to$`Prop`
  `init` : (x:α)(P x) → (Pstar x)
 | `add` : (x,y:α)(Pstar x) → (R x y) → (Pstar y)

Declarations. Declarations of the form:

Definition *name* := t.

are used to introduce *name* as an abreviation for the term t.

3 Description of p-Automata

Timed automata introduced by Alur and Dill [1] are widely used for the modelisation of real-time systems. In order to represent the ABR conformance algorithm, a generalization of timed automata called p-automata was introduced in [3]. In classical timed automata, states represent the current value of a finite set of clocks which can be reseted during a transition. In p-automata, there is a distinction between a universal clock which is never reseted and variables which are updated during transitions. The update of a variable may be non-deterministic (for instance we may assign to a variable an arbitrary value greater than 3, which is useful to model the behavior of the environment) and the update of one variable x may also depend on the value of another variable y in a linear way (for instance $x := y + 3$). The constraints used in location invariants may also involve a comparison between the values of two different variables (for instance $x \leq y + 3$).

In [6,7] this class of automata is precisely studied. In particular it is shown that deciding the emptiness of the language accepted by the automata is not

decidable in general. One need to restrict either the class of constraints used as invariants or the class of updates. Decidability of emptiness is guaranteed when constraints are diagonal free (ie they only compare a variable to a constant and not another variable) or when the updates are of a simple shape: assigning a constant, another variable or an arbitrary value less than a constant.

If we plan to manipulate p-automata into a general proof assistant, it is not necessary to restrict the class of variables, formula and updates. The general p-automata are specified by a set of locations with an invariant associated to each location and transitions given by two locations (the source and the target) the name of the action labelling the transition and a relation between the value of the variables before and after the transition.

4 Encoding p-Automata in Coq

4.1 Deep versus Shallow Embeddings

When encoding a theory inside a proof assistant there is always a choice between what is called a *deep embedding* and a *shallow embedding*. A deep embedding introduces the (abstract) syntax of the theory as a concrete type of the proof assistant. A shallow embedding uses the objects of the logic (data types or proposition) to represent a model of the theory.

For instance, assume we want to manipulate arithmetic formulas built on 0, 1, variables, the binary addition + and the equality relation. A deep embedding will introduce a data type for representing arithmetic expression with two constants for representing 0 and 1, an unary operation `var` to represent variables (indexed for instance by some given set I) and a binary operation to represent addition. An equation can be represented as a pair of arithmetic terms (t_1, t_2) of type `expr` $*$ `expr`.

```
Inductive expr : Set :=
  Zero : expr | One : expr | Var : I → expr | Plus : expr → expr → expr.
Definition equation := expr * expr.
```

In a shallow embedding, the Coq type of natural numbers can be used directly for interpreting arithmetic expressions. The variables of the object theory can be identified with Coq variables and the equality between terms can be represented using Coq equality.

It is not difficult to define a function going from the deep representation into the shallow one assuming we provide an interpretation of:

```
Fixpoint interp [i:I→nat; e:expr] : nat :=
  Cases e of Zero → O | One → (S O) | (Var x) → (i x)
        | (Plus e1 e2) → (plus (interp i e1) (interp i e2))
  end.
Definition interp_eq [i:I→nat] : equation → Prop :=
  [e]let (e1,e2) = e in (interp i e1)=(interp i e2).
```

It is also possible to go from a tCoq erm in the type `nat` or `Prop` to the concrete representation of type `expr` or `equation`. But these operations can only be done at the meta-level, and it is in general only a partial operation. The idea is to do an analysis on the syntax of a Coq expression t of type `nat` and try to find an interpretation i and an expression e such that the expression t is convertible with the expressions (`interp` i e). Any term t can be interpreted in a trivial way as a variable (`Var` x) and the interpretation i such that $(i\ x) = t$. But it is obviously not the case for equations, only terms of type `Prop` which are convertible to $t_1 = t_2$ with t_1, t_2 : `nat` can be transformed. This meta transformation can be done in Coq in a simple way using the new mechanism of `Tactic Definition` designed by D. Delahaye [9] which allows to define these transformations directly inside the Coq toplevel.

There are many possibilities to combine the two approaches, for instance it may be convenient, instead of specific constants for zero and one to introduce a general unary `Const` constructor which takes a natural number as an argument. Then the natural numbers at the meta-level are directly injected into the object-level arithmetic expressions.

4.2 The Type of p-Automata

As we explained earlier, the syntactic restriction on the shape of formulas is strongly related to the possibility to design decision procedures. This is not the major problem in an interactive proof assistant, so we prefer to define p-automata at a less syntactic level: invariants and transitions will be described using Coq predicates following a shallow embedding.

Before defining timed automata, we need a definition of time. We have chosen to introduce a type `Time` as a parameter and assume some properties. The type `Time` contains two constant elements 0 and 1; the available operations are the addition and the opposite. The relations are $<$, $\leq$ and $=$ with $x \leq y$ defined as $x < y \vee x = y$. The order is assumed to be total in a computational way. A model of this axiomatization can be built in Coq using either integers or rationals, in particular this axiomatization is consistent.

For building a p-automata, we need a type for the variables V, a type for locations L, a type for transition labels A. An invariant is given for each location, it depends on variables and the universal clock. An invariant is consequently an object of type $L \to$ `Time` $\to V \to$ `Prop`.

A transition takes place at some instant s, it is labeled by an action a, it goes from a location l to a location l' and transforms a variable v into a variable v'. The set of such transitions (a, s, l, v, l', v') is represented in Coq as a relation of type $A \to$ `Time` $\to L \to V \to L \to V \to$ `Prop`.

We may first introduce the type of automata built on the set of variable V, of locations L and actions A as a pair containing the invariant and the transition relations. It is also possible to see an automata as a package

$$\begin{aligned}\texttt{pAuto} \equiv <\ & V, L, A : \texttt{Set};\\ & \texttt{Inv} : L \to \texttt{Time} \to V \to \texttt{Prop};\\ & \texttt{Trans} : A \to \texttt{Time} \to L \to V \to L \to V \to \texttt{Prop}\ >\end{aligned}$$

containing the three types and both the invariant and the transition relations. Given an automata, it is possible to do a projection over each component. For instance one defines projections `V`, `L` and `A` from `pAuto` to `Set` which associates to each p-automata p respectively the type of its variables, locations and actions. There is also a projection `Inv` corresponding to the invariant which has type

$$\texttt{Inv} : (p : \texttt{pAuto})(\texttt{L}\ p) \to \texttt{Time} \to (\texttt{V}\ p) \to \texttt{Prop}$$

This is only possible because of the powerful type system of the Calculus of Inductive Constructions. It will give us the possibility to talk of an arbitrary family of automata, each one being built on its own types for variables, locations or actions.

4.3 Semantics

It is easy to associate to a p-automata a labeled transition system. The states are of the form (l, s, v) built from a location l, the time value s and the value of the variable v.

However there are a few choices to be made. There are at least two different ways to build the labeled transition system, the first one is to distinguish two kind of actions, the discrete ones labeled by the actions of the p-automata and the temporal ones parameterized by the time. Another possibility is to combine a temporal and a discrete transition into one transition parameterized by the action of the p-automata. This is possible because two consecutive temporal transitions of delay t and t' are always equivalent to one with delay $t + t'$. It is easy to define both interpretations.

Another subtle point is the role of invariants properties. A first point of view is to consider that transitions can only be done between states (called admissible) which satisfy the invariant property. However this invariant property is mainly used to restrict the amount of time spent in a location before taking a discrete transitions. The practical tools based on timed-automata have a restricted form of transitions, in order to be able to perform complicated operations on variables, it is necessary to introduce several successive transitions, the intermediate locations are called urgent because one does not want temporal transitions to take place in such places. Such urgent transitions or locations did not appear very natural in the model of p-automata, it was consequently decided to not consider this feature as a primitive one, but to encode it using invariants.

If we require discrete transitions to take place between admissible states then an urgent location should be such that the invariant is true exactly when we reach the state and becomes false immediately after. This can be achieved with an extra variable that we call `asap`. An urgent location will be such that the update $\texttt{asap} := s$ is added on each transition arriving on this place and the invariant of the location is $\texttt{asap} = s$. Another possibility which requires less encoding is to put on urgent location an invariant which is always false and to relax the condition on admissibility of states for discrete transitions. It leads to the following definition of transitions:

A discrete transition can take place between (l, s, v) and (l', s', v') if and only if $s = s'$ and there is a transition (a, s, l, v, l', v') in the p-automata.
A temporal transition can take place between (l, s, v) and (l', s', v') if and only if $l = l'$, $v = v'$ and $s \leq s'$ with the invariant of the p-automata satisfied for all states (l, t, v) with $s < t \leq s'$. We believe this second method will be easier to use in practice.

4.4 Synchronization

An important operation on automata is synchronization. With our modelisation, it was easy to define a very general synchronization mechanism.

We can start from a family of p-automata indexed by an arbitrary set I. It is represented in Coq as an object `PFam` of type $I \rightarrow$ `pAuto`.

It remains to understand what we expect as variables, transitions and actions for the synchronised automata. The easy case is locations. A location in the synchronised automata is just the product of the locations of each automata. This is expressed in Coq as:

Definition LocS := (i:I)(`L` (`PFam` i)).

A location l in type `LocS` is written in mathematical notations as $(l_i)_{(i:I)}$.

For the actions, we could do the same, however, it may be the case that we only want to synchronise a subset of the family of automata. It can easily be done by introducing a special action ϵ which corresponds to an identical transition $(\epsilon, s, l, v, l, v)$. If we name (`Eps` α) the type α extended with this extra element, the type of synchronised actions will be defined as:

Definition ActS := (i:I)(`Eps` (`A` (`PFam` i))).

For the variables, there is no canonical choice. Defining a variable of the synchronised automata as the product of the variables of the different systems corresponds to consider variables as local to each automata, this is not really convenient because variables are a mean to share information between automata. Another possibility would be to consider that variables are global. In that case, the complete family of p-automata should share the same set of variables that will also be the variable of the synchronised automata. Such a choice goes again modularity. For instance it will not be easy to introduce one automata and then to use the same automata up to the renaming of variables. For these reasons, it was decided, when designing the model of p-automata in the CALIFE project, to define a general notion of synchronisation, parameterised by a new set of variables V. What is only required is that one can define a family of projection functions $\texttt{proj}_i$ from V to each individual type of variable (`V` (`PFam` i)) and that this projections are injective. Namely if for all i, we have $(\texttt{proj}_i\ v) = (\texttt{proj}_i\ v')$ then $v = v'$. It is indeed the case for operations corresponding to global variables, local variables or renamings.

It remains to define the invariant `InvS` and the transition `TransS` of the synchronized automata. We define:

Definition InvS : LocS → Time → V → Prop
:= $[l, s, v](i : I)$(Inv (PFam i) (l i) s ($\mathtt{proj}_i$ v))
Definition TransS : ActS →Time → LocS → V → LocS → V → Prop
:= $[a, s, l, v, l', v']$
($i : I$)**if** (a i) = ϵ
then(l i) = (l' i) ∧ ($\mathtt{proj}_i$ v) = ($\mathtt{proj}_i$ v')
else(Trans (PFam i) (a i) s (l i) ($\mathtt{proj}_i$ v) (l' i) ($\mathtt{proj}_i$ v'))

These transitions are restricted to a subset of synchronised actions.

4.5 Representing a Specific Automata

The development we made is appropriate for studying the meta properties of p-automata, like establishing a correspondence between the transition system of the synchronized automata and synchronization of transition systems associated to each automata.

But the goal is to use also this modelisation for verifying practical instances of p-automata like the ABR conformance protocol we started with.

When we study a particular example, we usually draw a picture with names for locations, variables, actions and formulas for invariants, and transitions. It remains to do the link between this concrete presentation and the abstract model we have presented.

We propose to do it in a systematic way. Names for locations, variables and actions are introduced as enumerated sets (a particular class of inductive definition with only constant constructors). For locations and actions, it gives us the type L and A used in the model. For variables, we need more. Assume N is the type of names of variables $N = \{n_1, \ldots, n_p\}$. To each element n of N is associated a type T_n. We may build a family T of type $N \to \mathtt{Set}$ such that (T n) is defined to be T_n. This can easily be done by case analysis on n or using the combinator N_rect automatically defined by Coq after the inductive definition is declared. Now we define the type V of variables as $(n : N)(T\ n)$. Assume that v_n is a value of type T_n for each variable of name n, a value of type v is simply built as the Coq expression ($N\_rec\ T\ v_{n_1} \ldots\ v_{n_p}$). The projection of a variable v onto the component of name n is easily obtained by the expression (v n). With this representation, it is easy to define a generic notion of updates. Assume N is a set with a decidable equality, a family $T : N \to \mathtt{Set}$ and V defined to be $(n : N)(T\ n)$. The update relation can be defined as:

Definition update : $V \to (n_0 : N)(T\ n_0) \to V$
:= $[v, n_0, v_0, n]$if $n = n_0$ then v_0 else (v n)

In order for this expression to be well-typed, we need to transform the value v_0 of type (T n_0) into a value of type (T n) using the proof of $n = n_0$, this is a well-known difficulty when dealing with dependent types, but it can be handle with a little care.

An alternative possibility would be to define the type V as a record (an inductive definition with only one constructor with p arguments corresponding

to the p variables). But this representation with record does not allow for generic operations.

The invariant is defined by a case analysis on the location. If $\texttt{Inv}_l$ is the formula representing the invariant at location l and if the finite set of location is $\{l_1, \ldots, l_k\}$, then the Coq invariant will be defined as:

Definition Inv :L →Time →V →Prop
:= [l,s,v]**Cases** l **of**
l_1 ⇒$\texttt{Inv}_{l_1}[n \leftarrow (v\ n)]_{n:N}$
⋮
| l_k ⇒$\texttt{Inv}_{l_k}[n \leftarrow (v\ n)]_{n:N}$
end

For each concrete formula $\texttt{Inv}_l$, we only have to perform the substitution of the symbolic name n of the variable by the corresponding value $(v\ n)$ of the variable v. We have denoted this operation by $\texttt{Inv}_l[n \leftarrow (v\ n)]_{n:N}$.

For the transition relation, one possibility would be to a double case analysis on the location l and l'. However, this leads to consider k^2 different cases, many of them being irrelevant. So it is better to define the transition relation as the disjunction of the transitions given in the concrete representation. Assume we are given formulas $\texttt{Trans}_{l,l',a}$ for describing the transition between l and l' labelled by a, this formula uses variables names n and n' to denote the value of n after the transition. The definition of the abstract transition relation will have the following form, with one clause in the inductive definition for each transition identified in the concrete representation:

Inductive Trans [s:Time,v,v':V] :A →L →L →Prop :=
...
| t_i : $\texttt{Trans}_{l,l',a}[n \leftarrow (v\ n), n' \leftarrow (v'\ n)]_{n:N}$ →(Trans s v v' a l l')
...

The only requirement is that the expressions $\texttt{Inv}_{l_1}[n \leftarrow (v\ n)]_{n:N}$ and $\texttt{Trans}_{l,l',a}[n \leftarrow (v\ n), n' \leftarrow (v'\ n)]_{n:N}$ which are obtained by syntactic manipulations correspond to well-formed Coq propositions.

Graphical interface. In the CALIFE project, a graphical interface is currently designed by B. Tavernier from CRIL Technology for editing p-automata and do an automatic translation towards the verification tools.

5 Automation of Proofs

It is interesting to have the full power of the Coq specification language and the interactive proof system. However it will be useless if we could not provide help for automation of the proof.

For doing proofs without too much human interaction, there are at least two different approaches. The first one is to develop the problem and try to solve it

by computation. The second one is to abstract the problem and find a general theorem which can be applied.

What is interesting in Coq is the possibility to combine both approaches. For instance, abstract theorems can be used for establishing general properties of p-automata. For instance, it is easy to define a notion of bisimulation on labelled transitions systems, the use of primitive coinductive definitions of Coq is especially convenient for that.

Now it is possible to describe sufficient conditions on a relation on variables and time of a p-automata in order to build a bisimulation on the underlying transition system.

The first attempt to prove automatically the ABR conformance protocol was done by L. Fribourg using the Datalog system. The idea is to represent transitions using what is called gap-configurations. These configurations are disjunctions and conjunctions of atomic formulas of the form $x \leq y + k$ with x and y integer variables and c an integer constant. These configurations can be represented as directed graphs, the vertexes are the variables and the edges are labeled by the constants. The graph corresponds to a satisfiable formula if and only if there is no cycle in the graph with negative weight. In order to prove an arithmetic formula, it is enough to prove that the graph corresponding to the negation of the formula has a negative cycle. J. Goubault designed a Coq package for representing efficiently finite sets using a map structure indexed by addresses represented as binary integers. This structure has been used for certifying a BDD package [25] and building a symbolic model-checker on top of it. In his early development of map, J. Goubault developed a certified version of the algorithm to decide arithmetic formulas based on gap-configurations. This development is part of the Coq contribution library.

It uses the principle of computational reflection which was first introduced in Coq by S. Boutin [5]. The idea is to build a concrete type `Form` of arithmetic formula which contains variables indexed by addresses of type `adr`. A decision procedure `dec` of type `Form` $\rightarrow$ `bool` is constructed in Coq which translate the formula into a gap-configuration representing its negation and then try to find a negative cycle. An interpretation function `interp` of type (`adr` $\rightarrow$ `Z`) $\rightarrow$ `Form` $\rightarrow$ `Prop` is also defined. Given an interpretation of variables as integer values, and a formula, it builds its interpretation as a Coq proposition. This approach is a generalization of what we presented in the section 4.1. Now the key of the approach is to derive a proof of correctness of the decision procedure, namely a lemma proving:

Lemma `correct` : $(f : \texttt{Form})(\texttt{dec}\ f) = \texttt{true} \rightarrow (r : \texttt{adr} \rightarrow \texttt{Z})(\texttt{interp}\ r\ f)$

Now in order to prove an arithmetic goal G, it is enough to find a concrete formula f and an interpretation r such that (`interp` r f) is convertible to the goal G to be proven and to apply the lemma `correct`. Then one is left with the goal (`dec` f) = `true`. Because (`dec` f) is a closed term of type `bool` it reduces by computation to a value of type `bool`, namely `true` or `false`. If the property is true then (`dec` f) reduces to `true` and because Coq identifies terms up to computation, the trivial proof of `true` = `true` is also a proof (`dec` f) =

`true`. This method is a special case of combination of computation and abstract theorem. An important effort has to be done for the justification of the procedure, but the use of this method in particular cases relies only on computation done automatically by the system.

The Revesz procedure applies the method of gap-configurations for the computation of the transitive closure of a relation R. Assume the initial set P of states and the relation R are described by gap-formula and that we want to prove that a property Q is an invariant of the system. The main problem is usually to find an inductive invariant. We may define the `Pstar` formula inductively as it was done in section 2.3 on page 304. Given two predicates P and P' we write $P \subseteq P'$ to denote the fact that for all x, we have $(P\ x) \Rightarrow (P'\ x)$. Assume we want to prove that $\texttt{Pstar} \subseteq Q$. One may define for each natural number i the predicate P_i of objects that can be reached from P with at most i steps. It is easy to see that

$$(P_0\ x) \Leftrightarrow (P\ x) \qquad (P_{i+1}\ x) \Leftrightarrow ((P_i\ x) \vee \exists y.(P\ y) \wedge (R\ y\ x))$$

If P and R are represented by gap-configurations then P_i is also represented by a gap-configuration. In order to build this new gap-configuration, we need to formally program and justify the short-cut operation which implements the existential quantification.

If we can find i such that $P_{i+1} \subseteq P_i$ then it is easy to show that $\texttt{Pstar} \subseteq P_i$. Then we have a gap-configuration for representing `Pstar` and we can automatically check $\texttt{Pstar} \subset Q$.

An important problem is to ensure that the process which iteratively computes P_i will effectively reach a fixpoint. But this problem does not have to be formalized in Coq. In Coq, one proves a general theorem:

Lemma `fix` : $(i : \texttt{nat})(P_{i+1} \subseteq P_i) \rightarrow (P_i \subseteq Q) \rightarrow \texttt{Pstar} \subseteq Q$

It is enough to find with an external procedure an integer i, and to check in Coq (using for instance the gap procedure) the special instances $P_{i+1} \subseteq P_i$ and $P_i \subseteq Q$ in order to insure that Q is an inductive property of the transition system.

The main problem with this approach is the efficiency. Work is currently undertaken in order to compute with Coq term as efficiently as with functional programming languages like CAML. But the problem comes also from the efficiency of the procedures that are formalised, it is important to separate between what has to be done inside the proof assistant and what can be reused for the state-of-the-art automatic dedicated proof tools that are developed in the world.

6 Conclusion

In this paper, we have presented the main principles of a project to build on top of the Coq system a environment for the specification and proof of real-time systems represented by timed automata. We emphasized the features of Coq that makes it a good candidate among proof assistants to undertake this work.

This project shows a natural evolution of the proof assistant tools. Until now, doing proofs with these systems required to be able to translate a practical problem in the specific language of the proof-assistant. In this project, we use the full power of the language in order to design the general libraries and proof procedures. However, the end-user should mainly be involved with the specialized language of timed automata adapted to the class of problem (s)he is interested in. Coq becomes an environment for implementing dedicated tools for specification and verification. We can make an analogy with programs like the TEX word-processing tool or the emacs editor. Both of them provide a powerful meta-language, which is mainly used by a few experts; but the use of specialized TEX styles or emacs modes only requires ordinary skills. Like TEX or emacs, Coq is an open system, you can buy libraries or tactics designed by many different people without compromising the correctness of your own development. However, the adequation of this approach to industrial needs remains to be validated.

Acknowledgements

Many points presented in this paper were discussed during meeting of the CALIFE project. Emmanuel Freund participated to an early development of the library on p-automata in Coq.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 1994.
2. E. Beffara, O. Bournez, H. Kacem, and C. Kirchner. Verification of timed automata using rewrite rules and strategies. http://www.loria.fr/~kacem/AT/timed_automata.ps.gz, 2001.
3. B. Bérard and L. Fribourg. Automated verification of a parametric real-time program: the abr conformance protocol. In *11th Int. Conf. Computer Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 96–107. Springer-Verlag, 1999.
4. B. Bérard, L. Fribourg, F. Klay, and J.-F. Monin. A compared study of two correctness proofs for the standardized algorithm of abr conformance. *Formal Methods in System Design*, 2001. To appear.
5. Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Takahashi Ito and Martin Abadi, editors, *TACS'97*, volume 1281. LNCS, Springer-Verlag, 1997.
6. P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Are timed automata updatable? In *12th Int. Conf. Computer Aided Verification (CAV'2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 464–479, Chicago, IL, USA, July 2000. Springer-Verlag.
7. P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Expressiveness of updatable timed automata. In *25th Int. Symp. Math. Found. Comp. Sci. (MFCS'2000)*, volume 1893 of *Lecture Notes in Computer Science*, pages 232–242, Bratislava, Slovakia, August 2000. Springer-Verlag.
8. P. Castéran and D. Rouillard. Reasoning about parametrized automata. In *8th International Conference on Real-Time System*, 2000.

9. David Delahaye. A Tactic Language for the System coq. In *Logic for Programming and Automated Reasoning (LPAR' 00)*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95, Reunion Island, November 2000. Springer-Verlag.
10. C. Paulin & E. Freund. Timed automata and the generalised abr protocol. Contribution to the Coq system, 2000. http://coq.inria.fr.
11. L. Fribourg. A closed-form evaluation for extended timed automata. Research Report LSV-98-2, Lab. Specification and Verification, ENS de Cachan, Cachan, France, March 1998.
12. S. Graf and H. Saidi. Verifying invariants using theorem proving. In *8th Conf on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, USA, July 1996. Springer-Verlag.
13. E. L. Gunter. Adding external decision procedures to hol90 securely. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics 11th International Conference (TPHOLs '98)*, volume 1479 of *Lecture Notes in Computer Science*. Canberra, Australia, Springer-Verlag, September 1998.
14. T. Henzinger, P.-F. Ho, and H. Wong-Toi. A user guide to HYTECH. In *TACAS'95*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71, 1995.
15. D. Hirschkoff. A full formalisation of the π-calculus theory in the Calculus of Constructions. In E. Gunter and A. Felty, editors, *Theorem Proving in Higher-Order Logics*, volume 1275 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
16. F. Honsell, M. Miculan, and I. Scagnetto. Pi calculus in (co)inductive type theories. Technical report, Dipartimento di Matematica e Informatica, Universita' di Udine, 1998.
17. J.-F. Monin. Proving a real time algorithm for ATM in Coq. In *Types for Proofs and Programs (TYPES'96)*, volume 1512 of *Lecture Notes in Computer Science*, pages 277–293. Springer-Verlag, 1998.
18. J.-F. Monin and F. Klay. Correctness proof of the standardized algorithm for ABR conformance. In *Formal Methods'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 662–681. Springer-Verlag, 1999.
19. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
20. P. Z. Revesz. A closed-form evaluation for datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116:117–149, 1993.
21. M. Rusinowitch, S. Stratulat, and F. Klay. Mechanical verification of a generic incremental ABR conformance algorithm. In *12th Int. Conf. Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, Chicago, IL, USA, July 2000. Springer-Verlag.
22. C. Sprenger. A verified model-checker for the modal μ-calculus in Coq. In *TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, Lisbon, Portugal, 1998. Springer-Verlag.
23. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.0*, April 2001. http://coq.inria.fr.
24. K. N. Verma. Reflecting symbolic model checking in coq. Mémoire de dea, DEA Programmation, September 2000. http://www.lsv.ens-cachan.fr/Publis/PAPERS/Ver-dea2000.ps.

25. K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting bdds in coq. In *6th Asian Computing Science Conference (ASIAN'2000)*, volume 1961 of *Lecture Notes in Computer Science*, pages 162–181. Springer-Verlag, November 2000.
26. S. Yu and Z. Luo. Implementing a model-checker for LEGO. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97*, volume 1313 of *Lecture Notes in Computer Science*, pages 442–458, September 1997.