

GENERATE CERTIFIED TEST CASES BY COMBINING THEOREM PROVING AND REACHABILITY ANALYSIS*

Richard Castanet

Davy Rouillard

LaBRI, Bordeaux I University, 33405 TALENCE Cedex, FRANCE

{castanet,rouillard}@labri.u-bordeaux.fr

Abstract We present a test case generation method which conciliates theorem proving and model checking. Test purposes are expressed by timed regular expressions and then translated into a corresponding automaton using a certified function. This automaton is composed with the system specification and an execution is computed from this sub-specification by an automatic tool. The result is finally re-injected into the theorem prover to be checked.

Keywords: conformance testing, certified test purposes, timed automata, safe computations

1. INTRODUCTION

The life cycle of protocol engineering contains several steps: requirements definition, formal specification, simulation or verification, implementation and finally testing (conformance, interoperability, robustness and performance tests). It is widely recognized that the use of formal model is essential to allow automatic verification as well as automatic test generation. Proof techniques can also be used to validate the critical parts of a system but they are employed still rather rarely in the domain of protocols [14].

The aims of this paper are to show how the test cases generation can be compatible with the proof activity and to exhibit the benefits one may gain in developing a test environment based on a proof assistant. To achieve these goals, we propose to use the general framework *Cclair*

*Research partially supported by the French government for the RNRT project Calife.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35497-2_31](https://doi.org/10.1007/978-0-387-35497-2_31)

I. Schieferdecker et al. (eds.), *Testing of Communicating Systems XIV*

© IFIP International Federation for Information Processing 2002

[8] which supports both verification and testing activities on various kind of automata. For a description of the verification techniques supported by *CClair*, we refer the reader to [7].

The heart of *CClair* is the proof assistant *Isabelle*[17] and its implementation of the *Higher Order Logic*. This logic offers a very rich language in which it is possible to describe complex systems and properties. Thus, it becomes possible to work on systems for which automatic tools fail because of a classical state-explosion problem.

This high expressiveness is also a strong argument for the joint use of proof assistants and automatic procedures because tools such as *Isabelle* [17], *Coq*[16] or *PVS*[20] may work with a great variety of mathematic objects and they have therefore no problems to understand and manipulate the results computed by automatic tools.

Another advantage is that proof assistants deal with theorems i.e. certified objects. This means that all results computed by external procedures have to be checked before being integrated. Furthermore, complex pieces of reasoning may be formalized in order to combine these results. It results in an increased confidence in the validation/testing process since the unavoidable errors due to the merge of several formalisms are immediately detected.

This paper highlights all these concepts by proposing a new approach to generate certified test cases for real-time systems. We are only interested by deriving generic test case without considering a test method. This last aspect will be taken account in a next stage of the testing process, following the methodology of the standard *ISO9646*[13]. To model the specification systems, we use *Parameterized automata* (p-automata) that can be seen as extensions of timed automata or timed I/O automata. The latter have been used in recent works for test generation [12, 4]. The p-automata model has the advantage to allow the use of variables of any type and recent experiments [6] tend to prove it is a convenient formalism to model communication protocols.

The rest of the paper is organized as follows: In section 2, we present the main features of *Isabelle* and *CClair*. In section 3, we describe how *CClair* may be used to formalize the test generation starting from both a specification of a timed system under test and a test purpose that describes a set of behaviors to be tested. In section 4, we introduce a general method to generate test cases by translating test purpose into a p-automaton. This automaton is used as synchronous observer to guide the construction of an execution that satisfies the test purpose. This execution is computed by *Hytech* and then validated in *CClair*. This method is employed in section 5 to generate test cases for the ABR

algorithm used in ATM networks. Finally, the section 6 summarizes the results presented in the paper and outlines future works.

2. A GENERAL FRAMEWORK FOR STUDYING AUTOMATA

2.1 *Isabelle*

CClair is developed on top of *Isabelle*, which is a generic proof assistant in the sense that it supports reasoning in various logics : *First Order Logic*, *Zermelo-fraenkel set theory*, *High Order Logic* (HOL in short). In this paper, we are however only concerned by the latter instance of *Isabelle*.

HOL offers a great expressiveness that allows to formalize mathematical principles such as recursion and [co]induction.

Proofs, in *Isabelle*, may be built in a backward-chaining mode : a conjecture is stated and the goal is successively reduced to zero or more new subgoals by applying tactics. When no subgoals remain, the proof is complete and the resulted theorem may be used in further proof. The main tactic is resolution, which mimics the natural deduction: it unifies the conclusion of a proof rule with a selected subgoal and then replaces it by the instantiated premise of the rule. An other important feature is offered by rewriting tactics, which simplify a selected subgoal using possibly conditional directed equalities. Theorems used during rewriting are stored in *simplification sets* that can be extended by any necessary theorems.

Goals may contain *logical variables*, denoted with a question mark $x?$. Logical variables represent unknown values that can be instantiated by resolution during proofs. Rewriting tactics and logical variables may be combined to perform some computations. For instance,

$$\text{filter } (\lambda x. x = a) [a, b, b, c, a] = x?$$

will instantiate $x?$ with the list $[a, a]$ using a rewriting tactic.

2.2 *CClair*

The *CClair* project [7] aims to provide the user with an environment for working with transition systems of various kinds. Its kernel is a main theory of plain labeled transition systems, which allows to name states, action labels, transitions, traces and executions, and contains some tools for computing and reasoning. It intends to be *generic* in the same meaning as *Isabelle*: a general framework contains a theory of plain transition systems; various interesting kinds of transition systems, like Büchi, constrained or timed automata are defined in sub-theories.

Executions and traces of a system are formalized using the theory of *lazy* list developed by Paulson[18]. This theory allows us to represent finite as well infinite behaviors in a single representation and provides various usual functions on list such as concatenation and filtering. Executions are therefore naturally declared as lists of transitions.

Properties of executions and/or traces may be expressed via a set of temporal operators ($\bigcirc, \diamond, \square, \dots$) (see [9] for example) that define languages of potentially infinite words. Usual operations on languages are also provided, among them $\text{Concat } L_1 L_2$ denotes the concatenation of L_1 and L_2 , $\text{Star } L$ is the Kleene-star operation and $\text{Omega } L$ represents the set of words obtained by infinite concatenations of words in L .

3. TEST GENERATION METHODOLOGY

One of the goal of the *CClair* project is to put in the same framework, verification and testing activities. This section shows that test cases generation may be viewed as a verification task: proving that an unknown execution satisfies a given property. We begin by a description of the p-automata model and its implementation into *CClair*.

3.1 The p-automata model

To model real time systems, we use *Parameterised timed automata* [5]. This model was designed for the formalization and proof of algorithms used in telecommunications and constitutes the heart of the French RNRT scientific project *Calife*.

A p-automaton is a graph associated with a set of variables and a single non resettable clock. Each edge of the graph can be guarded by a condition on the variables (including the clock) and a relation which describes how the values of variables can be modified.

More precisely, a p-automata $P = \langle \mathcal{S}, \mathcal{A}, \mathcal{L}, \mathcal{W}, \mathcal{V}, \mathcal{M}, \mathcal{I} \rangle$ consists of the following components: \mathcal{S} is the *universal clock* that memorizes the elapse of time. \mathcal{A} is a set of *actions*. \mathcal{L} is a set of *locations* that are the vertices of the control graph. \mathcal{W} is a set of *parameters* i.e. variables with values fixed by initial constraints. \mathcal{V} is a set of variables. As usual, we use primed variables to denote the values after a modification and the set of primed of variables is denoted \mathcal{V}' . \mathcal{M} is a set of *moves*. Each move $m = \langle l, a, \theta, l' \rangle$ is composed of a source location $l \in \mathcal{L}$, a target location $l' \in \mathcal{L}$ and an action $a \in \mathcal{A}$. The *update relation* $\theta \subseteq (\mathcal{V} \times \{\mathcal{S}\}) \times \mathcal{V}'$ describes the guard condition and modifies the value of variables. Finally, the function \mathcal{I} assigns to each location a constraint on variables called *invariant*.

During an execution, the state of P is given by the triple (s, v, l) that is composed of the value of the clock, a location $l \in \mathcal{L}$ and a vector $v = (v_1, \dots, v_n)$ that represents a value a_i for each variable $x_i \in \mathcal{V}$. States must be *admissible* i.e. values of variables must satisfy the invariant associated with the location of the state.

P can change its state in two ways: a *discrete* transition associated to a move $\langle l, a, \theta, l' \rangle \in \mathcal{M}$ causes P to change the location from l to l' and the values of variables are updated according the update relation θ . Otherwise, a *delay transition* modifies only the value of the universal clock.

The way in which a p-automaton operates is described by an *execution*, i.e. a sequence of consecutive discrete and delay transitions. However, for reasoning about real-time systems, we are interesting in studying not only the actions that are performed when the system runs, but also the instant at which each action is executed. Therefore, an adequate way to describe the behavior of a p-automaton is given by the notion of *timed trace*. A timed trace is a sequence of pairs $[(a_1, t_1), \dots, (a_n, t_n)]$ where a_i denotes an action and t_i the instant of execution of a_i . A couple (a_i, t_i) is called a *dated* action.

Behind this general model, a subclass of p-automata called *restricted p-automata* has been identified in [5], where \mathcal{L} , \mathcal{A} , \mathcal{M} are finite sets, \mathcal{V} is a set of real-valued variables, and update relations and invariants are linear expressions. These constraints allow to translate restricted p-automata into linear hybrid automata [1, 3] on which tools for automatic analysis[11] can be applied.

Example: figure 1 depicts a restricted p-automaton which models a digicode with three keys A, B and C. The correct sequence of keys, i.e. the one which causes the door to open, is A, B, A. Since only three errors are tolerated, the variable *cpt* memorizes the number of errors. Moreover, some delays control the input: B must be entered at most 10 seconds after the key A. The last key A must be entered before 20 seconds. Four successive errors or a too long time between two keys cause the lock of the digicode. A correct timed trace of the digicode is, for instance, the sequence of dated actions $[(A, 3.2), (B, 5.4), (A, 6.1)]$.

3.2 P-automata in *CClair*

The formalization of the model of p-automata in our framework takes the form of an extension of the *CClair* theories on transition systems. It contains the definition of the polymorphic type (α, l, ν) pa which represents p-automata of actions of type α , locations of type l and variables

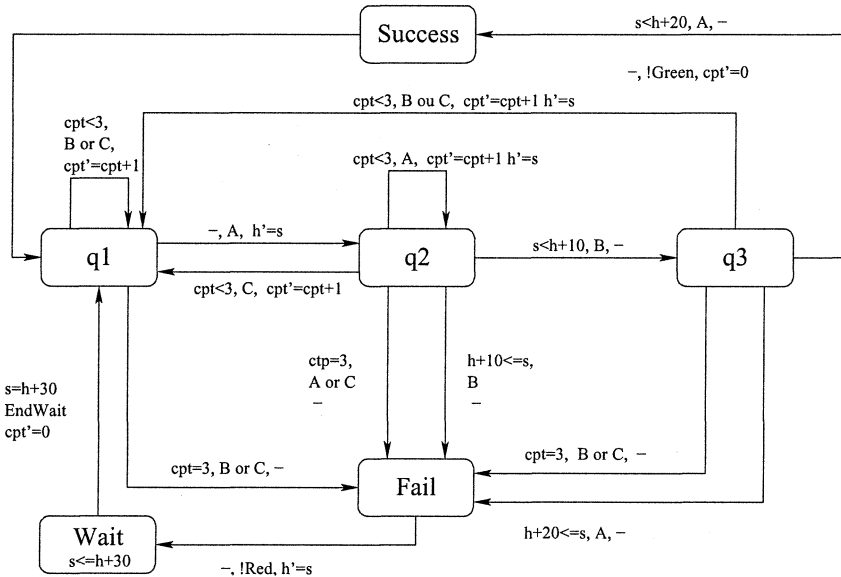


Figure 1. A example of p-automaton

of type ν . Because α, l and ν may be instantiated with any type, it is possible to represent systems with an infinite number of actions or locations. Moreover, variables can contain lists, sets or more complex data structures. A translation of p-automata into plain transition systems is also provided to represent p-automata during execution. Consequently, discrete transition and time delay need not to be defined: it is enough to consider transitions of the translation into the parent theory.

In order to make easier the communication between a finite execution and its associated timed trace, these notions are introduced simultaneous under the form of a predicat defined inductively: given a p-automaton P , $F_TTraces_P$ contains the objects of the form $s \xrightarrow[\xi, w]{P} s'$ where ξ is an execution of P from the state s to t together with its corresponding timed trace w . $s \xrightarrow[\xi, w]{P} s'$ is called a *run* of P . A run is built with the help of three rules, called *introduction rules*: a run is either empty or obtained by adding a discrete-transition or a delay-transition on the head of an already defined run.

A parallel composition operator, compatible with the *Hytech* composition[11], is provided to modelize complex systems from a collection of automata, each describing a modular component of the system. Consider two p-automata A and B , locations of the parallel composition $A||B$ are

the pairs of location (l_1, l_2) where l_1 is a location of A and l_2 is a location of B . The invariant function of $A||B$ assigns to (l_1, l_2) the conjunction of l_1 and l_2 's invariants.

A move of $A||B$ is either a move of A or B alone or a combined move if the action is common to A and B . In the latter case, the update relation is the conjunction of the update relation of each component.

Numerous results, about runs of p-automata and their composition, are proven in *CClair*. Among them, the following theorem will be used in section 4.5.

$$\begin{array}{l}
 H1 : \text{acts_of } w \subseteq \mathcal{A}_A \cap \mathcal{A}_B \wedge \\
 H2 : s \xrightarrow[P||Q]{x,w} t \\
 \hline
 \pi 1_s s \xrightarrow[A]{\pi 1_x x,w} \pi 1_s t \wedge \pi 2_s s \xrightarrow[A]{\pi 2_x x,w} \pi 2_s t
 \end{array} \tag{1}$$

Given an execution in the composed automaton, this theorem infers an execution with the same timed trace in A and B using the projections $\pi 1_s, \pi 2_s, \pi 1_x$ and $\pi 2_x$. These are used to project states and executions onto the first and second components. The additional assumption $H1$ requires that all actions mentioned in the trace (collected by the function acts_of) are common to A and B .

3.3 A proof approach of the test

Following the definition from standard ISO 9646[13], an implementation conforms to its specification if it satisfies all the conformance requirements which are explicitly mentioned in the protocol standard. Consequently, the process of conformance testing aims at deriving a set of tests cases which will check whether an implementation satisfies all these conformance requirements. In this paper, we focus on the first step of this process which consists in deriving *generic test cases* from the specification i.e. test cases which are independent of any implementation.

Conformance requirements have to be expressed in a formal language and therefore each requirement may involved one or many so-called *test purposes* which are formal descriptions of the constraints to impose on the behavior of the specification in order to decide whether the particular conformance requirements is satisfied. Using terms of theorem proving, we obtain an generic test case if we can prove that it corresponds to an execution of the specification that satisfies a test purpose. Since our aim is also to compute such a test case, we start with an incomplete statement where logical variables replace the desired execution.

Consequently, deriving a test case amounts to solve a statement of the following form:

$$s \xrightarrow{P} \xi^?, w^? \ s'^? \ \wedge \ Q(s, \xi^?, w^?, s'^?) \quad (2)$$

3.4 How to describe Test purposes

Translating conformance requirements into formal language is a non-obvious task: requirements are provided in natural language which often contains many ambiguities. If the formal language used for this translation is too poor, it may lead to errors on the meaning of the requirement. Since the property Q in the goal 2 is a formula of *HOL* logic, one disposes of a large expressive language. Moreover, *CClair* provides many operators, commonly used in verification and testing activities: LTL modalities, language constructors and of course automata. If it is not enough, user may develop his own operators together with a translation into concepts already defined in order to prove the soundness.

Let us emphasize that we are not limited to express test purpose about traces. The reason is that Q takes as argument the execution ξ and an execution contains all information about the successive states of the system. We can therefore formalize constraints on particular value of variables or location. Suppose, for instance, that we seek to obtain a test case where the counter exceeds the allowed value before the system reaches the location $q3$. Using the selectors `cpt` and `loc` to extract the value of the counter and the location from states, the property Q may be: $\xi^? \models \diamond((\text{cpt} > 3) \wedge \bigcirc \diamond(\text{loc} = q3))$

3.5 Deriving test cases

Deriving a test case amounts to providing a constructive proof of 2. The basic technique to build an execution is to use of simulation tactics in which introduction rules of the predicat `F_TTraces` are applied. Once a part of the execution is instantiated, the second part of the goal may be simplified through rewriting. If simplifications lead to the formula “False”, this means that an error is detected and another execution is searched using backtracking facilities.

This higher degree of interactivity is the price to pay to enable one to work with models known to be undecidable and therefore for which no automatic procedure exists. In this difficult cases, tactics are available to make the task easier. For instance, one may extract the *control graph* of a p-automaton, by ignoring all but action symbols and locations ; if this graph is finite, we can apply certified algorithms on graphs to

compute candidates for executions, which are confronted to invariants and update relation of the original automaton.

If automatic procedures may be used, then the generation of certified test cases is fully automatic as explained in section 4.

4. DERIVING TEST CASES FROM CERTIFIED TEST PURPOSE

This section focuses on the production of test cases for the class of restricted p-automata. Since a restricted p-automaton is a kind of linear hybrid automaton[2], we can invoke the verification tool *Hytech* from *CClair* in order to automatically compute an execution.

Our overall approach is as follows: a temporal test purpose T is expressed using a kind of regular expressions. Then, this expression is translated by means of rewriting rules into a timed-automaton T so that the traces of T satisfy the test purpose. The latter property is established by a pre-proven theorem. Next, T is used as a synchronous observer to guide the construction of an execution which is computed by *Hytech* and finally certified in *CClair*.

4.1 Final location and accepted language

The p-automata model does not permit to declare a final location. For the purpose of defining the language accepted by a p-automaton, we therefore need to extend the model. It is easily done under *CClair* by introducing a new type of automata, called **paf** (p-automata with final location) that adjoins a final location to each p-automata.

$$(\alpha, l, \nu) \text{ paf} = ((\alpha, l, \nu) \text{ pa} \times l)$$

A run is accepted by a given automaton (P, f) if there exists a corresponding run in P whose last location is f .

$$(s, w) \in \text{Accepts}_{(P, f)} \equiv \exists \xi t \mid s \xrightarrow{P, \xi, w} t \wedge (\text{loc } t = f)$$

4.2 A particular test purpose

Conformance requirements may be formalized by a set of test purposes where each test purpose describes a particular sequence of observable events that the system has to perform in order. As we study timed systems, it is moreover convenient to precise the instant at which each action must be observed. Such test purposes amount to provide a subword of a timed trace of the specification. *CClair* provides a dedicate operator, named **Subword** to express make this class of test purposes. To be

more precise, given a sequence of dated actions $l = [(a_1, t_1), \dots, (a_n, t_n)]$ ($a_i \in \mathcal{A}, t_i \in \mathbb{R}$), $\text{Subword}(l)$ specifies the set of timed traces which correspond to the regular expression $(\mathcal{A} \times \mathbb{R})^* \cdot (a_1, t_1) \cdot (\mathcal{A} \times \mathbb{R})^* \dots (a_n, t_n) \cdot (\mathcal{A} \times \mathbb{R})^*$ where $*$ denotes the Kleene-star on language and \cdot the concatenation. In other words, Subword allows us to select timed trace whose l is a subword. This means that between two given actions, the system is allowed to perform many other actions.

Using the digicode of the figure 1 as an illustration, $\text{Subword}([(A, 3.0), (B, 6.2), (A, 8.3)])$ represents the set of timed traces that contains the two following timed traces.

$$\begin{aligned} & [(B, 1.1), (A, 3.0), (A, 4.6), (B, 6.2), (A, 8.3)] \\ & [(A, 3.0), (C, 4.9), (A, 5.3), (A, 5.8), (B, 6.2), (A, 8.3)] \end{aligned}$$

In order to derive test cases, we are therefore interested in solving the testing statement of the section 3.3 where A is instantiated by Subword :

$$s \xrightarrow[P]{\xi^?, w^?} s' \wedge w^? \in \text{Subword}(l) \quad (3)$$

4.3 Timed regular expressions

Our purpose is to translate Subword into an automaton so as to use *Hytech* on it. It is a well-known result that regular expressions can be translated into a labeled transition system which accepts the language associated to the expression, and vice-versa. Consequently, it is natural to introduced regular expressions that can be translated into an equivalent automaton and then use these expressions to defined Subword .

A recent paper written by T.Nipkow[15] has explored the ability to formalize the Kleene's theorem into *Isabelle*. However this development cannot directly be reused here for several reasons:

For one, we need expressions that contain time informations and therefore we must translate them into p-automata, not into simple transition systems. For another, Nipkow uses an adhoc representation that differs from our actual formalization of p-automata. A third reason is that the construction presented in [15] is realized in two steps: a automaton with epsilon-transitions is first generated then it is transformed into a deterministic automaton. However, the latter construction involves a non-executable operator on set. Fortunately, to define Subword , we only need a subset of regular expressions, and we can therefore generate, directly from expression, an automaton without epsilon-transitions.

Hence, using the ML syntax, a timed expression on letters of type α is defined by the type α `texp`:

$$\begin{aligned} \text{datatype } \alpha \text{ } \text{texp} = & \quad \text{One_letter } \alpha \times \mathbb{R} \\ & | \quad \text{Any_letters} \\ & | \quad \text{Conc } (\alpha \text{ } \text{texp}) (\alpha \text{ } \text{texp}) \end{aligned}$$

Following the definition, an timed expression is constructed by concatenation of dated actions. A dated action is either explicitly given using the constructor `One_letter` or taken in the set $\mathcal{A} \times \mathbb{R}$ if the constructor `Any_letters` is used.

Given a regular expression, the primitive recursive function `lang` computes its corresponding set of dated actions `word`.

$$\begin{aligned} \text{lang } (\text{One_letter } (a, t)) &= \{[(a, t)]\} \\ \text{lang } (\text{Any_letters}) &= \text{Star} \left(\bigcup_{a \in \mathcal{A} \ 0 \leq t} \{[(a, t)]\} \right) \\ \text{lang } (\text{Conc } L_1 \ L_2) &= \text{Concat} (\text{lang } L_1) (\text{lang } L_2) \end{aligned}$$

Finally, `Subword` is defined as the language of the timed expressions built by the recursive function `sb_exp`.

$$\begin{aligned} \text{sb_exp}([]) &= \text{Any_letters} \\ \text{sb_exp}((a, t).l) &= \text{Conc } \text{Any_letters} \\ & \quad (\text{Conc } (\text{One_letter } (a, t)) (\text{sb_exp}(l))) \\ \text{Subword}(l) &= \text{lang } (\text{sb_exp}(l)) \end{aligned}$$

4.4 Deriving a certified automata

The translation of each component of a regular expression into a corresponding p-automaton is depicted on figure 2. The corresponding automata are very simple: locations are natural numbers and no variables, except the universal clock, are required. However, we need a type of variable to declare these automata in our typed framework. One solution is to use the special (ML like) type `unit` which contains the single value v .

Given an expression E of type `texp`, the result of the translation is the paf (T, f) where T is a p-automaton (the timed trace of which are in the language associated to the expression) and f its final location. The desired property is stated by the pre-proven theorem: ¹

$$\forall t_i \ t_f \ (t_i, v, 0) \xrightarrow[T]{\xi, w} (t_f, v, f) \longrightarrow w \in \text{lang } E \quad (4)$$

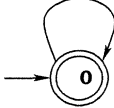
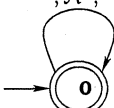
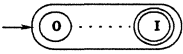
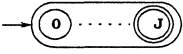
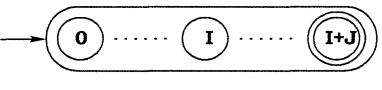
Expression	P-automata
<i>One_letter</i> (a,t)	$S = t, a, -$ 
<i>Any_letters</i>	$-, A^*, -$ 
<i>exp1</i>	
<i>exp2</i>	
<i>Conc exp1 exp2</i>	

Figure 2. Translation of expressions into p-automata

However, we are only interested in the instance of 4 where E is built using *Subword*.

$$\forall t_i t_f (t_i, v, 0) \xrightarrow[T]{\xi_i, w} (t_f, v, f) \longrightarrow w \in \text{Subword}(l) \tag{5}$$

Let us emphasize that it is an executable translation: definitions use mainly primitive recursions so that a automaton is automatically generated from a regular expression, thanks to the rewriting mechanism of *Isabelle*.

4.5 Validation of the observer technique

Now, if we put 5 on the goal 3, we obtain the following new subgoal:

$$s \xrightarrow[P]{\xi_1^?, w^?} s'^? \wedge (t_i^?, v, 0) \xrightarrow[T]{\xi_2^?, w^?} (t_f^?, v, f) \tag{6}$$

The next step consists to validate the well-known technique of observer automata: the basic idea is that the automaton T can be used as a synchronous observer i.e. it can be executed in parallel with the specification automaton in order to find an execution. This step is formalized from

to the rule given in 1 proven in section 3.2 and we obtain the following goal :

$$s_c \xrightarrow[P||T]{\xi^?, w^?} s_c' \quad (7)$$

Where s_c (resp. s_c') is the composition of the initial state (resp. final) of the runs in P and T . s_c is completely instantiated but we may take the liberty to let s_c' uninstantiated or to precise a part of the state. It is a means to precisely control the search by constraining the execution to end with particular values of variables. In this case, we have $s_c = (t_i, (v_i, v), (l_i, 0))$ and $s_c' = (t_f^?, (v_f, v), (l_f^?, f))$.

Notice that *Isabelle* keeps in memory the instantiation made during resolution of the goal 2 with 1 i.e. ξ , ξ_1 and ξ_2 are connected by the equations $\pi_{1_x} \xi = \xi_1$ and $\pi_{2_x} \xi = \xi_2$. When ξ is fully instantiated, it is obvious to compute the value of ξ_1 by rewriting.

4.6 An oracle called *Hytech*

Hytech[11] is a symbolic model checker for linear hybrid systems which may automatically generate a diagnostic trace that explains why a property is (or is not) satisfied by a system description. Therefore the tool *Hytech* can be used to search an execution in the composite specification+observer automaton.

To achieve this task, we need to build an input file that includes descriptions of automata and commands to compute an execution. Because we work with the restricted format of p-automata, the translation into hybrid automata is nearly obvious: all variables are real and declared as *discrete* variables. This ensures that they will not evolve except on explicit update. The universal clock is declared as a variable of type *clock* called *now*. The only difficulty concerns the translation of constraints (update relations and invariants) because *CClair* allows the declaration of any necessary constant to define it. As a consequence, the first step of the translation procedure is to compute a “normal form” which is done through rewriting. The user is responsible for giving all equations needed to expand constants by their definitions.

Two *Hytech* commands are needed to compute an execution in our composed automaton.

```
reached := reach forward from init_reg
print trace to final_reg using reached
```

One must first use forward reachability analysis to compute all states that are reachable from the initial region `init_reg` which contains the

single state s_c . The second command causes *Hytech* to generate an execution from `init_reg` to the final region `final_reg` $\{s'_c\}$.

4.7 Validation of the computed execution

Having obtained a desired execution, we then use this result to solve the goal 7. Two choices are offered to us, depending of the level of confidence we have about *Hytech* and our interface with *CClair* : we can admit the computation result as an axiom and directly apply it on 7. A safer alternative is to instantiate the variable $\xi^?$ with the computed execution and then validate it. Let us explain briefly how this latter step can be achieved.

Because all sets involved in the definition of a restricted p-automaton are finite, rewriting mechanism is powerful enough to decide whether a given move is in the set of moves. For instance, $\langle q_1, C, \text{cpt} = 3, \text{Fail} \rangle \in \mathcal{M}_{\text{digicode}}$ is simplified to `True` and therefore is accepted as a move of the digicode.

Concerning the data part of a transition, *Isabelle* provides a decision procedure for linear arithmetic. Therefore the value of variables computed by *Hytech* according to the update relations and invariants are automatically admitted or rejected. As a result, *CClair* provides a tactic which automatically checks the validity of the execution computed by *Hytech*.

5. A CASE STUDY : ABR CONFORMANCE ALGORITHMS

Available Bit Rate[10] is a special kind of connection defined in ATM network architecture : it allows the cell rate to vary during the same session. Therefore, a real-time reactive algorithm is necessary to control that data cells sent by user conform to the negotiated traffic parameters. Two of such algorithms are formalized in [6]. The first one, called \mathcal{I} , has the ideal desired behavior but its implementation is impossible in practice because of a too large memory cost. The second one, \mathcal{B}' , is therefore an approximation for which a proof of correctness with respect to the ideal algorithm is given. This proof has completely been formalized in our framework[19], including the description of algorithms in terms of p-automata, and proving a set of invariance assertions by induction on execution sequences.

The rate offered to the user depends of the instant at which resource management (RM) cells are received. The crucial point is therefore to test if \mathcal{B}' computes a rate greater or equal to the rate computed by \mathcal{I} for a given sequence of values carried by RM cells and their arrival

time. Thus, each sequence corresponds to a test purpose of the form: $\text{Subword}([(RM, t_1), \dots, (RM, t_n), (D, t_{n+1})])$ where the actions RM and D represent respectively the reception of a RM-cell and an user data cell. This allows to apply our *CClair/Hytech* approach for automatically generating certified test cases.

As an illustration, we present an execution obtained from the small test purpose $\text{Subword}([(RM, 3), (D, 30)])$.

Actions performed	$\delta(3)$	RM	I1	A7	$\delta(10)$	A9a	$\delta(17)$	D
Value of \mathcal{S}	0	3	3	3	3	13	13	30
Rate computed by \mathcal{I}	0	0	0	1	1	1	1	1
Rate computed by \mathcal{B}'	0	0	0	0	0	0	1	1

The first line presents the trace of the execution : δ denotes time elapse, I1 is an action of \mathcal{I} , A7 and A9a are actions of \mathcal{B}' . After each step, the value of the clock and the rates computed by \mathcal{I} and \mathcal{B}' are recorded.

We obtain a generic test case by filtering (through rewriting) the actions of the execution. The adaptation of this result according to the test architecture and the available PCO will be done in a further research.

6. CONCLUSION AND FUTURE WORKS

We have presented a general approach for generating generic test cases in the design of real-time systems using an interactive theorem prover. The formal framework is based on HOL specifications and verification environment that includes an expressive specification language and a powerful reasoning system. Furthermore, the use of the model of p-automata, which include truly real-valued variables allows a natural specification of systems.

We have described a technique for automatically deriving certified test cases from formal descriptions of test purposes expressed as a subset of regular timed expressions. This method significantly increases the confidence in the test case generation process since all reasoning steps and computation are checked by the theorem prover.

In future work, one should enrich the language of expressions to provide more refined constraints on the time part of dated actions. We also plan to study a new protocol called PGM².

Notes

1. We do not present here the proof of this theorem. All scripts are available online at http://dept-info.labri.u-bordeaux.fr/Davy_rouillar/CCLair/.

2. see http://www.cisco.com/warp/public/cc/pd/iosw/tech/_fr_xcst_ds.htm

References

- [1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
- [2] R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the 14th Annual Real-time Systems Symposium*, pages 2–11. IEEE Computer Society Press, 1993.
- [3] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *Lecture Notes in Computer Science*, 736:209–229, 1993.
- [4] B. Baumgarten. Timed systems behaviour and conformance testing - a mathematical framework. In *Proc. 8th PTS*, 1995.
- [5] B.Bérard, P.Castéran, E.Fleury, JF.Monin L.Fribourg, C.Paulin, A.Petit, and D.Rouillard. Automates temporisés calife, 2000. <http://www.loria.fr/projets/calife/>.
- [6] B. Bérard, L. Fribourg, F. Klay, and J.-F. Monin. A compared study of two correctness proofs for the standardized algorithm of ABR conformance. Research Report LSV-99-7, LSV, ENS de Cachan, Cachan, France, August 1999. 27 pages.
- [7] P. Castéran and Davy Rouillard. Reasoning about parametrized automata. In *Proceedings, 8th International Conference on Real-Time System*, volume 8, pages 107–119, 2000.
- [8] Pierre Castéran and Davy Rouillard. The cclair project, 1999. <http://dept-info.labri.u-bordeaux.fr/~casteran/CCclair/>.
- [9] E. Allen Emerson. *Handbook of Theoretical Computer Science (volume B)*, chapter Temporal and Modal Logic, pages 995–1072. Elsevier, 1990.
- [10] A. Forum. Atm forum traffic management specification version, 1996.
- [11] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *Lecture Notes in Computer Science*, 1254:460–463, 1997.
- [12] T. Higashino, A. Nakata, K. Taniguchi, and A. Cavalli. Generating test cases for a timed i/o automaton model. In *Proc. IFIP Int'l Work. Test Communicat. Syst. (IWTCS)*, pages 197–214. G. Csopaki, S. Dibuz, and K. Tarnay eds, 1999.
- [13] ISO. Information technology, open systems interconnection, conformance testing methodology and framework. *International Standard IS-9646*, CCITT X.290-X.294, 1991.
- [14] Jean-François Monin and Francis Klay. Correctness proof of the standardized algorithm for ABR conformance. In *FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 662–681. Springer, 1999.
- [15] Tobias Nipkow. Verified lexical analysis. In J. Grundy and M. Newey, editors, *Proc. of the 11th International Conference on Theorem Proving in Higher Order Logics*, pages 1–15. Springer-Verlag LNCS 1479, 1998.
- [16] Christine Paulin-Mohring. The coq project, 1999. <http://coq.inria.fr>.
- [17] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.
- [18] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Logic and Computation*, 7:175–204, 1997.

- [19] Davy Rouillard. Le modèle des p-automates dans CClair. Technical Report 1242-00, LaBRI, 2000.
- [20] N. Shankar, S. Owre, and J. Rushby. The pvs proof checker: A reference manual. Technical report, CSL, SRI International, Menlo Park CA, 1993.