

Mechanical Verification of a Generic Incremental ABR Conformance Algorithm

Michaël Rusinowitch, Sorin Stratulat, Francis Klay

▶ **To cite this version:**

Michaël Rusinowitch, Sorin Stratulat, Francis Klay. Mechanical Verification of a Generic Incremental ABR Conformance Algorithm. [Research Report] RR-3794, INRIA. 1999, pp.43. inria-00072865

HAL Id: inria-00072865
https://inria.hal.science/inria-00072865

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Mechanical Verification of a Generic Incremental ABR Conformance Algorithm

Michaël Rusinowitch , Sorin Stratulat , Francis Klay

N˚3794

Novembre 1999

THÈME 2



ISSN 0249-6399 ISRN INRIA/RR--3794--FR+ENG

Mechanical Verification of a Generic Incremental ABR Conformance Algorithm *

Michaël Rusinowitch† , Sorin Stratulat‡ , Francis Klay§

Thème 2 — Génie logiciel
et calcul symbolique
Projet PROTHEO

Rapport de recherche n ° 3794 — Novembre 1999 — 43 pages

Abstract: The Available Bit Rate protocol (ABR) for ATM networks is well-adapted to data traffic by providing minimum rate guarantees and low cell loss to the ABR source end system.The protocol relies on a contract between the operator who ensures a minimum rate and the source who must respect a rate that is dynamically allocated to him, according to the resources available in the networks. An ABR conformance algorithm for controlling the source rates through an interface has been defined by ATM Forum. A more efficient version of this algorithm has been designed by C. Rabadan and F. Klay. We present in this work the first complete mechanical verification of the equivalence between these two algorithms. The proof is rather involved and has been supported by the PVS theorem-prover. It has required many lemmas, case analysis and induction reasoning for the manipulation of non bounded scheduling lists. Previous works on the automated verification of ABR conformance protocols have only dealt with approximations of the algorithm we consider here since they assume that the scheduling lists contain at most two elements.

Key-words: automated verification, protocols, ATM networks, Available Bit Rate, theorem-proving

(Résumé : tsvp)

* Supported by CNET CTI 96 1B 008 and Action de Recherche Coopérative INRIA PRESYSA
† LORIA, INRIA B.P. 239, 54506 VANDŒUVRE-LES-NANCY CEDEX, FRANCE. email: rusi@loria.fr
‡ LORIA, Université Henri Poincaré B.P. 239, 54506 VANDŒUVRE-LES-NANCY CEDEX, FRANCE. email: stratula@loria.fr
§ France Telecom CNET DTL/MSV, Technopole Anticipa 2 av. Pierre Marzin, 22307, Lannion, France. email: Francis.Klay@cnet.francetelecom.fr

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : 03 83 59 30 30 - International : +33 3 3 83 59 30 30
Télécopie : 03 83 27 83 19 - International : +33 3 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ

Vérification automatisée d'un algorithme de contrôle de conformité incrémental et générique pour la capacité de transfert ABR

Résumé : Le protocole ABR pour les réseaux ATM est bien adapté au transport de données en assurant un débit minimum et en limitant les pertes. Le protocole repose sur un contrat entre l'opérateur qui fournit un débit minimum et une application qui respecte le débit qui lui est alloué de manière dynamique, selon les ressources disponibles du réseau. L'ATM Forum a défini un algorithme pour contrôler la conformité du débit de la source. C. Rabadan et F. Klay ont proposé une version incrémentale plus efficace de cet algorithme. Nous avons obtenu une preuve automatique complète de l'équivalence entre les deux algorithmes en utilisant le système PVS. La preuve nécessite de nombreux lemmes et une analyse par cas complexe. Les travaux antérieurs sur la vérification automatique de protocoles ABR traitaient seulement des approximations (à deux cellules) de l'algorithme que nous considérons ici.

Mots-clé : vérification automatique, protocoles, réseaux ATM, Available Bit Rate, preuve automatique

Introduction

The Available Bit Rate protocol (ABR) for ATM networks is well-adapted to data traffic by providing minimum rate guarantees and low cell loss to the ABR source end system. The protocol relies on a contract between the operator who ensures a minimum rate and the source who must respect a rate that is dynamically allocated to him, according to the resources available in the networks. Due to its flexibility the ABR service admits elaborated traffic management mechanisms . To avoid congestion the operator should control in real-time that the actual rate consumed by every ABR application is consistent with the allowed rate. Several algorithms for this conformance control are proposed and discussed in standardization committees.

It is essential for an operator to give evidence that the conformance control of the service he proposes does not jeopardize the quality of service (QoS) provided by the ATM network. For such a task formal validation through mathematical arguments is required. However conformance control verification often involves complex case analysis or inductions. This has motivated some operators to employ automated verification tools such as proof-assistants or model-checkers to process these proof obligations.

The algorithm A (also called Acr) that has been defined by ATM Forum is considered to give the optimal conformance control, in that it computes the minimal allowed rate among the others algorithms. An algorithm (B') for computing an approximation of the optimal control has been designed by C. Rabadan from France-Telecom [Rab97]. It is more efficient since the next two rates to be controlled are scheduled and are updated when receiving RM-cells. This incremental algorithm has been generalized in [RK97] to the scheduling of an arbitrary number of rates in the future. Our goal here is to derive a mechanical proof that this generic incremental algorithm is indeed equivalent to the reference algorithm A. By this we mean that every step in the equivalence proof has been verified mechanically. The theorem prover we use is PVS [ORS92]. Although PVS is interactive and operates under the direct control of the user it is capable of large autonomous deduction steps by appealing to decision procedures for arithmetics, to rewriting and induction.

Related works

Some ABR conformance protocols have been automatically verified in previous works. However these protocols are approximations of A. In particular unlike our case they assume a bound on the number of rates to be scheduled. For instance algorithm B', where the scheduling list is restricted to two elements, has been proved recently in [BF99] using temporized automata [HHWT97]. According to [MK99] the correctness proof of the ABR protocol B' of C. Rabadan [Rab97] *has been a key argument in the standardization process of ABR*. This proof is based on the calculus of weakest preconditions [Dij76] (inductive invariants) and has been completely formalized with the COQ proof-assistant [BBC+97]. Several proof techniques have been experimented in the FORMA project [1] for the validation of the ABR protocols. Model checking approaches have been hindered by the numerical parameters of the algorithm. As mentioned above some success has been reported in [BF99] by using the parameterized temporized automata of Hytech [HHWT97]. L. Fribourg has also obtained good results with extended timed automata [Fri98].

Layout of the paper

We first describe the principle of ABR Conformance in Section 1. Then we introduce the two algorithms Acr and Acr1 for controlling the ABR Conformance in Section 2 and 3 respectively. The rest of the paper is devoted to the equivalence proof of this two functions. We first introduce the PVS definitions for Acr and Acr1 in Section 4 and then give an overview of the proof in Section 6. Since the PVS proof is too complex to be presented in extenso we have given a skeleton that follows closely the mechanical proof. In particular we refer to explicit instances of lemmas that were derived during the PVS proof construction. The whole list of lemmas is given in tables 1 and 2. The set of key lemmas is described in Appendix A. The full PVS code of the specification is listed in Appendix B. The last two appendices present a full PVS proof of a nontrivial lemma.

[1] http://www-verimag.imag.fr

1 ABR Conformance Control

ATM (Asynchronous Transfer Mode) technology allows networks to transmit on the same media various applications whose needs are different in term of dataflow rate or quality of services. ATM is a connection-oriented technology since users should declare service requirements and traffic parameters to all intermediates switches when initializing connections. They also may agree to control these parameters on demand. In order to guarantee QoS a traffic contract specifying a traffic mode is negotiated when the connection is set up. Traffic management should ensure that users get their desired QoS although traffic demand is constantly varying. In other words traffic management should ensure that all contracts are met.

In order to solve the critical issue of *congestion control* the effective rate of cells emitted by user applications is controlled by a *conformance algorithm* called GCRA (Generic Control of Cell Rate Algorithm).

Among the possible traffic modes, Constant Bit Rate (CBR) and Variable Bit Rate (VBR) where designed mainly for traffic like voice and video. The Available Bit Rate (ABR) service class is especially adapted for standard data traffic, where timing constraints are not tight. Target applications for ABR are email, WWW, file transfer and variable quality video and voice. The principle of ABR is to divide the available bandwidth [2] fairly among active traffic sources so that the network should provide each user with the best rate that is compatible with the current traffic (best-effort service principle). In ABR connections the allowed cell rate (ACR) is determined by the network from load information and may vary during the same connection. The network informs periodically the user about the new rate he can apply by sending him *Resource Management (RM)* cells. Hence the source rate is dynamically adjusted according to the available resources of the network by a *feedback control loop*. Since the allocated rate varies during a connection with ABR mode, the conformance control is performed by a dynamic GCRA (DGCRA).

Several ABR conformance algorithms have been proposed to the normalization committees. The algorithm Acr [BBF95a] can be viewed as a reference for defining the control of the user dataflow in the case of ABR. Algorithm A computes for each data cell outcoming from the source into the control interface the rate that should be applied to it. The computational cost induced by A has been considered too high and there was several proposals to alleviate it.

For instance it was noticed that the rate change is determined only by the departure of RM-cells from the control interface toward the application source and these RM-cells are much less frequent than data cells. Hence there is much improvement in scheduling rate changes in advance when receiving RM-cells. This is the motivation for the so-called incremental algorithm Acr1, that maintains a list of planned rates that is updated on receiving a new RM cell. Also controlling the rates with the scheduled list of Acr1 seems to be less expensive than computing the maximum of a list of rates with Acr.

For efficiency reasons several approximation algorithms have been designed. The algorithm B' of C. Rabadan [Rab97] considers only two RM-cells at a time and computes an approximated value for Acr. It has been selected by ITU (Recommendation I-371.1) and ATM Forum. Before it was accepted as a standard, it has been necessary to provide evidence that the flow control generated by B' is never unfair to the user, i.e. the rate allowed by B' is always greater than the reference rate computed with Acr. However it was noticed that approximation algorithms often overestimate too much the rate to be controlled by the interface.

2 Dataflow Control Definition with ABR (Algorithm Acr)

We shall give the dataflow control definition, called here Acr, that has been first introduced in ATM Forum by [BBF95b] and since then has been considered as a reference for the other algorithms.

The principle of the algorithm A running in the interface is to manage the list of RM-cells received from the network in order to determine the rate that has to be controlled at the current time. We shall associate with each RM-cell a couple (t, er) where t is the time when the RM-cell leaves the interface (where conformance control is performed) towards the application source, and er is the new rate requested by the network. For our discussion we identify the cell and its associated couple. The control device receives the new expected rate value before the application. Hence to take into account the transmission delays in the networks, the control device should apply at time t a value received at time $t - \tau$, where τ represents a propagation delay equal to the time taken by a RM-cell to go from the interface to the application and back to the interface. However the

[2] left-over by CBR,VBR, e.g.

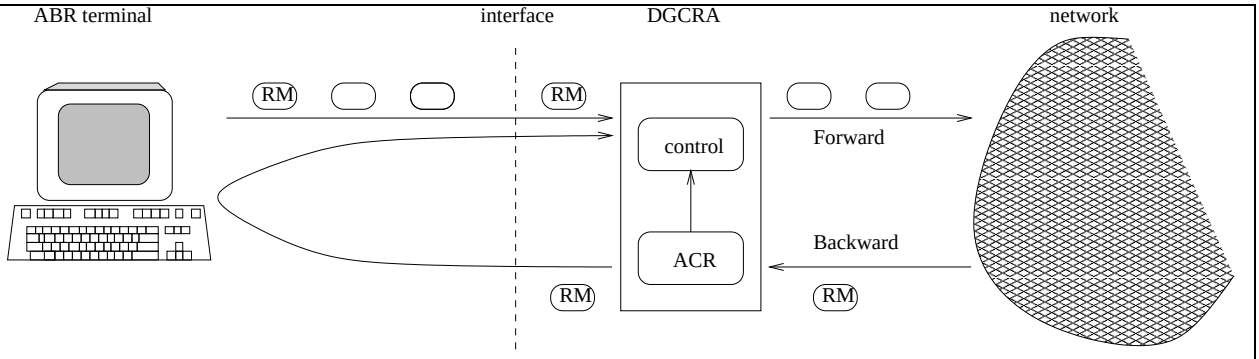


Figure 1: Time delays between the network and an ABR terminal

propagation times in the network may vary according to the traffic load. In order to take into account variations of these delays, the ITU-T has proposed that the rate to be controlled at time t is computed as the maximum among the rates received by the interface within a temporal window $t-\tau_2 \ldots t-\tau_3$ and the one received just before $t-\tau_2$. The *window parameters* τ_2, τ_3 satisfying $\tau_2 > \tau_3$ have been negotiated during the establishment of the traffic contract.

More formally, let $l =< (t_i, er_i) >_{i=1\ldots n}$ be the list of RM-cells that have been conveyed to the control interface. We assume that the list is sorted in decreasing order on time: $i < j \Rightarrow t_i \geq t_j$. Hence t_0 is the most recent reception time of a RM-cell. In order to handle limit cases we shall define a special virtual cell whose reception time is: $t_{-1} = \infty$. The rate that has to be controlled at time t w.r.t. the list of received RM-cells l is:

$$Acr(l,t) = \begin{cases} MaxEr(Wind(l,t)) & \text{if } Wind(l,t) \neq \emptyset, \\ 0 & \text{otherwise} \end{cases}$$

where

$$\begin{array}{rcl} Wind(l,t) & = & \{(t_i, er_i) \in l |\ (t-\tau_2 < t_i \leq t-\tau_3) \text{ or } (t_i \leq t-\tau_2 < t_{i-1})\} \\ MaxEr(s) & = & max\{er | (t,er) \in s\} \end{array}$$

For instance, with this rate control policy the end user can benefit at time $t+\tau_3$ from a rate increase received (on a forward cell) at time t by the interface, that is, as soon as possible. On the other hand a rate decrease will be taken into account only at time $t+\tau_2$, that is, as late as possible. Hence this is a policy in favour of the user and based on worst-case situations. It can be noticed that for a fixed l, $Acr(l,t)$ is decreasing on t after $t+\tau_3$ since the window is decreasing (non strictly) for the inclusion relation.

3 Incremental Conformance Algorithm (Algorithm Acr1)

We now introduce a generic algorithm Acr1 for conformance control. Unlike Acr, the algorithm Acr1 computes a list of rates to be controlled in the future. Hence it maintains a list $Prog(l)$ (l is as above) of the future rates to be scheduled together with their validity time. Similar to l, $Prog(l)$ is a list of cells $(time, rate)$ that is sorted in decreasing order on time. This list is updated when receiving an RM-cell. The gain over Acr is due to the fact that the RM-cells are much less frequent than the datacells. The default ratio $\#RM\text{-}cells/\#datacells$ recommended by the ATM Forum is $1/32$ [3]. Also with $Acr1$ the current rate to be controlled is obtained more efficiently than with the maximum computation of Acr. Let us assume that the list $Prog(l)$ is constructed. We shall show later that it sorted in decreasing order w.r.t. time as for l. Then the rate to be applied at time t is:

$$Acr1(l,t) = ProgAt(Prog(l), t)$$

where $ProgAt$ is a function that computes by extrapolation the rate at time t from a list of scheduled rates p. This function simply extracts from p the first rate value that is scheduled at or immediately before the target

[3] ATM Forum Traffic Management Specification Version 4.0. ftp://ftp.atmforum.com/pub/approved-specs/af-tm-0056.000.ps

time t. This rate is the one to be controlled at t:

$$ProgAt(p,t) = \begin{cases} e_i & \text{if exists } (t_i,e_i) \in p \text{ s.t. } t \geq t_i \text{ and there is no } (t_j,e_j) \in p \text{ with} \\ & \quad t \geq t_j,\ i > j \text{ and } i,j \in [1..n] \\ 0 & \text{otherwise} \end{cases}$$

We now describe how the list $Prog(l)$ is updated. When the cell (t,er) received at time t has a bigger rate er than the one that is currently scheduled at time $t+\tau_3$ then er will be better after $t+\tau_3$ than any other possible rate and it is the more recently received one. Hence it can be scheduled safely at $t+\tau_3$ and all subsequent cells deleted. If er is smaller than the rate er_i of the first cell in $Prog(l)$ then er will win over er_i only after $t+\tau_2$ when the window contains the unique cell (t,er); hence er can be scheduled at $t+\tau_2$. If (t_i,e_i) is the first cell such that $er < e_i$ and (t_{i-1},e_{i-1}) exists, at t_{i-1} the rate controlled at the interface is er_{i-1} hence smaller than er; moreover we can show that all subsequent scheduled rates will be smaller than er_{i-1}. Then it is more advantageous for the user that the control interface cancel all rates scheduled at t_{i-1} and later and replace them by a unique cell (t_{i-1},er).

$$Prog((t,er).l) = \begin{cases} \text{if} & er \geq ProgAt(Prog(l,t+\tau_3)) \text{ then} \\ & \text{- delete from } l \text{ the cells with time} > t+\tau_3, \text{ and} \\ & \text{- insert } (t+\tau_3,er). \\ \text{else} & \text{let } (t_i,e_i) \text{ be the cell with minimal } i \text{ such that } er < e_i \text{ and} \\ & \text{let } t' \text{ be } t_{i-1} \text{ if it is defined and } t+\tau_2 \text{ otherwise. Then} \\ & \text{- delete from } l \text{ the cells with time} \geq t' \text{ and} \\ & \text{- insert } (t',er). \end{cases}$$

Our goal in the remaining of the paper is to show that the incremental algorithm Acr1 delivers the same rate values as the reference algorithm Acr, i.e.

$$\forall t\ \forall l \quad Acr1(l,t) = Acr(l,t)$$

4 The PVS Specifications of the Acr and Acr1 Algorithms

We describe in this section the PVS specifications of the Acr and Acr1 algorithms defined in Section 2 and 3. Although PVS admits a higher-order specification language we deliberately restricted ourselves to apply only first-order features of this language. The reason is that our next goal is to prove the equivalence theorem with a first-order theorem-prover in a more automatic way, i.e. with less inputs from the user. In the specification, the type of non-negative reals is denoted by nonneg_real, a cell is represented as a pair of non-negative reals, the variables τ_2, τ_3, t, e are non-negative reals, l is a list of cells and a is a cell. The reference to a function from the PVS specification is denoted by employing `truetype` fonts.

```
cell : TYPE = [nonneg_real, nonneg_real]

τ2, τ3, t, e : VAR nonneg_real
l : VAR list[cell]
a : VAR cell
```

The functions presented in this section implement the algorithms Acr and Acr1 under the conditions: *i*) the cells occurring in the list l have the time-values sorted in decreasing order, and *ii*) the window parameter τ_2 is greater than τ_3. The sets will be implemented as lists. The window parameters τ_2 and τ_3 are added as arguments of the functions *Wind*, *Acr*, *Prog* and *Acr1* presented in Section 2 and 3.

4.1 Common Functions

The functions *Time*, *Er* and *SortedT* are used in the definitions of Acr and Acr1. The function time returns the first component of a cell that we call its time-value;

```
Time((t, e)) : nonneg_real = t
```

The function *Er* returns the second component of a cell called its rate-value;

```
Er((t, e)) : nonneg_real = e
```

SortedT(l) is a predicate that returns TRUE iff the time-values of the cells in l are sorted in decreasing order. In this case, we will say that l is a *time-decreasing* list.

In the PVS language, the constructors of a list are `null` and `cons`. The keyword RECURSIVE specifies that the defined function is recursive while the keyword MEASURE indicates that the expression following it is the measure function to be used during the proof obligations that should be discharged in order to show that the recursive function is well-defined. The measure function should be defined previously. Its signature matches that of the defined recursive function with the exception that its range type is implicitly `nat` or `ordinals`, if no order relation is explicitly specified after the keyword MEASURE.

In the recursive definitions listed in this section, the employed measure is length(l). The function `length`, i.e. the function that returns the length of a list, is predefined in the PVS prelude theory.

```
SortedT(l) :
 RECURSIVE bool = CASES l OF
   null : TRUE,
   cons(a, p1) :
     (CASES p1 OF
       null : TRUE,
       cons(o2, p2) :
         IF Time(a) ≥ Time(o2) THEN SortedT(cons(o2, p2)) ELSE FALSE ENDIF
       ENDCASES)
   ENDCASES
 MEASURE length
```

4.2 Defining Acr

The Acr function is defined from two auxiliary functions: *Wind* and *MaxEr*.

$Wind(l, t, \tau_2, \tau_3)$ recursively defines the $Wind$ function from Section 2. It returns in a list the first cells, let a be such a cell, whose time-value $Time(a)$ satisfies the relation : $Time(a)+\tau_3 \leq t < Time(a)+\tau_2$. We will add at the end of the returned list the first cell from l that occurs at a time less or equal than $t-\tau_2$, if such a cell exists.

```
Wind(l, t, τ2, τ3) :
 RECURSIVE list[cell] = CASES l OF
   null : null,
   cons(a, l) :
     (IF Time(a) + τ3 > t
        THEN Wind(l, t, τ2, τ3)
      ELSIF Time(a) + τ2 ≤ t THEN cons(a, null) ELSE cons(a, Wind(l, t, τ2, τ3)) ENDIF)
   ENDCASES
 MEASURE length(l)
```

$MaxEr(l)$ recursively defines the maximal rate-value of the cells in l. If l is empty, it returns 0.

```
MaxEr(l) :
 RECURSIVE nonneg_real = CASES l OF
    null : 0,
    cons(a, l) :
       (IF MaxEr(l) ≤ Er(a) THEN Er(a) ELSE MaxEr(l) ENDIF)
    ENDCASES
  MEASURE length
```

$Acr(l, t, \tau_2, \tau_3)$ defines the function Acr from Section 2; it computes the maximal rate-value of $Wind(l, t, \tau_2, \tau_3)$ if the cells occurring in the list l have the time-values sorted in decreasing order, and the window parameter τ_2 is greater than τ_3. Otherwise, it returns 0.

Note that when $Wind(l, t, \tau_2, \tau_3)$ is empty, $MaxEr(Wind(l, t, \tau_2, \tau_3))$ is 0.

```
Acr(l, t, τ2, τ3) : nonneg_real =
 IF SortedT(l) ∧ (τ2 > τ3) THEN MaxEr(Wind(l, t, τ2, τ3)) ELSE 0 ENDIF
```

4.3 Defining Acr1

The *Acr1* function is built from the specific functions *InsAt*, *InsIn*, *ProgAt* and *Prog*.

$InsAt(l, t, e)$ inserts in the list l the rate-value e at time t. The cells with time-value greater than t are deleted;

```
InsAt(l, t, e) :
 RECURSIVE list[cell] = CASES l OF
    null : cons((t, e), null),
    cons(a, l) :
       (IF Time(a) ≤ t THEN cons((t, e), cons(a, l)) ELSE InsAt(l, t, e) ENDIF)
    ENDCASES
  MEASURE length(l)
```

$InsIn(l, t, e)$ first deletes all the cells from l with the rate-value smaller or equal to e and then inserts e at the time-value of the last deleted cell. If no deletion operation is performed, e is inserted at time t.

```
InsIn(l, t, e) :
 RECURSIVE list[cell] = CASES l OF
    null : cons((t, e), null),
    cons(a, l) :
       (IF Er(a) ≤ e THEN InsIn(l, Time(a), e) ELSE cons((t, e), cons(a, l)) ENDIF)
    ENDCASES
  MEASURE length(l)
```

$ProgAt(l, t)$ recursively defines the function $ProgAt$ from Section 3; it returns the rate-value of the first cell from l that occurs at a time smaller or equal than t, if it exists. Otherwise, the function returns 0.

```
ProgAt(l, t) :
 RECURSIVE nonneg_real = CASES l OF
    null : 0,
    cons(a, l) :
       (IF Time(a) ≤ t THEN Er(a) ELSE ProgAt(l, t) ENDIF)
    ENDCASES
  MEASURE length(l)
```

$Prog(l, \tau_2, \tau_3)$ implements the function $Prog$ from Section 3 and recursively defines it. The insertion/deletion operations are performed with the help of the functions `InsIn` and `InsAt`.

```
Prog(l, τ2, τ3) :
 RECURSIVE list[cell] = CASES l OF
   null : null,
   cons(a, l) :
     (IF
        ProgAt(Prog(l, τ2, τ3),
          (Time(a) + τ3)) ≤
         Er(a)
       THEN
       InsAt(Prog(l, τ2, τ3),
             (Time(a) + τ3), Er(a))
       ELSE
       InsIn(Prog(l, τ2, τ3),
             (Time(a) + τ2), Er(a))
       ENDIF)
   ENDCASES
  MEASURE length(l)
```

$Acr1(l, t, \tau_2, \tau_3)$ defines the function $Acr1$ from Section 3 if the cells occurring in the list l have the time-values sorted in decreasing order and the window parameter τ_2 is greater than τ_3. Otherwise, it returns 0.

```
Acr1(l, t, τ2, τ3) : nonneg_real = IF SortedT(l) ∧ (τ2 > τ3) THEN ProgAt(Prog(l, τ2, τ3), t) ELSE 0 ENDIF
```

5 Auxiliary Functions

The specification from Section 4 is extended by new definitions in order to express properties that are essential for proving the equivalence of Acr and Acr1. Hence in order to complete the equivalence proof we have added the following ten functions, listed in alphabetical order:

$Erl(l)$ is the rate-value of the first cell occurring in l if l is nonempty and 0 otherwise;

```
Erl(l) : nonneg_real = CASES l OF null : 0, cons(a, l) : Er(a) ENDCASES
```

The functions $FirstAt(l, t)$ and $ListAt(l, t)$ send back the first cell and the sublist starting with the first cell, respectively, of the list l that occurs at a time smaller or equal than t. They are defined in a similar way as the function `ProgAt` from Section 4.3.

$ListUpTo(l, t)$ returns the sublist up to the first cell of the list l that occurs at a time smaller or equal than t.

```
ListUpTo(l, t) :
 RECURSIVE list[cell] = CASES l OF
   null : null,
   cons(a, l) :
     (IF Time(a) ≤ t THEN cons(a, null) ELSE cons(a, ListUpTo(l, t)) ENDIF)
   ENDCASES
  MEASURE length(l)
```

The function $MemberC(a, l)$ tests whether a is a cell from l.

$MemberE(e, l)$ returns TRUE if there exists a cell in l whose rate-value equals e. $MemberC$ and $MemberE$ are defined in a similar way as the function `MemberT` from below.

The function $MemberT(t, l)$ tests whether there exists a cell in l whose time-value equals t.

```
MemberT(tcrt, l) :
 RECURSIVE bool = CASES l OF
    null : FALSE,
    cons(o1, p1) : (IF tcrt = Time(o1) THEN TRUE ELSE MemberT(tcrt, p1) ENDIF)
    ENDCASES
  MEASURE length(l)
```

$SortedE(l)$ returns TRUE if l has its rate-values sorted in strictly increasing order. In this case, we will say that l is a *strictly rate-increasing* list.

```
SortedE(l) :
 RECURSIVE bool = CASES l OF
    null : TRUE,
    cons(a, p1) :
       (CASES p1 OF
          null : TRUE,
          cons(o2, p2) :
             IF Er(a) < Er(o2) THEN SortedE(cons(o2, p2)) ELSE FALSE ENDIF
          ENDCASES)
    ENDCASES
  MEASURE length
```

The function $TimeAt(l, t)$ sends back the time-value of the first cell from the list l that occurs at a time smaller or equal than t.

```
TimeAt(l, tcrt) :
 RECURSIVE nonneg_real = CASES l OF
    null : 0,
    cons(O, p) : (IF Time(O) ≤ tcrt THEN Time(O) ELSE TimeAt(p, tcrt) ENDIF)
    ENDCASES
  MEASURE length(l)
```

$Timel(l)$ is the time-value of the first cell occurring in l if l is nonempty and 0 otherwise;

```
Timel(l) : nonneg_real = CASES l OF null : 0, cons(a, l) : Time(a) ENDCASES
```

6 Overview of the Proof

In this section, we will show the equivalence of the algorithms Acr and Acr1. We present only the main steps of the proof since it is impossible to present all the inferences performed in the formal proof due to they large number. Some statistics concerning the overall proof are presented in Table 1 and 2. Hence, during the proof overview, we will sometimes consolidate our argumentation by referring to a relatively small subset of the lemmas (defined in Appendix B) which we consider as *key* lemmas. They are listed in alphabetical order in Fig. 2 and informally described in Appendix A. For those that we consider to be more complex or interesting, we give hints in the same appendix. In the proofs detailed in this section, we refer to lemmas by their labels as they appear in Fig. 2.

We will employ the following notations. Assume that x is a cell and τ_2, τ_3 the two window parameters. Then $T_2(x)$ stands for $Time(x) + \tau_2$ and $T_3(x)$ for $Time(x) + \tau_3$. We will also adopt the notation $x.y$ for

leftmax : THEOREM
$\text{SortedT}(l) \wedge \tau_2 > \tau_3 \wedge \text{tcrt} \geq \text{Timel}(l) + \tau_2 \Rightarrow \text{ProgAt}(\text{Prog}(l, \tau_2, \tau_3), \text{tcrt}) = \text{Erl}(l)$

leftmax_max : THEOREM
$\text{SortedT}(l) \wedge \tau_2 > \tau_3 \wedge \text{tcrt} \geq \text{Timel}(l) + \tau_2 \Rightarrow \text{MaxEr}(\text{Wind}(l, \text{tcrt}, \tau_2, \tau_3)) = \text{Erl}(l)$

member_t_insat : LEMMA $\text{MemberT}(\text{tcrt}, \text{InsAt}(l, t, e)) \Rightarrow \text{tcrt} = t \vee \text{MemberT}(\text{tcrt}, l)$

member_t_insin : LEMMA $\text{MemberT}(\text{tcrt}, \text{InsIn}(l, t, e)) \Rightarrow \text{tcrt} = t \vee \text{MemberT}(\text{tcrt}, l)$

null_wind2 : LEMMA
$\tau_2 > \tau_3 \wedge \text{Time}(a) + \tau_3 \leq \text{tcrt} \wedge \text{null?}(\text{Wind}(l, \text{tcrt}, \tau_2, \tau_3)) \wedge \text{SortedT}(\text{cons}(a, l)) \Rightarrow \text{null?}(l)$

progat_insat1 : LEMMA $\text{SortedT}(l) \wedge t \leq \text{tcrt} \Rightarrow \text{ProgAt}(\text{InsAt}(l, t, e), \text{tcrt}) = e$

right_prog : LEMMA
$\text{SortedT}(\text{cons}(a, l)) \wedge \text{Time}(a) + \tau_3 > \text{tcrt} \wedge \tau_2 > \tau_3 \Rightarrow$
$\text{ProgAt}(\text{Prog}(\text{cons}(a, l), \tau_2, \tau_3), \text{tcrt}) = \text{ProgAt}(\text{Prog}(l, \tau_2, \tau_3), \text{tcrt})$

right_wind : LEMMA
$\text{SortedT}(\text{cons}(a, l)) \wedge \text{Time}(a) + \tau_3 > \text{tcrt} \wedge \tau_2 > \tau_3 \Rightarrow$
$\text{MaxEr}(\text{Wind}(\text{cons}(a, l), \text{tcrt}, \tau_2, \tau_3)) = \text{MaxEr}(\text{Wind}(l, \text{tcrt}, \tau_2, \tau_3))$

sorted_e_progat_prog_two_nil : LEMMA
$\text{SortedT}(l) \wedge \tau_2 > \tau_3 \Rightarrow \text{SortedE}(\text{ListUpTo}(\text{Prog}(l, \tau_2, \tau_3), \text{Timel}(l) + \tau_3))$

sorted_insat1 : LEMMA $\text{SortedT}(l) \Rightarrow \text{SortedT}(\text{InsAt}(l, t, e))$

sorted_insin2 : LEMMA $\text{SortedT}(l) \wedge \text{Timel}(l) \leq t \Rightarrow \text{SortedT}(\text{InsIn}(l, t, e))$

sorted_prog_conj : THEOREM $\text{SortedT}(l) \wedge \tau_2 > \tau_3 \Rightarrow \text{SortedT}(\text{Prog}(l, \tau_2, \tau_3))$

sorted_sorted : LEMMA $\text{SortedT}(\text{cons}(a, l)) = \text{TRUE} \Rightarrow \text{SortedT}(l) = \text{TRUE}$

timel_prog_conj1 : THEOREM
$\text{SortedT}(l) \wedge \text{Timel}(l) \leq \text{tcrt} \wedge \tau_2 > \tau_3 \wedge \text{MemberT}(\text{Time}(a), \text{Prog}(l, \tau_2, \tau_3)) \Rightarrow$
$(\text{Time}(a) \leq \text{tcrt} + \tau_2)$

Figure 2: Key lemmas

$cons(x, y)$ and ϵ for *null*. We consider *null*? as the monadic predicate satisfied when the list given as argument is empty, and *car* the selector of the first element of a non empty list. To simplify the notation, we will drop the parameters τ_2 and τ_3 from the list of arguments of the functions `Wind`, `Acr`, `Prog` and `Acr1`. Note that the model checking approaches with MEC [Arn90] and UPPAAL [BLL+96] had to assign values to these parameters in order to proceed. We will only assume all over the proof that $\tau_2 > \tau_3$ without any further mention of this hypothesis since otherwise the proof is immediate. We also omit the quantifier prefixes of the formulas since all the variables are considered as universally quantified.

6.1 Key Lemmas

The first challenging lemma that we encountered was `sorted_prog_conj`. It is itself based on 4 lemmas: `sorted_sorted`, `sorted_insat1`, `sorted_insin2` and `timel_prog_conj1`. The first three lemmas are quite straightforward. As the last one is more difficult, we present its proof after the proof of `sorted_prog_conj`.

Lemma 6.1 (sorted_prog_conj) *Given a time-decreasing cell list l, the list $Prog(l)$ is also time-decreasing.*

Proof The proof of $SortedT(l) \Rightarrow SortedT(Prog(l))$ is done by induction on the length of l: The *base case* is easy since $SortedT(Prog(\epsilon))$ is true. For the *step case*, we know that $SortedT(l) \Rightarrow SortedT(Prog(l))$ is true and we try to prove that

$$SortedT(a'.l) \Rightarrow SortedT(Prog(a'.l)).$$

The conditions of the induction hypothesis are discharged to conclude that $SortedT(Prog(l))$: we obtain $SortedT(l)$ by the instance $SortedT(a'.l) \Rightarrow SortedT(l)$ of `sorted_sorted` and the hypothesis $SortedT(a'.l)$;

We distinguish the following cases that are generated by expanding the definition of $Prog(a'.l)$ in the induction conclusion:

1. The formula

$$\begin{array}{ll} ProgAt(Prog(l), T_3(a')) \leq Er(a') & \wedge \\ SortedT(l) & \wedge \\ SortedT(Prog(l)) & \wedge \\ SortedT([a', l]) & \Rightarrow \\ SortedT(InsAt(Prog(l), T_3(a'), Er(a'))) & \end{array}$$

   is true since the conclusion is deduced from the instance $SortedT(Prog(l)) \Rightarrow SortedT(InsAt(Prog(l), T_3(a'), Er(a'))$ of the lemma `sorted_insat1`

2. The conclusion of

$$\begin{array}{ll} SortedT(l) & \wedge \\ SortedT(Prog(l)) & \wedge \\ SortedT([a', l]) & \wedge \\ ProgAt(Prog(l), T_3(a')) > Er(a') & \Rightarrow \\ SortedT(InsIn(Prog(l), T_2(a'), Er(a'))) & \end{array}$$

   is true if the conclusion of the following instance of `sorted_insin2` is true since they represent the same formula:

$$\begin{array}{ll} SortedT(Prog(l)) & \wedge \\ Timel(Prog(l)) \leq T_2(a') & \Rightarrow \\ SortedT(InsIn(Prog(l), T_2(a'), Er(a'))) & \end{array}$$

   $SortedT(Prog(l))$ is discharged. It remains to prove that $Timel(Prog(l)) \leq T_2(a')$. In order to prove it, we consider the instance of lemma `timel_prog_conj1`:

$$\begin{array}{ll} SortedT(l) & \wedge \\ Timel(l) \leq Time(a') & \wedge \\ MemberT(Time(car(Prog(l))), Prog(l)) & \Rightarrow \\ (Time(car(Prog(l))) \leq T_2(a')) & \end{array}$$

Its hypotheses are trivially discharged since for any cell list l, we have $Time(car(l)) = Timel(l)$ and $MemberT(Timel(l), l)$ is true.

□

Lemma 6.2 (timel_prog_conj1) *If the maximal time-value of a time-decreasing list l is smaller or equal than a time-value t, then the time-value of any cell from $Prog(l)$ is smaller or equal than $t + \tau_2$.*

Proof The proof of the lemma `timel_prog_conj1`

$$\begin{array}{ll} SortedT(l) & \wedge \\ Timel(l) \leq t & \wedge \\ MemberT(Time(o1), Prog(l)) & \Rightarrow \\ Time(o1) \leq t + \tau_2 & \end{array}$$

is based on the easy lemmas `sorted_sorted`, `member_t_insin` and `member_t_insat`.

We apply the induction on the cell list l. The *base case* when l is empty is immediate. For the *step case*, we suppose that the lemma is valid for l and we try to prove it for $a'.l$. From $SortedT(a'.l)$ we obtain that $Time(a') \geq Timel(l)$ and that $SortedT(l)$, by instantiating the lemma `sorted_sorted` as in the previous proof. After expanding `Prog`, it remains to prove two cases:

1. We add to the hypotheses of the formula

$$\begin{array}{ll} ProgAt(Prog(l), T_3(a')) \leq Er(a') & \wedge \\ MemberT(Time(a), InsAt(Prog(l), T_3(a'), Er(a'))) & \wedge \\ Time(a') \geq Timel(l) & \wedge \\ SortedT(l) & \wedge \\ SortedT([a', l]) & \wedge \\ Time(a') \leq t & \Rightarrow \\ MemberT(Time(a), Prog(l)) & \vee \\ Time(a) \leq \tau_2 + t & \end{array}$$

   the conclusion of the instance

$$MemberT(Time(a), InsAt(Prog(l), T_3(a'), Er(a'))) \quad \Rightarrow \quad \begin{array}{l} Time(a) = T_3(a') \vee \\ MemberT(Time(a), Prog(l)) \end{array}$$

   of the lemma `member_t_insat`. The proof of this case is derived by arithmetic reasoning.

2. Similarly with the previous case, the proof of the formula

$$\begin{array}{ll} ProgAt(Prog(l), T_3(a')) \leq Er(a') & \wedge \\ MemberT(Time(a), InsIn(Prog(l), T_2(a'), Er(a'))) & \wedge \\ Time(a') \geq Timel(l) & \wedge \\ SortedT(l) & \wedge \\ SortedT([a', l]) & \wedge \\ Time(a') \leq t & \Rightarrow \\ ProgAt(Prog(l), T_3(a')) \leq Er(a') & \vee \\ MemberT(Time(a), Prog(l)) & \vee \\ Time(a) \leq \tau_2 + t & \end{array}$$

   is trivial by adding to its hypotheses the conclusion of the instance

$$MemberT(Time(a), InsIn(Prog(l), T_2(a'), Er(a'))) \quad \Rightarrow \quad \begin{array}{l} Time(a) = T_2(a') \vee \\ MemberT(Time(a), Prog(l)) \end{array}$$

   of the lemma `member_t_insin`.

□

6.2 The Main Proof

The two functions, Acr and Acr1, are equivalent if

$$Acr1(l,t) = Acr(l,t)$$

The proof of the above conjecture is immediate if we prove the main lemma $P(l)$ where:

$$P(l): \quad SortedT(l) \Rightarrow ProgAt(Prog(l),t) = MaxEr(Wind(l,t))$$

We detail now the proof of `main_lemma` which is at the core of the proof. It is done by induction on the length of l:

a. *base case:* when l is empty, the proof is immediate.

b. *step case:* we suppose $P(l)$ and we try to prove $P(a'.l)$).

To be more specific, we should prove that $ProgAt(Prog(a'.l),t) = MaxEr(Wind(a'.l,t))$ follows from $SortedT(a'.l)$. From the hypothesis saying that $SortedT(a'.l)$ is true we deduce that $SortedT(l)$ is also true, by using the instance $SortedT(a'.l) \Rightarrow SortedT(l)$ of the lemma `sorted_sorted`. Hence, by induction hypothesis, we can assume:

$$ProgAt(Prog(l),t) = MaxEr(Wind(l,t))$$

The arguments for proving the step case are also based on the following properties:

Property 6.1 In the interval $[TimeAt(Prog(l),T_3(a'))$, $T_2(a'))$, the rate-values of $Prog(l)$ cells are strictly increasing.

Proof The property is captured by the instance $SortedT(l) \Rightarrow SortedE(ListUpTo(Prog(l),Timel(l)+\tau_3))$ of `sorted_e_progat_prog_two_nil` which states that any time-decreasing list is strictly rate-increasing on the interval $[TimeAt(Prog(l),Timel(l)+\tau_3)$, $T_2(a'))$. As $SortedT(l)$ is true, we deduce that $Prog(l)$ is strictly rate-increasing on $[TimeAt(Prog(l),Timel(l)+\tau_3)$, $T_2(a'))$. From $SortedT(a'.l)$, it results that $Time(a') \geq Timel(l)$. Hence the interval $[TimeAt(Prog(l),Time(a')+\tau_3)$, $T_2(a'))$ $Prog(l)$ is included in $[TimeAt(Prog(l),Timel(l)+\tau_3)$, $T_2(a'))$. It concludes that $ListUpTo(Prog(l),Time(a')+\tau_3)$ is also strictly rate-increasing, as a sublist of $ListUpTo(Prog(l),Timel(l)+\tau_3)$. □

Property 6.2 Let us define T_m as the instant $TimeAt(Prog(l),t)$. Then $t \geq T_m$ and no cell from the list $Prog(l)$ has a time-value in the interval $(T_m, t]$.

Proof By the definition of `TimeAt`, it results that T_m is the time-value of the first cell of $Prog(l)$ that is smaller or equal than t. □

Let us now prove the step case of the main lemma.

We will denote by $Slots(l')$ the set of time-values $\{T_2(t),T_3(t) \mid MemberT(t,l')\}$, where l' is a list of cells. In all the figures below, the time-value where the value $Er(a')$ is to be inserted in $Prog(l)$, denoted by $I_{a'}$, is a member of the set of time-values in $Slots(a'.l)$ that are located in the shadowed time domain.

We perform a case analysis according to the position of the current time t w.r.t. the values $T_2(a')$ and $T_3(a')$:

1. If $T_2(a') \leq t$ (see Fig. 3) then it means that t is located on the right-hand side of $Wind(a'.l,t)$. Since we know that $T_2(a') = Timel(a'.l) + \tau_2$ from the definition of `Timel`, and that $Erl(a'.l) = Er(a')$ from the definition of `Erl` we will use the two instances from below

   - $SortedT(a'.l) \wedge t \geq Timel(a'.l) + \tau_2 \Rightarrow ProgAt(Prog(a'.l),t) = Erl(a'.l)$ of `leftmax` which states that, for any time-decreasing list, the rate-value of the first cell in its associated scheduling list occurring at the current time is the rate-value of the first cell in the list if the current time is greater or equal than the time-value of the first cell from the list augmented with τ_2.

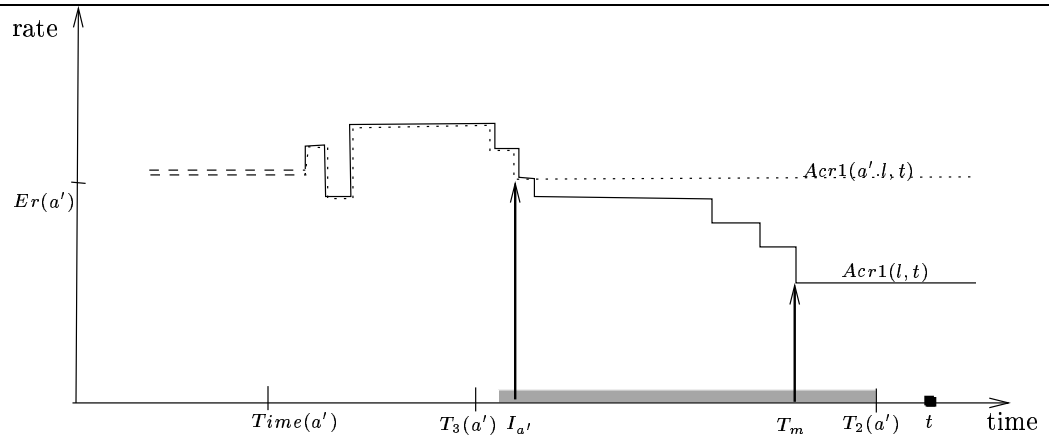


Figure 3: $T_2(a') \leq t$

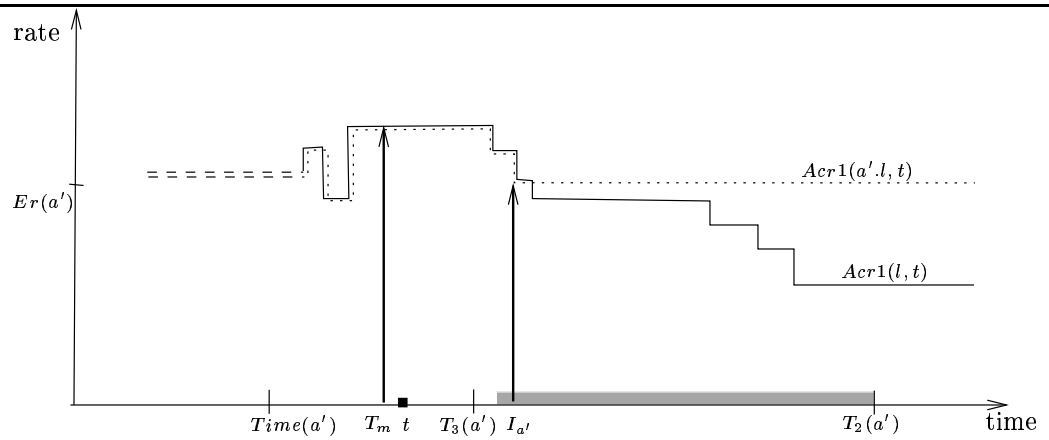


Figure 4: $T_3(a') > t$

- $SortedT(a'.l) \wedge t \geq Timel(a'.l) + \tau_2 \Rightarrow MaxEr(Wind(a'.l,t)) = Erl(a'.l)$ of `leftmax_max` which states that, for any time-decreasing list, the maximal rate-value of the cells from the window defined by in the associated scheduling and the current time is the rate-value of the first cell in the list if the current time is greater or equal than the time-value of the first cell of the list augmented with τ_2.

to show that

$$Acr1(a'.l,t) = Acr(a'.l,t)(= Er(a'))$$

2. If $T_3(a') > t$ (see Fig. 4) then

   - a' is not member of $Wind(a'.l,t)$. We obtain $Acr(a'.l,t) = Acr(l,t)$, by the instance $SortedT(a'.l) \wedge T_3(a') > t \Rightarrow MaxEr(Wind(a'.l,t)) = MaxEr(Wind(l,t))$ of `right_wind`, and
   - $Acr1(a'.l,t) = Acr1(l,t)$ from the instance $SortedT(a'.l) \wedge T_3(a') > t \Rightarrow ProgAt(Prog(a'.l),t) = ProgAt(Prog(l),t)$ of `right_prog`.

   By the induction hypothesis, $Acr1(l,t) = Acr(l,t)$. It results that $Acr1(a'.l,t) = Acr1(a'.l,t)$. Therefore the proof of this case is completed.

3. If $T_2(a') > t \geq T_3(a')$ we deduce that a' is an element of $Wind(a'.l,t)$ by definition of `Wind`.

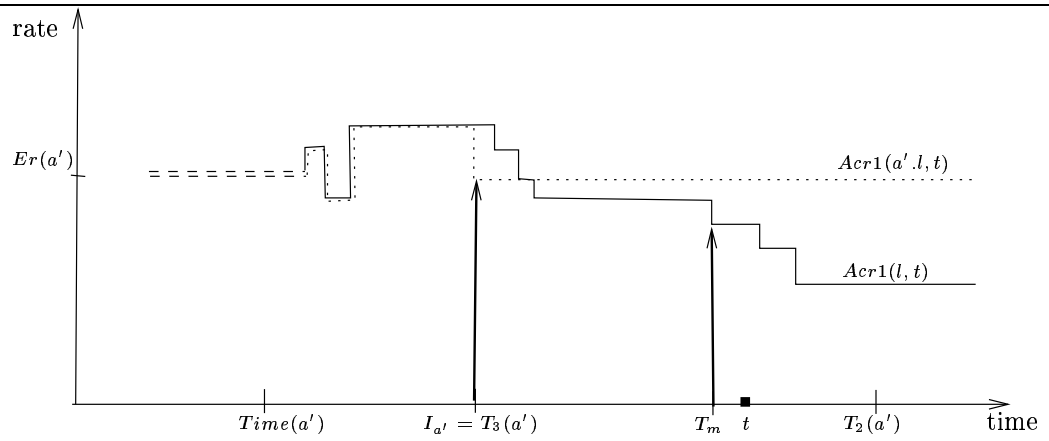


Figure 5: $T_2(a') > t \geq T_3(a')$ and $Acr1(l, T_3(a')) \leq Er(a')$

3.1. If $Wind(l,t)$ is empty then the list l is empty too. This result is captured by the instance $Time(a') + \tau_3 \leq t \wedge null?(Wind(l,t)) \wedge SortedT(a'.l) \Rightarrow null?(l)$ of the lemma `null_wind2`. It is worth to notice the necessity of the condition $SortedT(a'.l)$ for asserting that all the cells from l occur a time smaller or equal $t - \tau_3$. Therefore, by the definition of $Wind$, the first cell of l is part of the window. Since the window is empty, we deduce that l is also empty.

The rate-value $Er(a')$ is inserted in the empty list l by using an `InsAt` insertion/deletion operation because the application condition

$$Er(a') \geq Acr1(\epsilon, t)(= 0)$$

is satisfied (see the definition of `Prog`). Therefore the time-value where $Er(a')$ is inserted is $T_3(a')$. Since $t \geq T_3(a')$ we conclude that $Acr1(a'.\epsilon, t) = Er(a') = Acr(a'.\epsilon, t)$.

3.2. From now on we will analyze the case when $Wind(l,t)$ is not empty. Let ER_m be the maximal rate-value of the $Wind(l,t)$ cells. Note that it is equal to $Acr(l,t)$. According to the induction hypothesis the rate-value of the $Prog(l)$ cell situated at T_m is ER_m.

The rate-value $Er(a')$ can be inserted in the list $Prog(l)$ either by an `InsIn` or an `InsAt` operation (see the definition of `Prog`):

3.2.1. If the condition

$$Acr1(l, T_3(a')) \leq Er(a') \qquad (Cond.1)$$

is true (see Fig. 5), then we have applied an `InsAt` operation for the insertion of $Er(a')$, i.e. $Prog(a'.l) = InsAt(Prog(l), T_3(a'), Er(a'))$.

By the instance $SortedT(Prog(l)) \wedge T_3(a') \leq t \Rightarrow ProgAt(InsAt(Prog(l), T_3(a'), Er(a')), t) = Er(a')$ of `progat_insat1`, we show that $Acr1(a'.l, t) = Er(a')$ since $T_3(a') \leq t$ and $SortedT(Prog(l))$ is true. The last fact is inferred from the instance $SortedT(l) \Rightarrow SortedT(Prog(l))$ of `sorted_prog_conj`.

It remains to prove that $Acr(a'.l, t)$ also equals $Er(a')$, or, equivalently, $ER_m \leq Er(a')$.

Again from $T_3(a') \leq t$ together, this time, with the fact that on the interval $[TimeAt(Prog(l), T_3(a')),\ T_2(a'))$, the rate-values of $Prog(l)$ cells are strictly increasing according to *Property 6.1*, we deduce

$$Acr1(l,t) \leq Acr1(l, T_3(a'))$$

Therefore, by the condition $Cond.1$ and the induction hypothesis, $ER_m \leq Er(a')$.

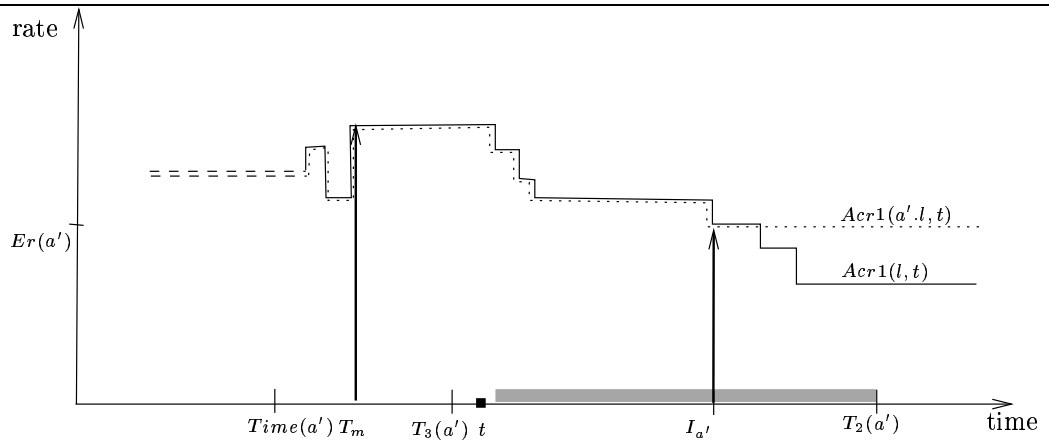


Figure 6: $T_2(a') > t \geq T_3(a')$, $Acr1(l, T_3(a')) > Er(a')$ and $T_m \leq T_3(a')$

3.2.2. If the condition

$$Acr1(l, T_3(a')) > Er(a') \qquad (Cond.2)$$

is true, then we have applied an `InsIn` operation for introducing $Er(a')$ in $Prog(l)$, with $Prog(a'.l) = InsIn(Prog(l), T_2(a'), Er(a'))$. Therefore, the insertion time for $Er(a')$ in $Prog(l)$ is $I_{a'} = Timel(InsIn(Prog(l), T_2(a'), Er(a'))$.

From *Property 6.1* and from the fact that the insertion operation starts from $T_2(a')$ ($> T_3(a')$), we obtain $I_{a'} > T_3(a')$. We distinguish the following cases:

3.2.2.1 $T_m \leq T_3(a')$ (see Fig. 6). Since $T_3(a') \leq t$, it results that $t \geq T_3(a') \geq T_m$. There is no cell in the list $Prog(l)$ whose time-value is in the interval $(T_m, t]$, according to *Property 6.2*, hence $Acr1(l, T_3(a')) = ER_m$. On the one hand, by the condition $Cond.2$, we have $ER_m > Er(a')$. Consequently, $Acr(a'.l, t) = ER_m$. On the other hand, from the facts $t \geq T_3(a') \geq T_m$ and *Property 6.2* we can deduce that no $Prog(l)$ cell is situated in the interval $(T_m, t]$. The instant $I_{a'}$ can be either $T_2(a')$ or a time-value of a $Prog(l)$ cell, according to the definition of `InsIn`. $T_2(a')$ also cannot be inside the interval $(T_m, t]$, since $T_2(a') > t$. If $I_{a'}$ corresponds to a time-value of a $Prog(l)$ cell such that $I_{a'} > T_3(a')$, we deduce $I_{a'} > t$ because there is no $Prog(l)$ cell situated in the interval $(T_m, t]$. Therefore $Acr1(a'.l, t)$ is also equal to ER_m.

3.2.2.2. $T_m > T_3(a')$ (see Fig. 7). It results that $t \geq T_m > T_3(a')$. We have the following cases to consider:

$-Er(a') \geq ER_m$. Since the rate-values of $Prog(l)$ cells are strictly increasing in the interval $[TimeAt(Prog(l), T_3(a')),\ T_2(a'))$ according to *Property 6.1*, we obtain $I_{a'} \leq T_m$. By the insertion/deletion operation, we delete all the cells of $Prog(l)$ whose rate-value is smaller than $Er(a')$, including the cell (T_m, ER_m). It results that $Acr1(a'.l, t) = Acr1(a'.l, T_m) = Er(a')$.

$-Er(a') < ER_m$. Again from *Property 6.1* we deduce that the cell obtained after the `InsIn` insertion operation and whose rate-value is $Er(a')$ is more recent than (T_m, ER_m). The instant $I_{a'}$ is greater or equal than T_m and, by the definition of `InsIn`, it can be either

- $T_2(a')$. In this case, $I_{a'} > t$; or
- a time-value of $Prog(l)$. Then $Er(a')$ will be inserted at a time greater than t, since there is no $Prog(l)$ cell in the interval $(T_m, t]$, according to *Property 6.2*.

We finally obtain that $I_{a'} > t$ in all cases and conclude that

$$Acr1(a'.l, t) = Acr1(l, t)$$

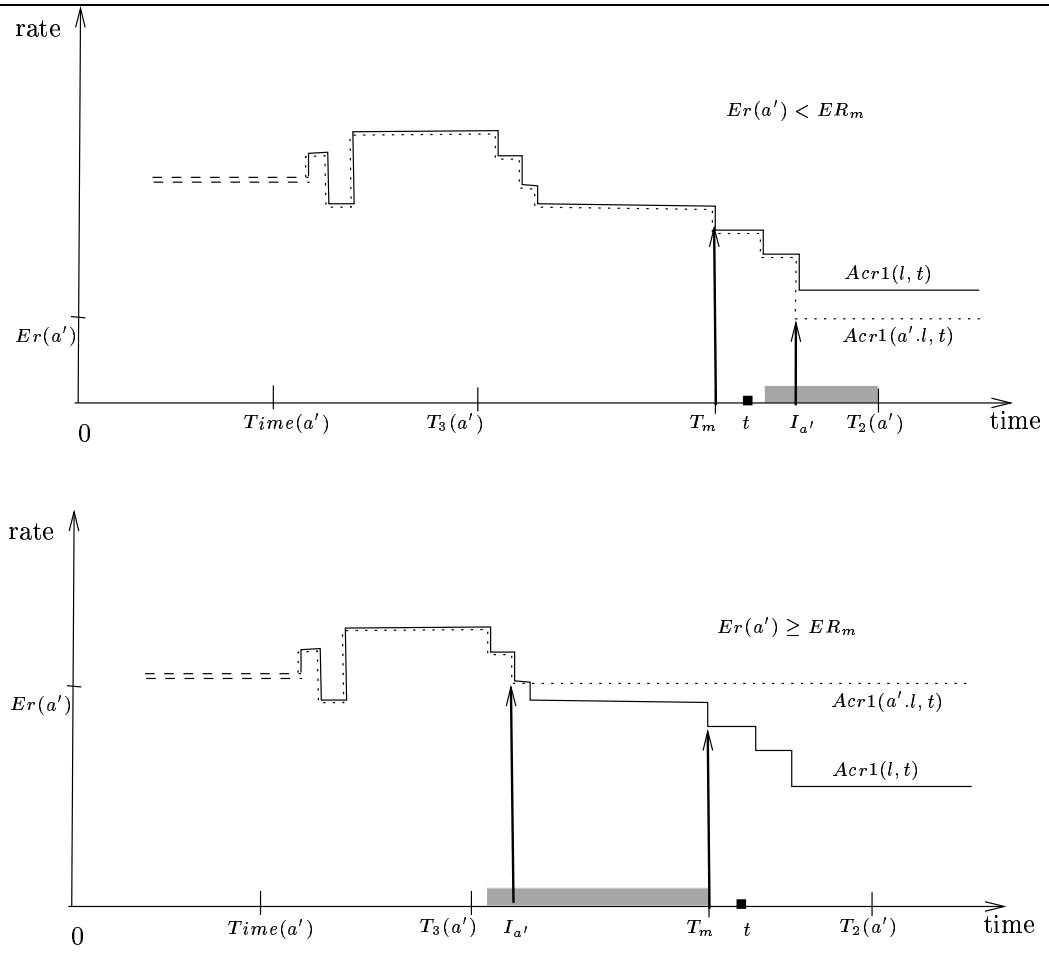


Figure 7: The two cases when $T_2(a') > t \geq T_3(a')$, $Acr1(l, T_3(a')) > Er(a')$ and $T_m > T_3(a')$

7 Comments on the Formal Proof

The first difficult step we encountered was to show that if a list of cells l is time-decreasing then its associated scheduling list $Prog(l)$ is also time-decreasing: this property is expressed by the `sorted_prog_conj` lemma. We have tried to apply induction on the length of the list but it failed because the scheduling list in the induction conclusion may be completely different w.r.t. the scheduling list in the induction hypothesis; we may insert a new cell and/or delete cells of the scheduling list used in the induction hypothesis, by the `InsIn` and `InsAt` insertion/deletion operations. Therefore, the direct application of the induction hypothesis was impossible. After a closer analysis, we discovered that the auxiliary lemmas we need concern time-value properties of cell lists. However the initial functions were not sufficient to express them. Hence an important decision was to enrich the initial specification with the auxiliary functions `MemberT` and `Timel`, described in Section 5.

The formal verification of `sorted_prog_conj` follows very closely the informal proof from Section 6. By the rules ASSERT and GRIND, the theorem prover PVS applies its decision procedures to perform arithmetic reasoning, employs congruence closure for equality reasoning, installs theories and rewrite rules along with all the definitions relevant to a certain goal in order to prove the trivial cases, simplify complex expressions and definitions, and perform matching. In detail, the formal proof requires 32 user-steps in PVS, six of which are ASSERT, five are GRIND and four are LEMMA, which introduce instances of previously proved lemmas as new formulas. The four lemmas quoted in the proof are `sorted_sorted`, `sorted_insat1`, `sorted_insin2` and `timel_prog_conj1`. Finally, `sorted_prog_conj` has been invoked nine times in the further proofs: for two times by `right_prog`, `sorted_cons_two` and `sorted_e_progat_prog`, and for one time by `timel_prog`, `progat_prog_two` and `main_conj`.

The proof of the `main_conj` lemma is the most complex that was elaborated for this specification. It follows the skeleton of the informal proof described in Section 6. We have applied an INDUCT rule to perform induction on the length of the cell list. The basic case was completed by a GRIND operation. However, the proof of the step case has needed 27 invocations of lemmas, some of them presented in Subsection A, and for six times the application of the CASE rule, to perform case reasoning. The analysis of each particular case was a source for the development of new lemmas that in turn may require other new lemmas for their proofs. The maximal depth of the lemmas dependency graph is seven. An example of a maximal path in the graph is : `final` $\rightarrow$ `main_conj` $\rightarrow$ `progat_prog_two` $\rightarrow$ `timel_prog` $\rightarrow$ `sorted_prog_conj` $\rightarrow$ `sorted_insin2` $\rightarrow$ `sorted_sorted`, where $a \rightarrow b$ means that a invokes b. The lemma `final` corresponds to the equivalence conjecture of the two algorithms, Acr and Acr1.

We have also devised the 8 remaining auxiliary functions from Section 5 to express additional properties derived from the analysis of some particular cases presented in the proof of the `main_conj` lemma. For instance, the auxiliary function `ListUpTo` introduces the notion of sublist of a list whose cells satisfy a certain (temporal) condition, while `SortedE` is necessary to test that a (sub)list is strictly rate-increasing, as we captured with the lemma `sorted_e_progat_prog_two_nil`. Some of the cases follow very closely the corresponding cases from the informal proof, as 1, 2, 3.1 and 3.2.1. The cases 3.2.2.1 and 3.2.2.2 are more complex and have required 17 lemma invocations. The final proof consists of 120 user-guided steps, 7 of which are GRIND operations and 29 are ASSERT commands which indicate that the arithmetic and equality reasoning have been intensively used.

In Table 1 and 2, we summarize some important measures that may give an idea of the size and difficulty of building proofs for the lemmas of this specification. The order of lemmas in the table is that from the PVS proof and satisfies the following constraint: if lemma a uses lemma b then b has smaller index than a in the table. The first two columns contain the current number and the name of the lemma concerned. The next five columns display the number of invoked lemmas #LEMMAS and the number of INDUCT, ASSERT, GRIND and CASE operations respectively. The last two columns show the total number of user interventions and the number of calls by the previous lemmas respectively. At the bottom line, we give the grand total for the last seven columns. The first nine lemmas represent proof obligations that must be discharged before the theory be considered typechecked. They subsume other ten proof obligations that have been generated during the typechecking process. Note they were proved in a completely automatic way.

The full machine-readable PVS specification and the PVS proofs of the lemmas are available from the authors.

#	name of lemma	#LEMMA	#INDUCT	#ASSERT	#GRIND	#CASE	#u.i.	#calls
1	MemberC_TCC1	0	0	0	0	0	0	0
2	MemberT_TCC1	0	0	0	0	0	0	0
3	MemberE_TCC1	0	0	0	0	0	0	0
4	SortedT_TCC1	0	0	0	0	0	0	0
5	SortedE_TCC1	0	0	0	0	0	0	0
6	ListUpTo_TCC1	0	0	0	0	0	0	0
7	Wind_TCC1	0	0	0	0	0	0	0
8	Wind_TCC2	0	0	0	0	0	0	0
9	MaxEr_TCC1	0	0	0	0	0	0	0
10	firstat_timeat	0	1	0	1	0	9	1
11	firstat_progat	0	1	0	1	0	9	2
12	sorted_sorted	0	1	0	2	0	4	48
13	sorted_insat1	1	1	2	2	0	11	2
14	sorted_insin2	1	1	0	2	0	6	2
15	sorted_e_two	0	1	0	1	0	6	9
16	sorted_e_insin	1	1	2	2	0	16	1
17	sorted_e_member	2	1	2	3	0	17	1
18	member_t_insin	0	1	2	4	0	19	1
19	member_t_insat	0	1	0	3	0	8	1
20	member_less	1	1	3	2	0	14	2
21	member_insin_time	2	1	4	3	0	27	1
22	member_firstat	0	1	3	1	0	16	2
23	timel_insat_t	0	1	0	2	0	5	2
24	timel_insin	2	1	0	3	0	15	7
25	erl_insin	0	1	0	2	0	5	1
26	erl_insat	0	1	0	2	0	5	1
27	erl_prog	2	1	0	3	0	12	1
28	time_progat_er	0	1	0	1	0	10	2
29	timeat_tcrt	1	1	1	1	0	11	2
30	timeat_timel	1	1	2	2	1	14	6
31	timel_timeat_max	1	1	3	2	1	15	1
32	sorted_timeat	2	1	5	2	1	23	1
33	sorted_timel_timeat	2	1	5	2	1	20	1
34	timel_prog_conj1	3	1	4	2	1	23	1
35	sorted_prog_conj	4	1	6	5	0	32	9
36	timel_prog	4	1	6	2	1	27	6
37	null_insat	0	1	1	1	0	4	1
38	null_insin	0	1	1	1	0	4	1
39	null_prog	2	1	5	2	0	20	3
40	null_listat	0	1	2	1	0	12	1
41	null_listat1	0	1	0	1	0	3	1
42	cons_insat	0	1	0	2	0	11	1
43	cons_listat	0	1	0	1	0	3	1
44	progat_two_timeat	5	1	9	2	2	41	2
45	progat_timel_erl	1	1	1	2	0	14	1

Table 1: Some statistics concerning the overall proof

#	name of lemma	#LEMMA	#INDUCT	#ASSERT	#GRIND	#CASE	#u.i.	#calls
46	progat_insat	1	1	1	3	0	16	1
47	progat_insat1	1	1	1	1	0	18	3
48	progat_insin_timeat	1	1	5	2	1	26	5
49	progat_insin	2	1	6	2	1	32	3
50	listat_insin_tcrt	2	1	6	2	1	32	1
51	progat_insin_t	1	1	6	1	0	22	1
52	listupto1_erl	0	1	2	1	0	11	1
53	null_listupto	0	1	1	2	0	11	1
54	listupto_t_insat	1	1	3	1	0	19	1
55	listupto_insin_tcrt	3	1	9	2	0	50	1
56	sorted_e_listupto	4	1	9	4	3	39	3
57	timel_listupto	0	1	0	1	0	6	1
58	sorted_listupto	2	1	5	3	0	37	1
59	progat_listupto	1	1	4	1	0	28	2
60	leftmax	3	0	2	0	0	9	1
61	leftmax_max	1	1	2	3	1	16	1
62	right_prog	14	0	14	2	1	54	1
63	right_wind	0	0	0	0	0	5	1
64	time_listat	1	1	1	1	0	12	2
65	listat_listupto	2	1	6	2	0	39	1
66	sorted_cons_listat	1	1	4	1	0	17	1
67	progat_member_time	7	1	16	3	2	49	1
68	sorted_cons_two	11	1	12	2	1	48	1
69	sorted_cons_two_nil	1	1	1	1	0	7	1
70	sorted_e_progat_prog_two	10	1	9	4	2	47	2
71	sorted_e_progat_prog_two_nil	1	1	1	1	0	7	1
72	sorted_e_member_two	6	1	11	3	2	42	1
73	new_listat_prog	4	1	10	3	1	47	1
74	progat_prog_two	17	1	23	5	5	83	1
75	null_wind1	0	1	3	1	0	10	1
76	null_wind2	1	0	1	1	1	7	1
77	member_t_timel	0	1	4	1	0	17	1
78	timeat_greater	2	1	7	2	1	28	1
79	timel_insin1	0	0	0	1	0	1	1
80	null_listupto1	0	1	0	2	0	6	1
81	sorted_e_cons	0	1	1	1	0	6	1
82	erl_cons	0	1	0	1	0	5	1
83	no_time	0	1	4	1	0	16	1
84	prev_time	3	1	10	3	1	36	1
85	monoton_e	7	1	11	5	1	55	1
86	monoton_e1	4	1	10	4	1	49	1
87	main_conj	27	1	29	7	6	120	1
88	final	1	0	1	0	0	10	0
	Total	**181**	**73**	**320**	**158**	**40**	**1686**	**181**

Table 2: Some statistics concerning the overall proof (cont.)

8 Conclusions

Our objective was to derive the first mechanical proof of the most general incremental ABR conformance algorithm. Manual proofs of the algorithm have been designed before. However the problem is intricate and it was not clear that all cases were considered and/or handled properly. In this respect our machine proof gives more evidence of the correctness of the algorithm since every single step has been verified by PVS.

The specification of the algorithm as a recursive function in PVS was relatively easy. The proof we obtained on the other hand was rather involved. It has required about 80 intermediate lemmas and the introduction of auxiliary definitions. We feel that there is a lot of space for optimization. Many steps can possibly be simplified. It is also our plan to search for a proof with more automated tools. We expect to have entirely automatic proofs for easy lemmas such as `sorted_sorted`, `member_t_insin`, `member_t_insat`, `sorted_insat1`, `sorted_insin2`, `time_progat_er` or `right_wind`. In another direction we also think about using higher-order specification and reasoning techniques to derive a proof that is more synthetic and therefore easier to grasp.

We believe that much of the methodology we have developed and the experience we have gained while proving the equivalence of the two algorithms, Acr and Acr1, can be reused for the validation of implementations of future conformance algorithms.

References

[Arn90] A. Arnold. MEC: A system for constructing and analysing transition systems. In J. Sifakis, editor, *Proceedins of the International Workshop on Automatic Verification Methods for Finite State Systems*, number 407 in LNCS, pages 117–132. Springer Verlag, 1990.

[BBC+97] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.

[BBF95a] A. Berger, F. Bonomi, and K. Fendick. The ABR conformance definition: more motivation and optimality. Technical Report 95-0481, ATM Formum Traffic Management Group, 1995.

[BBF95b] A. Berger, F. Bonomi, and K. Fendick. Proposed TM baseline text on an ABR conformance definition. Technical Report 95-0212R1, ATM Formum Traffic Management Group, 1995.

[BF99] B. Bérard and L. Fribourg. Automated verification of a parametric real-time program: the ABR conformance protocol. In *Proc. 11th Int. Conf. Computer Aided Verification (CAV'99)*, number 1633 in LNCS, pages 96–107, Trento, Italy, July 1999.

[BLL+96] J. Bengtsson, K. G. Larser, F. Larsson, P. Pettersson, and Wang Yi. UPPAAL: a tool suite for the automatic verification of real-time systems. In R. Alur, T. A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, number 1066 in LNCS, pages 232–243, 1996.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Fri98] L. Fribourg. A closed-form evaluation for extended timed automata. Technical Report LSV-98-2, Lab. Specification and Verification, ENS de Cachan, Cachan, March, France 1998. 17 pages.

[HHWT97] T.A. Henzinger, P.H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. In *CAV'97*, number 1254 in LNCS, pages 460–463. Springer-Verlag, 1997.

[Jai96] R. Jain. Congestion control and traffic management in ATM networks: Recent advances and a survey. *Computer Networks and ISDN Systems*, 28:1723–1738, 1996. ftp://ftp.netlab.ohio-state.edu/pub/jain/papers/cnis/index.html.

[MK99] J.F. Monin and F. Klay. Correctness proof of the standardized algorithm for ABR conformance. In J. Wing, J. Woodcock, and J. Davies, editors, *Formal Methods (FM) '99*, number 1709 in LNCS, pages 662–681. Springer Verlag, 1999.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer Verlag, 1992.

[Rab97] C. Rabadan. L'ABR et sa conformité. Technical report, NT DAC/ARP/034, CNET, 1997.

[RK97] C. Rabadan and F. Klay. Un nouvel algorithme de contrôle de conformité pour la capacité de transfert "Available Bit Rate". Technical Report NT/CNET/5476, CNET, 1997

A Index of Key Lemmas

The proof of each of the following lemmas usually consists in one induction step followed by several simplification operations. We expect them to be entirely automatic using automated theorem provers like *Spike*.

- `sorted_sorted` states that, if any non-empty cell list $cons(a, l)$ is time-decreasing, then l is time-decreasing, too.

sorted_sorted : LEMMA $\text{SortedT}(\text{cons}(a, l)) = \text{TRUE} \Rightarrow \text{SortedT}(l) = \text{TRUE}$

- `member_t_insin` states that any time-values of $InsIn(l, t, e)$ cells is either a time-value of an l cells or is equal to t.

member_t_insin : LEMMA $\text{MemberT}(\text{tcrt}, \text{InsIn}(l, t, e)) \Rightarrow \text{tcrt} = t \vee \text{MemberT}(\text{tcrt}, l)$

- `member_t_insat` states that any time-values of $InsAt(l, t, e)$ either a time-value of an l cells or is equal to t.

member_t_insat : LEMMA $\text{MemberT}(\text{tcrt}, \text{InsAt}(l, t, e)) \Rightarrow \text{tcrt} = t \vee \text{MemberT}(\text{tcrt}, l)$

- `sorted_insat1` states that the time-decreasing property is an invariant w.r.t. the `InsAt` insertion/deletion operation.

sorted_insat1 : LEMMA $\text{SortedT}(l) \Rightarrow \text{SortedT}(\text{InsAt}(l, t, e))$

- `sorted_insin2` states that the time-decreasing property is an invariant w.r.t. the `InsIn` insertion/deletion operation, under the condition that, given a cell to be inserted in a time-decreasing list, its time-value is greater or equal than the maximal time-value of the list.

sorted_insin2 : LEMMA $\text{SortedT}(l) \wedge \text{Timel}(l) \leq t \Rightarrow \text{SortedT}(\text{InsIn}(l, t, e))$

- `right_wind` states that, given a time-decreasing list l, the maximal rate-value of $Wind(l, tcrt)$ does not change after the insertion of a cell a in l if $T_3(a)$ is greater than $tcrt$ and τ_2 greater than τ_3.

right_wind : LEMMA
$\text{SortedT}(\text{cons}(a, l)) \wedge \text{Time}(a) + \tau_3 > \text{tcrt} \wedge \tau_2 > \tau_3 \Rightarrow$
$\text{MaxEr}(\text{Wind}(\text{cons}(a, l), \text{tcrt}, \tau_2, \tau_3)) = \text{MaxEr}(\text{Wind}(l, \text{tcrt}, \tau_2, \tau_3))$

- `progat_insat1` states that, given a list obtained from the `InsAt` insertion/deletion operation of a rate-value e, starting with the time-value t, in a time-decreasing list, the rate-value of its first cell, whose time-value is smaller than or equal to $tcrt$, equals e if $tcrt$ is greater or equal than t.

progat_insat1 : LEMMA $\text{SortedT}(l) \wedge t \leq \text{tcrt} \Rightarrow \text{ProgAt}(\text{InsAt}(l, t, e), \text{tcrt}) = e$

The lemmas presented below are considered as non-trivial and require interaction with the user for guiding the proof. We give hints for proving each of these lemmas.

- `null_wind2` states that a list l is empty if the list $Wind(l, tcrt, \tau_2, \tau_3)$ is empty, $\tau_2 > \tau_3$, $a'.l$ is time-decreasing and $tcrt \geq T_3(a')$.

null_wind2 : LEMMA
$\tau_2 > \tau_3 \wedge \text{Time}(a) + \tau_3 \leq \text{tcrt} \wedge \text{null?}(\text{Wind}(l, \text{tcrt}, \tau_2, \tau_3)) \wedge \text{SortedT}(\text{cons}(a, l)) \Rightarrow \text{null?}(l)$

The idea: if l is not empty, $Wind(l, tcrt)$ should contain a cell whose rate-value is $Erl(l)$. Contradiction with the hypothesis that it is empty.

- `sorted_e_progat_prog_two_nil` states that, on the interval $[TimeAt(Prog(l), Timel(l) + \tau_3), +\infty)$, the time-decreasing list $Prog(l)$ is strictly rate-increasing if $\tau_2 > \tau_3$.

sorted_e_progat_prog_two_nil : LEMMA
$\text{SortedT}(l) \wedge \tau_2 > \tau_3 \Rightarrow \text{SortedE}(\text{ListUpTo}(\text{Prog}(l, \tau_2, \tau_3), \text{Timel}(l) + \tau_3))$

The idea: the cells from this interval are the exclusive result of `InsIn` insertion/deletion operations. The result of any `InsIn` operation in a strictly rate-increasing list is a strictly rate-increasing list, too.

- `right_prog` states that the insertion of a rate-value $Er(a)$ in a time-decreasing list l does not change the rate-value of the first cell occuring at a time less or equal than $tcrt$ if $T_3(a)$ is greater than $tcrt$ and τ_2 greater than τ_3. $ProgAt(Prog(a'.l), tcrt) = ProgAt(Prog(l), tcrt)$ if l is a time-decreasing list, $T_3(a') > tcrt$ and $\tau_2 > \tau_3$.

right_prog : LEMMA
$\text{SortedT}(\text{cons}(a, l)) \wedge \text{Time}(a) + \tau_3 > \text{tcrt} \wedge \tau_2 > \tau_3 \Rightarrow$
$\text{ProgAt}(\text{Prog}(\text{cons}(a, l), \tau_2, \tau_3), \text{tcrt}) = \text{ProgAt}(\text{Prog}(l, \tau_2, \tau_3), \text{tcrt})$

The idea:

1. if we use an `InsAt` insertion/deletion operation, then $Er(a')$ is inserted in l at $T_3(a')$ in l. It results that, for any time-value $t < T_3(a')$ we have $ProgAt(Prog(a'.l), tcrt) = ProgAt(Prog(l), tcrt)$. Since $tcrt < T_3(a')$, the value $ProgAt(Prog(a'.l), tcrt)$ equals $ProgAt(Prog(l), tcrt)$.
2. as in the previous case, if we use an `InsIn` insertion/deletion operation, $Er(a')$ is inserted at a time-value greater than $T_3(a')$. We derive it from Property 6.1 and the application condition of the `InsIn` operation (see *Cond.*2 from Section 6.2).

- Given a time-decreasing list l, a current time-value $tcrt$ such that $tcrt \geq Timel(l) + \tau_2$, the lemma `leftmax` states that we have the equality $ProgAt(Prog(l), tcrt) = Erl(l)$.

leftmax : THEOREM
SortedT(l) $\wedge$ τ_2 $>$ τ_3 $\wedge$ tcrt $\geq$ Timel(l) $+$ τ_2 $\Rightarrow$ ProgAt(Prog(l, τ_2, τ_3), tcrt) = Erl(l)

The idea: The time-value of the first cell of $Prog(l, \tau_2, \tau_3)$ is smaller or equal than $Timel(l) + \tau_2$, hence smaller or equal than *tcrt*. We can show that the rate-values of the first cell of $Prog(l, \tau_2, \tau_3)$ and l coincide.

- Given a time-decreasing list l, a current time-value *tcrt* such that $tcrt \geq Timel(l) + \tau_2$, the lemma `leftmax_max` states that we have the equality $Maxer(Wind(l, tcrt, \tau_2, \tau_3)) = Erl(l)$.

leftmax_max : THEOREM
SortedT(l) $\wedge$ τ_2 $>$ τ_3 $\wedge$ tcrt $\geq$ Timel(l) $+$ τ_2 $\Rightarrow$ MaxEr(Wind(l, tcrt, τ_2, τ_3)) = Erl(l)

The idea: By case reasoning, the proof is trivial when l is empty. Otherwise, the unique element of $Wind(l, tcrt, \tau_2, \tau_3)$ is $car(l)$.

The proof of the lemma `sorted_prog_conj` and `timel_prog_conj1` are fully described in Section 6.

B The *PVS* Code of *ABR* Algorithm

```
atm : Theory
  BEGIN

    tuptype : TYPE = [nonneg_real, nonneg_real]

    tcrt, t2, t3, t, to, e, e1, ta, tb : VAR nonneg_real

    l, p : VAR list[tuptype]

    O, o1, oa : VAR tuptype

    Time(t, e) : nonneg_real = t

    Er(t, e) : nonneg_real = e

    Timel(l) : nonneg_real = CASES l OF null : 0, cons(O, p) : Time(O) ENDCASES

    Erl(l) : nonneg_real = CASES l OF null : 0, cons(O, p) : Er(O) ENDCASES

    MemberC(O, l) :
     RECURSIVE bool = CASES l OF
        null : FALSE,
        cons(o1, p1) : (IF O = o1 THEN TRUE ELSE MemberC(O, p1) ENDIF)
        ENDCASES
      MEASURE length(l)

    MemberT(tcrt, l) :
     RECURSIVE bool = CASES l OF
        null : FALSE,
        cons(o1, p1) : (IF tcrt = Time(o1) THEN TRUE ELSE MemberT(tcrt, p1) ENDIF)
        ENDCASES
      MEASURE length(l)

    MemberE(e, l) :
     RECURSIVE bool = CASES l OF
        null : FALSE,
        cons(o1, p1) : (IF e = Er(o1) THEN TRUE ELSE MemberE(e, p1) ENDIF)
```

```
    ENDCASES
  MEASURE length(l)

SortedT(l) :
 RECURSIVE bool = CASES l OF
    null : TRUE,
    cons(o1, p1) :
      (CASES p1 OF
         null : TRUE,
         cons(o2, p2) :
           IF Time(o1) ≥ Time(o2) THEN SortedT(cons(o2, p2))
            ELSE FALSE
            ENDIF
       ENDCASES)
    ENDCASES
  MEASURE length

SortedE(l) :
 RECURSIVE bool = CASES l OF
    null : TRUE,
    cons(o1, p1) :
      (CASES p1 OF
         null : TRUE,
         cons(o2, p2) :
           IF Er(o1) < Er(o2) THEN SortedE(cons(o2, p2))
            ELSE FALSE
            ENDIF
       ENDCASES)
    ENDCASES
  MEASURE length

ListUpTo(l, t) :
 RECURSIVE list[tuptype] = CASES l OF
    null : null,
    cons(O, p) :
      (IF Time(O) ≤ t THEN cons(O, null)
        ELSE cons(O, ListUpTo(p, t))
        ENDIF)
    ENDCASES
  MEASURE length(l)

Wind(l, tcrt, t2, t3) :
 RECURSIVE list[tuptype] = CASES l OF
    null : null,
    cons(O, p) :
      (IF Time(O) + t3 > tcrt THEN Wind(p, tcrt, t2, t3)
        ELSIF Time(O) + t2 ≤ tcrt THEN cons(O, null)
        ELSE cons(O, Wind(p, tcrt, t2, t3))
        ENDIF)
    ENDCASES
  MEASURE length(l)

MaxEr(l) :
 RECURSIVE nonneg_real = CASES l OF
    null : 0,
    cons(O, p) : (IF MaxEr(p) ≤ Er(O) THEN Er(O) ELSE MaxEr(p) ENDIF)
    ENDCASES
  MEASURE length

Acr(l, tcrt, t2, t3) : nonneg_real = IF SortedT(l) ∧ (t2 > t3) THEN MaxEr(Wind(l, tcrt, t2, t3)) ELSE 0 ENDIF

InsAt(l, t, e) :
```

```
    RECURSIVE list[tuptype] = CASES l OF
      null : cons((t, e), null),
      cons(O, p) :
        (IF Time(O) ≤ t THEN cons((t, e), cons(O, p))
         ELSE InsAt(p, t, e)
         ENDIF)
      ENDCASES
    MEASURE length(l)

  InsIn(l, t, e) :
    RECURSIVE list[tuptype] = CASES l OF
      null : cons((t, e), null),
      cons(O, p) :
        (IF Er(O) ≤ e THEN InsIn(p, Time(O), e)
         ELSE cons((t, e), cons(O, p))
         ENDIF)
      ENDCASES
    MEASURE length(l)

  ProgAt(l, tcrt) :
    RECURSIVE nonneg_real = CASES l OF
      null : 0,
      cons(O, p) : (IF Time(O) ≤ tcrt THEN Er(O) ELSE ProgAt(p, tcrt) ENDIF)
      ENDCASES
    MEASURE length(l)

  TimeAt(l, tcrt) :
    RECURSIVE nonneg_real = CASES l OF
      null : 0,
      cons(O, p) : (IF Time(O) ≤ tcrt THEN Time(O) ELSE TimeAt(p, tcrt) ENDIF)
      ENDCASES
    MEASURE length(l)

  FirstAt(l, tcrt) :
    RECURSIVE tuptype = CASES l OF
      null : (0, 0),
      cons(O, p) : (IF Time(O) ≤ tcrt THEN O ELSE FirstAt(p, tcrt) ENDIF)
      ENDCASES
    MEASURE length(l)

  ListAt(l, tcrt) :
    RECURSIVE list[tuptype] = CASES l OF
      null : null,
      cons(O, p) :
        (IF Time(O) ≤ tcrt THEN cons(O, p) ELSE ListAt(p, tcrt) ENDIF)
      ENDCASES
    MEASURE length(l)

  Prog(l, t2, t3) :
    RECURSIVE list[tuptype] = CASES l OF
      null : null,
      cons(O, p) :
        (IF ProgAt(Prog(p, t2, t3), (Time(O) + t3)) ≤ Er(O)
           THEN InsAt(Prog(p, t2, t3), (Time(O) + t3), Er(O))
         ELSE InsIn(Prog(p, t2, t3), (Time(O) + t2), Er(O))
         ENDIF)
      ENDCASES
    MEASURE length(l)

  Acr1(l, tcrt, t2, t3) : nonneg_real = IF SortedT(l) ∧ (t2 > t3) THEN ProgAt(Prog(l, t2, t3), tcrt) ELSE 0 ENDIF

  firstat_timeat : THEOREM Time(FirstAt(l, t)) = TimeAt(l, t)
```

firstat_progat : THEOREM $\mathrm{Er}(\mathrm{FirstAt}(l,t)) = \mathrm{ProgAt}(l,t)$

sorted_sorted : THEOREM $\mathrm{SortedT}(\mathrm{cons}(O,p)) = \mathrm{TRUE} \Rightarrow \mathrm{SortedT}(p) = \mathrm{TRUE}$

sorted_insat1 : THEOREM $\mathrm{SortedT}(l) \Rightarrow \mathrm{SortedT}(\mathrm{InsAt}(l,t,e))$

sorted_insin2 : THEOREM $\mathrm{SortedT}(l) \wedge \mathrm{Timel}(l) \leq t \Rightarrow \mathrm{SortedT}(\mathrm{InsIn}(l,t,e))$

sorted_e_two : THEOREM $\mathrm{SortedE}(\mathrm{cons}(O,p)) \Rightarrow \mathrm{SortedE}(p)$

sorted_e_insin : THEOREM $\mathrm{SortedE}(l) \Rightarrow \mathrm{SortedE}(\mathrm{InsIn}(l,t,e))$

sorted_e_member : THEOREM $\mathrm{SortedE}(l) \wedge \mathrm{MemberC}(O,l) \Rightarrow \mathrm{Erl}(l) \leq \mathrm{Er}(O) \vee \mathrm{null?}(l)$

member_t_insin : THEOREM $\mathrm{MemberT}(\mathrm{tcrt}, \mathrm{InsIn}(l,t,e)) \Rightarrow \mathrm{tcrt} = t \vee \mathrm{MemberT}(\mathrm{tcrt}, l)$

member_t_insat : THEOREM $\mathrm{MemberT}(\mathrm{tcrt}, \mathrm{InsAt}(l,t,e)) \Rightarrow \mathrm{tcrt} = t \vee \mathrm{MemberT}(\mathrm{tcrt}, l)$

member_less : THEOREM $\mathrm{SortedT}(l) \wedge \mathrm{MemberC}(O,l) \Rightarrow \mathrm{Time}(O) \leq \mathrm{Timel}(l)$

member_insin_time : THEOREM
$\mathrm{SortedT}(l) \wedge \mathrm{MemberC}(o_1,l) \wedge e < \mathrm{Er}(o_1) \wedge t \geq \mathrm{Timel}(l) \Rightarrow$
$\mathrm{Timel}(\mathrm{InsIn}(l,t,e)) \geq \mathrm{Time}(o_1)$

member_firstat : THEOREM $\mathrm{cons?}(\mathrm{ListAt}(l,\mathrm{tcrt})) \Rightarrow \mathrm{MemberC}(\mathrm{FirstAt}(l,\mathrm{tcrt}),l)$

timel_insat_t : THEOREM $\mathrm{Timel}(\mathrm{InsAt}(l,t,e)) = t$

timel_insin : THEOREM $\mathrm{SortedT}(l) \wedge \mathrm{Timel}(l) \leq \mathrm{tcrt} \Rightarrow \mathrm{Timel}(\mathrm{InsIn}(l,\mathrm{tcrt},e)) \leq \mathrm{tcrt}$

erl_insin : THEOREM $\mathrm{Erl}(\mathrm{InsIn}(l,t,e)) = e$

erl_insat : THEOREM $\mathrm{Erl}(\mathrm{InsAt}(l,t,e)) = e$

erl_prog : THEOREM $\mathrm{Erl}(\mathrm{Prog}(l,t_2,t_3)) = \mathrm{Erl}(l)$

time_progat_er : THEOREM $\mathrm{Timel}(l) \leq \mathrm{tcrt} \Rightarrow \mathrm{ProgAt}(l,\mathrm{tcrt}) = \mathrm{Erl}(l)$

timeat_tcrt : THEOREM $\mathrm{SortedT}(l) \Rightarrow \mathrm{TimeAt}(l,\mathrm{tcrt}) \leq \mathrm{tcrt}$

timeat_timel : THEOREM $\mathrm{SortedT}(l) \Rightarrow \mathrm{TimeAt}(l,\mathrm{tcrt}) \leq \mathrm{Timel}(l)$

timel_timeat_max : THEOREM $\mathrm{SortedT}(l) \wedge t \geq \mathrm{Timel}(l) \Rightarrow \mathrm{TimeAt}(l,t) = \mathrm{Timel}(l)$

sorted_timeat : THEOREM $\mathrm{SortedT}(l) \wedge t_2 \geq t_3 \Rightarrow \mathrm{TimeAt}(l,t_2) \geq \mathrm{TimeAt}(l,t_3)$

sorted_timel_timeat : THEOREM $\mathrm{SortedT}(l) \wedge \mathrm{Timel}(l) \leq \mathrm{TimeAt}(l,t) \Rightarrow t \geq \mathrm{Timel}(l)$

timel_prog_conj1 : THEOREM
$\mathrm{SortedT}(l) \wedge \mathrm{Timel}(l) \leq \mathrm{tcrt} \wedge t_2 > t_3 \wedge \mathrm{MemberT}(\mathrm{Time}(o_1), \mathrm{Prog}(l,t_2,t_3)) \Rightarrow$
$(\mathrm{Time}(o_1) \leq \mathrm{tcrt} + t_2)$

sorted_prog_conj : THEOREM $\mathrm{SortedT}(l) \wedge t_2 > t_3 \Rightarrow \mathrm{SortedT}(\mathrm{Prog}(l,t_2,t_3))$

timel_prog : THEOREM $\mathrm{SortedT}(l) \wedge t_2 > t_3 \Rightarrow (\mathrm{Timel}(\mathrm{Prog}(l,t_2,t_3)) \leq \mathrm{Timel}(l) + t_2)$

null_insat : THEOREM $\mathrm{null?}(\mathrm{InsAt}(l,t,e)) \Rightarrow \mathrm{null?}(l)$

null_insin : THEOREM $\mathrm{null?}(\mathrm{InsIn}(l,t,e)) \Rightarrow \mathrm{null?}(l)$

null_prog : THEOREM $\mathrm{null?}(\mathrm{Prog}(l,t_2,t_3)) \Rightarrow \mathrm{null?}(l)$

null_listat : THEOREM null?(ListAt(l,t)) $\Rightarrow$ ProgAt$(l,t) = 0$

null_listat1 : THEOREM null?(l) $\Rightarrow$ null?(ListAt(l,t))

cons_insat : THEOREM cons?(InsAt(l,t,e))

cons_listat : THEOREM cons?(ListAt(l,t)) $\Rightarrow$ cons?(l)

progat_two_timeat : THEOREM SortedT(l) $\Rightarrow$ ProgAt$(l,$ TimeAt$(l,t)) =$ ProgAt(l,t)

progat_timel_erl : THEOREM SortedT(l) $\Rightarrow$ ProgAt$(l,$ Timel$(l)) =$ Erl(l)

progat_insat : THEOREM SortedT$(l) \wedge t >$ tcrt $\Rightarrow$ ProgAt(InsAt(l,t,e), tcrt) = ProgAt$(l,$ tcrt$)$

progat_insat1 : THEOREM SortedT$(l) \wedge t \leq$ tcrt $\Rightarrow$ ProgAt(InsAt(l,t,e), tcrt$) = e$

progat_insin_timeat : THEOREM
SortedT$(l) \wedge$ ProgAt$(l,t) > e \wedge$ to $\geq$ Timel$(l) \wedge t <$ to $\Rightarrow$ Timel(InsIn$(l,$ to$,e)) > t$

progat_insin : THEOREM
SortedT$(l) \wedge$ Timel(InsIn$(l,t,e)) >$ tcrt $\Rightarrow$ ProgAt(InsIn(l,t,e), tcrt$) =$ ProgAt$(l,$ tcrt$)$

listat_insin_tcrt : THEOREM
SortedT$(l) \wedge$ Timel(InsIn$(l,t,e)) >$ tcrt $\Rightarrow$ ListAt(InsIn(l,t,e), tcrt$) =$ ListAt$(l,$ tcrt$)$

progat_insin_t : THEOREM
SortedT$(l) \wedge$ Timel(InsIn$(l,t,e)) \leq$ tcrt $\Rightarrow$ ProgAt(InsIn(l,t,e), tcrt$) = e$

listupto1_erl : THEOREM
SortedT$(l) \wedge t \leq$ to $\Rightarrow$
Erl(ListUpTo$(l,t)) =$ Erl(ListUpTo$(l,$ to$)) \vee$
null?(ListUpTo$(l,$ to$)) \vee$ null?(ListUpTo(l,t))

null_listupto : THEOREM $t \leq$ to $\wedge$ SortedT$(l) \wedge$ null?(ListUpTo$(l,t)) \Rightarrow$ null?(ListUpTo$(l,$ to$)$)

listupto_t_insat : THEOREM SortedT$(l) \Rightarrow$ ListUpTo(InsAt$(l,t,e),t) =$ cons$((t,e),$ null$)$

listupto_insin_tcrt : THEOREM
SortedT$(l) \wedge$ tcrt $<$ Timel(InsIn$(l,t,e)) \Rightarrow$
ListUpTo(InsIn(l,t,e), tcrt$) =$ InsIn(ListUpTo$(l,$ tcrt$),t,e)$

sorted_e_listupto : THEOREM
SortedT$(l) \wedge t \leq$ to $\wedge$ SortedE(ListUpTo$(l,t)) \Rightarrow$ SortedE(ListUpTo$(l,$ to$)$)

timel_listupto : THEOREM Timel(ListUpTo$(l,t)) =$ Timel(l)

sorted_listupto : THEOREM SortedT$(l) \Rightarrow$ SortedT(ListUpTo(l,t))

progat_listupto : THEOREM SortedT$(l) \wedge$ ta $\geq$ tb $\Rightarrow$ ProgAt(ListUpTo$(l,$ tb$),$ ta$) =$ ProgAt$(l,$ ta$)$

leftmax : THEOREM
SortedT$(l) \wedge t_2 > t_3 \wedge$ tcrt $\geq$ Timel$(l) + t_2 \Rightarrow$ ProgAt(Prog(l,t_2,t_3), tcrt$) =$ Erl(l)

leftmax_max : THEOREM
SortedT$(l) \wedge t_2 > t_3 \wedge$ tcrt $\geq$ Timel$(l) + t_2 \Rightarrow$ MaxEr(Wind$(l,$ tcrt$,t_2,t_3)) =$ Erl(l)

right_prog : THEOREM
SortedT(cons$(o_1,p)) \wedge$ Time$(o_1) + t_3 >$ tcrt $\wedge t_2 > t_3 \Rightarrow$
ProgAt(Prog(cons$(o_1,p),t_2,t_3)$, tcrt$) =$ ProgAt(Prog(p,t_2,t_3), tcrt$)$

right_wind : THEOREM

$\text{SortedT}(\text{cons}(o_1, p)) \wedge \text{Time}(o_1) + t_3 > \text{tcrt} \wedge t_2 > t_3 \Rightarrow$
$\text{MaxEr}(\text{Wind}(\text{cons}(o_1, p), \text{tcrt}, t_2, t_3)) = \text{MaxEr}(\text{Wind}(p, \text{tcrt}, t_2, t_3))$

time_listat : THEOREM $\text{SortedT}(l) \wedge t \geq \text{Timel}(l) \Rightarrow \text{ListAt}(l, t) = l$

listat_listupto : THEOREM
$\text{SortedT}(l) \wedge \text{ta} \leq \text{tb} \wedge \text{cons?}(\text{ListAt}(l, \text{ta})) \Rightarrow \text{cons?}(\text{ListAt}(\text{ListUpTo}(l, \text{ta}), \text{tb}))$

sorted_cons_listat : THEOREM $\text{SortedT}(l) \wedge \text{cons?}(\text{ListAt}(l, t_3)) \wedge t_2 \geq t_3 \Rightarrow \text{cons?}(\text{ListAt}(l, t_2))$

progat_member_time : THEOREM
$\text{cons?}(\text{ListAt}(l, t)) \wedge$
$\text{ProgAt}(l, t) = 0 \wedge$
$\text{SortedT}(l) \wedge \text{SortedE}(l) \wedge \text{MemberC}(O, l) \wedge \text{Time}(O) > t \Rightarrow$
$\text{Er}(O) = 0$

sorted_cons_two : THEOREM
$\text{SortedT}(\text{cons}(o_1, p)) \wedge t_2 > t_3 \Rightarrow \text{cons?}(\text{ListAt}(\text{Prog}(\text{cons}(o_1, p), t_2, t_3), \text{Time}(o_1) + t_3))$

sorted_cons_two_nil : THEOREM
$\text{SortedT}(l) \wedge t_2 > t_3 \Rightarrow \text{cons?}(\text{ListAt}(\text{Prog}(l, t_2, t_3), \text{Timel}(l) + t_3)) \vee \text{null?}(l)$

sorted_e_progat_prog_two : THEOREM
$\text{SortedT}(\text{cons}(o_1, p)) \wedge t_2 > t_3 \Rightarrow \text{SortedE}(\text{ListUpTo}(\text{Prog}(\text{cons}(o_1, p), t_2, t_3), \text{Time}(o_1) + t_3))$

sorted_e_progat_prog_two_nil : THEOREM
$\text{SortedT}(l) \wedge t_2 > t_3 \Rightarrow \text{SortedE}(\text{ListUpTo}(\text{Prog}(l, t_2, t_3), \text{Timel}(l) + t_3)) \vee \text{null?}(l)$

sorted_e_member_two : THEOREM
$\text{SortedT}(l) \wedge \text{SortedE}(l) \wedge t_2 \geq t_3 \Rightarrow$
$\text{ProgAt}(l, t_2) \leq \text{ProgAt}(l, t_3) \vee \text{ProgAt}(l, t_3) = 0$

new_listat_prog : THEOREM
$\text{SortedT}(l) \wedge \text{SortedE}(l) \wedge \text{cons?}(\text{ListAt}(l, t)) \wedge \text{ProgAt}(l, t) \leq e \wedge t \leq \text{tcrt} \Rightarrow$
$\text{ProgAt}(l, \text{tcrt}) \leq e$

progat_prog_two : THEOREM
$\text{SortedT}(\text{cons}(o_1, p)) \wedge$
$t_2 > t_3 \wedge$
$\text{ProgAt}(\text{Prog}(p, t_2, t_3), \text{Time}(o_1) + t_3) \leq \text{Er}(o_1) \wedge$
$\text{tcrt} \geq \text{Time}(o_1) + t_3 \Rightarrow$
$\text{ProgAt}(\text{Prog}(p, t_2, t_3), \text{tcrt}) \leq \text{Er}(o_1)$

null_wind1 : THEOREM $t_2 > t_3 \wedge \text{Timel}(l) + t_3 \leq \text{tcrt} \wedge \text{null?}(\text{Wind}(l, \text{tcrt}, t_2, t_3)) \Rightarrow \text{null?}(l)$

null_wind2 : THEOREM
$t_2 > t_3 \wedge \text{Time}(o_1) + t_3 \leq \text{tcrt} \wedge \text{null?}(\text{Wind}(l, \text{tcrt}, t_2, t_3)) \wedge \text{SortedT}(\text{cons}(o_1, l)) \Rightarrow$
$\text{null?}(l)$

member_t_timel : THEOREM $\text{MemberT}(\text{Timel}(\text{InsIn}(l, t, e)), l) \vee \text{Timel}(\text{InsIn}(l, t, e)) = t$

timeat_greater : THEOREM $\text{SortedT}(l) \wedge \text{MemberT}(t, l) \wedge t > \text{TimeAt}(l, \text{tcrt}) \Rightarrow t > \text{tcrt}$

timel_insin1 : THEOREM $e < \text{Erl}(l) \Rightarrow \text{Timel}(\text{InsIn}(l, t, e)) = t$

null_listupto1 : THEOREM $\text{null?}(\text{ListUpTo}(l, t)) \Rightarrow \text{null?}(l)$

sorted_e_cons : THEOREM $\text{SortedE}(\text{cons}(O, l)) \Rightarrow \text{Er}(O) < \text{Erl}(l) \vee \text{null?}(l)$

erl_cons : THEOREM $\text{Erl}(\text{ListUpTo}(l, t)) = \text{Erl}(l)$

no_time : THEOREM $\text{ProgAt}(l, t) = e \wedge t \geq \text{ta} \wedge \text{ta} \geq \text{TimeAt}(l, t) \Rightarrow \text{ProgAt}(l, \text{ta}) = e$

prev_time : THEOREM
SortedT(l)∧
Timel(InsIn(l, t_2, e)) > t_3∧
$t_2 \geq$ Timel(l)∧
t_2 > tcrt ∧ tcrt $\geq t_3 \wedge t_3 \geq$ TimeAt(l, tcrt) ⇒
Timel(InsIn(l, t_2, e)) > tcrt ∨ null?(l)

monoton_e : THEOREM
SortedT(l)∧
SortedE(ListUpTo(l, t_3))∧
t_2 > tcrt∧
tcrt $\geq t_3$∧
t_3 < TimeAt(l, tcrt)∧
$e \geq$ ProgAt(l, tcrt) $\wedge\ t_2 \geq$ Timel(l) ⇒
Timel(InsIn(l, t_2, e)) $\leq$ TimeAt(l, tcrt) ∨ null?(l)

monoton_e1 : THEOREM
SortedT(l)∧
SortedE(ListUpTo(l, t_3))∧
t_2 > tcrt∧
tcrt $\geq t_3$∧
t_3 < TimeAt(l, tcrt)∧
e < ProgAt(l, tcrt) $\wedge\ t_2 \geq$ Timel(l) ⇒
Timel(InsIn(l, t_2, e)) > TimeAt(l, tcrt) ∨ null?(l)

main_conj : THEOREM SortedT(l) $\wedge\ t_2 > t_3 \Rightarrow$ ProgAt(Prog(l, t_2, t_3), tcrt) = MaxEr(Wind(l, tcrt, t_2, t_3))

final : THEOREM Acr(l, tcrt, t_2, t_3) = Acr1(l, tcrt, t_2, t_3)

END atm

C The PVS Proof of `sorted_prog_conj`

Verbose proof for `sorted_prog_conj`.

`sorted_prog_conj:`

$$\{1\}\quad (\forall\ (l:\ \text{list[cell]}, t_2:\ \text{nat}, t_3:\ \text{nat}):\ \text{SortedT}(l) \wedge t_2\ >\ t_3 \Rightarrow \text{SortedT}(\text{Prog}(l, t_2, t_3)))$$

`sorted_prog_conj:`

$$\{1\}\quad (\forall\ (l:\ \text{list[cell]}, t_2:\ \text{nat}, t_3:\ \text{nat}):\ \text{SortedT}(l) \wedge t_2\ >\ t_3 \Rightarrow \text{SortedT}(\text{Prog}(l, t_2, t_3)))$$

Inducting on l

,

we get 2 subgoals:

`sorted_prog_conj.1:`

$$\{1\}\quad (\forall\ (t_2:\ \text{nat}, t_3:\ \text{nat}):\ \text{SortedT}(\text{null}) \wedge t_2\ >\ t_3 \Rightarrow \text{SortedT}(\text{Prog}(\text{null}, t_2, t_3)))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `sorted_prog_conj.1`.

`sorted_prog_conj.2:`

$$\{1\}\quad (\forall\ (\text{cons1_var}:\ \text{cell}, \text{cons2_var}:\ \text{list[cell]}):\ (\forall\ (t_2:\ \text{nat}, t_3:\ \text{nat}):\ \text{SortedT}(\text{cons2_var}) \wedge t_2\ >\ t_3 \Rightarrow \text{SortedT}(\text{Prog}(\text{cons2_var}, t_2, t_3))) \supset (\forall\ (t_2:\ \text{nat}, t_3:\ \text{nat}):\ \text{SortedT}(\text{cons}(\text{cons1_var}, \text{cons2_var})) \wedge t_2\ >\ t_3 \Rightarrow \text{SortedT}(\text{Prog}(\text{cons}(\text{cons1_var}, \text{cons2_var}), t_2, t_3))))$$

Repeatedly Skolemizing and flattening,

`sorted_prog_conj.2:`

$$\begin{array}{ll}\{\text{-}1\} & (\forall\ (t_2:\ \text{nat}, t_3:\ \text{nat}):\ \text{SortedT}(\text{cons2_var}') \wedge t_2\ >\ t_3 \Rightarrow \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2, t_3)))\\ \{\text{-}2\} & \text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))\\ \{\text{-}3\} & t_2' > t_3'\\ \hline \{1\} & \text{SortedT}(\text{Prog}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), t_2', t_3'))\end{array}$$

Expanding the definition of Prog

,

`sorted_prog_conj.2:`

$\{-1\}$ $(\forall\ (t_2 :\ \text{nat}, t_3 :\ \text{nat}) :$
$\quad \text{SortedT}(\text{cons2_var}') \wedge t_2\ >\ t_3 \Rightarrow$
$\quad\ \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2, t_3)))$
$\{-2\}$ $\text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$
$\{-3\}$ $t_2'\ >\ t_3'$

$\{1\}$ IF
$\quad \text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}')\ +\ t_3')) \leq$
$\quad\ \text{Er}(\text{cons1_var}')$
$\quad$ THEN
$\quad \text{SortedT}(\text{InsAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'),$
$\qquad\qquad (\text{Time}(\text{cons1_var}')\ +\ t_3'), \text{Er}(\text{cons1_var}')))$
$\quad$ ELSE
$\quad \text{SortedT}(\text{InsIn}(\text{Prog}(\text{cons2_var}', t_2', t_3'),$
$\qquad\qquad (\text{Time}(\text{cons1_var}')\ +\ t_2'), \text{Er}(\text{cons1_var}')))$
$\quad$ ENDIF

Applying sorted_sorted

`sorted_prog_conj.2:`

$\{-1\}$ $(\forall\ (O :\ \text{cell}, p :\ \text{list[cell]}) :$
$\quad \text{SortedT}(\text{cons}(O, p)) = \text{TRUE} \Rightarrow \text{SortedT}(p) = \text{TRUE})$
$\{-2\}$ $(\forall\ (t_2 :\ \text{nat}, t_3 :\ \text{nat}) :$
$\quad \text{SortedT}(\text{cons2_var}') \wedge t_2\ >\ t_3 \Rightarrow$
$\quad\ \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2, t_3)))$
$\{-3\}$ $\text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$
$\{-4\}$ $t_2'\ >\ t_3'$

$\{1\}$ IF
$\quad \text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}')\ +\ t_3')) \leq$
$\quad\ \text{Er}(\text{cons1_var}')$
$\quad$ THEN
$\quad \text{SortedT}(\text{InsAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'),$
$\qquad\qquad (\text{Time}(\text{cons1_var}')\ +\ t_3'), \text{Er}(\text{cons1_var}')))$
$\quad$ ELSE
$\quad \text{SortedT}(\text{InsIn}(\text{Prog}(\text{cons2_var}', t_2', t_3'),$
$\qquad\qquad (\text{Time}(\text{cons1_var}')\ +\ t_2'), \text{Er}(\text{cons1_var}')))$
$\quad$ ENDIF

Instantiating the top quantifier in -1 with the terms: (cons1_var!1
cons2_var!1
),

`sorted_prog_conj.2:`

$$
\begin{array}{|ll}
\{\text{-}1\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) = \mathrm{TRUE} \Rightarrow \\
& \quad \mathrm{SortedT}(\mathrm{cons2_var}') = \mathrm{TRUE} \\
\{\text{-}2\} & (\forall\ (t_2 :\ \mathrm{nat}, t_3 :\ \mathrm{nat}) : \\
& \quad \mathrm{SortedT}(\mathrm{cons2_var}') \wedge t_2\ >\ t_3 \Rightarrow \\
& \quad\ \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t_2, t_3))) \\
\{\text{-}3\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\{\text{-}4\} & t'_2\ >\ t'_3 \\
\hline
\{1\} & \mathrm{IF} \\
& \quad \mathrm{ProgAt}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), (\mathrm{Time}(\mathrm{cons1_var}')\ +\ t'_3)) \leq \\
& \quad\ \mathrm{Er}(\mathrm{cons1_var}') \\
& \quad \mathrm{THEN} \\
& \quad \mathrm{SortedT}(\mathrm{InsAt}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), \\
& \qquad\qquad (\mathrm{Time}(\mathrm{cons1_var}')\ +\ t'_3), \mathrm{Er}(\mathrm{cons1_var}'))) \\
& \quad \mathrm{ELSE} \\
& \quad \mathrm{SortedT}(\mathrm{InsIn}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), \\
& \qquad\qquad (\mathrm{Time}(\mathrm{cons1_var}')\ +\ t'_2), \mathrm{Er}(\mathrm{cons1_var}'))) \\
& \quad \mathrm{ENDIF}
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

`sorted_prog_conj.2:`

$$
\begin{array}{|ll}
\{\text{-}1\} & \mathrm{SortedT}(\mathrm{cons2_var}') \\
\{\text{-}2\} & (\forall\ (t_2 :\ \mathrm{nat}, t_3 :\ \mathrm{nat}) : \\
& \quad \mathrm{SortedT}(\mathrm{cons2_var}') \wedge t_2\ >\ t_3 \Rightarrow \\
& \quad\ \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t_2, t_3))) \\
\{\text{-}3\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\{\text{-}4\} & t'_2\ >\ t'_3 \\
\hline
\{1\} & \mathrm{IF} \\
& \quad \mathrm{ProgAt}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), (\mathrm{Time}(\mathrm{cons1_var}')\ +\ t'_3)) \leq \\
& \quad\ \mathrm{Er}(\mathrm{cons1_var}') \\
& \quad \mathrm{THEN} \\
& \quad \mathrm{SortedT}(\mathrm{InsAt}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), \\
& \qquad\qquad (\mathrm{Time}(\mathrm{cons1_var}')\ +\ t'_3), \mathrm{Er}(\mathrm{cons1_var}'))) \\
& \quad \mathrm{ELSE} \\
& \quad \mathrm{SortedT}(\mathrm{InsIn}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), \\
& \qquad\qquad (\mathrm{Time}(\mathrm{cons1_var}')\ +\ t'_2), \mathrm{Er}(\mathrm{cons1_var}'))) \\
& \quad \mathrm{ENDIF}
\end{array}
$$

Instantiating the top quantifier in -2 with the terms: (t2!1
t3!1
),

`sorted_prog_conj.2:`

$$
\begin{array}{|ll}
\{\text{-}1\} & \mathrm{SortedT}(\mathrm{cons2_var}') \\
\{\text{-}2\} & \mathrm{SortedT}(\mathrm{cons2_var}') \wedge t'_2\ >\ t'_3 \Rightarrow \\
& \quad \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3)) \\
\{\text{-}3\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\{\text{-}4\} & t'_2\ >\ t'_3 \\
\hline
\{1\} & \mathrm{IF} \\
& \quad \mathrm{ProgAt}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), (\mathrm{Time}(\mathrm{cons1_var}')\ +\ t'_3)) \leq \\
& \quad\ \mathrm{Er}(\mathrm{cons1_var}') \\
& \quad \mathrm{THEN} \\
& \quad \mathrm{SortedT}(\mathrm{InsAt}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), \\
& \qquad\qquad (\mathrm{Time}(\mathrm{cons1_var}')\ +\ t'_3), \mathrm{Er}(\mathrm{cons1_var}'))) \\
& \quad \mathrm{ELSE} \\
& \quad \mathrm{SortedT}(\mathrm{InsIn}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), \\
& \qquad\qquad (\mathrm{Time}(\mathrm{cons1_var}')\ +\ t'_2), \mathrm{Er}(\mathrm{cons1_var}'))) \\
& \quad \mathrm{ENDIF}
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

`sorted_prog_conj.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{SortedT}(\text{cons2_var}') \\
\{\text{-}2\} & \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \\
\{\text{-}3\} & \text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}')) \\
\{\text{-}4\} & t_2' > t_3' \\
\hline
\{1\} & \text{IF} \\
 & \quad \text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}') + t_3')) \leq \\
 & \quad\ \text{Er}(\text{cons1_var}') \\
 & \ \text{THEN} \\
 & \ \text{SortedT}(\text{InsAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), \\
 & \qquad (\text{Time}(\text{cons1_var}') + t_3'), \text{Er}(\text{cons1_var}'))) \\
 & \text{ELSE} \\
 & \ \text{SortedT}(\text{InsIn}(\text{Prog}(\text{cons2_var}', t_2', t_3'), \\
 & \qquad (\text{Time}(\text{cons1_var}') + t_2'), \text{Er}(\text{cons1_var}'))) \\
 & \text{ENDIF}
\end{array}
$$

Splitting conjunctions,

we get 2 subgoals:

`sorted_prog_conj.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{SortedT}(\text{cons2_var}') \\
\{\text{-}2\} & \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \\
\{\text{-}3\} & \text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}')) \\
\{\text{-}4\} & t_2' > t_3' \\
\hline
\{1\} & \text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}') + t_3')) \leq \\
 & \quad \text{Er}(\text{cons1_var}') \supset \\
 & \ \text{SortedT}(\text{InsAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), \\
 & \quad (\text{Time}(\text{cons1_var}') + t_3'), \\
 & \quad \text{Er}(\text{cons1_var}')))
\end{array}
$$

Applying disjunctive simplification to flatten sequent,

`sorted_prog_conj.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}') + t_3')) \leq \\
 & \quad \text{Er}(\text{cons1_var}') \\
\{\text{-}2\} & \text{SortedT}(\text{cons2_var}') \\
\{\text{-}3\} & \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \\
\{\text{-}4\} & \text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}')) \\
\{\text{-}5\} & t_2' > t_3' \\
\hline
\{1\} & \text{SortedT}(\text{InsAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), \\
 & \qquad (\text{Time}(\text{cons1_var}') + t_3'), \text{Er}(\text{cons1_var}')))
\end{array}
$$

Applying sorted_insat1

`sorted_prog_conj.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & (\forall\ (e:\ \text{nat}, l:\ \text{list[cell]}, t:\ \text{nat}): \\
 & \quad \text{SortedT}(l) \Rightarrow \text{SortedT}(\text{InsAt}(l, t, e))) \\
\{\text{-}2\} & \text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}') + t_3')) \leq \\
 & \quad \text{Er}(\text{cons1_var}') \\
\{\text{-}3\} & \text{SortedT}(\text{cons2_var}') \\
\{\text{-}4\} & \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \\
\{\text{-}5\} & \text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}')) \\
\{\text{-}6\} & t_2' > t_3' \\
\hline
\{1\} & \text{SortedT}(\text{InsAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), \\
 & \qquad (\text{Time}(\text{cons1_var}') + t_3'), \text{Er}(\text{cons1_var}')))
\end{array}
$$

Instantiating the top quantifier in -1 with the terms: (Er(cons1_var′)
Prog(cons2_var′, t'_2, t'_3)
(Time(cons1_var′) + t'_3)
),

`sorted_prog_conj.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3)) \Rightarrow \\
 & \quad \mathrm{SortedT}(\mathrm{InsAt}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), \\
 & \qquad (\mathrm{Time}(\mathrm{cons1_var}') + t'_3), \\
 & \qquad \mathrm{Er}(\mathrm{cons1_var}'))) \\
\{\text{-}2\} & \mathrm{ProgAt}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), (\mathrm{Time}(\mathrm{cons1_var}') + t'_3)) \leq \\
 & \quad \mathrm{Er}(\mathrm{cons1_var}') \\
\{\text{-}3\} & \mathrm{SortedT}(\mathrm{cons2_var}') \\
\{\text{-}4\} & \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3)) \\
\{\text{-}5\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\{\text{-}6\} & t'_2 > t'_3 \\
\hline
\{1\} & \mathrm{SortedT}(\mathrm{InsAt}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), \\
 & \qquad (\mathrm{Time}(\mathrm{cons1_var}') + t'_3), \mathrm{Er}(\mathrm{cons1_var}')))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `sorted_prog_conj.2.1`.

`sorted_prog_conj.2.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & \mathrm{SortedT}(\mathrm{cons2_var}') \\
\{\text{-}2\} & \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3)) \\
\{\text{-}3\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\{\text{-}4\} & t'_2 > t'_3 \\
\hline
\{1\} & \neg\mathrm{ProgAt}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), \\
 & \qquad (\mathrm{Time}(\mathrm{cons1_var}') \\
 & \qquad\quad + t'_3)) \leq \\
 & \quad \mathrm{Er}(\mathrm{cons1_var}') \supset \\
 & \quad \mathrm{SortedT}(\mathrm{InsIn}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), \\
 & \qquad (\mathrm{Time}(\mathrm{cons1_var}') + t'_2), \\
 & \qquad \mathrm{Er}(\mathrm{cons1_var}')))
\end{array}
$$

Applying disjunctive simplification to flatten sequent,

`sorted_prog_conj.2.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & \mathrm{SortedT}(\mathrm{cons2_var}') \\
\{\text{-}2\} & \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3)) \\
\{\text{-}3\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\{\text{-}4\} & t'_2 > t'_3 \\
\hline
\{1\} & \mathrm{ProgAt}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), (\mathrm{Time}(\mathrm{cons1_var}') + t'_3)) \leq \\
 & \quad \mathrm{Er}(\mathrm{cons1_var}') \\
\{2\} & \mathrm{SortedT}(\mathrm{InsIn}(\mathrm{Prog}(\mathrm{cons2_var}', t'_2, t'_3), \\
 & \qquad (\mathrm{Time}(\mathrm{cons1_var}') + t'_2), \mathrm{Er}(\mathrm{cons1_var}')))
\end{array}
$$

Applying timel_prog_conj1

`sorted_prog_conj.2.2:`

{-1}	$(\forall\ (l:\ \text{list[cell]}, o_1:\ \text{cell}, t_2:\ \text{nat}, t_3:\ \text{nat}, \text{tcrt}:\ \text{nat}):$ $\text{SortedT}(l) \wedge$ $\text{Timel}(l) \leq \text{tcrt} \wedge$ $t_2\ >\ t_3 \wedge$ $\text{MemberT}(\text{Time}(o_1), \text{Prog}(l, t_2, t_3)) \Rightarrow$ $(\text{Time}(o_1) \leq \text{tcrt}\ +\ t_2))$
{-2}	$\text{SortedT}(\text{cons2_var}')$
{-3}	$\text{SortedT}(\text{Prog}(\text{cons2_var}', t_2', t_3'))$
{-4}	$\text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$
{-5}	$t_2'\ >\ t_3'$
{1}	$\text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}')\ +\ t_3')) \leq$ $\text{Er}(\text{cons1_var}')$
{2}	$\text{SortedT}(\text{InsIn}(\text{Prog}(\text{cons2_var}', t_2', t_3'),$ $(\text{Time}(\text{cons1_var}')\ +\ t_2'), \text{Er}(\text{cons1_var}')))$

Applying sorted_insin2

`sorted_prog_conj.2.2:`

{-1}	$(\forall\ (e:\ \text{nat}, l:\ \text{list[cell]}, t:\ \text{nat}):$ $\text{SortedT}(l) \wedge \text{Timel}(l) \leq t \Rightarrow \text{SortedT}(\text{InsIn}(l, t, e)))$
{-2}	$(\forall\ (l:\ \text{list[cell]}, o_1:\ \text{cell}, t_2:\ \text{nat}, t_3:\ \text{nat}, \text{tcrt}:\ \text{nat}):$ $\text{SortedT}(l) \wedge$ $\text{Timel}(l) \leq \text{tcrt} \wedge$ $t_2\ >\ t_3 \wedge$ $\text{MemberT}(\text{Time}(o_1), \text{Prog}(l, t_2, t_3)) \Rightarrow$ $(\text{Time}(o_1) \leq \text{tcrt}\ +\ t_2))$
{-3}	$\text{SortedT}(\text{cons2_var}')$
{-4}	$\text{SortedT}(\text{Prog}(\text{cons2_var}', t_2', t_3'))$
{-5}	$\text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$
{-6}	$t_2'\ >\ t_3'$
{1}	$\text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}')\ +\ t_3')) \leq$ $\text{Er}(\text{cons1_var}')$
{2}	$\text{SortedT}(\text{InsIn}(\text{Prog}(\text{cons2_var}', t_2', t_3'),$ $(\text{Time}(\text{cons1_var}')\ +\ t_2'), \text{Er}(\text{cons1_var}')))$

Instantiating the top quantifier in -1 with the terms: ($\text{Er}(\text{cons1_var}')$
$\text{Prog}(\text{cons2_var}', t_2', t_3')$
$(\text{Time}(\text{cons1_var}')\ +\ t_2')$
),

`sorted_prog_conj.2.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'))\wedge \\
& \quad \mathrm{Timel}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3')) \leq \\
& \quad\quad (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2') \Rightarrow \\
& \quad \mathrm{SortedT}(\mathrm{InsIn}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'), \\
& \quad\quad (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2'), \\
& \quad\quad \mathrm{Er}(\mathrm{cons1_var}'))) \\
\{\text{-}2\} & (\forall\ (l : \ \mathrm{list[cell]}, o_1 : \ \mathrm{cell}, t_2 : \ \mathrm{nat}, t_3 : \ \mathrm{nat}, \mathrm{tcrt} : \ \mathrm{nat}) : \\
& \quad \mathrm{SortedT}(l)\wedge \\
& \quad\quad \mathrm{Timel}(l) \leq \mathrm{tcrt}\wedge \\
& \quad\quad\quad t_2 \ > \ t_3\wedge \\
& \quad\quad\quad\quad \mathrm{MemberT}(\mathrm{Time}(o_1), \mathrm{Prog}(l, t_2, t_3)) \Rightarrow \\
& \quad\quad (\mathrm{Time}(o_1) \leq \mathrm{tcrt} \ + \ t_2)) \\
\{\text{-}3\} & \mathrm{SortedT}(\mathrm{cons2_var}') \\
\{\text{-}4\} & \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3')) \\
\{\text{-}5\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\{\text{-}6\} & t_2' \ > \ t_3' \\
\hline
\{1\} & \mathrm{ProgAt}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'), (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_3')) \leq \\
& \quad \mathrm{Er}(\mathrm{cons1_var}') \\
\{2\} & \mathrm{SortedT}(\mathrm{InsIn}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'), \\
& \quad\quad\quad\quad\quad (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2'), \mathrm{Er}(\mathrm{cons1_var}')))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

`sorted_prog_conj.2.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & (\forall\ (l : \ \mathrm{list[cell]}, o_1 : \ \mathrm{cell}, t_2 : \ \mathrm{nat}, t_3 : \ \mathrm{nat}, \mathrm{tcrt} : \ \mathrm{nat}) : \\
& \quad \mathrm{SortedT}(l)\wedge \\
& \quad\quad \mathrm{Timel}(l) \leq \mathrm{tcrt}\wedge \\
& \quad\quad\quad t_2 \ > \ t_3\wedge \\
& \quad\quad\quad\quad \mathrm{MemberT}(\mathrm{Time}(o_1), \mathrm{Prog}(l, t_2, t_3)) \Rightarrow \\
& \quad\quad (\mathrm{Time}(o_1) \leq t_2 \ + \ \mathrm{tcrt})) \\
\{\text{-}2\} & \mathrm{SortedT}(\mathrm{cons2_var}') \\
\{\text{-}3\} & \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3')) \\
\{\text{-}4\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\{\text{-}5\} & t_2' \ > \ t_3' \\
\hline
\{1\} & \mathrm{Timel}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3')) \leq (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2') \\
\{2\} & \mathrm{ProgAt}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'), (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_3')) \leq \\
& \quad \mathrm{Er}(\mathrm{cons1_var}') \\
\{3\} & \mathrm{SortedT}(\mathrm{InsIn}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'), \\
& \quad\quad\quad\quad\quad (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2'), \mathrm{Er}(\mathrm{cons1_var}')))
\end{array}
$$

Instantiating the top quantifier in -1 with the terms: (cons2_var!1
car(Prog(cons2_var′, t2′, t3′))
t2!1
t3!1
Time(cons1_var′)
),

we get 2 subgoals:

`sorted_prog_conj.2.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{SortedT}(\text{cons2_var}')\wedge \\
& \quad \text{Timel}(\text{cons2_var}') \leq \text{Time}(\text{cons1_var}')\wedge \\
& \quad\ t_2' > t_3'\wedge \\
& \quad\quad \text{MemberT}(\text{Time}(\text{car}(\text{Prog}(\text{cons2_var}', \\
& \quad\quad\quad t_2', t_3'))), \\
& \quad\quad\ \text{Prog}(\text{cons2_var}', t_2', t_3')) \Rightarrow \\
& \quad (\text{Time}(\text{car}(\text{Prog}(\text{cons2_var}', t_2', t_3'))) \leq \\
& \quad\ t_2' + \text{Time}(\text{cons1_var}')) \\
\{\text{-}2\} & \text{SortedT}(\text{cons2_var}') \\
\{\text{-}3\} & \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \\
\{\text{-}4\} & \text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}')) \\
\{\text{-}5\} & t_2' > t_3' \\
\hline
\{1\} & \text{Timel}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \leq (\text{Time}(\text{cons1_var}') + t_2') \\
\{2\} & \text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}') + t_3')) \leq \\
& \quad \text{Er}(\text{cons1_var}') \\
\{3\} & \text{SortedT}(\text{InsIn}(\text{Prog}(\text{cons2_var}', t_2', t_3'), \\
& \quad\quad\quad\quad (\text{Time}(\text{cons1_var}') + t_2'), \text{Er}(\text{cons1_var}')))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

`sorted_prog_conj.2.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{Timel}(\text{cons2_var}') \leq \text{Time}(\text{cons1_var}')\wedge \\
& \quad \text{MemberT}(\text{Time}(\text{car}(\text{Prog}(\text{cons2_var}', t_2', t_3'))), \\
& \quad\ \text{Prog}(\text{cons2_var}', t_2', t_3')) \Rightarrow \\
& \quad (\text{Time}(\text{car}(\text{Prog}(\text{cons2_var}', t_2', t_3'))) \leq \\
& \quad\ \text{Time}(\text{cons1_var}') + t_2') \\
\{\text{-}2\} & \text{SortedT}(\text{cons2_var}') \\
\{\text{-}3\} & \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \\
\{\text{-}4\} & \text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}')) \\
\{\text{-}5\} & t_2' > t_3' \\
\hline
\{1\} & \text{Timel}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \leq (\text{Time}(\text{cons1_var}') + t_2') \\
\{2\} & \text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}') + t_3')) \leq \\
& \quad \text{Er}(\text{cons1_var}') \\
\{3\} & \text{SortedT}(\text{InsIn}(\text{Prog}(\text{cons2_var}', t_2', t_3'), \\
& \quad\quad\quad\quad (\text{Time}(\text{cons1_var}') + t_2'), \text{Er}(\text{cons1_var}')))
\end{array}
$$

Splitting conjunctions,

we get 3 subgoals:

`sorted_prog_conj.2.2.1.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & (\text{Time}(\text{car}(\text{Prog}(\text{cons2_var}', t_2', t_3'))) \leq \text{Time}(\text{cons1_var}') + t_2') \\
\{\text{-}2\} & \text{SortedT}(\text{cons2_var}') \\
\{\text{-}3\} & \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \\
\{\text{-}4\} & \text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}')) \\
\{\text{-}5\} & t_2' > t_3' \\
\hline
\{1\} & \text{Timel}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \leq (\text{Time}(\text{cons1_var}') + t_2') \\
\{2\} & \text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}') + t_3')) \leq \\
& \quad \text{Er}(\text{cons1_var}') \\
\{3\} & \text{SortedT}(\text{InsIn}(\text{Prog}(\text{cons2_var}', t_2', t_3'), \\
& \quad\quad\quad\quad (\text{Time}(\text{cons1_var}') + t_2'), \text{Er}(\text{cons1_var}')))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

`sorted_prog_conj.2.2.1.1:`

$$
\begin{array}{|ll}
\{\text{-}1\} & (\mathrm{Time}(\mathrm{car}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'))) \leq \mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2') \\
\{\text{-}2\} & \mathrm{SortedT}(\mathrm{cons2_var}') \\
\{\text{-}3\} & \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3')) \\
\{\text{-}4\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\{\text{-}5\} & t_2' \ > \ t_3' \\
\hline
\{1\} & \mathrm{Timel}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3')) \leq (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2') \\
\{2\} & \mathrm{ProgAt}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'), (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_3')) \leq \\
 & \quad \mathrm{Er}(\mathrm{cons1_var}') \\
\{3\} & \mathrm{SortedT}(\mathrm{InsIn}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'), \\
 & \qquad\qquad (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2'), \mathrm{Er}(\mathrm{cons1_var}')))
\end{array}
$$

Hiding formulas: -2, -3, -4, 2, 3,

`sorted_prog_conj.2.2.1.1:`

$$
\begin{array}{|ll}
\{\text{-}1\} & (\mathrm{Time}(\mathrm{car}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'))) \leq \mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2') \\
\{\text{-}2\} & t_2' \ > \ t_3' \\
\hline
\{1\} & \mathrm{Timel}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3')) \leq (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2')
\end{array}
$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `sorted_prog_conj.2.2.1.1`.

`sorted_prog_conj.2.2.1.2:`

$$
\begin{array}{|ll}
\{\text{-}1\} & \mathrm{SortedT}(\mathrm{cons2_var}') \\
\{\text{-}2\} & \mathrm{SortedT}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3')) \\
\{\text{-}3\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\{\text{-}4\} & t_2' \ > \ t_3' \\
\hline
\{1\} & \mathrm{Timel}(\mathrm{cons2_var}') \leq \mathrm{Time}(\mathrm{cons1_var}') \\
\{2\} & \mathrm{Timel}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3')) \leq (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2') \\
\{3\} & \mathrm{ProgAt}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'), (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_3')) \leq \\
 & \quad \mathrm{Er}(\mathrm{cons1_var}') \\
\{4\} & \mathrm{SortedT}(\mathrm{InsIn}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3'), \\
 & \qquad\qquad (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2'), \mathrm{Er}(\mathrm{cons1_var}')))
\end{array}
$$

Hiding formulas: -1, -2, -4, -2, 3, 4,

`sorted_prog_conj.2.2.1.2:`

$$
\begin{array}{|ll}
\{\text{-}1\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\hline
\{1\} & \mathrm{Timel}(\mathrm{cons2_var}') \leq \mathrm{Time}(\mathrm{cons1_var}') \\
\{2\} & \mathrm{Timel}(\mathrm{Prog}(\mathrm{cons2_var}', t_2', t_3')) \leq (\mathrm{Time}(\mathrm{cons1_var}') \ + \ t_2')
\end{array}
$$

Hiding formulas: 2,

`sorted_prog_conj.2.2.1.2:`

$$
\begin{array}{|ll}
\{\text{-}1\} & \mathrm{SortedT}(\mathrm{cons}(\mathrm{cons1_var}', \mathrm{cons2_var}')) \\
\hline
\{1\} & \mathrm{Timel}(\mathrm{cons2_var}') \leq \mathrm{Time}(\mathrm{cons1_var}')
\end{array}
$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `sorted_prog_conj.2.2.1.2`.

`sorted_prog_conj.2.2.1.3:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{SortedT}(\text{cons2_var}') \\
\{\text{-}2\} & \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \\
\{\text{-}3\} & \text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}')) \\
\{\text{-}4\} & t_2' > t_3' \\
\hline
\{1\} & \text{MemberT}(\text{Time}(\text{car}(\text{Prog}(\text{cons2_var}', t_2', t_3'))), \\
 & \qquad \text{Prog}(\text{cons2_var}', t_2', t_3')) \\
\{2\} & \text{Timel}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \leq (\text{Time}(\text{cons1_var}') + t_2') \\
\{3\} & \text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}') + t_3')) \leq \\
 & \text{Er}(\text{cons1_var}') \\
\{4\} & \text{SortedT}(\text{InsIn}(\text{Prog}(\text{cons2_var}', t_2', t_3'), \\
 & \qquad (\text{Time}(\text{cons1_var}') + t_2'), \text{Er}(\text{cons1_var}')))
\end{array}
$$

Expanding the definition of MemberT

,

`sorted_prog_conj.2.2.1.3:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{SortedT}(\text{cons2_var}') \\
\{\text{-}2\} & \text{SortedT}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \\
\{\text{-}3\} & \text{SortedT}(\text{cons}(\text{cons1_var}', \text{cons2_var}')) \\
\{\text{-}4\} & t_2' > t_3' \\
\hline
\{1\} & \text{CASES Prog}(\text{cons2_var}', t_2', t_3') \text{ OF} \\
 & \text{null}: \text{FALSE}, \\
 & \text{cons}(o_1, p_1): \\
 & \quad (\text{IF} \\
 & \qquad \text{Time} \\
 & \qquad\quad (\text{car} \\
 & \qquad\qquad (\text{Prog}(\text{cons2_var}', \\
 & \qquad\qquad\quad t_2', t_3'))) = \\
 & \qquad\quad \text{Time}(o_1) \\
 & \qquad \text{THEN TRUE} \\
 & \quad \text{ELSE} \\
 & \qquad \text{MemberT} \\
 & \qquad\quad (\text{Time} \\
 & \qquad\qquad (\text{car}(\text{Prog}(\text{cons2_var}', \\
 & \qquad\qquad\qquad t_2', t_3'))), \\
 & \qquad\quad p_1) \\
 & \quad \text{ENDIF}) \\
 & \text{ENDCASES} \\
\{2\} & \text{Timel}(\text{Prog}(\text{cons2_var}', t_2', t_3')) \leq (\text{Time}(\text{cons1_var}') + t_2') \\
\{3\} & \text{ProgAt}(\text{Prog}(\text{cons2_var}', t_2', t_3'), (\text{Time}(\text{cons1_var}') + t_3')) \leq \\
 & \text{Er}(\text{cons1_var}') \\
\{4\} & \text{SortedT}(\text{InsIn}(\text{Prog}(\text{cons2_var}', t_2', t_3'), \\
 & \qquad (\text{Time}(\text{cons1_var}') + t_2'), \text{Er}(\text{cons1_var}')))
\end{array}
$$

Lifting IF-conditions to the top level,

`sorted_prog_conj.2.2.1.3:`

$\{-1\}$ SortedT(cons2_var′)
$\{-2\}$ SortedT(Prog(cons2_var′, t'_2, t'_3))
$\{-3\}$ SortedT(cons(cons1_var′, cons2_var′))
$\{-4\}$ $t'_2 > t'_3$

$\{1\}$ IF null?(Prog(cons2_var′, t'_2, t'_3)) THEN FALSE
ELSE
(IF
Time(car(Prog(cons2_var′, t'_2, t'_3))) =
Time(car(Prog(cons2_var′, t'_2, t'_3)))
THEN TRUE
ELSE
MemberT(Time(car(Prog(cons2_var′, t'_2, t'_3))),
cdr(Prog(cons2_var′, t'_2, t'_3)))
ENDIF)
ENDIF
$\{2\}$ Timel(Prog(cons2_var′, t'_2, t'_3)) $\leq$ (Time(cons1_var′) + t'_2)
$\{3\}$ ProgAt(Prog(cons2_var′, t'_2, t'_3), (Time(cons1_var′) + t'_3)) $\leq$ Er(cons1_var′)
$\{4\}$ SortedT(InsIn(Prog(cons2_var′, t'_2, t'_3),
(Time(cons1_var′) + t'_2), Er(cons1_var′)))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `sorted_prog_conj.2.2.1.3`.

`sorted_prog_conj.2.2.2:`

$\{-1\}$ SortedT(cons2_var′)
$\{-2\}$ SortedT(Prog(cons2_var′, t'_2, t'_3))
$\{-3\}$ SortedT(cons(cons1_var′, cons2_var′))
$\{-4\}$ $t'_2 > t'_3$

$\{1\}$ cons?[cell](Prog(cons2_var′, t'_2, t'_3))
$\{2\}$ Timel(Prog(cons2_var′, t'_2, t'_3)) $\leq$ (Time(cons1_var′) + t'_2)
$\{3\}$ ProgAt(Prog(cons2_var′, t'_2, t'_3), (Time(cons1_var′) + t'_3)) $\leq$ Er(cons1_var′)
$\{4\}$ SortedT(InsIn(Prog(cons2_var′, t'_2, t'_3),
(Time(cons1_var′) + t'_2), Er(cons1_var′)))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `sorted_prog_conj.2.2.2`.

Q.E.D.

D The Proof Script for `sorted_prog_conj`

```
(|sorted_prog_conj| "" (INDUCT "l")
  (("1" (GRIND)  )
   ("2" (SKOSIMP*)
    (("2" (EXPAND "Prog" +)
      (("2" (LEMMA "sorted_sorted")
        (("2" (INSTANTIATE -1 ("cons1_var!1" "cons2_var!1"))
          (("2" (ASSERT)
            (("2" (INSTANTIATE -2 ("t2!1" "t3!1"))
              (("2" (ASSERT)
                (("2" (SPLIT)
                  (("1" (FLATTEN)
                    (("1" (LEMMA "sorted_insat1")
                      (("1"
                        (INSTANTIATE -1
                         ("Er(cons1_var!1)" "Prog(cons2_var!1, t2!1, t3!1)"
                          "(Time(cons1_var!1) + t3!1)"))
                        (("1" (ASSERT)  )) ))
                      ))
                     )
                    ("2" (FLATTEN)
                     (("2" (LEMMA "timel_prog_conj1")
                       (("2" (LEMMA "sorted_insin2")
                         (("2"
                           (INSTANTIATE -1
                            ("Er(cons1_var!1)" "Prog(cons2_var!1, t2!1, t3!1)"
                             "(Time(cons1_var!1) + t2!1)"))
                           (("2" (ASSERT)
                             (("2"
                               (INSTANTIATE -1
                                ("cons2_var!1"
                                 "car(Prog(cons2_var!1,t2!1,t3!1))" "t2!1"
                                 "t3!1" "Time(cons1_var!1)"))
                               (("1" (ASSERT)
                                 (("1" (SPLIT -1)
                                   (("1" (ASSERT)
                                     (("1" (HIDE -2 -3 -4 2 3)
                                       (("1" (GRIND)  )) ))
                                     )
                                    ("2" (HIDE -1 -2 -4 -2 3 4)
                                     (("2" (HIDE 2) (("2" (GRIND)  ))
                                       ))
                                     )
                                    ("3" (EXPAND "MemberT" 1)
                                     (("3" (LIFT-IF) (("3" (GRIND)  ))
                                       ))
                                     ))
                                   ))
                                 ))
                               ))
                             ))
                           ))
                         ))
                       ))
                     ))
                   ))
                 ))
               ))
             ))
           ))
         ))
       ))
     ))
   ))
```



Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399