Compact Timed Automata for PLC Programs

H.X. Willems*

University of Nijmegen
Computing Science Institute
P.O.Box 9010
6500 GL Nijmegen, The Netherlands

November 24, 1999

Abstract

In this work a set of tools is developed to convert programs for Programmable Logic Controllers (PLCs) into timed automata in order to facilitate the verification of such programs. It is shown that our timed automata models of PLC programs can be dissected into a timed and an untimed part. Typically, the untimed part is much larger than the timed part and can be reduced in size by using the CADP toolset. The reduction in state space is substantial, even for small PLC programs.

Keywords: Programmable Logic Controllers, PLC-Automata, Timed Automata

AMS Subject Classification (1991): 68N20, 68Q05, 68Q55, 68Q60

CR Subject Classification (1994): C.3, D.2.4, D.2.5, D.3.2, D.3.4, F.3.1

*Permanent address: Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands, Rik.Willems@philips.com

1 Introduction

Programmable Logic Controllers (PLCs) are increasingly used for safety critical applications in a variety of industrial settings. The purpose of the work reported here is the verification of programs for PLC applications. Because many of the processes that are controlled by PLCs are time critical, time is considered to be an integral part of control as exerted by PLCs. The formalism of timed automata seems to be appropriate for modelling PLC systems because it allows to include real–time aspects. Furthermore, a number of model checking tools is available to facilitate verifying of systems of timed automata [1, 2, 3].

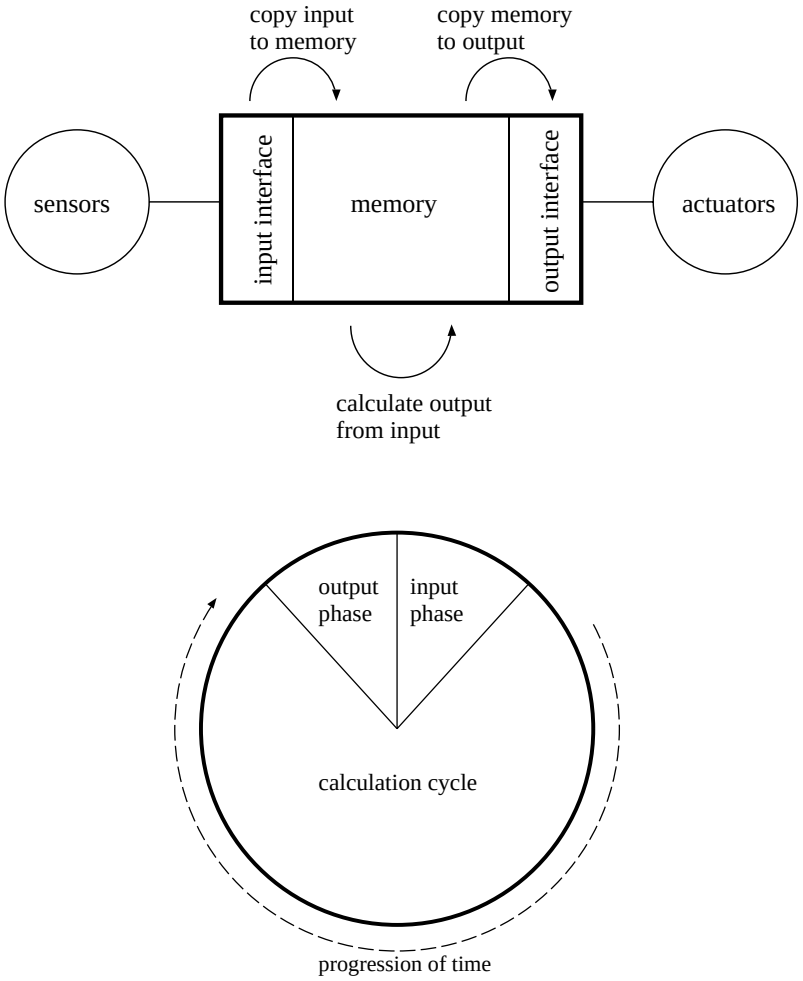

Figure 1: The operational structure of a PLC.

The PLCs considered here have an operational structure as depicted in figure 1. At the start of each cycle, the PLC reads all sensors from the environment and places their actual state in a memory location accessible to the PLC program. Subsequently the instructions that are programmed into the PLC are executed and the results of that computation are written to (other) memory locations. In a third step, these results are mapped to the actuators controlled by the PLC. After this output is completed, a new cycle can start. In this work, we consider

only simple PLCs, i.e. PLCs that do not support multitasking and interrupts, because these features would make the problems encountered even more complex.

One of the problems encountered when verifying PLC applications is the state space explosion. If we have for instance a PLC that controls 30 valves (which is not uncommon) and each valve can be open or closed, the state space resulting from the valve positions alone comprises 2^{30} states. However, many of these states may be inaccessible because certain combinations of valve positions can not occur. To reduce the state space a number of algorithms have been described [4, 5, 6, 7]. State space reduction is supported by the Caesar/Aldebaran Development Package (CADP toolset) which can be obtained from `http://www.inrialpes.fr/vasy/cadp.html`. However, these algorithms (and the toolset) do not take into account aspects of time and therefore state space reduction as offered by CADP can not be used directly for the state space reduction of PLC systems. In this work, we split the PLC systems into a timed and an untimed part and use state reduction on the latter. In this respect it is fortunate that of the two parts the untimed part turns out to be the largest by far.

In the next section a subset of a PLC language is presented that can be translated to timed automata and the construction of a compiler for that translation is discussed. Section 3 focuses on the use of CADP for the reduction of state spaces of PLC systems and the compilers that have been developed for the generation of reduced timed automata. Section 4 presents and discusses some results obtained on PLC systems with the compilers as developed in the preceding sections, section 5 gives the conclusions drawn from this research and section 6 offers some recommendations for further research.

2 Compiler construction

This section introduces the Instruction List (IL) language as defined in IEC standard 1131-3 [8] along with some timer constructs. In section 2.2 some restrictions on the base language are made to simplify the construction of the compilers. Also introduced is the model checker Uppaal which in this work is used to perform the model checking on the timed automata. Compilers to convert from IL to an intermediate format and from that format to Uppaal Timed Automaton format are described along with the design decisions taken. The Uppaal Timed Automaton format will be indicated with TA in the remainder of the text (in Uppaal it is referred to as the '.ta' format). The descriptions (timed automatons) generated with the compilers are indicated as 'PLC automatons' in the remainder of the text.

2.1 Base language: Instruction List

IL is a low level language which has a structure similar to a simple machine assembler. The International Electro-technical Commission (IEC) has embraced IL [8] because it is 'simple, easy to learn [and] ideal for solving small straight-forward problems where there are few decision points and where there is a limited number of changes in program execution flow' [9]. Furthermore it is sometimes regarded as 'the base language of an IEC compliant PLC, in which all other languages [defined in the standard] can be converted' [9]. A small program written in IL is given in figure 2.

```
            LD      Cycle      (* Load Cycle and  *)
            ADD     1          (* increase by 1   *)
            ST      Cycle      (* Store Cycle     *)
            LD      Speed      (* Load Speed and  *)
            GT      1000       (* test if > 1000  *)
            JMPNC   VOLTS_OK   (* Jump if not     *)
            LD      Volts      (* Load Volts and  *)
            SUB     10         (* reduce by 10    *)
            ST      Volts      (* Store Volts     *)
            LD      1          (* Load 1          *)
            R       Ready      (* Reset Ready     *)
            JMP     END        (* Go to End       *)
VOLTS_OK:   LD      1          (* Load 1          *)
            S       Ready      (* Set Ready       *)
END:        LD      0          (* Don't care      *)
```

Figure 2: A simple program written in IL.

An IL program consists of a series of instructions where each instruction is on a new line. An instruction consists of (at least) an operator and an operand; the operator can be preceded by a label, and its meaning can be changed by appending so–called *modifiers* to it. An overview of the operators that can be used in IL and the modifiers and operands allowed for each operator are given in table 2.1. Finally, each operand can optionally be followed by a comment.

Table 1: An overview of IL operators

operator	Modifiers	Operand	Semantics
LD	N	any	Load operator into actual register
ST	N	any	Store actual register into operand
S		BOOL[a]	Set operand to *true* if actual register is 1[b]
R		BOOL	Reset operand to *false* if actual register is 1[b]
AND	N,(	BOOL	Boolean AND
&	N,(	BOOL	Boolean AND (equivalent to AND operator)
OR	N,(	BOOL	Boolean OR
XOR	N,(	BOOL	Boolean exclusive OR
ADD	(	any	Addition
SUB	(	any	Subtraction
MUL	(	any	Multiplication
DIV	(	any	Division
GT	(	any	Comparison greater than
GE	(	any	Comparison greater than or equal
EQ	(	any	Comparison equal
NE	(	any	Comparison not equal
LE	(	any	Comparison less than or equal
LT	(	any	Comparison less than
JMP	N,C	LABEL	Jump to label
CAL	N,C	NAME[c]	Call function block
RET	N,C		Return from function block
)			Execute the last suspended operator

[a] Boolean type

[b] Ref [9] forgets to mention the condition.

[c] The name of a function block

Some syntactical details: a label starts with a letter, consists of letters and digits and ends with a colon (`VOLTS_OK:` and `END:` in figure 2 are labels). The modifiers recognized are 'N' (for –bitwise– negation), 'C' (for conditional) and '(', (indicating a 'suspended operation', i.e. the part between the parentheses has to be evaluated first). The operands that are present in an IL instruction can be boolean values, integers or variable names. In IL, labels and variable names are not case sensitive. The comments that can be present in an instruction start with '(*' and end with '*)' and do not contain 'newlines'.

To obtain a program for a PLC the IL code is enveloped in a 'program type definition' which starts with the keyword PROGRAM and is terminated with the keyword END_PROGRAM. The definition contains a list of input, output and internal variable declarations in addition to the (IL) program text. A sample program type definition is given in figure 3.

The semantics of IL programs are described by Mader and Wupper for boolean variables [10]. Extension of this work to encompass integers is straightforward. However, there is one problem:

```
PROGRAM SpeedCheck

VAR_INPUT
     Speed:  INT;
END_VAR
VAR
     Cycle:  INT;
END_VAR
VAR_OUTPUT
     Ready:  BOOL;
     Volts:  INT;
END_VAR

Here for instance the text of figure 2

END_PROGRAM
```

Figure 3: A sample program type definition

there is a special variable called *the actual register* (AE) which represents the value in the accumulator of the central processing unit of the PLC, and it should hold both boolean and integer values. As we don't want to use overloaded variables, this register should have a single type associated with it. It was decided to use an integer for the type of the register and to represent *true* with '1' and *false* with '0'.

2.2 Restrictions on the base language

With respect to timers: this work considers only on-delay timers ('TON' timers) which are standard timers described in IEC 1131-3 that have two inputs ('IN' and 'PT') and two outputs ('Q' and 'ET'). The layout of such a timer is given in figure 4 (upper part). The IN input determines whether the clock is running; if it is made *true*, the timer will wait for a time given by PT before it will make Q *true*. During this waiting time, the ET parameter gives the time since the timer started; if the waiting time is over, ET has the value of PT. In figure 4 (lower part) the relations between the terminals of a TON timer are given. This kind of timer can be thought of as representing a kitchen alarm: if it is set it will ring after a specified time.

This type of timer is realised by a software–construction (not by hardware) and as a result of this delays are introduced compared to ideal timers. This inaccuracy is not addressed explicitly in the remainder of the text. Furthermore, *the output values of the timer do not change unless the timer is called*: the timer does not influence the variables directly but a `CAL timer` statement has to be used to update the Q and ET values.

As a result of the parsing used here, the inputs of the timer have to be loaded before the call and an input list (which can be used in standard IL) can not be used. Because the target formats do not support variable names containing points, these points are translated to underscores (TIMER.PT is translated to TIMER_PT). If a TON timer is found in the input, the program

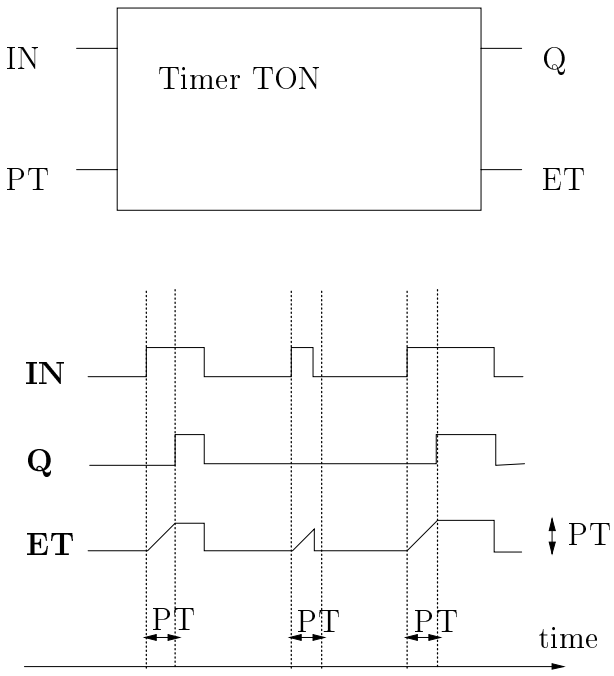

Figure 4: The inputs and outputs of a TON timer (top) and the relation between these signals (bottom). See text for further explanation.

adds four variables to its variable list: `<timer name>_IN`, `<timer name>_PT`, `<timer name>_ET` and `<timer name>_Q`. The `<timer name>_PT` variable should have a single value associated with it for each program. The reason for this will become clear in section 2.4.2.

Other types of timers could be treated similarly. Mader and Wupper show how such a timer can be converted into a timed automaton. For details the reader is referred to the original article [10]. In this conversion the parameter ET is not used, so we also do not use it.

The 'N' modifier for operators (see table 2.1) should be a bitwise negation according to the standard. In this work this is interpreted differently for booleans and integers: for booleans it is interpreted as as a logical negation (e.g. `LDN B1` –where B1 is a boolean variable with the value *true*– would result in the value 0 in the actual register) while for integers it is interpreted as changing the sign of the integer (e.g. `LDN 1` results in AE $=-1$). Note that the use of *true* and *false* as operand is not permitted in the translator developed here: these boolean values remain unscathed in the translation to intermediate format (see the next paragraph), but will cause problems further on (because they will be interpreted as –undeclared– identifiers); instead, 1 and 0 should be used. However, this can lead to problems if the original program contains one of the statements `LDN true` or `LDN false`, because `LDN 1` will lead to AE $=-1$ instead of AE $=0$. Therefore the use of the operator LDN should not be used with *true* or *false* (use LD with 0 or 1 instead).

The translation used here recognizes only three base types (BOOL, INT and TON) and three use types (INPUT, OUTPUT and NORMAL) while the standard defines more types for both.

The reason for restricting the base types is that a lot can be done with integers and booleans and that a lot of model checkers do not support more elaborate types like for instance reals for efficiency reasons. The reason for restricting the use types is that as far as verification is concerned a PLC does not need to have variables other than those three: the other types (e.g. 'retain' variables) can be easily incorporated if needed, but will not yield additional problems.

Furthermore, functions are not recognized, except when a TON timer is called (this implies that the CAL operator is only implemented for timers and that the RET operator is not implemented). This was done because for verification it does not matter whether the code is sequential or contains function calls and all PLC programs with function calls can be rewritten into programs without function calls. Generally, programs containing functions calls can not be rewritten simply in sequential programs. However, because recursive functions are not permitted in applications with a bounded cycle time [11] (such as for instance PLC programs[1]), functions degenerate into simple macros and rewriting the programs becomes possible.

Finally, absolute adresses defined in the standard (for instance `%IW40`) are not recognized, and identifiers should be used instead.

2.3 From instruction list to intermediate format

Initially, it was expected that we could use an Extended Affix Grammar (EAG) compiler to convert IL to a timed automaton, but this approach resulted in a compiler that was too slow and this approach was abandoned further on. We then decided to use a C program for the translation and as we already had the EAG parser we used it to generate an intermediate format that could be read easily by a C program.

The intermediate format has the name of the program on the first line (in ASCII), the number of declarations on the next line followed bij the declarations. Each declaration line has the format `variable_name : base_type : use_type` where the `base_type` is `BOOL`, `INT` or `TON` and the `use_type` is `INPUT`, `OUTPUT` or `NORMAL`. The next line contains the number of instructions of the IL file, followed by the instructions in the format `sequence_number : level : label : operator : modifiers : operand`, each on a new line.

While the latter elements of this format are self-explanatory, the *sequence number* and the *level* may need some introduction. The *level* is the number of leading parentheses for a certain statement. The reason for introducing such a level is that we intend to model nested parentheses not with a stack but with different variables for each level. We do this because we want to end up with a finite automaton (the number of nested parentheses can be determined statically). The construction depicted in figure 5a is translated in that depicted in figure 5b. It can be observed that the sequence of the statements in 5b ('postfix' notation) is different from those in 5a 'prefix' notation). The postfix notation of figure 5b is necessary to treat the bracket construction appropriately. The structure in 5b will later (see sections 2.4 and 3) be modelled with variables `ae0` and `ae1` for the main level and the nested level respectively.

See the end of the report for the availability of the literal text of the EAG grammar (and related files). The EAG compiler itself can be obtained from `ftp://ftp.cs.kun.nl/pub/eag`.

[1] Actually, recursive functions are forbidden in the standard.

```
Lbl1: LD    Top          1 : 0 : LBL1 : LD : <no_entry> : TOP
      ADDN( 5            2 : 1 : <no_entry> : LD : <no_entry> : 5
      MUL   MID          3 : 1 : <no_entry> : MUL : <no_entry> : MID
      )                  4 : 0 : <no_entry> : ADD : N : <SUSPENDED>
      ST    Bottom       5 : 0 : <no_entry> : ST : <no_entry> : BOTTOM
```

a b

Figure 5: The IL text (a) and the corresponding intermediate text (b).

2.4 From intermediate format to TA format

The model checker Uppaal (`http://www.docs.uu.se/docs/rtmv/uppaal/`) was chosen because it supports integers and most of the elements of the IL language (assignments, arithmetic and comparisons). Furthermore, real–time constructs can be modelled, and as indicated above this is important for PLC applications. The package does not support booleans (only integers) and this coincides nicely with the choice of implementation of the actual register as discussed above. An additional advantage is that the TA input format that is supported by this package is fairly simple (note: the DIV operator is not implemented to maintain similarity with the i2lotos compiler, see section 3.1).

2.4.1 Main translation characteristics

The main characteristics of the translation are

- Every statement in the IL program is translated into a (number of) transition(s) between two states. These two states can be thought of as 'before the execution of the statement' and 'after the execution of the statement'. It is evident that the 'result state' of one statement is the 'begin state' of the subsequent statement.
- The actual number of transitions and the conditions for each transition are determined by the operator and the operand of that statement. For instance the 'LD' operator with an integer operand results in a single transition with the action *the actual register acquires the value of the operand* (e.g. `ae0:=TOP`, where TOP is the name of a variable), while the same operator with a boolean operand gives rise to two transitions, one in which the actual register acquires the value 1 and another in which it acquires the value 0. In the latter case, **both** transitions are needed, because *at the time of translation* it can not be determined whether the boolean *at the time of execution* is *true* or *false*.
- An additional transition is introduced to model the input phase of the PLC. In this transition the variables that are in the input list of the PLC (the variables with usetype 'INPUT') are copied from their values in the environment. The value of a variable in the environment is modelled with an additional variable; the name of that additional environment variable is `<variable name>_env` (see also next point). This transition is added to the beginning of the description of the PLC automaton. The associated state is referred to as the input state in the remainder of this text.

- The environment is modelled with an additional automaton, which manipulates the `<variable name>_env` variables. For each boolean variable there are two transitions in this automaton representing *set boolean to true* and *set boolean to false*. For each integer variable there is a transition which increases the value of the variable and one which decreases its value. To obtain correspondence with the reduced automata (see section 3) the values of the integer values are restricted to the interval [-9..9], which is achieved by adding guards to the transitions[2].

- An additional transition is introduced in the PLC automaton to model the output phase of the PLC. Because the present model does not use the output values of the PLC, no real actions are performed in this transition; if actions in the output phase are required for instance at a later stage of the research it is easy to add them here. This transition is added to the end of the description of the PLC automaton. The associated state is referred to as the output state in the remainder of this text. Furthermore, the transition can be used to control the timing behaviour of the PLC (see section 2.4.2).

- The last extra transition that is added to the PLC automaton is the initialisation of the PLC. This transition models the event of starting the program for the first time, e.g. after it has been downloaded from a PC; in that case, all variables are initialised to zero. So-called 'Retain variables' defined in the standard are not supported, because they are not important for the present verification effort. These variables could be added if consequences of power failures are to be studied.

2.4.2 Timing aspects

For each TON timer present in the PLC code, a separate timer automaton is added to the system. This timer automaton has a structure as depicted in figure 6 [10]. The timer is synchronized with the rest of the system through the CALxy_TIMER! synchronization statements, where the symbols xy are replaced by ff, tf or tt. In these statements, the value of x indicates the value of IN that is sent to the timer and y indicates the value of Q that is returned.

The time–out value for each timer (the PT value) has a fixed value for each timer automaton. This restriction is imposed partly by good programming practice (semantics of changing a PT value of a running timer are not defined) and partly because Uppaal does not allow variables in state invariants (and the PT value is used in a state invariant for one of the timer states). Note (1) that the use of PT in a state invariant is due to the particular translation that we use here (from [10]): it is not a fundamental problem, (2) that the PT value is treated as a variable in the program, and therefore values can be assigned to it during program execution. It is difficult to extract PT values from the IL program text without imposing strict coding rules on the assignment of values to PT variables, which doesn't appeal to us at this moment. Therefore the PT values have to be entered by hand in each produced TA file when the code comprises at least one TON timer.

The description so far has not taken into account the timing aspects of the PLC itself. To circumvent the problems of Zenoness (the possibility of performing infinitely many transitions

[2] If another environmental behaviour is more appropriate, the entire environment automaton can be replaced with a new automaton by using an ASCII editor.

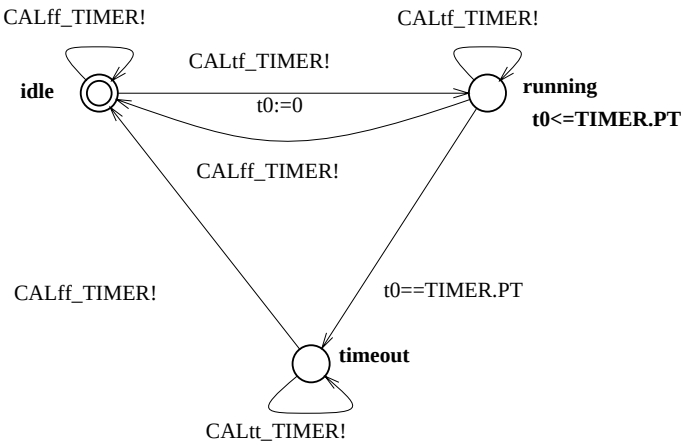

Figure 6: A timed automaton for a TON timer (cf. [10]).

without the progression of time) two different approaches have been incorporated in the compiler: the 'epsilon model' and the 'change event model'.

In the epsilon model, the minimum and the maximum cycle time of the PLC are explicitly given (`epsilon_1` and `epsilon_2` respectively). This is more or less in accordance with reality, where minimum and maximum cycle times of control programs can be determined. To this end, a clock `PLCsystime` is added to the system and this clock is set to zero upon initialisation of the PLC. Every state in the PLC automaton except the initialisation state now has an invariant `PLCsystime<=epsilon_2` to limit the maximum time spent in each state. The transition from the output state to the input state is now guarded with `PLCsystime>=epsilon_1` and performs the assignment `PLCsystime:=0`. The compiler sets the values of these constants to 1 (`epsilon_1`) and 2 (`epsilon_2`) respectively. If other values are required, the text in the TA file has to be edited (as these files are in ASCII, this can be done quite simply). This model has the disadvantage that an additional clock is added to the system, which enlarges the state space.

In the change event model it is assumed that the computational steps can be performed in zero time. However, between two consecutive program executions the input of the PLC has to change. The implicit assumption is then that the changes in the environment do not occur in zero time and that therefore the problem of Zenoness is circumvented. The change event model comes in two flavours. The first model has additional guards on the transition from output state to input state, the second also incorporates an additional variable to explicitly record a change in the environment.

The first model has a guard `<variable name>!=<variable name>_env` for each input variable (a separate transition is supplied for each variable because Uppaal does not support logical 'OR' constructs). As long as nothing changes in the environment, no return to the input phase is possible. These guards could also be added to the transition *from* the input state (that would be more in accordance with reality), but as there are no real output actions in the model used here these transitions can be added here, which is simpler. Note that for each input variable a separate transition from the output state emerges.

The second model has a guard `PLC_env_change==1` and an extra assignment `PLC_env_change:=0`

on the input transition. In addition, it has the assignment `PLC_env_change:=1` on each transition in the environment model. Compared to the previous model this has the disadvantage of an additional (boolean) variable, but the advantage is that a similar construction can be used on the reduced automata produced with CADP (section 3.2), which is not possible with the first construction (see section 3.3).

Both change event models are only correct for the class of PLC programs where the same input always leads to the same output ('history free'). A commonly used algorithm to control temperature in for instance a furnace is the so-called 'PID' controller: this controller does not only use the temperature reading itself but also uses the integral and the derivative of this value (hence its name). Such an algorithm is not 'history free'. In general, programs which contain (TON) timers are not history free because the output of the program will be different for differing states of the timer. Because the timer could generate an event going from its *running* state to its *timeout* state, this problem could be circumvented (we did not do that in this work because we wanted the timers to have an identical structure for the three types of translation, but the addition of an additional assignment `PLC_env_change:=1` to the transition mentioned would solve the problem for the second change event model).

The compiler generates a system with an epsilon model by default. Change event models can be obtained by specifying `-opt` or `-ce` on the command line (for the first and second model respectively). The option `-eps` generates a system with an epsilon model (same as no option specified).

An illustrative example of a translation from intermediate format to TA format is rather space–consuming and will therefore be omitted. A comprehensive example of a translation is given in section 4 where a graphical representation of the (TA) automatons is used which is more concise.

3 State space reduction with Caesar/Aldebaran

State space reduction can be performed with the CADP toolset. However, this toolset can not handle timed automata. Therefore we split the PLC system in a timed and an untimed part and use state reduction on the latter. For that purpose, a compiler was constructed that abstracts from timing and is able to produce a description of the PLC automaton in LOTOS, which is one of the input languages supported by the CADP toolset. The reduced automaton can be saved in a number of formats, of which the AUT format (the standard CADP output, extension '.aut') is the most simple. These outputfiles can then be translated to TA format and in this process the timing information can be added again. The translation from intermediate format to LOTOS is described in section 3.1, the translation from AUT to TA in section 3.3.

3.1 From intermediate format to LOTOS format

The syntax of LOTOS is a little bit more complicated than the syntax of for instance TA. A typical outline of a LOTOS program obtained from a IL program is given in figure 7. The identifiers between square brackets (Read, Write, XTIMER) are the externally visible *gates* (communication channels) of the PLC automaton. In the *Process description* part we find a description of the states and transitions analogous to the description in the TA file (but of course in LOTOS' own dialect). For details about the LOTOS syntax see [12].

The main characteristics of the translation are:

```
specification PLCspec[Read, Write, XTIMER]: noexit
library
       X_INTEGER, X_NATURAL, X_BOOLEAN
type PLCstates is Boolean
       sorts  State_Type
       opns   state_0 (*!  constructor *), state_1 (*!  constructor *),...
              state_n (*!  constructor *) :  -> State_Type
              _eq_:  State_Type, State_Type -> Bool
       eqns   forall x,y:  State_Type
              ofsort Bool
              x eq x = true;[3]
              x eq y = false
endtype
behaviour PLC [Read, Write, XTIMER] (state_0, false, 0 of Int,...)
where
Process description
endproc
endspec
```

Figure 7: A typical outline of a LOTOS program describing a PLC.

[3] This construction may not be completely fail safe [13].

- Just as in the transition to TA format (section 2.4) every statement is translated into one or more transitions and the number of transitions and the conditions for each transition are determined by the operator and the operand of that statement.
- Both input and output are now explicitly modelled with an additional state and synchronisation statements.
- The initialisation of the PLC is not an additional state but is contained in the `behaviour` clause in the LOTOS file.
- As LOTOS has a library which supports booleans, boolean variables are now translated as booleans. This generates slightly more complex code, but this is needed to keep the state space small.
- The DIV operator present in the IL language has no corresponding construct in the LOTOS library and is therefore not implemented.
- The integer library of LOTOS only supports integers in the range [-9..9] which is rather limited. However, this also helps to keep the state space small while making available 19 levels for input or output variables. Doing arithmetic with this limited value range could possibly be dangerous.
- The clock functionality is abstracted away, but the communication with the clocks remains in the description of the PLC automaton. In this translation the `CAL Timer` statement is converted into a sequence of events: first the value of `TIMER_IN` is written to the timer gate (`XTIMER` in figure 7); then the value of `TIMER_Q` is read from the timer gate and finally the PLC is 'promoted' to the next state. In an execution of such a PLC program there are four ways in which this statement can be executed: both `TIMER_IN` and `TIMER_Q` can be either *true* or *false*. This is of importance when we try to translate the minimised automaton to TA format (see section 3.3).

3.2 State space reduction of the timeless automaton

The automaton described with the LOTOS file does not contain timers and can thus be processed by the CADP toolset to reduce the state space. If the file produced with i2lotos has for instance the name `proc2.lotos` then we need two additional files with names `proc2.t` and `proc2.f`; their contents are given in figure 8. The tool Eucaliptus can be used for easy manipulation of the files and as an interface for the Caesar/Aldebaran tools. Alternatively these tools can be used from the command line, which is more in accordance with the pipe-and-filter architecture of the other programs; however, the command line syntax is rather complex and you may not want to use it. See the documentation included in the toolset for further information on CADP and Eucalyptus.

3.3 From Caesar/Aldebaran AUT format to Uppaal TA format

The AUT files produced by CADP contain a Labeled Transition System (LTS). An example of an AUT file is given in figure 9. The first line of such a file gives the start state, the total number of transitions and the total number of states. The other lines each describe one transition:

```
#define CEASAR_ADT_EXPERT_F 4.4
```

a

```
#define CAESAR_ADT_EXPERT_T 4.4

#include "X_BOOLEAN.h"
#include "X_NATURAL.h"
#include "X_INTEGER.h"
```

b

Figure 8: The contents of files `proc2.f` (a) `proc2.t` (b).

```
des (0, 8, 6)
(0, "READ !FALSE", 2)
(0, "READ !TRUE",3)
(1, "WRITE !TRUE",0)
(2, "XTIMER !FALSE",4)
(3, "XTIMER !TRUE",4)
(4, "XTIMER !FALSE",5)
(4, "XTIMER !TRUE",1)
(5, "READ !FALSE",0)
```

Figure 9: The contents of a sample file in AUT format.

subsequently the originating state, the label of the transition and the resulting state are given. It can be observed that the original structure of the IL program is completely lost.

Because the structure of this file is different from the original program, the resulting TA file does not have completely the same style as the TA file obtained directly from the IL program.

The differences with i2ta and the design decisions taken in developing this compiler are:

- The variables do not have their original names. This is caused mainly by the fact that CADP removes the variables that are not used by the program from its variable set and reconstruction becomes tiresome. However, the sequence of variables in input and output statements seems to be undisturbed compared to the LOTOS inputfile, which in turn uses the sequence of the declarations of the variables and thus the ordering of the variable declarations can possibly be used to reassign variables their original names if required. The program assigns names to the variables in order of appearance; input variables are assigned names `IVAR0`, `IVAR1`, `IVAR2`, ... etc and output variables are assigned names `OVAR0`, `OVAR1`, `OVAR2`, ... etc.
- There is no explicit input state. The input actions are coded in the `"READ !FALSE !TRUE..."` labels of the LTS. These are translated in the TA program as guards `IVAR0==0, IVAR1==1,...` (remember that there are no boolean variables in TA).
- The Environment automaton now acts on the input variables themselves, i.e. it contains assignments for each input variable (for instance `IVAR0:=1` and `IVAR0:=0`).

AUT contents:

```
...
(2, "XTIMER !FALSE",4)
(3, "XTIMER !TRUE",4)
(4, "XTIMER !FALSE",5)
(4, "XTIMER !TRUE",1)
...
```

TA contents:

```
...
state_2 -> state_5 {
  sync CALff_XTIMER?;
},
state_3 -> state_5 {
  sync CALtf_XTIMER?;
},
state_3 -> state_1 {
  sync CALtt_XTIMER?;
},
...
```

Figure 10: An example of the translation of a timer from AUT to TA format.

- There is no single output state. The output actions are coded in the `"WRITE !FALSE !TRUE..."` labels of the LTS. These are translated in the TA program as assignments `OVAR0:=0, OVAR1:=1,...`.
- As indicated earlier (section 3) the timers are abstracted away in the LOTOS file, but the communications with the timers remain in the description. This communication behaviour is still present in the reduced automaton. For each timer, there is a state that can be reached in two ways (`TIMER_IN` is *true* or *false*) and that can be left in two ways (`TIMER_Q` is *true* or *false*). Because each timer has its own communication channel (gate) these four transitions can be identified for each timer. We can now replace these four transitions by three transitions that use the same synchronisation events as the timers used earlier (section 2.4). The loss of one transition is brought about by the fact that for a TON timer it is impossible to respond a `TIMER_IN` that is *false* with a `TIMER_Q` that is *true* (CALft_XTIMER does not exist). As a result of this replacement the 'central timer state' is removed from the automaton. To avoid problems with transitions the remaining states are not renumbered and the central timer states are simply removed from the declaration of states. Figure 10 shows an example.
- The timer constants (XTIMER_PT) are now coded as constants and the value of each timer constant should be set to its appropriate value in the TA file (default value is 1). The code for the timers is identical to the code for the timers produced by i2ta.
- The Zenoness of the resulting automaton again needs to be established. To handle this, two models have been incorporated: an epsilon model analogous to the epsilon model described in section 2.4 and a change event model analogous to the second model described in the same section. The first change event model can not be used because there is no difference here between the variables that are handled by the environment and the internal variables of the program (as a matter of fact, the PLC program actually has no internal variables anymore).

4 Results and discussion

To show the working of the compilers we give here an example of an (extremely simple) IL based PLC program, that has one input (a button) and one output (a light). If the button is pushed long enough (i.e. longer than the time-out of the timer, which in this case is 5) the light will come on. The IL and intermediate texts are given in figure 11. It is assumed that a pushed button and a burning light are represented as *true*, while an unpressed button and an extinguished light are represented as *false*.

Original program:

```
PROGRAM TimeResp
VAR_INPUT
     start:  BOOL;
END_VAR
VAR
     timer:  TON;
END_VAR
VAR_OUTPUT
     led:  BOOL;
END_VAR
     LD   5
     ST   timer.PT
     LD   start
     ST   timer.in
     CAL  timer
     LD   timer.q
     ST   led
END_PROGRAM
```

Intermediate program:

```
TIMERESP
3
START : INPUT : BOOL
TIMER : NORMAL : TON
LED : OUTPUT : BOOL
7
0 : 0 : <no_entry> : LD : <no_entry> : 5
1 : 0 : <no_entry> : ST : <no_entry> : TIMER_PT
2 : 0 : <no_entry> : LD : <no_entry> : START
3 : 0 : <no_entry> : ST : <no_entry> : TIMER_IN
4 : 0 : <no_entry> : CAL : <no_entry> : TIMER
5 : 0 : <no_entry> : LD : <no_entry> : TIMER_Q
6 : 0 : <no_entry> : ST : <no_entry> : LED
```

Figure 11: The 'TimeResp' program in IL and intermediate format.

This program can be converted into TA format with the program `i2ta`. The timed PLC automaton that is the result of this conversion (in this case with the epsilon model) is given in figure 12. The complete system consists of this automaton, a timer, an automaton representing the environment and some declarations. The timer and the environment automaton are shown in figure 13.

From figure 12 it can be seen that the i2ta translator as produced in this work is not optimal: the coding of (IL) statements that contain booleans is rather inefficient (see for instance the transition from state_3 to state_4). This is caused by the fact that the i2ta translator was developed from the i2lotos translator, where conversion between boolean variables and (integer type) actual registers has to be performed.

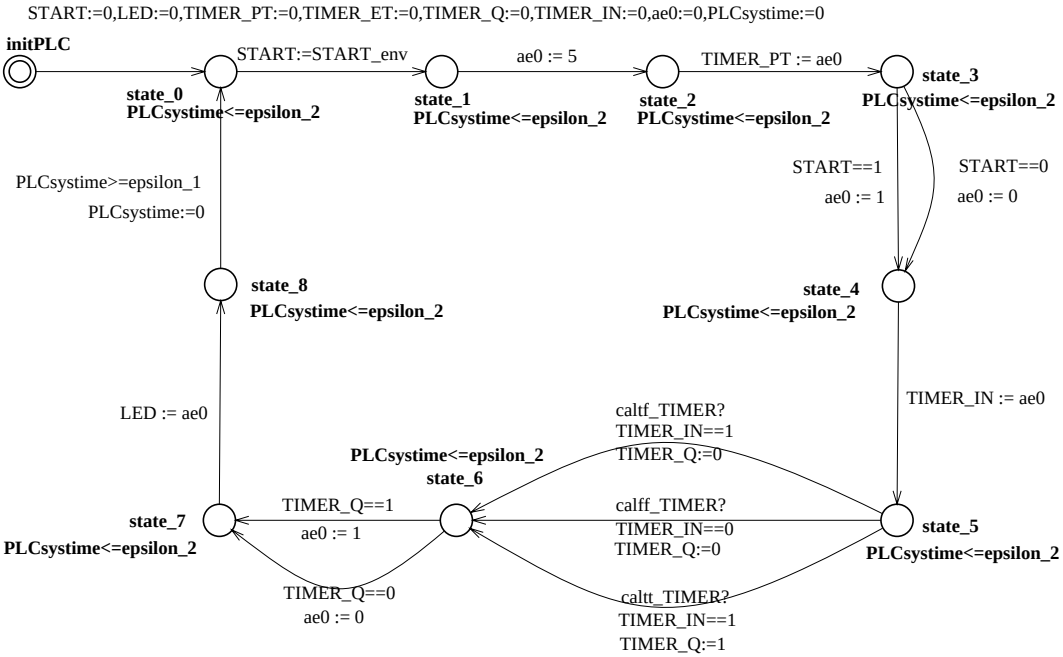

Figure 12: PLC automaton in Uppaal format resulting from direct translation of the intermediate text of the 'TimeResp' program.

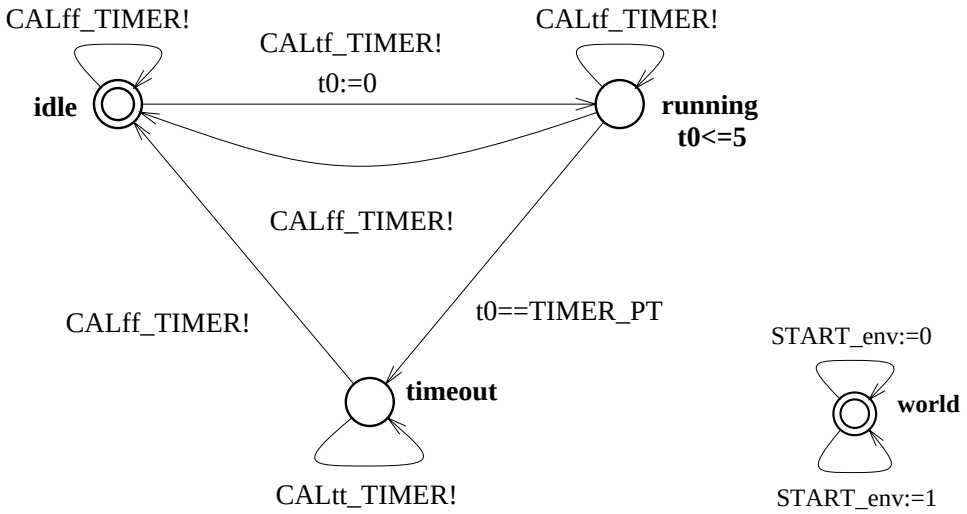

Figure 13: The automata for the timer (left) and the environment (right) in Uppaal format ('TimeResp' program).

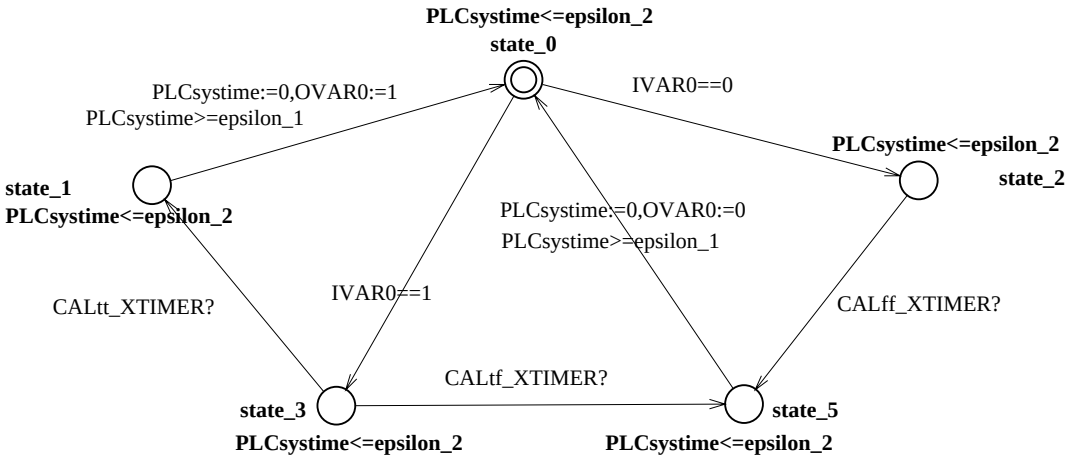

Figure 14: Reduced PLC automaton in Uppaal format ('TimeResp' program).

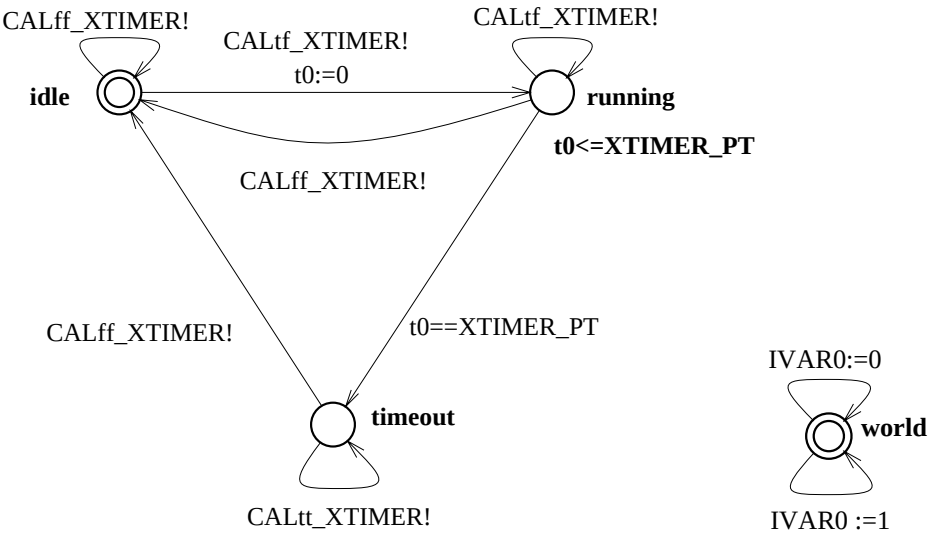

Figure 15: The automata for the timer (left) and the environment (right) in Uppaal format (reduced 'TimeResp' program).

Table 2: State space reduction of PLC automata with the CADP toolset.

Program name	Original automaton		Reduced automaton	
	States	Transitions	States	Transitions
TimeResp	25	34	5[a]	7[a]
VHS_CS1_B2	279	496	10	12

[a] Actually, the reduced automaton contains 6 states and 8 transitions in the AUT file, but *aut2ta* reduces both by one (see also figure 10).

To be able to use the CADP toolset, the same intermediate format can be translated to LOTOS; in that process the timer functionality is removed from the description (only the communication with the timer remains). It is fortunate that of the two parts the untimed part turns out to be the largest by far. If the reduced automaton produced by the CADP toolset is translated to Uppaal format (with `aut2ta`), the automaton shown in figure 14 is obtained instead of the automaton shown in figure 12. The timer (reintroduced by `aut2ta`) and the environment are given in figure 15.

To give an impression of the state space reduction obtained, table 2 gives some data for the original and the reduced automaton. The 'Original automaton' is the finite state machine constructed by the CADP toolset from the LOTOS file (note that the 'Original automaton' has more states and transitions than the *extended* finite state machine from the TA file). The same table also contains data for a somewhat more elaborate IL program; this examples was taken from a case study in the field of PLC verification [14]. It can be observed that the state space reduction is substantial, even for these small examples.

5 Conclusions

The compilers produced in this work are able to perform the translations given in figure 16: *il2i* converts IL into intermediate format and this can be translated directly into TA by *i2ta* or via *i2lotos* to input for CADP, the output of which can be translated to TA with *aut2ta*.

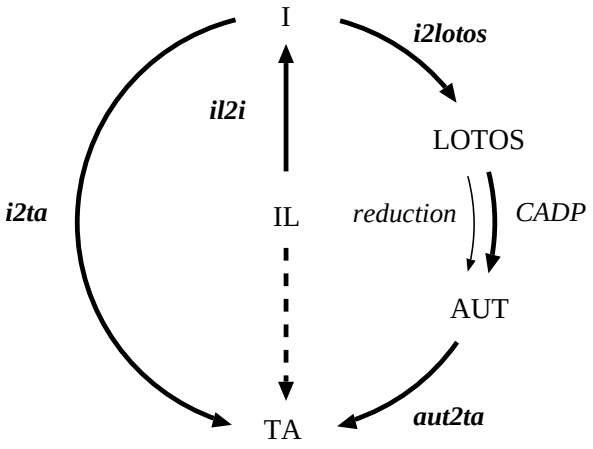

Figure 16: Translations from IL to TA.

The method of dissecting a PLC program into a timed and an untimed part can be performed quite elegantly. The untimed part is much larger than the timed part and can be reduced in size by using the CADP toolset. The reduction in state space is substantial, even for small PLC programs.

6 Recommendations for further research

First some more experience has to be gathered with size reduction as presented here, especially for larger and more realistic PLC programs. To be able to handle more realistic programs, the compilers will most likely have to be able to handle function invocation and DIV / MOD operations. As the latter is dependent on the (integer) library used with LOTOS, it may be necessary to write a more complete library for this purpose. Function invocation could for instance be treated as expansion of the sequential program or it could be treated as message passing to a separate 'function automaton'. The former seems to be rather straightforward while the latter holds the promise of a more elegant solution (but maybe at the cost of greater complexity).

Another elaboration is the use of other PLC programming languages from the IEC standard, like for instance Structured Text (ST) or Sequential Function Chart (SFC). For this work, additional compilers should be written, either to convert from ST, SFC etc. to the intermediate format or to IL (if that is really possible for all languages). It is also possible to use the i2ta and i2lotos compilers developed here as a basis and incorporate the support for different languages here (not recommended, as it increases the complexity and decreases the modularity of the compilers).

It should be (formally ?) verified that the (TA) timed automatons produced via the two ways described here (direct translation versus translation with reduction) are in fact equivalent.

The method outlined here can possibly be used for the verification of a large variety of control tasks where timing is important. Especially the power to decrease the state space of programs in this way seems to be promising.

Availability

The translators are coded as C programs (except for il2i, which is an EAG grammar). Because these translators are written in ANSI C they can be compiled with any general C compiler (gcc, cc etc). The executables can be used from the command line. The programs read their input from STDIN and write their output to STDOUT.

This document and the source codes of the compilers is available from
`http://www.cs.kun.nl/~mader/rik/rik_vhs.html`.

Acknowledgement

Thanks are due to Angelika Mader for her help with the semantics of PLC programs, to Judi Romijn for her help with the CADP toolset and LOTOS and to Frits Vaandrager for initializing this research and for usefull questions and comments. Marco Hollenberg (Philips Research) is acknowledged for his comments on the manuscript. This work was carried out in the context of the ESPRIT project 26270, Verification of Hybrid Systems (VHS).

References

[1] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. *UPPAAL: a tool suite for the automatic verification of real–time systems*, pages 232–243. Volume 1066 of Alur et al. [15], 1996.

[2] C. Daws, A Olivero, S. Tripakis, and S. Yovine. *The tool KRONOS*, pages 208–219. Volume 1066 of Alur et al. [15], 1996.

[3] T. Henzinger and P.-H. Ho. HyTech: The Cornell HYbrid TECHnology tool. In U. Engberg, K. Larsen, and A. Skou, editors, *Proceedings of the workshop on tools and algorithms for the construction and analysis of systems*, volume NS-95-2 of *BRICS Notes Series*, pages 29–43. Department of Computer Science, University of Aarhus, Denmark, may 1995.

[4] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[5] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6), December 1987.

[6] J-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3):219–236, 1990.

[7] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, and P. Raymond. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–269, June 1992.

[8] IEC International Standard 1131-3, Programmable Controllers, Part 3: Programming Languages, 1993.

[9] R. W. Lewis. *Programming industrial control systems using IEC 1131-3*, volume 50 of *IEEE Control Engineering Series*. The Institution of Electrical Engineers, London, UK, 1995.

[10] A. Mader and H. Wupper. Timed automaton models for simple programmable logic controllers. In preparation; an earlier version of this paper has appeared in the Proceedings of the Euromicro Conference on Real-Time Systems that was held in York (UK) on june 9-11, 1999.

[11] P. Pushner and C. Koza. Calculating the maximum execution time of real time programs. *Real–Time Systems*, 1(2):159–176, 1989.

[12] E. Brinksma, editor. Information processing systems – Open systems interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour. Technical Report ISO/TC 97/SC 21, International Organisation for Standardization, 1987.

[13] Private communication Dr. J. M. T. Romijn.

[14] Verification of Hybrid Systems (ESPRIT project 26270), case study 1, program B2. More information on the VHS project can be obtained from `http://www-verimag.imag.fr//VHS/main.html`.

[15] R. Alur, T. Henzinger, and E. Sontag, editors. *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*. Springer, 1996.

A Abbreviations and acronyms

ANSI	American National Standardisation Institute
ASCII	American Standard Code for Information Interchange ('plain text')
AUT	Caesar/Aldebaran (.aut) format
CADP	Caesar/Aldebaran Development Package
EAG	Extended Affix Grammar
IEC	International Electro-technical Commission
IL	Instruction List
LOTOS	Language Of Temporal Ordering Specification
LTS	Labeled Transition System
PC	Personal Computer
PLC	Programmable Logic Controller
SFC	Sequential Function Chart
ST	Structured Text
TA	Timed Automaton (.ta) language of Uppaal
TON	On-delay timer
Uppaal	Model checker developed by the universities of Uppsala and Aalborg