Synthesizing Controllers from Real-Time Specifications

Henning Dierks

Abstract—We present an algorithm for synthesizing real-time controllers specified in a subset of the interval temporal logic duration calculus. The synthesized controllers are given in terms of programmable logic controller (PLC)-automata, which are an abstract description of programs of polling machines. PLC-automata can be implemented directly on PLC’s, a special kind of polling real-time controllers that are often used in industry to control production cells and batch processes. We prove the correctness of the algorithm by the duration calculus semantics for PLC-automata. Furthermore, the set of specifications on which the algorithm terminates with a well-formed PLC-automaton coincides with the set of specifications that are implementable in principle. Hence, the algorithm also decides whether a specification given in this subset of the duration calculus is implementable. We demonstrate the behavior of the algorithm by an example and apply the algorithm to the well-known “gasburner” case study.

Index Terms—Controller synthesis, duration calculus, embedded systems, formal methods, formal verification, programmable logic controller (PLC), timing verification.

I. INTRODUCTION

Today’s methods for proving correctness of programs for embedded real-time systems can roughly be classified into three categories. Model checking can be applied when both a specification and a proposed implementation are available, whereas refinement and synthesis construct—interactively or automatically—implementations that are guaranteed to be correct with respect to a given specification. Of these three methods, model checking places the greatest burden on the programmer in the sense that it provides no assistance with arriving at an implementation. Automatic synthesis is, of course, the most desirable alternative, but it applies only to a restricted class of problems.

The well-known ProCoS project [1] advocates a refinement approach that starts from a problem-oriented specification language (the duration calculus [2]) and proceeds through various intermediate languages to implementations in either hardware or transputer machine code. The first stage of transformations is performed entirely within the duration calculus (a logic for reasoning about interval functions). If successful, it reduces the given specification to a certain normal form—a conjunction of so-called implementables [3]—which determines the logical architecture of the final implementation. The effectiveness of the first transformation stage has been demonstrated in a large number of nontrivial case studies, e.g., [4]–[6]. Conjunctions of implementables still need to be implemented, and two target mechanisms have been investigated, namely, clocked circuits (realized on field-programmable gate arrays) and transputer programs. Unfortunately, neither of these transformations is automatic, nor are the target mechanisms widely used for industrial embedded systems. One of the problems is that not every collection of implementables is in fact implementable because it may not be consistent.

The contribution of this paper is a synthesis algorithm that takes as input an arbitrary set of implementables, checks it for consistency, and, if successful, translates it to source code for a programmable logic controller (PLC). PLC’s are widely used for controlling critical real-time systems such as railway crossings and production lines.

Our algorithm plays an essential role for the UniForM project [7], which is a continuation of the ProCoS project. Unlike its predecessor, UniForM has industrial participation and provides tools for PLC’s [8]. The result of the algorithm in the case of a consistent specification is given in terms of a PLC-automaton [9], an automaton-like notion for PLC programs. PLC-automata can be directly translated into PLC source code, and they are equipped with a duration calculus semantics. This semantics describes the behavior of a PLC executing the source code of the PLC-automaton and allows us to prove that the synthesis result meets the specification. The most difficult task of the algorithm is to cope with the timing constraints given in the set of implementables including required stabilities and maximal reaction times.

II. DURATION CALCULUS

In this paper, we use duration calculus (DC), a dense time-interval temporal logic developed by Chaochen and others [2], [3], [10], as the predicate language to describe properties of real-time systems. This choice is mostly motivated by our previous experience and acquired fluency in this logic, but also by the convenience with which the interval and continuous time aspects of DC allow us to express and reason about reaction times of components and durations of states.

A. Motivation

We consider the gasburner case study of the ProCoS project to illustrate the usage of DC as a high-level specification.
language. The gasburner [11] is triggered by a thermostat; it can directly control a gas valve and monitor the flame (Fig. 1).

This physical system is modeled by three Boolean observables: “hr” (heat request) represents the state of the thermostat, “fl” (flame) represents the presence of a flame at the gas valve, and “gas” represents the state of the gas valve. One of the top-level requirements is that in every period shorter than 30 s, gas must not leak for more than 4 s.

This is expressed by the following DC formula:

\[
\neg \square (\ell \leq 30) \Rightarrow \int (gas \land \neg fl) \leq 4
\] (1)

Note that the integral \( \int \) operator accumulates all durations of leaks (\( \equiv gas \land \neg fl \)) over a given interval. Hence, (1) can be read as follows: for every interval \( \square \) that is longer than 30 s (\( \ell > 30 \)), the sum of all leak durations within that interval is at most 4 s. (\( \int (gas \land \neg fl) \leq 4 \)).

The \( \int \) operator is the main advantage of the DC because it allows one to reason about the sum of specific durations, which is not possible in other real-time logics like TCTL [12] for timed automata [21]. Thus, it is not difficult to specify the quasi-fairness of two processes \( P_1 \) and \( P_2 \), which are to enter a critical section \( cS_1 \) and \( cS_2 \), respectively, in a mutually exclusive way

\[
\neg \square (\ell \geq 10) \Rightarrow \left| \int (P_1 = cS_1) - \int (P_2 = cS_2) \right| \leq \frac{\ell}{10}
\]

This formula forbids that one process occupy its critical section substantially longer (i.e., 10%) than the other one during an interval of at least 10 s. It says that in every interval \( \square \) that is longer than 10 s (\( \ell \geq 10 \)), the distance \( \{ \cdots \cdots \} \) between the occupation times of \( P_1 \) and \( P_2 \) (\( \int P_i = cS_i \), \( i = 1, 2 \)) is at most 10% of the time (\( \leq (\ell/10) \)).

Mutual exclusion is simply specified by

\[
\int (P_1 = cS_1 \land P_2 = cS_2) = 0
\]

which means that the accumulated duration of the simultaneous occupations is zero. Note that this formula means that \( P_1 \) and \( P_2 \) are in their critical section simultaneously for at most finitely many points in every interval. Due to the integration, finitely many points do not play a role for the validity of a DC formula.

To specify low-level requirements for real-time systems, a subset of DC called “implementables” was developed. This subset has been introduced in [3] as a stepping stone for the specification of distributed controllers. A fully developed theory can be found in [3] on how to obtain implementables from general DC formulas. With the help of these formulas, one could formally specify the following informal operational requirements for the gasburner.

- Start with \( \neg gas \).
- Hold \( \neg gas \) reading \( \{ \neg hr \} \).
- Hold \( \neg gas \) for 30 s.
- Leave \( \neg gas \) reading \( \{ hr \} \) within 30 + \( \varepsilon' \) s.

These requirements constrain the \( \neg gas \) phases of the system: we want the system to start with \( \neg gas \) and keep \( \neg gas \) as long as no heat request occurs. Moreover, we want the system to keep \( \neg gas \) for at least 30 s. The final requirement expresses that we want to react to a heat request by leaving \( \neg gas \) and that this has to happen within 30 + \( \varepsilon' \) s. Note that the last requirement specifies an upper bound for the reaction time, and this upper bound must not be less than 30 s, otherwise this could cause a conflict with the third requirement.

In the ProCoS project, a high-level specification in DC was refined by logical reasoning into implementables. Afterwards, logical reasoning was used to refine these implementables into implementations via several intermediate languages without tool support. The contribution of this paper is that after reaching the level of implementables, now automatic procedures are applicable to get an implementation.

**B. Syntax**

Formally, the syntax of duration calculus distinguishes terms, duration terms, and duration formulas. Terms \( \tau \) have a certain type and are built from time-dependent observables \( obs \) like gas or track, rigid variables \( x \) representing time-independent variables, and are closed under typed operators \( op \)

\[
\tau ::= obs[x] \mid op(\tau)
\]

where \( \tau \) is a vector of terms. Terms of Boolean type are called state assertions. We use \( S \) for a typical state assertion.

**Duration terms** \( \theta \) are of real type but their values depend on a given time interval. The simplest duration term is the symbol \( \ell \), denoting the length of the given interval. The name “duration calculus” stems from the fact that for each state assertion \( S \), there is a duration term \( \int S \) measuring the duration of \( S \), i.e., the accumulated time \( S \) holds in the given interval. Formally

\[
\theta ::= \ell \int S \mid op_{real}(\theta)
\]

where \( op_{real} \) is a real-valued operator and \( \theta \) a vector of duration terms.

**Duration formulas** denote truth values depending on a given time interval. They are built from the constants true and false, relations rel applied to duration terms, and are closed under the chop operator (denoted by “;”), propositional connectives \( op_{DC} \), and quantification \( \forall x \in [\mathbb{V}, \mathbb{Z}] \) over rigid variables \( x \). We use \( F \) for a typical duration formula

\[
F ::= rel(\theta) \mid F_1 ; F_2 \mid op_{DC}(F) \mid Q x . F
\]

where \( F \) is a vector of duration formulas.
C. Semantics

The semantics of duration calculus is based on an interpretation \( \mathcal{I} \) that assigns a fixed meaning to each observable, rigid variable and operator symbol of the language. To an observable \( \text{obs} \), the interpretation \( \mathcal{I} \) assigns a function

\[
\text{obs}_{\mathcal{I}} : \text{Time} \rightarrow D_{\text{obs}}
\]

with \( \text{Time} = \mathbb{R}_{\geq 0} \). This induces inductively the semantics of terms and hence state assertions. For a state assertion \( S \), it is a function

\[
S_{\mathcal{I}} : \text{Time} \rightarrow \text{Bool}
\]

where \( \text{Bool} \) is identified with the set \{0, 1\}.

The semantics of a duration term \( \theta \) is denoted by \( \mathcal{I}(\theta) \) and yields a real value depending on a given time interval \([b, c] \subseteq \text{Time}\). In particular, \( \ell \) denotes the length of \([b, c]\) and \( \int S \) the duration of the state assertion \( S \) in \([b, c]\) as given by the integral. Formally

\[
\mathcal{I}(\ell)[b, c] = c - b \\
\mathcal{I}(\int S)[b, c] = \int_b^c S_{\mathcal{I}}(t) \, dt.
\]

The semantics of a duration formula \( F \) denotes a truth value depending on \( \mathcal{I} \) and a given time interval \([b, c]\). We write \( \mathcal{I}, [b, c] \models F \) if that truth value is true for \( \mathcal{I} \) and \([b, c]\). The definition is by induction on the structure of \( F \). The cases of relations, propositional connectives, and quantification are handled as usual. For example, \( \mathcal{I}, [b, c] \models \int S \leq k \) if the duration \( \int_b^c S_{\mathcal{I}}(t) \, dt \) is at most \( k \). For \( F_1 ; F_2 \) (read as \( F_1 \) chop \( F_2 \)), we define \( \mathcal{I}, [b, c] \models F_1 ; F_2 \) if the interval \([b, c]\) can be “chopped” into two subintervals \([b, m]\) and \([m, c]\) such that \( \mathcal{I}, [b, m] \models F_1 \) and \( \mathcal{I}, [m, c] \models F_2 \).

Since in our application to the design of real-time systems the initial values of observables are important, we especially consider time intervals starting at time zero and define the following: a duration formula \( F \) holds in an interpretation \( \mathcal{I} \) if \( \mathcal{I}, [0, t] \models F \) for all \( t \in \text{Time} \). To formalize requirements in \( \text{DC} \), one states a number of suitable duration formulas and considers all interpretations for which the conjunction of the \( \text{DC} \) formulas holds in this sense.

D. Abbreviations and Precedence Rules

Besides this basic syntax, various abbreviations are used for duration formulas

- state assertions \( v \), \( \square v \) \( \allowbreak \Delta v \)
- point interval \( [\cdot] \)
- everywhere \( [P] \)
- somewhere \( \Diamond F \)
- always \( \Box F \)
- followed-by \( F \rightarrow [P] \)
- timed leads-to \( F \overset{\ell}{\rightarrow} [P] \)
- timed up-to \( F \overset{\leq \ell}{\rightarrow} [P] \)

To avoid parentheses, the following precedence rules are used:

1) \( \bar{f} \);  2) real operators;  3) real predicates;  4) \( \neg, \Box, \Diamond \);  5) \( ; \);  6) \( \wedge, \vee \);  7) \( \Rightarrow, \rightarrow, \Leftarrow, \rightarrow, \Leftarrow, \rightarrow \);  8) quantification.

III. IMPLEMENTABLES

One result of the ProCoS project is that a certain subset of \( \text{DC} \) formulas is useful to specify the behavior of controllers. These formulas are called implementables [3]. They speak about an input observable with domain \( \Sigma \) and an output observable with a finite domain \( \Pi \). The name “implementables” stems from the intuition that these kinds of formulas are more operational than general \( \text{DC} \) formulas and therefore are nearer to an implementation. In this paper, we prove this intuition to be right since we construct an implementable system that fulfills a given set of implementables provided that the specification is implementable at all. We explain the implementables using the following example.

Example III.1: Consider a sensor for a track segment with output \( \{\text{n}, t\} \), where \( \text{n} \) stands for “no train detected” and \( t \) stands for “train detected.” We want to build a system that recognizes trains and raises an error whenever a train remains in the track segment for longer than 10 s. The output of the system should be \( \{\text{n}, \text{t}, \text{e}\} \), standing for “no train on track,” “train on track,” and “error.”

The following types of formulas are “implementables.”

1) Initialization:

\[
[\cdot] \lor [\pi_0]; \text{true}
\]

with \( \pi_0 \in \Pi \) says that the output is initially \( \pi_0 \). We want our example system to start in state \( \text{n} \). Thus, we specify \( [\cdot] \lor [\text{n}]; \text{true} \).

2) Sequencing:

\[
[\pi] \rightarrow [\pi \lor \Gamma]
\]

with \( \pi \in \Pi \) and \( \Gamma \subseteq \Pi \) postulates that if the output is \( \pi \), it can only change to an output in \( \Gamma \). In the example, we want that after \( \text{n} \) only \( \text{t} \) is possible: \( [\text{n}] \rightarrow [\text{n} \lor \{\text{t}\}] \). Furthermore, we wish that the \( \text{e} \) state holds forever: \( [\text{e}] \rightarrow [\text{e} \lor \emptyset] \). Last, we specify that all states are possible when leaving \( \text{t} \): \( [\text{t}] \rightarrow [\text{t} \lor \{\text{n}, \text{e}\}] \).

3) Unbounded Stability:

\[
[\neg \pi]; \pi \land \phi \rightarrow [\pi \lor \Phi]
\]
with \( \pi \in \Pi \), \( \emptyset \neq \phi \subseteq \Sigma \) and \( \Phi \subseteq \Pi \) says that if all read inputs since the change of the output to \( \pi \) are in \( \phi \), then the output can only change to an output in \( \Phi \). In the example, we can use this to specify that \( N \) holds as long as \( n \) is read \( ([\lnot N] ; [N \land \{n\}] \rightarrow [N \lor \emptyset]) \) and to specify that \( E \) is not possible after \( T \) if only \( n \) is read: \( [\lnot T] ; [T \land \{n\}] \rightarrow [T \lor \{E\}] \).

Furthermore, we want that in state \( T \) only changes to \( E \) are possible if only \( t \) is read: \( [\lnot T] ; [T \land \{t\}] \rightarrow [N \lor \{E\}] \). By these three examples, it becomes clear that this kind of formula can be used in two ways: either we specify that we want to keep a state provided a certain input \( \phi \) holds (i.e., an unbounded stability that may end when the input is \( \lnot \phi \), like in the first example) or we constrain the set of successors of the current state provided input \( \phi \) holds (like the latter examples).

4) **Bounded Stability:**

\[
[\lnot \pi] ; [\pi \land \psi] \xrightarrow{\text{s}} [\pi \lor \psi]
\]

with \( \pi \in \Pi \), \( t > 0 \), \( \emptyset \neq \psi \subseteq \Sigma \) and \( \Psi \subseteq \Pi \). This restricts changes from \( \pi \) like the unbounded stability, but just for the first \( t \) after the change to \( \pi \). This implementable can be used to require that \( T \) hold at least 10 s if only \( t \) is signaled by the sensor: \( [\lnot T] ; [T \land \{t\}] \xrightarrow{\leq 10} [T \lor \emptyset] \). With the help of bounded stabilities, we can constrain the minimal amount of time an output state is kept provided a certain input holds.

5) **Synchronization:**

\[
[\pi \land \varphi] \xrightarrow{\text{s}} [\lnot \pi]
\]

with \( \pi \in \Pi \), \( s > 0 \), and \( \emptyset \neq \varphi \subseteq \Sigma \) says that if the input is in \( \varphi \) for \( s \) s while the output is \( \pi \), then the output changes. In the example, we want our system to react after at most \( \gamma \) s for some parameter \( \gamma > 0 \). Therefore, we specify that we leave \( N \) (respectively, \( T \)) when reading \( t \) (respectively, \( n \)): \( [N \land \{t\}] \xrightarrow{\gamma} [\lnot N] \) and \( [T \land \{n\}] \xrightarrow{\gamma} [\lnot N] \). Furthermore, we want to leave \( T \) after at most \( 10 + \gamma \): \( [T \land \{n, t\}] \xrightarrow{\leq 10 + \gamma} [\lnot T] \).

With synchronizations, we are able to specify upper bounds for the reaction time of a system in a certain state reading a certain input.

Since these implementables themselves can serve as a specification language, it is useful to provide a little bit of "syntactic sugar" that makes a specification more readable. To this end, one can think of the following representation of the implementables above (in the same order).

- **Start with \( N \).**
- **\( N \)-successors are: \{\( T \)\}.**
- **\( E \) has no successors.**
- **\( T \)-successors are: \{\( N, E \)\}.**
- **Hold \( N \) reading \{\( n \)\}.**
- **\( T \)-successors reading \( n \) are: \{\( N \)\}.**
- **\( T \)-successors reading \( t \) are: \{\( E \)\}.**
- **Hold \( T \) reading \{\( t \)\} for 10 s.**
- **Leave \( N \) reading \{\( t \)\} within \( \gamma \) s.**
- **Leave \( T \) reading \{\( n \)\} within \( \gamma \) s.**
- **Leave \( T \) reading \{\( n, t \)\} within \( 10 + \gamma \) s.**

IV. PLC-AUTOMATA

In the UniForM project [7], an automaton-like notion of polling real-time systems has been developed to enable a formal verification of PLC programs. Basically, PLC’s, the hardware aim of the project, can be viewed as simple computers with a special real-time operating system. They have features for making the design of time- and safety-critical systems easier.

- PLC’s have input and output channels where sensors and actuators, respectively, can be plugged in.
- They behave in a cyclic manner where every cycle consists of the following phases.
  - Poll all inputs and store the read values.
  - Compute the new values for the outputs.
  - Update all outputs.

The repeated execution of this cycle is managed by the operating system. The only part the programmer has to adapt is the computing phase.

- Depending on the program and on the number of inputs and outputs, there is an upper time bound for a cycle that can be used to calculate an upper time bound for the reaction time.
- Convenient standardized libraries are given to simplify the handling of time.

In the UniForM project [7], we developed automaton-like pictures which can serve as a common basis for computer scientists and engineers. The latter have an intuitive understanding of these pictures. Therefore, we formalized the notion of “PLC-automaton” and defined a formal semantics for it in \( \mathcal{DC} \) (see Appendix I) that is consistent with the translation of PLC-automata into PLC source code. But PLC-automata are not especially tailored to PLC’s. In fact, PLC-automata are an abstract representation of a machine that periodically polls the input and has the possibility of measuring time. In [9], a compilation function to PLC source code is given.

Fig. 2 gives an example of a PLC-automaton. It shows an implementation of Example III.1 with three states \{\( q_0, q_1, q_2 \)\} and outputs \{\( N, T, E \)\} that reacts to inputs of the alphabet \{\( n, t \)\}. Every state has two annotations in the graphical representation. The upper one denotes the output of the state. Thus in state \( q_0 \) the output is \( N \), and in state \( q_2 \) the output is \( E \), denoting an “error.” The lower annotation is either zero or a pair consisting of a real number \( d > 0 \) and a nonempty subset \( S \) of inputs.

The operational behavior is as follows: if the second annotation of a state \( q \) is zero, the PLC-automaton reacts in every cycle to the inputs that are read and behaves according to the transition relation. If the second annotation of \( q \) is a pair \((d, S)\), the PLC-automaton checks in every cycle the input \( i \)
according to these parameters. If $i$ is not in $S$, the automaton reacts immediately according to the transition relation. If $i$ is in $S$ and the current state does not hold longer than $\delta$, the input will be ignored and the automaton remains in state $q$. If $i$ is in $S$ and state $q$ is held longer than $\delta$, the PLC-automaton will react to $i$ according to the transition relation.

The PLC-automaton in Fig. 2 thus behaves as follows: it starts in state $q_0$ and remains there as long as it reads only the input $\mathbf{n}$. The first time it reads $\mathbf{t}$, it changes to state $q_1$. In $q_1$, the automaton reacts to the input $\mathbf{n}$ by changing the state back to $q_0$ independently of the time it stayed in state $q_1$. It reacts to the input $\mathbf{t}$ by changing the state to $q_2$ provided that $q_1$ holds longer than $10$ s. If this transition takes place, the automaton enters $q_2$ and remains there forever. Hence we know that the automaton changes its output to $\mathbf{E}$ when $\mathbf{t}$ holds a little bit longer than $10$ s (the cycle time has to be considered).

We formalize this graphic notation using an automaton-like structure extended by some components.

**Definition IV.1:** A tuple $A = (Q, \Sigma, \delta, q_0, \varepsilon, S_t, S_e, \Omega, \omega)$ is a PLC-automaton if:

- $Q$ is a nonempty, finite set of states;
- $\Sigma$ is a nonempty, finite set of inputs;
- $\delta$ is a function of type $Q \times \Sigma \rightarrow Q$ (transition function);
- $q_0 \in Q$ is the initial state;
- $\varepsilon > 0$ is the upper time bound for a cycle;
- $S_t$ is a function of type $Q \rightarrow \mathbb{R}_{\geq 0}$ assigning to each state $q$ a delay time, how long the inputs contained in $S_e(q)$ should be ignored;
- $S_e$ is a function of type $Q \rightarrow \mathcal{P}(\Sigma) \setminus \{\emptyset\}$ assigning to each state a set of delayed inputs [If $S_i(q) = \emptyset$, the set $S_e(q)$ can be arbitrarily chosen] (output function);
- $\omega$ is a function of type $Q \rightarrow \Omega$ (output function); and it holds for all $q \in Q$ and $a \in \Sigma$.

$$S_t(q) > 0 \land a \notin S_e(q) \Rightarrow \delta(q, a) \neq q. \quad (2)$$

The components $Q$, $\Sigma$, $\delta$, and $q_0$ are as usual for usual finite-state automata. The additional components are needed to model PLC behavior and to enrich the language for dealing with real-time aspects. The $\varepsilon$ represents the upper time bound for a cycle of a PLC and enables us to model this cycle in the semantics. The delay functions $S_t$ and $S_e$ represent the annotations of the states. In the case of $S_t(q) = 0$, no delay time is given and the value $S_e(q)$ is arbitrary. If the delay time $S_t(q)$ is greater than zero, the set $S_e(q)$ denotes the set of inputs for which the delay time is valid. The restriction (2) is introduced to enable the application of the compilation schema given in [9]. Note that this is no restriction to the expressiveness of a PLC-automaton: only loops are affected by (2). If we have a loop—say, $\delta(q, a) = q$—we want the system to hold $q$ when reading input $a$. If we add $a$ to the set $S_e(q)$, the difference is that the system will ignore the loop for the first $S_t(q)$ s, which is equivalent to executing the loop.

### V. THE SYNTHESIS ALGORITHM

In this section, we present an algorithm that computes a PLC-automaton from a set of implementables provided that this set is consistent. We demonstrate the operations of the algorithm by Example III.1 with the set of implementables given in Section III.

The idea of the algorithm is that we collect for each output state $\pi$ the implementables speaking about $\pi$. We sort the time bounds appearing in all bounded stabilities for $\pi$ and build a cascade of internal automaton states, all with output $\pi$. The first internal state of the cascade has to consider all bounded stabilities, the second one only those bounded stabilities with bounds that are not the smallest one, etc. The last internal state of the cascade has no bounded stability to consider.

After building this cascade, we build up two tables. The first table represents the possible transitions in an internal state depending on the read input. The second table contains the information if an input is a delayed input in an internal state. All implementables are now used to make changes in the table in order to ensure that the cascade fulfills the requirements. Thus, the implementables are translated into requirements for the tables (respectively, for the cascade). Last, we compute for $\pi$ an upper bound of the cycle time from the timed constraints.

In more detail, the algorithm works as follows: except for the initialization, all implementables speak about one certain state $\pi$ of a controller. Hence we construct a part of a PLC-automaton that implements the behavior of the state $\pi$. Assume that the specification of the controller expressed in implementables contains these formulas for $\pi$

$$\forall i \in I: [\neg \pi]; [\pi \land \phi_i] \quad \Rightarrow \quad [\pi \lor \phi_i]$$

$$\forall j \in J: [\neg \pi]; [\pi \land \psi_j] \quad \Rightarrow \quad [\pi \lor \psi_j]$$

$$\forall k \in K: [\pi \land \varphi_k] \quad \Rightarrow \quad [\neg \pi].$$

The sets $I$, $J$, $K$ are finite (and possibly empty). We assume that $\Phi(\psi_i)$ and $\Psi(\psi_i)$ are subsets of $\pi$. Assume that $\{t_1, \cdots, t_{n(\pi)}\} = \{t^i_j | j \in J\}$ with $t_1 < t_2 < \cdots < t_{n(\pi)}$. We put $t_0 = 0$ and $t_{n(\pi)+1} = \infty$. In the example, we have $n(\pi) = n(E) = 0$ and $n(T) = 1$ with $t_1 = 10$. The part of the PLC-automaton we construct for $\pi$ is a cascade of internal states $q_{\pi, t_1}, \cdots, q_{\pi, t_{n(\pi)}}$; $q_{\pi, \infty}$. The idea is that the state $q_{\pi, t}$ represents the knowledge that $\pi$ has held at least $t-1$ s and at most $t + 2\varepsilon$ s with $\varepsilon > 0$ representing the cycle time of the PLC-automaton. Note that in the worst case it lasts one cycle to recognize that the delay time is up and another cycle to react to the input. Hence in the worst case, we pick up in each internal state of the cascade $2\varepsilon$ time units.

Whenver the PLC-automaton changes its output to $\pi$, it starts in the state $q_{\pi, t_1}$, and for all $q_{\pi, t_i}$ with $t_i \neq \infty$, transitions to a $q_{\pi, t_j}$ with $j \leq i$ or $j > i + 1$ are not allowed. For $q_{\pi, \infty}$, transitions to a $q_{\pi, t_j}$ with $j < n(\pi)$ are forbidden. Hence the skeleton of this part of the PLC-automaton looks as follows (with $n(\pi)$)}:
We now have to determine how the transition function works and which inputs are delayed.

**Step 0:** In this step, we consider \([\pi] \rightarrow [\pi \lor \Gamma]\): build the following tables \(\delta^{\pi}\) and \(S^{\pi}\) [suppose that the set of inputs \(\Sigma = \{\kappa_1, \ldots, \kappa_m\}\) and let \(\Gamma = \Gamma \cup \{\pi\}\)]

The \(\delta^{\pi}(\kappa_i, t_k)\) entry represents the possible transitions from state \(q_n, t_k\) when reading the input \(\kappa_i\). The \(S^{\pi}(\kappa_i, t_k)\) entry contains the information whether in state \(q_n, t_k\) the input \(\kappa_i\) is delayed or not. We know that no more input is delayed after \(t_n(\pi)\) s because there is no bounded stability with a larger bound. Hence we omit a \(t_n(\pi) + 1\) column in this table. Initially, we allow only transitions to \(\Gamma_\pi\) and delay all inputs. In our example, we have these sequencings

\[
\begin{align*}
[N] & \rightarrow [N \lor \{T\}] \\
[E] & \rightarrow [E \lor \emptyset] \\
[T] & \rightarrow [T \lor \{N, E\}].
\end{align*}
\]

Hence, we get these tables

The other implementables are taken into account in the following steps.

**Step 1:** Replace for all \([\pi] \rightarrow [\pi \land \varphi]\) all entries \(\delta^{\pi}(\kappa_i, t_j)\) in the \(\delta^{\pi}\) table by \(\delta^{\pi}(\kappa_i, t_j) \land \varphi(\kappa_i)\). This avoids transition from \(\pi\) to a state not in \(\varphi\) when reading an input in \(\varphi\). In the example, we have three implementables of this kind

\[
\begin{align*}
[N \land \{n\}] & \rightarrow [N \lor \emptyset] \\
[T \land \{n\}] & \rightarrow [T \lor \{N\}] \\
[T \land \{t\}] & \rightarrow [T \lor \{E\}].
\end{align*}
\]

This step yields the following \(\delta^{\pi}\) tables of Example III.1.

**Step 2:** Replace for all \([\pi] \rightarrow [\pi \land \psi]\) all entries \(\delta^{\pi}(\kappa_i, t_j)\) in the \(\delta^{\pi}\) table by \(\delta^{\pi}(\kappa_i, t_j) \land \psi(\kappa_i)\). This avoids transition from \(\pi\) to a state not in \(\psi\) when reading an input in \(\psi\). In the example, we have only one bounded stability

\[
[\neg T]; [T \land \{t\}] \leq 10 [T \lor \emptyset].
\]

Thus, we get now

**Step 3:** Replace for all \([\pi \land \varphi_k] \rightarrow [\pi]\) and for all \(\kappa_i \in \varphi_k\) all entries \(\delta^{\pi}(\kappa_i, t_j)\) in the \(\delta^{\pi}\) table by \(\delta^{\pi}(\kappa_i, t_j) \land \varphi(\kappa_i)\) provided that \(s_k \leq t_j\). This avoids that the state \(\pi\) is held reading input \(\kappa_i\) in the case that too much time has already elapsed. Three synchronizations are given in the example

\[
[N \land \{t\}] \rightarrow [\neg N] \\
[T \land \{n\}] \rightarrow [\neg T] \\
[T \land \{n, t\}] \rightarrow [\neg T].
\]

Now the \(\delta^{\pi}\) tables for the example look like this (assuming that \(\gamma \leq 10\))

**Step 4:** Last, set for all \([\pi \land \varphi_k] \rightarrow [\pi]\) and for all \(\kappa_i \in \varphi_k\) the \(S^{\pi}(\kappa_i, t_j)\) entry in the \(S^{\pi}\) table to false provided that \(s_k \leq t_j\). This is the only step that modifies the \(S^{\pi}\) table. It avoids that an input in \(\varphi_k\) is delayed after \(s_k\) s have elapsed. So we get

Note that except for step 0, the order of the other steps is arbitrary.

**Lemma V.1:** For all \(\pi\) and for all \(\kappa_i, t_k\), the following holds: \(\pi \in \delta^{\pi}(\kappa_i, t_k) \iff S^{\pi}(\kappa_i, t_k) = \text{true} \).

This lemma is obvious due to the construction steps (3) and (4). It guarantees that a transition from \(q_n, t_k\) to \(q_{n+1}, t_{k+1}\) is only possible after the delay time of \(q_n, t_k\). This preserves our intuition that in state \(q_n, t_k\) the output \(\pi\) held at least \(t_k\) s and at most \(t_k + 2\varepsilon\) s. 
We define now $\xi(\pi) \in \mathbb{R}_{\geq 0}$, which represents the upper bound for the cycle time of the PLC-automaton to guarantee the timing constraints for $\pi$. It is the maximal number $\bar{\xi}$ such that

$$
\xi \leq \frac{s_k}{2}
$$

holds for all $k \in K$; and for all $j \in J$ with $s_k > t_j$, it holds

$$
\xi \leq \frac{s_k - t_j}{2j + 1}.
$$

If $K = \emptyset$, we set $\xi(\pi) = \infty$. Bound (3) is needed because in the worst case, a PLC-automaton needs two cycles to react to an input. In the presence of bounded stabilities, we have to take into account that state $q_{n, t_j}$ only represents the knowledge that $\pi$ held at most $t_j + 2j$ s. In this case, the automaton needs one more cycle to react, which leads us to the inequality $t_j + (2j + 1)\varepsilon \leq s_k$ in order to react in time. Hence we get bound (4). In the example, we have $\varepsilon(\Pi) = \gamma/2$, $\varepsilon(\Sigma) = \gamma/3$, and $\varepsilon(\Pi) = \infty$. Note that $\xi(\pi) > 0$ always holds.

Hence we get for each state $\pi$ a pair of tables $\mathcal{S}$ and $\mathcal{S}_c$. We now construct a PLC-automaton out of these tables supposing that $\Pi$ is the set of output states the implementables are speaking about.

**Definition V.2:** Given a set $\Pi$ of output states of a controller and a set $SPEC$ of implementables speaking about $\Pi$ using the input observable $\text{input}$, we call each PLC-automaton $A = (Q, \Sigma, \delta, q_0, \varepsilon, S_t, S_c, \Pi, \omega)$ a SPEC-automaton if it has the following properties:

$$Q = \{q_{n, t_j} | \pi \in \Pi, 1 \leq j \leq n(\pi) + 1\}$$

$$\varepsilon < \min\{\varepsilon(\pi) | \pi \in \Pi\}$$

$$S_t(q_{n, t_j}) = \begin{cases} 0, & j = n(\pi) + 1 \\ \{t_j - t_j, \delta, 1 \leq j \leq n(\pi)\} \\ \{s \in \Sigma | S_c(q_{n, t_j}), t_j = \text{true}\} \end{cases}$$

$$\omega(q_{n, t_j}) = \pi.$$ 

The transition relation $\delta$ has to fulfill the following: for all $\delta_k \in \Sigma$ and for all $\pi \in \Pi$, $1 \leq j \leq n(\pi) + 1$ and $q_{n, t_j} = \delta(q_{n, t_j}, \delta_k)$ holds

$$\pi' \in \delta(\pi, \delta_k, t_j)$$

$$\bigwedge_{\pi \neq \pi'} \Rightarrow j' = 1$$

$$\bigwedge_{\pi = \pi'} \Rightarrow (j' = j + 1 \lor j' = j = n(\pi) + 1).$$

If there is an initial constraint $[\pi] \lor [\pi_0]; \text{true}$ in $SPEC$, the initial state $q_0$ is $q_{\pi_0, t_1}$. Otherwise, $q_0$ is in $\{q_{n, t_1} | \pi \in \Pi\}$. □

In the rest of this section, we will define a consistent specification (Definition V.3) and show that for every consistent specification SPEC, a SPEC-automaton exists (Theorem V.4). Last, Theorem V.5 shows that every SPEC-automaton satisfies SPEC. Fig. 3 shows an automaton with cycle time $\varepsilon \leq \gamma/3$, which is obtained by the above synthesis algorithm from the implementables of Section III. Note that this automaton differs slightly from the automaton in Fig. 2. It is not guaranteed that the algorithm produces an optimal result in the sense that there may be a PLC-automaton with less states and greater $\varepsilon$. But the states $q_{\pi, \infty}$ are the only possible superfluous states introduced by the algorithm. All other superfluous states are introduced by the specification itself when containing superfluous bounded stabilities.

**Definition V.3:** We call a specification by implementables consistent if all initialization requirements are identical and for every synchronization $[\pi \land \phi] \rightarrow [\pi; \phi]$, the following holds: for any input $i \in \phi$, there is a state $\pi' \neq \pi$ such that:

- for every $[\pi] \rightarrow [\pi \lor \Gamma]$, it holds $\pi' \in \Gamma$;
- for every $[\neg \pi]; [\pi \land \phi] \rightarrow [\pi \lor \Phi(\phi)]$, it holds $i \in \phi \Rightarrow \pi' \in \Phi(\phi)$;
- for every $[\neg \pi]; [\pi \land \psi] \rightarrow [\pi \lor \Psi(\psi)]$, it holds $i \in \psi \land s \Rightarrow \pi' \in \Psi(\psi)$.

To justify this definition, we should explain why specifications that are not consistent in the sense of Definition V.3 are problematic. It is clear why specifications with at least two nonidentical initializations cause problems: they contradict each other. In the case of a synchronization requirement $[\pi \land \phi] \rightarrow [\pi; \phi]$ with $i \in \phi$, we have to leave the output state $\pi$. What is needed is a succeeding output state $\pi' \neq \pi$, which may depend on $i$. Basically, the second part of Definition V.3 says that in the case of a synchronization regarding $\pi$ and $i$, not all other $\pi' \neq \pi$ are forbidden by the specification. In other words, if the specification requires a change, then it allows at least one direction.

**Theorem V.4:** For each consistent specification SPEC of a controller that speaks about $\Pi$ with the input observable input, there exists a SPEC-automaton.

**Proof:** Only three things have to be checked: if restriction (2) holds if $\varepsilon > 0$ and if for every $\pi, n_k$, and $t_k$ is the $\delta(\pi, \delta_k, t_k)$ entry not empty. Restriction (2) is fulfilled due to the definition of $\delta$ and $S_t$ in Definition V.2: loops at a state $q_{n, t_j}$ are only possible when $t_j = \infty$, but in this case, $S_t(q_{n, t_j}) = 0$ holds. The second obligation is trivial by the definition of $\varepsilon(\pi)$, which is always greater than zero for all $\pi \in \Pi$. Therefore, the minimum of these bounds is also greater than zero because we have a finite set of output states. The last obligation can be seen as follows: suppose $\pi \notin \delta(\pi, t_k)$. Then we know that there was a synchronization constraint $[\pi \land \phi] \rightarrow [\pi; \phi]$ with $k \in \phi$ and $s \leq t_k$. By the assumption that SPEC is consistent, we get a $\pi'$ that is by construction in $\delta(\pi, t_k)$. □

**Theorem V.5:** Given a consistent specification SPEC of a controller with the input observable input and the set of output states $\Pi$, any SPEC-automaton $A$ implements SPEC.

A sketch of the proof can be found in Appendix II.

VI. THE CASE STUDY “GASBURNER”

Consider the gasburner of Section II-A again. In the ProCoS project [1], this case study was transformed first from duration
calculus into implementables. This yielded a specification of a controller using four states—idle (id), purge (pg), ignite (ig), and burn (bn)—fulfilling the following assertions with a $\varepsilon' > 0$ as shown in (4a) at the bottom of the page. The gas valve should be opened iff $\text{state}$ is in $\{\text{bn}, \text{ig}\}$.

In the ProCoS project, this specification was transformed via several steps and interfacing languages to hardware. Here we need not perform these transformations. It is sufficient to apply the algorithm. We show the $\delta$ tables after steps $1–3$.

<table>
<thead>
<tr>
<th>step (1)</th>
<th>id, $\infty$</th>
<th>pg, $30$</th>
<th>pg, $\infty$</th>
<th>ig, $1$</th>
<th>ig, $\infty$</th>
<th>bn, $\infty$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg hr \land \neg fl$</td>
<td>id, pg</td>
<td>pg, ig</td>
<td>pg, ig</td>
<td>ig, bn</td>
<td>ig, bn</td>
<td>bn, id</td>
</tr>
<tr>
<td>$hr \land \neg fl$</td>
<td>id, pg</td>
<td>pg, ig</td>
<td>pg, ig</td>
<td>ig, bn</td>
<td>ig, bn</td>
<td>bn, id</td>
</tr>
<tr>
<td>$hr \land fl$</td>
<td>id, pg</td>
<td>pg, ig</td>
<td>pg, ig</td>
<td>ig, bn</td>
<td>ig, bn</td>
<td>bn, id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>step (2)</th>
<th>id, $\infty$</th>
<th>pg, $30$</th>
<th>pg, $\infty$</th>
<th>ig, $1$</th>
<th>ig, $\infty$</th>
<th>bn, $\infty$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$hr \land \neg fl$</td>
<td>id, pg</td>
<td>pg, ig</td>
<td>pg, ig</td>
<td>ig, bn</td>
<td>ig, bn</td>
<td>bn, id</td>
</tr>
<tr>
<td>$hr \land fl$</td>
<td>id, pg</td>
<td>pg, ig</td>
<td>pg, ig</td>
<td>ig, bn</td>
<td>ig, bn</td>
<td>bn, id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>step (3)</th>
<th>id, $\infty$</th>
<th>pg, $30$</th>
<th>pg, $\infty$</th>
<th>ig, $1$</th>
<th>ig, $\infty$</th>
<th>bn, $\infty$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$hr \land \neg fl$</td>
<td>id, pg</td>
<td>pg, ig</td>
<td>pg, ig</td>
<td>ig, bn</td>
<td>ig, bn</td>
<td>bn, id</td>
</tr>
<tr>
<td>$hr \land fl$</td>
<td>id, pg</td>
<td>pg, ig</td>
<td>pg, ig</td>
<td>ig, bn</td>
<td>ig, bn</td>
<td>bn, id</td>
</tr>
</tbody>
</table>

The $S(x)$ tables are all unchanged and thus omitted here. Fig. 4 shows the result of the algorithm applied to the gasburner specification.

VII. COMPLEXITY

In this section, we discuss some complexity issues of the synthesis algorithm. To this end, we introduce the following constants:

$O$ number of output values;
$I$ number of input values;
$S$ number of implementables;
$S(\pi)$ number of implementables for output $\pi$;
$BS$ number of bounded stabilities;
$BS(\pi)$ number of bounded stabilities for output $\pi$.

From the construction of the tables and from the definition of the $SPEC$-automaton, it is obvious that the number of states in the $SPEC$-automaton is bounded by $O + BS$.

The space consumption of the algorithm depends on the size of the tables in use. Their size can be easily bounded by $(O + BS) \cdot I \cdot O$ (number of columns · number of rows · size of entries). In the case of several input variables $I_1, \ldots, I_n$ with domains $D_{1}, \ldots, D_{n}$, the number of possible input values $I$ is $\prod_{i=1}^{n} |D_i|$. If this size is a problem, one could change the algorithm in such a way that it distinguishes only input values that are distinguished by the specification. For example, in the case study “gasburner,” the specification does not distinguish any input in the context of the pg or ig output. With this idea, one could improve the algorithm in such a way that the number of rows in the tables grows with the specification. However, the number of distinguishable input values grows exponentially with the number of implementables.

The time consumption depends on the number of actions that are made in the tables for each implementable $s$ in the specification $SPEC$ speaking over $\pi(s)$. This number can be bounded as follows: the number of columns where changes possibly have to been made is bounded by $1 + BS(\pi(s))$. The number of rows that are affected is in the worst case $I$. The number of changes on the each entry is bounded by $O$. Hence we have an upper bound of actions by

$$\sum_{s \in SPEC} (1 + BS(\pi(s))) \cdot I \cdot O \leq I \cdot O \cdot \sum_{s \in SPEC} (1 + BS) = I \cdot O \cdot S \cdot (1 + BS).$$

VIII. RELATED WORK

In the following, we discuss related work grouped in three areas. The first one gives an overview about other results on PLC-automata. After that, we compare our result with other research on the synthesis of implementables. Last, some other approaches to real-time programming and verification are discussed.

```latex
[\cdot] \lor [id]; \text{true}

[id] \rightarrow [id \lor pg]

[ig] \rightarrow [ig \lor bn]

[\neg id]; [id \land \neg hr] \rightarrow [id]

[\neg pg]; [pg] \leq 30 [pg]

[pg] \rightarrow [pg \lor ig]

[bn] \rightarrow [bn \lor id]

[\neg bn]; [bn \land hr \land fl] \rightarrow [bn]

[\neg ig]; [ig] \leq 1 [ig]

[ig] \rightarrow [ig]

[bn \land \neg hr] \rightarrow [\neg bn]

[bn \land \neg fl] \rightarrow [\neg bn]
```

(4a)
A. PLC-Automata

PLC-automata have been successfully applied to both kinds of case studies: academic ones [6], [13] and ones that are not toy examples [14]. Reference [14] contains the case study of the UniForM project, which was completely designed by the synthesis we presented in this paper. The problem was to design a system that implements a mutual exclusion of trams for a single-tracked railway line on distributed PLC’s. Real-time techniques were especially needed to cope with the required reaction times of the system and to handle unreliable track sensors. The system consists of 14 automata synthesized out of 89 implementables by Moby/PLC [14]–[16]. The resulting PLC source code for all automata together consists of approximately 700 lines of code. Even bigger case studies have been implemented by PLC-automata with the help of the tool Moby/PLC, e.g., a controller for the fault-tolerant production cell [17]. Moby/PLC enables hierarchical design, verification, simulation, and compilation of PLC-automata.

Recently, another semantics in terms of timed automata has been developed [18]. This kind of semantics is more operational in the sense that the timed automaton describing the PLC-automaton models a PLC with the corresponding program. An interesting side effect is that now model checkers for timed automata are applicable to verify designs using PLC-automata. Corresponding export functions are already implemented in Moby/PLC.

B. Implementables

There are two approaches to translating implementables into other formalisms [19], [20]. Both translate the implementables in variants of timed automata. Since PLC-automata have a timed automaton semantics [18], our approach belongs also to this category. The main difference is that our result is immediately implementable by real machines, whereas purely timed automata are in general not implementable.

C. Other Real-Time Approaches

Many approaches to real-time systems have been developed in recent years. They can be categorized into formal models, which are intended to be specification languages like timed automata [21] and duration calculus [2] or to be nearer to implementation like, for example, the group of synchronous languages such as statecharts [22], [23], esterel [24], lustre [25], and signal [26]. The main difference between these synchronous approaches and PLC-automata is the absence of a “perfect synchrony hypothesis” in the latter case. That means that the synchronous languages have the possibility to produce source code from their model but are obliged afterwards to justify the hypothesis that the synchronization of events happens without the consumption of time. In the case of PLC-automata, communication is asynchronous, and the reaction time of the system to an input ranges from > 0 to ≤ 2ε. Hence after producing source code, one only has to assure that the cycle time of the system is ≤ ε in the case of PLC-automata.

IX. Conclusion and Future Work

The main advantage of the synthesis algorithm presented is that we gained a new methodology for correct design of real-time systems combining refinement and synthesis. In comparison to this, the ProCoS method would refine from top-level specifications down to transputers via many intermediate refinement steps, which is hardly applicable to realistic problems.

Currently, we are interested in improvements of the synthesis algorithm in order to cope with bigger subsets of DC formulas than implementables. Some minor generalizations are worked out already, but we think that it is more important to recognize which kinds of formulas are useful in the UniForM project than to find slight generalizations of both the algorithm and implementables.

APPENDIX I

SEMANTICS OF PLC-AUTOMATA

The duration calculus semantics \([\mathcal{A}]_{DC}\) of a PLC-automaton \(A = (Q, \Sigma, \delta, q_0, \varepsilon, S_e, S_t, \Omega, \omega)\) is given by the conjunction of the following predicates regarding the observables state : Time \(\rightarrow Q\), input : Time \(\rightarrow \Sigma\) and output : Time \(\rightarrow \Omega\). First, the starting of the automaton in the proper initial state is expressed by

\[ \lbrack \emptyset \rbrack \lor \lbrack q_0 \rbrack; \text{true}. \quad (5) \]

Note that \(\lbrack q_0 \rbrack\) is an abbreviation of \(\lbrack \text{state} = q_0 \rbrack\). Next, we want to describe the behavior of the automaton in state \(q\). The cyclic behavior of PLC’s has to be reflected in the semantics to achieve a realistic modeling. One question the semantics should answer is: When a state \(q\) is entered, what kind of input can influence the behavior of the PLC? The answer to this question is:

- only the inputs after entering \(q\);
- only the inputs during the last cycle time \(\varepsilon\).

This is expressed by the following predicates, where \(\mathcal{A}\) ranges over all sets of inputs with \(\emptyset \neq \mathcal{A} \subseteq \Sigma\):

\[ [-q]; [q \land A] \rightarrow [q \lor \delta(q, A)] \quad (6) \]

\[ [q \land A] \xrightarrow{\varepsilon} [q \lor \delta(q, A)]. \quad (7) \]

Statement (6) formalizes the fact that after a change of the automaton’s state to \(q\), only the set of inputs \(A\) that is valid after the change can have an effect on the behavior in the future. Statement (7) represents the formalization of the cyclic behavior of PLC’s. A PLC reacts only on inputs during the last cycle. Preceding inputs are forgotten and cannot influence the PLC anymore.

The quantification over all nonempty subsets of the input alphabet was motivated by the behavior of the PLC’s. The more we know about the inputs during the last cycle, the more we know about the actions of the PLC. For example, it is necessary that an input \(a\) is held for at least \(\varepsilon\) s to assure that the PLC reads this at least once. This is directly reflected in the semantics as well. If there is an interval of length \(\varepsilon\),

\[ \text{in the formula, we use } \mathcal{A} \text{ as an abbreviation for } \text{input} \in A \text{ respective to } \delta(q, a) \text{ for } \text{state} \in \{\delta(q, a)\} | a \in A. \]
predicate (7) can be applied with $A = \{a\}$ to this interval. This implies that after this interval is over, the state either changes to $\delta(q, a)$ or remains unchanged. For states without a stability requirement, we expect a change to $\delta(q, a)$ or, more generally, we expect that the automaton reacts accordingly in at most $2\varepsilon$ s. For states with a stability requirement, we expect this behavior after the required period of time. This leads us to additional statements in the semantics

$$S_t(q) = 0 \implies [q \land A] 2\varepsilon \delta(q, A)$$

(8)

$$S_t(q) > 0 \implies [q\land A] 2\varepsilon \delta(q, A)$$

(9)

$$S_t(q) = 0 \land q \notin \delta(q, A) \implies [-q\land A] \varepsilon \implies [-q].$$

(10)

Statement (8) says that after at most $2\varepsilon$ s, the automaton reacts on the input accordingly if there is no stability required for $q$. Note that $\varepsilon$ s are needed to assure that the PLC reads the input at least once, and $\varepsilon$ s are needed to react on this input in the worst case. Formula (9) states this behavior after $S_t(q)$ s: if $S_t(q)$ s have elapsed, the automaton reacts on inputs in at most $2\varepsilon$ s. In the case that we know that the automaton has just changed the state, then we want to be able to exploit the information that within the next $\varepsilon$ s another reaction on the input $A$ has to occur. This is formalized by (10).

Next we want to describe the automaton’s behavior if it is in a state $q$ where a stability is required and the $S_t(q)$ s have not elapsed. Then we want to hold this state provided that during this phase only inputs in $S_v(q)$ are read. That means inputs in $S_v(q)$ cannot cause a change of state during the first $S_t(q)$ s

$$S_t(q) > 0 \implies [-q]; [q\land A] \varepsilon \delta(q, A) S_v(q)]].$$

(11)

However, we have to take into account the cyclic behavior of the hardware again. In particular, we should require that if $q$ is left during the stability phase, then there has to be an input not contained in $S_v(q)$ at most $\varepsilon$ s ago

$$S_t(q) > 0 \implies [-q]; [q\land A] \varepsilon \delta(q, A) S_v(q)]].$$

(12)

Furthermore, we know that the automaton reacts according to the input if there is a set $A$ that is valid for the last $2\varepsilon$ s and disjoint from $S_v(q)$

$$S_t(q) > 0 \land A \cap S_v(q) = \emptyset \implies [q \land A] 2\varepsilon \delta(q, A)$$

(13)

$$\implies [-q]; [q\land A] \varepsilon \implies [-q].$$

(14)

Formula (14) corresponds to (10). Due to (2), we know $q \notin \delta(q, A)$. Note that (6), (10)–(12), and (14) require a change from $[-q]$ to $[q]$ to restrict the possible behavior. But for the initial state, there is no change, and therefore the assertions are not applicable in this case. This can be expressed by five corresponding assertions suitable for the initial state, which for brevity are omitted here as they do not provide any new insight into the PLC-automaton.

Last, the relation between the observables $state$ and $output$ is established by

$$\square([q] \implies [\omega(q)]).$$

(15)

We state two simple properties that are easy to see for a PLC-automaton $A = (Q, \Sigma, \delta, q_0, \varepsilon, S_t, S_v, \Omega, \omega)$ with $\pi \in \Omega$ and $\emptyset \neq \varphi \subseteq \Sigma$.

**Lemma A.1:** $[\pi] \implies [\pi \lor \omega(\delta(\pi^{-1}(\pi), \Sigma))].$

**Lemma A.2:** $[\pi \land \varphi] \implies [\pi \lor \omega(\delta(\pi^{-1}(\pi), \varphi))].$

### APPENDIX II

**CORRECTNESS OF THE ALGORITHM (THEOREM V.5)**

#### A. Sketch of the Proof

1. **Initialization:** This is obvious by the construction of $A$ and by (5) and (15).

2. **Sequencing:** Suppose $SPEC$ contains a sequencing constraint $[\pi] \implies [\pi \lor \Gamma]$. We initialized the $\delta^\pi$ table with $\Gamma_\pi$ and no step adds entries. Hence by Definition V.2 and Lemma A.1 we know that this is true.

3. **Unbounded Stability:** Suppose $SPEC$ contains an unbounded stability. By step 1 and with Lemma A.2, we can conclude that this also holds.

4. **Bounded Stability:** Suppose that $SPEC$ contains a bounded stability $[-\pi]; [\pi \land \varphi] \varepsilon S_v \pi \lor \Psi(\varphi)$. By step 2, we know that only changes to $\Psi(\varphi)$ are allowed for the first states $q_{\pi, t_1}, \ldots, q_{\pi, t_k}$ with $t_k = \emptyset$ when reading inputs in $\psi$. By Lemma V.1, we know that the transition from $q_{\pi, t}$ to $q_{\pi, t_{k+1}}$ can only happen after $S_t(q_{\pi, t})$ s. By Definition V.2, we know that $\sum_{i=1}^{k} S_t(q_{\pi, t_i}) = t$ holds.

5. **Synchronization:** Suppose $SPEC$ contains a synchronization $[\pi \land \varphi] \varepsilon [-\pi]$. The negation of this requirement is:

$$true; [\pi \land \varphi] \varepsilon [-\pi].$$

(16)

By finite variability and (15), we can find a finite sequence $q_{\pi, t_1}, \ldots, q_{\pi, t_k}$ with $1 \leq j \leq k \leq n(\pi) + 1$ such that

$$true; [\pi]; \ldots; [q_{\pi, t_k}] \land [\pi \land \varphi] \varepsilon [\pi]; true.$$

(15)

holds. Due to the construction of the PLC-automaton, we know that there is no transition from $q_{\pi, t}$ to $q_{\pi, t_i}$ supposed that $i < k$. Hence we can conclude by (9) that all $[q_{\pi, t_i}]$ phases cannot last longer than $t_i - t_{i-1} + 2\varepsilon$ s.

In the case of $s > t_k$, we can assign to each $[q_{\pi, t_i}]$ phase an upper time bound by $t_i - t_{i-1} - 2\varepsilon$ s. Hence we can conclude by (9) that all $[q_{\pi, t_i}]$ phases cannot last longer than $t_i - t_{i-1} + 2\varepsilon$ s.

We know by (8) or (13) in conjunction with $S_t^\pi(\varphi, t_k) = false$ that the $[q_{\pi, t_k}]$ phase could not last longer than $2\varepsilon$ s. This leads...
to:\ \text{true}:[\pi \land [\varphi]^s \land [q_{\pi,t_k}] \leq 2\varepsilon];[\pi];\text{true}.\ \text{This can be weakened to } s \leq 2\varepsilon, \text{ which is a contradiction to the definition of } \varepsilon \text{ and (3).}

In the case that \( j < k \) and \( t_k \geq s > t_{k-1} \) holds, we know by (10) or (14) in conjunction with \( \delta^s(\varphi, t_k) = \text{false} \) that the \([q_{\pi,t_k}]\) phase could not last longer than \( \varepsilon \) s. This leads to
\[
\text{true};\left(\left[q_{\pi,t_1}\right] \leq \varepsilon \land t_1 < 2\varepsilon; \cdots \left[q_{\pi,t_{k-1}}\right] \leq \varepsilon \land t_{k-1} < t_k - t_{k-1} + 2\varepsilon; \left[q_{\pi,t_k}\right] \leq \varepsilon \land [\pi \land [\varphi]^s];[\pi];\text{true}.
\]

This can be weakened to

\[
s \leq \varepsilon + \sum_{i=0}^{k-1} (t_i - t_{i-1} + 2\varepsilon).
\]

which is a contradiction to the definition of \( \varepsilon \) and (4).

Suppose now \( j < k \) and \( s < t_{k-1} \). If \( j < k-1 \) holds, (16) is a contradiction to (6) with \( A = \varphi \) because of \( \pi \notin \delta^s(\varphi, t_{k-1}) \).

If \( j = k-1 \), \varepsilon s for the same reason for the same time that the \([q_{\pi,t_{k-1}}]\) phase must be shorter than \( \varepsilon \) s due to (7). The \([q_{\pi,t_k}]\) phase could not last longer than \( \varepsilon \) s, as in the previous case. Hence we get \( s < 2\varepsilon \), which contradicts (3) again.

\section*{Acknowledgment}

The author would like to thank E.-R. Olderog, C. Dietz, B. von Karger, S. Kleuker, M. Schenke, J. Tapken, and H. Wehrheim for detailed comments and various discussions on the subject of this paper. Furthermore, the author thanks the anonymous referees for their helpful suggestions and advice.

\section*{References}


