Compilation and Synthesis for Real-Time Embedded Controllers

Martin Fränzle[1] and Markus Müller-Olm[2]

[1] Carl v. Ossietzky Universität, Department of Computer Science,
26111 Oldenburg, Germany
Martin.Fraenzle@Informatik.Uni-Oldenburg.DE

[2] University of Dortmund, Department of Computer Science, FB 4, LS 5,
44221 Dortmund, Germany
Markus.Mueller-Olm@cs.uni-dortmund.de

Abstract. This article provides an overview over two constructive approaches to provably correct hard real-time code generation where hard real-time code is *generated* from abstract requirements rather than *verified* against the timing requirements *a posteriori*. The first, more pragmatic approach is concerned with translation of imperative programs, extended by hard real-time commands which allow one to specify upper bounds for the execution time of basic blocks. In the second approach, Duration Calculus, a metric-time temporal logic, is used as the source language. Duration Calculus allows one to specify real-time systems at a very high level of abstraction.

1 Introduction

Due to rapidly dropping costs and the increasing power and flexibility of embedded digital hardware, digital control is becoming ubiquitous in technical systems encountered in everyday life. Modern means of transport rely on digital hardware even in vital sub-systems like anti-locking brakes, fly-by-wire systems, or signaling hardware. Medical equipment gains such a boost in functionality from exploiting the flexibility of computer control that more classical technology is replaced by digital control even in such critical applications as life-support systems or radiation treatment. Correct behavior of digital systems has thus become crucial to the safety of human life.

Formal methods are mathematical techniques developed to aid in the design of software systems. These techniques can provide correctness guarantees that are not otherwise available. Formal methods thus complement more traditional approaches for ensuring quality of software, like testing and certification by code inspection. Classical program verification is concerned with functional input/output specifications of stand-alone programs. In the application scenarios sketched above, however, correctness typically does not only depend on functional requirements but also on the time at which inputs are read and outputs are provided. Moreover, the digital system typically controls an environment of non-digital nature, i.e. these systems belong to the class of *real-time embedded controllers*.

The omnipresence of hard real-time systems in the realm of embedded systems urgently calls for techniques that support analysis, design, and reliable implementation with respect to both facets of their functionality, which are *logical correctness* and *timeliness* of service. As these aspects are not independent, techniques that can deal with both aspects simultaneously are particularly desirable. For early design phases like requirements capture and functional specification, this has led to the development of prototypical formalisms that tightly integrate algorithmic descriptions and timing. Prominent examples are timed automata [1] on the more operational side and metric-time temporal logics [46, 49] on the declarative side.

For later design phases, in particular code generation, it is, however, still state of the art to keep algorithmic aspects and timing separate. While compilers are generally used for generating code that is logically correct, timing behaviour is mostly analysed *a posteriori*, using profiling tools or even machine-code inspection. We suggest that the chain of formalisms and tools that support an integrated approach to functionality and timing may be extended down to the implementation level. As functionality is nowadays generally dealt with constructively by compilers or synthesis procedures, rather than through *a posteriori* analysis, this necessitates an incorporation of constructive methods for implementing hard real-time constraints into compilers or synthesizers, as well as suitably expressive source languages for these procedures.

In this article, we survey research in this direction that has been pursued by the authors in the scope of the ProCoS project [8]. On the more practical side, this was translation of a hard real-time imperative programming language which has a semantic model that — for the sake of nice algebraic properties — fully abstracts from the runtime of non-communicating statements. Here, compilation exploits invisibility of internal state changes for implementing such instantaneous statements through non-instantaneous sequential code. This consideration is part of more comprehensive work on rigorous verification of compilers which is surveyed in Sect. 2. On the more theoretical side, we have studied automatic synthesis of embedded controllers from rich subsets of Duration Calculus, an interval-based metric-time temporal logic introduced in [49]. Again, observational constraints of the environment are crucial as synthesis exploits band-limitedness or, for still richer subsets of Duration Calculus, even synchronicity properties of the environment for overcoming undecidability of Duration Calculus. This line of work is summarized in Sect. 3.

2 Compiler Verification

Although the idea of mathematically verified compilers dates back at least to the sixties [28] the complete verification of realistic compilers is still a challenge. Most documented work on compiler verification heads for a mathematical understanding of typical or semantically intricate implementation mechanisms illustrated by toy source and target languages [33, 43, 24, 38]. Target code for commercially available processors is only seldomly formally investigated. The work at CLI (Computational Logic Inc.) on the 'small stack' [7], a hierarchy of languages with mechanically proved translations between them, particularly Moore's work on the verified translation of the PITON assembly language to the binary machine code of the FM 9001 chip [31], is one of the rare exceptions. Even the impressive work on VLisp, a verified translator for Scheme, [17] ends at the level of an abstract machine that, given the abstractness of the source language Scheme, is rather close to actual hardware but is still more abstract than commercially available processors.

Apart from mathematical insight there is, however, a more practical motivation for an interest in compiler verification that certainly calls for an investigation of actual machine code: the justification of compiler-generated code from the correctness of the source code. In particular in the area of safety-critical systems, trusted verifed compilers would allow to certify control software on the source code level which would be less time-consuming and thus less costly than the current practice of inspecting machine code [39]. Moreover it would encourage a good documentation or even formal verification of the source code.

In this section we highlight an approach to verifying translations to machine code of actual processors. As a major case study we investigated the Transputer manufactured by the British company INMOS (now part of SGS-THOMSON Microelectronics Ltd) as target architecture. The source language is a prototypical hard real-time language, which allows the programmer to explicitly state upper bound requirements for the execution time of basic blocks. Such a code generator correctness proof may easily become monolithic, aimed at a narrow source language with a specific code generator for the given target processor. A proof of this kind would have little interest beyond the particular application, and might still require a large effort. We have pursued a more modular approach that should adapt to both extensions of the source and modifications to the target which justifies the effort. Like most literature on compiler verification we concentrate on the correctness proof for code generators here because construction of scanners and parsers is well-understood.

2.1 Prologue

It is natural to think of instructions of von Neumann machines as denoting assignments to machine components like accumulators and store. Hence, the effect of machine instructions can conveniently be described by imperative programs. The Transputer instruction $\mathtt{ldc}(1)$, for instance, which loads the constant value 1 to the accumulator called $\mathtt{A}$, and moves $\mathtt{A}$'s contents to accumulator $\mathtt{B}$, as well as $\mathtt{B}$'s contents to accumulator $\mathtt{C}$ (in the Transputer the registers $\mathtt{A}$, $\mathtt{B}$ and $\mathtt{C}$ are used in a stack-like manner), can be represented by the multiple assignment

$$E(\mathtt{ldc}(1)) \stackrel{\mathrm{def}}{=} \mathtt{A}, \mathtt{B}, \mathtt{C} := 1, \mathtt{A}, \mathtt{B} \ .$$

Similarly, the effect of $\mathtt{stl}(x)$, writing $\mathtt{A}$'s contents to variable x, moving $\mathtt{B}$'s value to $\mathtt{A}$, $\mathtt{C}$'s value to $\mathtt{B}$, and an unspecified value to $\mathtt{C}$, can be described by

$$E(\mathtt{stl}(x)) \stackrel{\text{def}}{=} x, \mathtt{A}, \mathtt{B} := \mathtt{A}, \mathtt{B}, \mathtt{C} \; ; \; \mathtt{C} := ?\ ,$$

where $\mathtt{C} := ?$ denotes the nondeterministic choice between all possible assignments to $\mathtt{C}$. To specify a machine by a high level program is of course not a new idea; it already underlies the concept of micro-programming [47], for instance. Such descriptions can also be used as starting point for hardware design [27, 10, 19] and thus provide a good interface to lower levels of abstraction.

If semantics of machine instructions is captured by imperative program fragments, refinement algebra [21], which provides semantics-preserving or refining program transformation rules, can be used to show that certain machine instruction sequences implement certain source programs. The following calculation proves, for example, that the code sequence $\langle \mathtt{ldc}(1), \mathtt{stl}(x) \rangle$, assumed to have the same meaning as the sequential composition of the effects $E(\mathtt{ldc}(1))$ and $E(\mathtt{stl}(x))$, is correct target code for the assignment $x := 1$; for the moment the additional effect on the accumulator $\mathtt{C}$ is taken to be irrelevant:

$$\begin{array}{ll}
& E(\mathtt{ldc}(1)) \; ; \; E(\mathtt{stl}(x)) \\
= & \qquad \text{[Definitions above]} \\
& \mathtt{A}, \mathtt{B}, \mathtt{C} := 1, \mathtt{A}, \mathtt{B} \; ; \; x, \mathtt{A}, \mathtt{B} := \mathtt{A}, \mathtt{B}, \mathtt{C} \; ; \; \mathtt{C} := ? \\
= & \qquad \text{[(Combine-assign), (Identity-assign)]} \\
& x, \mathtt{A}, \mathtt{B}, \mathtt{C} := 1, \mathtt{A}, \mathtt{B}, \mathtt{B} \; ; \; \mathtt{C} := ? \\
= & \qquad \text{[(Cancel-assign), (Identity-assign)]} \\
& x := 1 \; ; \; \mathtt{C} := ?\ .
\end{array}$$

In this proof we have used the following three assignment laws where x and y stand for disjoint lists of variables, e and f for lists of expressions of corresponding type, and $f[e/x]$ denotes substitution of e for x in expression f.

$$\begin{array}{lr}
\text{(Identity-assign)} & (x := e) = (x, y := e, y) \\
\text{(Combine-assign)} & (x := e \; ; \; x := f) = (x := f[e/x]) \\
\text{(Cancel-assign)} & (x, y := e, f \; ; \; y := ?) = (x := e \; ; \; y := ?)
\end{array}$$

The above calculation illustrates a basic idea of our approach to compiler verification: to use an imperative meta-language and refinement laws as proposed by Hoare [20, 22]. We write $\geq$ for the refinement relation; intuitively $P \geq Q$ means that P is better than Q in serving every purpose served by Q in every context.[1]

[1] Due to lack of space we cannot present a formal definition of the meta-language in this article. For the purpose of this overview an informal understanding is sufficient. Let us just mention that the imperative meta-language is interpreted by predicate transformers in the tradition of Dijkstra's wp-calculus [12] and the refinement calculus [34, 5, 32] but extended to communicating programs. For performing the abstractions we use a variant of the data refinement theory of Back [4], Gardiner & Morgan [16], and Morris [35]. Details can be found in the monograph [37].

The exposition up to now is of course oversimplified. Firstly, the model of the instruction's effect is too abstract. For example, the Transputer instructions reference memory locations basically, not variable identifiers as we assumed in the description of $\mathtt{stl}(x)$, and a machine program basically is not a separate entity but the executed instructions are taken from the memory. Secondly, a number of unformalized assumptions have been made in the surrounding text.

An idealized abstract model of the target processor simplifies the compiler verification. But if considerations are based on such a model alone there is a severe danger of unsoundness because the postulated model might fail to provide a safe abstraction of the processor's actual behavior. To ban this danger we interface directly to the Transputer's documentation by starting from a semi-formal model of its execution cycle provided by INMOS, the manufacturer of the Transputer, in [23]. A direct employment of this model in a compiler proof, however, would result in very long and tedious calculations which would seriously affect credibility of the proofs. How can we combine simplicity and conciseness of proofs with realistic modeling of the processor?

The idea is to derive a hierarchy of mutually consistent, increasingly abstract views to the Transputer's behavior, starting from bit-code level up to assembly levels with symbolic addressing. In each of the abstraction steps one particular phenomenon can be tackled in isolation. Afterwards we can choose for each proof task the model that allows the simplest proof or even mix reasoning at different abstraction levels without risking inconsistencies or unsoundness.

In the remaining parts of this section we describe the Transputer base model, the derived more abstract models, and the technique by which the abstraction is performed. Then we show how this hierarchy can advantageously be employed in the translation correctness proof for an imperative (un-timed) source language. Afterwards we indicate the generalization to a timed language. Due to lack of space we cannot present all formal details but invite the reader to enjoy the presentation with an informal understanding. A complete treatment can be found in the monograph [37].

2.2 Transputer Base Model

Appendix F of the Transputer Instruction Manual [23] contains a semi-formal model of the Transputer's behavior. Essentially it describes the Transputer as a state machine that communicates via four bi-directional synchronous channels called *links*, and works on a state consisting of three accumulators `A`, `B` and `C` (used as a small stack by most instructions), an operand register `Oreg`, a workspace pointer `Wptr`, an error flag `EFlag`, an instruction pointer `IP`, and an addressable memory `Mem`. We reformulate this model using the notation of the imperative meta-language mentioned above. The state components are represented by likewise named variables of appropriate type and the links by four input channels $\mathtt{In}_0, \ldots, \mathtt{In}_3$ and four output channels $\mathtt{Out}_0, \ldots, \mathtt{Out}_3$.

We introduce a process *Run* that is constrained by axioms in the form of refinement formulas. *Run* models the complete behavior of the running phase which is entered by the Transputer after the initialization phase that follows a reset. (The initialization phase need not be formally captured for the purpose of compiler verification.) In order to describe *Run* in a modular fashion we introduce auxiliary processes *Step* and *Fetch* and for each Transputer instruction *instr* a process $E_0(instr)$. *Step* models the behavior of a complete execution cycle, *Fetch* the instruction fetch phase and $E_0(instr)$ the instruction specific part of the execution cycle.

The basic property of *Run* is that it cyclically executes steps. This is captured by the axiom

$$Run \;=\; Step \;;\; Run \;. \tag{1}$$

Step is characterized by the family of axioms (one for each Transputer instruction *instr*)

$$Step \;\geq\; \{CurFct(instr)\} \;;\; Fetch \;;\; E_0(instr) \;\;,^{2} \tag{2}$$

where $CurFct(instr) \stackrel{\text{def}}{=} (\mathsf{Byte}(\mathtt{Mem},\mathtt{IP})\ \mathsf{bitand}\ \$\mathtt{F0}) = \mathsf{InstrCode}(instr)$. Intuitively, $CurFct(instr)$ is a predicate that holds true if and only if the memory location pointed to by the instruction pointer `IP` contains just the instruction code of the instruction *instr*. The above property of *Step* means that *Step* behaves like the sequential composition of the *Fetch* phase and the process $E_0(instr)$ if activated in a state where $CurFct(instr)$ holds. *Fetch* is completely described by the axiom

$$Fetch \;=\; \mathtt{Oreg}, \mathtt{IP} := \mathtt{Oreg}\ \mathsf{bitor}\ (\mathsf{Byte}(\mathtt{Mem},\mathtt{IP})\ \mathsf{bitand}\ \$\mathtt{0F}), \mathtt{IP}+1 \;\;.$$

The effect of the single instructions is described by Z-like schemata in appendix F.3 of [23]. (The fetch phase too is described by a schema called 'InstrDecode'.) The schema for the load constant instruction `ldc`, for example, which loads the contents of the operand register `Oreg` to the evaluation stack, looks as follows [23, Page 132]:

ldc	#4_	load constant
$\mathsf{Areg}' = \mathsf{Oreg}^0$		
$\mathsf{Breg}' = \mathsf{Areg}$		
$\mathsf{Creg}' = \mathsf{Breg}$		
$\mathsf{Oreg}' = 0$		
$\mathsf{Iptr}' = NextInst$		

Primed names represent the values of the corresponding register after execution and unprimed names the values before. Oreg^0 stands for the operand register's contents after the fetch phase. In our framework the effect of `ldc` is captured by a refinement axiom about $E_0(\mathtt{ldc})$. As we are using an imperative programming notation we need not use primed and superscripted variables for distinguishing

2 For a predicate ϕ, the *assertion* $\{\phi\}$ is a process that terminates immediately without state change if ϕ holds, and behaves chaotically, i.e. is completely unconstrained otherwise. (Formally, $\{\phi\}$ is the predicate transformer defined by $\{\phi\}(\psi) = \phi \wedge \psi$.) As a consequence, (2) constrains – for a given instruction *instr* – the behavior of *Step* only for initial states in which $CurFct(instr)$ holds.

register values at different stages of execution; we differentiate between them implicitly by the places at which they appear in the formulas. The register updates caused by `ldc` can be represented simply by a multiple assignment statement:

$$E_0(\mathtt{ldc}) \;\geq\; \mathtt{A},\mathtt{B},\mathtt{C},\mathtt{Oreg} := \mathtt{Oreg},\mathtt{A},\mathtt{B},0 \;. \tag{3}$$

This refinement formula can be interpreted as an axiom on `ldc`'s behavior. Comparing it with the Z-like schema above, the careful reader will notice that incrementation of the instruction pointer is missing. We have decided to move it consistently for all instructions to the fetch phase as this is slightly more convenient for the following exposition.

The behavior of the other instructions can be captured by similar axioms. This keeps the modularity of the description in [23]. Each instruction is described separately by one (or more) formula, which leads in the abstractions to short proofs for single instructions instead of one monolithic proof for the entire instruction set. In addition, it allows to reason formally with only a partial formalization of the instruction set; only the instructions actually used in the compiler need be considered.

The axioms on $E_0(\mathit{instr})$ claim refinement, not equality. This has a particular benefit: it allows to approximate the actual effect safely if it is not completely known. As an example, let us consider the following axiom for the instruction `stl` (store-local). Intutitively, `stl` stores the top value of the register mini-stack to a certain location in the memory.

$$\begin{aligned} E_0(\mathtt{stl}) \;\geq\; & \{\mathsf{Index}(\mathtt{Wptr},\mathtt{Oreg}) \in \mathsf{Addr}\} \;; \\ & \mathtt{Mem}[\mathsf{Index}(\mathtt{Wptr},\mathtt{Oreg})],\mathtt{A},\mathtt{B},\mathtt{Oreg} := \mathtt{A},\mathtt{B},\mathtt{C},0 \;;\; \mathtt{C} := ?\;. \end{aligned}$$

Addr denotes the set of valid word addresses and contains only those addresses for which memory is actually available. The assertion $\{\mathsf{Index}(\mathtt{Wptr},\mathtt{Oreg}) \in \mathsf{Addr}\}$ ensures that the inequality is trivial if the referenced memory address $\mathsf{Index}(\mathtt{Wptr},\mathtt{Oreg})$ is invalid. This means that the axiom doesn't tell how `stl` behaves in this situation. The nice effect is that we must ensure, when reasoning about correctness of code, that `stl` is not used under such circumstances as we cannot otherwise show correctness.

The non-deterministic assignment $\mathtt{C} := ?$ captures that the contents of register `C` after a `stl` location is left unspecified by the Transputer designers. Clearly, the `C` register will contain a certain value but our axiom does not say which. So any reasoning based on this axiom cannot rely on any specific assumption about the final value of `C`. Without the use of non-determinism and refinement we would be forced either to describe the effect of `stl` on the register `C` and `stl`'s behavior when it accesses invalid addresses completely, or to work with an idealized model. The first solution is impractical as such information is not available; the second solution might lead to unsafe reasoning.

2.3 A Hierarchy of Views

The base model of the Transputer is on a rather low level of abstraction. It has not even an explicit notion of an executed machine program but instructions are taken from the memory. In principle it is possible to use this model directly in a correctness predicate relating machine code with source programs but this leads to a complicated definition and to complicated proofs. Therefore, we stepwise derive more abstract views to the Transputer's behavior. These views are successively concerned with the following topics, which in this way can be treated in isolation:

– symbolic representation of the control point, which results in a treatment of the executed machine program as a separate entity;
– word-size operands for direct functions;
– convenient access to the workspace;
– symbolic variables instead of workspace addresses; the mapping of these variables to the workspace is described by a *dictionary* δ;
– hiding of registers.

Each abstraction level comprises a collection of processes. The abstractions are performed by defining the collection of processes for the more abstract view in terms of the collection for the more concrete one. Afterwards sufficiently strong theorems are established that allow one to reason with the abstract family of processes alone, without referring to the concrete family. This is essential for meeting the objective of abstraction, viz. to increase tractability.

Table 1 shows the drastic reduction of the complexity of the terms describing the instruction `ldl` (load local) at the various abstraction levels. As mentioned, the assertion $\{\mathsf{Index}(\mathtt{Wptr}, \mathtt{Oreg}) \in \mathsf{Addr}\}$ present at the lower abstraction levels make the inequalities applicable only if the referenced address is valid. From Level 3 onwards this is ensured by a global assumption about the storage allocated for the workspace. The models in the hierarchy are consistent by construction, i.e. by definition and calculation.

In the following we discuss the first two abstractions in more detail and sketch the other abstractions.

Symbolic Representation of Programs. It is quite natural to think of a machine program as a separate entity consisting of a sequence of instructions. The point of execution, that is represented on the machine by the instruction pointer's contents, can be modeled by a distinguished position in that instruction sequence. More elegantly we can use a pair of instruction sequences (a, b), where a stands for the part of the machine program before the distinguished position and b for the part thereafter, i.e. the complete machine program is $a \cdot b$ and the next instruction to be executed is the first instruction of b. Progress of execution can be elegantly expressed by partitioning the sequence $a \cdot b$ differently.

Formalizing this idea we define, based on *Run*, a family of processes $I_1(a, b)$ (parameterized by the pair of instruction sequences a, b). $I_1(a, b)$ describes the

Table 1. Illustration of abstraction levels

The letters v and w range over words; v additionally satisfies $1 \leq v \leq l_{\mathrm{W}}$, i.e., it is a valid workspace address. x is a variable of type word; it is assumed to be in the domain of δ. adr_x is the address of the memory cell allocated for x.

$$\begin{array}{rcl}
E_5^{\delta}(\mathtt{ldl}, adr_x) & \geq & \mathtt{SKIP} \\
E_4^{\delta}(\mathtt{ldl}, adr_x) & \geq & \mathtt{A}, \mathtt{B}, \mathtt{C} := x, \mathtt{A}, \mathtt{B} \\
E_3(\mathtt{ldl}, v) & \geq & \mathtt{A}, \mathtt{B}, \mathtt{C} := \mathtt{Wsp}(v), \mathtt{A}, \mathtt{B} \\
E_2(\mathtt{ldl}, w) & \geq & \{\mathsf{Index}(\mathtt{Wptr}, w) \in \mathsf{Addr}\} \ ; \\
& & \mathtt{A}, \mathtt{B}, \mathtt{C} := \mathtt{Mem}(\mathsf{Index}(\mathtt{Wptr}, w)), \mathtt{A}, \mathtt{B} \\
E_1(\mathtt{ldl}) & \geq & \{\mathsf{Index}(\mathtt{Wptr}, \mathtt{Oreg}) \in \mathsf{Addr}\} \ ; \\
& & \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{Oreg} := \mathtt{Mem}(\mathsf{Index}(\mathtt{Wptr}, \mathtt{Oreg})), \mathtt{A}, \mathtt{B}, 0 \\
E_0(\mathtt{ldl}) & \geq & \{\mathsf{Index}(\mathtt{Wptr}, \mathtt{Oreg}) \in \mathsf{Addr}\} \ ; \\
& & \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{Oreg} := \mathtt{Mem}(\mathsf{Index}(\mathtt{Wptr}, \mathtt{Oreg})), \mathtt{A}, \mathtt{B}, 0
\end{array}$$

total behavior resulting from starting $a \cdot b$ with the first instruction of b:

$$I_1(a, b) \ \stackrel{\text{def}}{=} \ \mathtt{var\ IP}\ ; [\mathit{Loaded}(a \cdot b) \wedge \mathit{IPAfter}(a)] \ ; \mathit{Run} \ ; \mathtt{end\ IP} \ .$$

The outer block $\mathtt{var\ IP} \ldots \mathtt{end\ IP}$ hides the instruction pointer $\mathtt{IP}$ that is no longer needed because the point of execution is symbolically represented from now on. The assumption[3] $[\mathit{Loaded}(a \cdot b) \wedge \mathit{IPAfter}(a)]$ ensures that Run is started in a state where $a \cdot b$ is loaded and the instruction pointer $\mathtt{IP}$ points just after a. The predicates $\mathit{Loaded}(u)$ and $\mathit{IPAfter}(v)$ are defined by

$$\begin{array}{rcl}
\mathit{Loaded}(u) & \stackrel{\text{def}}{=} & |u| \leq l_P \ \wedge \\
& & \langle \mathsf{Byte}(\mathtt{Mem}, s_P), \ldots, \mathsf{Byte}(\mathtt{Mem}, s_P + |u| - 1) \rangle = \mathsf{Code}(u) \\
\mathit{IPAfter}(v) & \stackrel{\text{def}}{=} & \mathtt{IP} = s_P + |v| \ \ ,
\end{array}$$

where s_P and l_P are the start address and the length of the *program memory*, i.e. that region of memory allocated to hold the program.

We also define abstractions of the effect processes $E_0(\mathit{instr})$. These ensure by a final assertion and by taking the greatest lower bound over all possible instruction sequences a, b that neither the loaded program (whatever it might be) nor the position of execution is changed or, more precisely, that changes lead to chaotic behavior:

$$\begin{array}{rcl}
E_1(\mathit{instr}) & \stackrel{\text{def}}{=} & \bigwedge\limits_{a,b} \mathtt{var\ IP}\ ; \ [\mathit{Loaded}(a \cdot b) \wedge \mathit{IPAfter}(a)] \ ; E_0(\mathit{instr}) \ ; \\
& & \qquad\qquad \{\mathit{Loaded}(a \cdot b) \wedge \mathit{IPAfter}(a)\} \ ; \mathtt{end\ IP} \ .
\end{array}$$

[3] For a predicate ϕ, the *assumption* $[\phi]$ is a process that — like the assertion $\{\phi\}$ — terminates immediately without state change if ϕ holds but leads to miraculous success otherwise. Formally, $[\phi]$ is the predicate transformer $[\phi](\psi) = (\phi \Rightarrow \psi)$.

From these definitions and the axioms of the base model we can prove the following theorem:

Theorem 1 (I1-instruction theorem).
$I_1(a, \mathit{instr}(n) \cdot b) \;\geq\; \mathtt{Oreg} := \mathtt{Oreg}\ \mathsf{bitor}\ n\ ;\ E_1(\mathit{instr})\ ;\ I_1(a \cdot \mathit{instr}(n), b)\,.$

Here we write $\mathit{instr}(n)$ for the code sequence consisting of the single instruction *instr* with four-bit operand n, $0 \leq n < 16$. This theorem formally reflects that a machine program that is loaded and started at some instruction *instr* with four-bit operand n behaves as follows: n is bitwisely or-ed with the value in the operand register (this loads n to the least four bits of the operand register since every previous instruction leaves these bits cleared), and then the (abstracted) effect of *instr* is executed. Afterwards it behaves as if the same program is executed starting at the next instruction.

Moreover, we prove for each instruction *instr* approximations for $E_1(\mathit{instr})$ that do not refer to E_0. An example for the `ldl` instruction can be found in Table 1.

Large Operands for Instructions. The purpose of the Transputer's operand register `Oreg` is to provide word-size operands for the instructions. The idea is that the operand register is filled with the operand in portions of four bits by a sequence of `pfix` and `nfix` instructions preceding the instruction for the actual function, the operand part of which provides the least significant four bits only. This special purpose and use of the operand register, however, is not directly reflected in the behavioral description of the Transputer we have available up-to-now, where `Oreg` is treated like any other register. The second abstraction, therefore, provides an understanding of leading `pfix` and `nfix` sequences together with a trailing non-`pfix` and non-`nfix` instruction as a multi-byte instruction.

We define a new view $I_2(a, b)$ based on $I_1(a, b)$ and an abstraction $E_2(\mathit{instr}, w)$ of an instruction *instr*'s effect together with a *word operand* w.[4] We then can prove a new version of the instruction theorem:

Theorem 2 (I2-instruction theorem).
$I_2(a, \mathit{instr}(w) \cdot b) \;\geq\; E_2(\mathit{instr}, w)\ ;\ I_2(a \cdot \mathit{instr}(w), b)\,.$

Now $\mathit{instr}(w)$ stands for the *instruction sequence* resulting from the standard scheme for generating `pfix` and `nfix` instructions described in [23, Chapter 4]. Furthermore we prove approximations for E_2 as shown for the `ldl`-instruction in Table 1. Note that the I2-instruction theorem is easier to apply than the old I1-instruction theorem as it requires no explicit calculations with the operand register.

[4] The range of word operands differs for the different processors from the Transputer family. A typically value is $-2^{31} \leq w < 2^{31}$ for 32-bit Transputers but there are also 16-bit Transputers for which the range is $-2^{15} \leq w < 2^{15}$.

Further Abstractions. Due to lack of space we can only briefly sketch the remaining abstraction levels:

- The third level I_3 replaces the memory variable `Mem` by an array `Wsp` (the 'workspace') of a fixed length l_W. A fixed value s_W of the workspace pointer is assumed and the workspace is mapped to the sequence of memory locations just above s_W. The abstraction is based on a global assumption that this memory region is disjoint from the program storage and contains only valid addresses. It allows us to reason more easily about Transputer code that accesses the memory via the usual workspace mechanism because reasoning on this level need not reflect the mechanism by which the workspace is mapped to memory. This is done only once while performing the abstraction.
- The next level I_4^δ replaces the workspace by a list of symbolic variables. The set of introduced variables as well as their representation is described by a dictionary δ.
- The final level I_5^δ hides the remaining registers `A`, `B`, `C` and `EFlag` and provides a view in which only the communications via the links and the symbolic variables are visible.

The choice of the abstraction levels is not accidental. They correspond very well to the intuitive concepts used in informal reasoning about Transputer code and can be interpreted as semantical analogues to increasingly abstract assembler languages. A formally justified counterpart to the intuitive understanding of each abstraction level is provided by an instruction theorem like Theorem 1, special theorems for jumps and conditional jumps, and theorems about the single instructions.

2.4 Incremental Specification of Code

We benefit from the hierarchy of views to the Transputer's behavior when defining the correctness relation between source and target programs. Consider as an example a simple imperative programming language with syntactic categories of programs, statements and expressions and assume that we are globally only interested in the communication behavior of complete programs. Then it is sensible to call a Transputer instruction sequence m correct code for a program *prog* iff

$$I_5^\emptyset(\varepsilon, m) \ \geq \ \mathsf{MP}(\mathit{prog}) \ ,$$

where $\mathsf{MP}(\mathit{prog})$ is the meaning of *prog* interpreted in the space of processes, ε stands for the empty code sequence, and $\emptyset$ represents the empty dictionary.

Translation of statements, however, has to take representation of variables into account. Therefore, a reasonable correctness condition relating a statement *stat* to code m must employ a non-trivial dictionary δ for the variables appearing in *stat*. When comparing m with *stat* we cannot simply use the predicate given by

$$I_5^\delta(\varepsilon, m) \ \geq \ \mathsf{MS}(\mathit{stat}) \ ,$$

since — inherited from *Run* — $I_5^\delta(\varepsilon, m)$ does not terminate. (For complete programs this problem does not arise: we assume that programs stop on termination which can be achieved by a `stopp` instruction in m). The main purpose of termination of *stat* is to transfer control to its sequential successor. Therefore, we do not expect that the machine stops after execution of m but rather that control is transferred to the code just following m. Formalizing this idea we use the following predicate as notion of correct implementation of statements: m implements *stat* w.r.t. dictionary δ, $\mathsf{CS}(m, stat, \delta)$ for short, iff

$$I_5^\delta(a, m \cdot b) \ \geq \ \mathsf{MS}(stat) \ ; I_5^\delta(a \cdot m, b) \ \ ,$$

for all code sequences a, b. A correctness predicate for expression translation must refer to the more concrete view I_4^δ because correct expression code is expected to leave the expression's value in register `A` that is not visible on Level 5.

Having defined correctness predicates, we establish for each source language constructor a theorem that shows how correct code for the composed construct can be obtained from correct code of the components. From a comprehensive set of such theorems a code generator program can be implemented without further semantic consideration. As a first example we present the theorem for sequential composition here; further examples can be found in Sect. 2.6.

Not surprisingly, the sequential composition $\mathbf{seq}(stat_1, stat_2)$ of two statements $stat_1$ and $stat_2$ can simply be implemented by concatenating machine code implementing $stat_1$ and $stat_2$.

Theorem 3 (Sequential composition translation). *Suppose* $\mathsf{CS}(m_1, stat_1, \delta)$ *and* $\mathsf{CS}(m_2, stat_2, \delta)$ *hold. Then* $\mathsf{CS}(m_1 \cdot m_2, \mathbf{seq}(stat_1, stat_2), \delta)$.

The proof is by a little calculation that applies to arbitrary code sequences a and b; of course, sequential composition in the source language corresponds to the sequential composition operator ; of the meta-language:

$$\begin{array}{ll} & I_5^\delta(a, m_1 \cdot m_2 \cdot b) \\ \geq & \qquad [\mathsf{CS}(m_1, stat_1, \delta)] \\ & \mathsf{MS}(stat_1) \ ; I_5^\delta(a \cdot m_1, m_2 \cdot b) \\ \geq & \qquad [\mathsf{CS}(m_2, stat_2, \delta)] \\ & \mathsf{MS}(stat_1) \ ; \mathsf{MS}(stat_2) \ ; I_5^\delta(a \cdot m_1 \cdot m_2, b) \\ = & \qquad [\text{Definition of semantics of } \mathbf{seq}] \\ & \mathsf{MS}(\mathbf{seq}(stat_1, stat_2)) \ ; I_5^\delta(a \cdot m_1 \cdot m_2, b) \ \ . \end{array}$$

The simplicity of this proof, which looks almost too straightforward to be of interest, stems from the use of the adequate abstraction level. If we had defined CS with direct reference to the Transputer base model (which is easily done by unfolding the definition of I_5^δ) the proof would be far more complex as all invariants that are kept by the code had to be explicitly treated. Now they are treated incrementally during the derivation of the abstraction levels. Moreover, we would not have such a clear way of speaking about the control point but had to refer to its coding in the instruction pointer.

2.5 Real-Time Programs

The ideas described up to now have been applied to a prototypic real-time language. We considered the language of while-programs extended by (synchronous) input/output statements and timing constructs. The timing constructs separate two fundamentally different aspects of timing of computer programs. On the one hand, means for explicitly specifying that certain actions happen at certain time instants are needed. On the other hand, the delay caused by the execution of statements must be controlled to stay in safe limits.

Time instants may be specified either relative or absolut. In a controller for an elevator, for example, we might want to specify that the elevator door opens one second after the chosen floor has been reached, which is a *relative* specification. *Absolute* time instants are needed, for example, to express the requirement that the elevator is to be shut off after closing-time, 6 o'clock p.m. (Of course this requirement needs careful refinement in order to prohibit starvation of people entering the elevator at 5:59...) To perform tasks at absolute time instants is typically delegated to the operating system (consider, for example, the UNIX `at` daemon); thus absolute timing constructs are not crucial for a real-time language. Relative timing, however, is indispensable in control programs. Common means for specifying relative timing are WAIT statements and time-out clauses. For implementing these constructs, a compiler typically exploits specific features of the implementing hardware, like timers, or applies some service provided by the operating system. Therefore, they don't provide a particular challenge for a compiler. Our work focussed thus on mastering the execution delay.

From the specification point of view the delay caused by statement execution is an undesirable inconvenience but it is an inevitable companion of computation. As it is typically orders of magnitude smaller than the timing requirements of the application, it is often not explicitly adressed in real-time formalisms, e.g., in synchronous languages [18, 6]. If computation load is high, however, the implementing code must be analysed in order to guarantee the timing requirements of the application. We are heading for an approach that avoids such an *a posteriori* analysis but remains as convenient as possible for the programmer.

The idea is to allow the programmer to specify upper bounds for the tolerated execution time of basic blocks in the source language. It is the obligation of the compiler to check whether these upper bounds are met by the generated code. There is no point in specifying lower bounds on execution delay: faster execution should always be an improvement. Note that faster execution does not affect explicit delays as in WAIT statements, as these are implemented by primitives that are not influenced by processor speed. To specify only upper bounds has the benefits that the compiler has to perform just a worst-case timing analysis and that migration to faster processors without recompilation is possible. It would, however, be rather inconvenient for the programmer, if any statement must be guarded by an upper-bound. We offer the idealization that internal computation, like assignments and evaluation of guards of loops and conditionals, proceeds in zero time. Thus only input/output statements must be guarded and smooth program transformation laws are valid. Assignments, for example, can be moved in and out of time bounds, and logically redundant assignments can freely be introduced or eliminated without affecting the timing behavior of programs. This idealization, which is akin to that found in synchronous languages, can be justified by exploiting that internal computation is not directly observable. We describe in Sect. 2.6 how this challenge was met in a semantic compiler proof.

Let us illustrate both aspects of timing by means of a small program fragment taken from a control program for the ProCoS gas burner (cf. Sect. 3.6):

```
. . .
heatreq := false
WHILE not(heatreq) DO
  SEQ
    WAIT 1/2
    UPPERBOUND 1/8
      input(hr, heatreq)
. . .
```

We use an OCCAM-like concrete syntax of programs, where indentation indicates block structure; `SEQ` is the sequential composition operator, the other constructors are self-explaining.

The statement `input(hr, heatreq)` reads the current state of a thermostat, which is connected to the program via channel `hr`, and stores it to the variable `heatreq`. The purpose of the above program fragment is to poll the thermostat until it reports that heat is requested. Each iteration of the loop first waits half a second and then questions channel `hr` for the current status of the thermostat. The input statement is guarded by the `UPPERBOUND 1/8` clause, which ensures that it takes at most 1/8 seconds to read in the current state of the thermostat.[5] The programmer can safely assume that all other activity is instantaneous, the initial assignment to variable `heatreq` as well as the evaluation and checking of the loop's guard `not(heatreq)`. He can also assume that `WAIT 1/2` waits precisely half a second. The actual execution time of the code implementing these statements, as well as the deviation of the implementation of the `WAIT` statement from the ideal timing is shifted by the compiler to the `input` statement and then settled with the upperbound; thus the total overhead of each iteration is bounded by 1/8 seconds. Consequently, the polling loop ensures that a change to the heat request state of the thermostat is detected after at most 5/8 seconds.

The idealized timing properties supported by the compiler allow one to specify such a reactivity requirement in a simple way. Otherwise, all statement would have to be bounded but it would be quite difficult to guess adequate bounds because the generated code is not known in advance.

[5] Communication is synchronous; hence both input and output statements stall until the corresponding communication partner becomes ready. The duration of this stalling cannot be bounded by the compiler; it is totally dependent on the environment. By convention, it is not included into the time bounded by `UPPERBOUND`; the `UPPERBOUND` only refers to the time used for preparation of the communication. In the example program fragment, we assume that the thermostat is always ready to output its current state on channel `hr` such that there is no stalling.

2.6 Time Shifts

In this section we discuss the additional ingredients of the compiler correctness proof for the timed language.

First of all, the imperative meta-language is provided with a timed interpretation and extended by delays $\Delta\, d$ and time bounds $|\, P\,| \preceq d$, where $d \in \mathbb{R}_{\geq 0} \cup \{\infty\}$. This is done in such a way that all constructs except delays execute in zero time. The bound operator constrains the time consumed by the enclosed process P (not counting the time that P stalls waiting for communication partners to become ready); $\Delta\, d$ executes in at most d time units without state change. For simplicity, we use the duration of one execution cycle of the target processor as the unit of time in this section.

We add delays to the axioms describing the effect of instructions. For example, the timed version of Axiom (3) looks as follows:

$$E_0(\mathtt{ldc}) \;\geq\; \Delta\, 1 \;;\; \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{IP}, \mathtt{Oreg} := \mathtt{Oreg}, \mathtt{A}, \mathtt{B}, \mathtt{IP} + 1, 0 \;\;.$$

This expresses that `ldc` uses at most one execution cycle for execution. The timing information is taken from the tables in [23, Appendix D].

A particularly interesting aspect of the code generator proof is that it justifies the idealization of instantaneous execution for all internal constructs of the timed source language, i.e. all constructs except input/output statements. Of course, the code implementing, say, an assignment needs time to execute. The idea is to shift such excess time of code implementing internal activity to a sequentially successive process that is compiled to a machine program needing less time for execution than allowed by the source [15]. This can be accomplished by adding two parameters L and R to the correctness predicate for statements, where L states the excess time of the sequential predecessor that is absorbed and R states the excess time that is handed over to the sequential successor for absorption. A third new parameter E is introduced that asserts a time bound for the source statement. This leads to the following definition: a machine program m implements source statement *stat*, absorbing excess time L from its sequential predecessor, exporting excess time R to its sequential successor, under time bound E, iff for all instruction sequences a and b

$$\Delta\, L \;;\; I_5^{\delta}(a, m \cdot b) \;\geq\; |\, \mathsf{MS}(\mathit{stat})\,| \preceq E \;;\; \Delta\, R \;;\; I_5^{\delta}(a \cdot m, b) \;\;.$$

For brevity, we denote this implementation property by $\mathsf{CS}'(m, \mathit{stat}, \delta, L, R, E)$. Let us now have a look at some example translation theorems.

The theorem that allows to translate time bounds in the source program looks as follows:

Theorem 4 (Translation of time bounds). *Suppose* $\mathsf{CS}'(m, \mathit{stat}, \delta, L, R, E)$. *If* $E \leq t$ *then* $\mathsf{CS}'(m, \mathbf{upperbound}(P, t), \delta, L, R, E)$.

Thus, a compiler encountering an upper bound operator in the source statement needs only check whether the required time bound is more liberal than the one asserted upon the code generated for the enclosed statement. If it is, then no further action is necessary, as the real-time requirement expressed by the bound is met. If it is less liberal, on the other hand, the source statement cannot be adequately compiled for the given target hardware with the given code generation strategy, and should be rejected (or perhaps another code generator should be activated). The proof of Theorem 4 is quite simple, given the following law.

$$\text{(Multiple-bound)} \qquad (|\,|\,P\,| \preceq t_1\,| \preceq t_2) = (|\,P\,| \preceq \min(t_1, t_2))$$

Let us now have a look at the theorem for the translation of an assignment statement $\mathbf{assign}(x, e)$. In the theorem below, CE is the correctness condition for expression translation. Intutitively, $\mathsf{CE}(e, m_{,1}, \delta, E_1)$ holds, if m_1 is a piece of code that evaluates e assuming that variables are represented according to the dictionary δ. $\mathsf{CE}(e, m_{,1}, \delta, E_1)$ also asserts that m_1 executes in at most E_1 time units.

Theorem 5 (Assignment translation).
Suppose $\mathsf{CE}(e, m_1, \delta, E_1)$*,* $m = m_1 \cdot \mathtt{stl}(adr_x)$*, and* $R \geq L + E_1 + ldt(adr_x) + 1$*.*
Then $\mathsf{CS}'(\mathbf{assign}(x, e), m, \delta, L, R, E)$ *.*

The proposed code m is composed from evaluation code for the expression e and a `stl`-instruction that stores the result value to the location allocated for x. Its execution time is the sum of the time E_1 needed for evaluating the expression e, the time $ldt(adr_x)$ needed for loading the operand of the final `stl` instruction, i.e. the address of x, and one additional time unit for the execution of the `stl` instruction itself. Note that the entire execution delay $E_1 + ldt(adr_x) + 1$ of m is handed over via the parameter R to the sequential successor for absorption together with the time L to be absorbed. Note also that there is no condition on E. Hence, any time bound can be asserted for an assignment. This justifies the idealization of immediate execution.

For communication statements, on the other hand, immediate execution cannot be assumed as their effect is visible to the environment. Also the time absorbed from predecessor code becomes visible here. Let us have a look at the theorem concerned with translation of an output statment $\mathbf{output}(\mathtt{Out}_i, expr)$, which, intuitively, outputs the value of expression $expr$ on link i ($i = 0, \ldots, 3$).

Theorem 6 (Output translation).
Suppose $\mathsf{CE}(expr, m_1, \delta, E_1)$*,* $m = m_1 \cdot \mathtt{mint} \cdot \mathtt{ldc}(i) \cdot \mathtt{bcnt} \cdot \mathtt{add} \cdot \mathtt{rev} \cdot \mathtt{outword}$*,*
$R \geq outdelay_2$*, and* $E \geq L + E_1 + outdelay_1 + 8$*.*
Then $\mathsf{CS}'(\mathbf{output}(\mathtt{Out}_i, expr), m, \delta, L, R, E)$ *.*

The code is composed from expression evaluation code m_1, a piece of code $\mathtt{mint} \cdot \mathtt{ldc}(i) \cdot \mathtt{bcnt} \cdot \mathtt{add} \cdot \mathtt{rev}$ (executing in 8 cycles) that fills the Transputer registers with adequate parameters, and a final `outword` instruction which initiates the actual output. The execution of `outword` consists of three phases. After the first phase, which executes in (at most) $outdelay_1$ time units, the commuicated value is available on the link i. In the second phase the Transputer waits for its communication partner to become ready (recall that communication is synchronous) and exchanges the actual value. The time spent during this phase cannot be controlled by a process but depends solely on the environment; it is ignored in the semantic model used. The third phase, which executes in (at most) $outdelay_2$ time units, is concerned with some operation after the communication commenced.

The time $E_1 + outdelay_1 + 8$ spent by the code before the communicated value becomes available to the environment cannot be handed over to the successor as the communication is visible to the environment. Together with the time L to be absorbed from the predecessor it poses a constraint on time bounds that can be guaranteed for the output statement. This is expressed by the condition on E. The time $outdelay_2$ spent after the communication takes place, on the other hand, is handed over to the sequential successor for absorption via parameter R.

Let us finally have a look at the timed version of Theorem 3.

Theorem 7 (Sequential composition translation).
Suppose $\mathsf{CS}'(stat_1, m_1, \delta, L_1, R_1, E_1)$, $\mathsf{CS}'(stat_2, m_2, \delta, L_2, R_2, E_2)$, *and* $R_1 \leq L_2$. *Then* $\mathsf{CS}'(\mathbf{seq}(stat_1, stat_2), m_1 \cdot m_2, \delta, L_1, R_2, E_1 + E_2)$.

Here, the premise $R_1 \leq L_2$ expresses that the code m_2 for the second component $stat_2$ must be able to absorb at least the excess time R_1 handed over from the first component. Note that we can assert for a sequential composition the sum of time bounds for its components.

From a collection of such theorems that describe construction of correct code for each operator of the source language we have developed a code generator written in the functional language Standard ML [30]. By adding a frontend, we have constructed a prototypical compiler [36].

This concludes this overview on the modular verified design of code generators. The modularity of the approach facilitates the construction of code generators and assists rigorous control procedures because it allows to split both tasks into relatively small, independent sub-tasks. Moreover, it enables reuse. The derived views to the Transputer, e.g., can be exploited for different source languages or even used when verifying boot programs or operating systems. To use some form of refinement as underlying notion of correctness instead of semantic equivalence allows a proper treatment of under-specification in the target and the source language, which allows, e.g., to give a proper meaning to uninitialized variables. Moreover it accommodates modularization. Like the work at CLI (Computational Logic Inc.) on the 'small stack' [7] we have put emphasis on consistent interfaces to higher and lower levels of abstraction.

3 Synthesis of Embedded Real-Time Controllers

We will now turn to the problem of directly synthesizing real-time embedded controllers from metric-time temporal logic specifications. In comparison to imperative programming languages, such logics provide a very abstract means of specifying what the system should do rather than saying how to achieve this. Thus, using such logics as source languages for compilation or synthesis methods for embedded controllers would be desirable. This is particularly true for those logics that provide a very high level of abstraction from operational detail, like e.g. the monadic second order logic of temporal distance [46] or the Duration Calculus [49].

Unfortunately, in the realm of dense metric time, logics featuring rich metric-time vocabulary and full negation tend to become undecidable (e.g., above two are), which has direct impact on the feasibility of sound and complete automatic synthesis procedures. However, the argument which makes decidability a necessary condition for feasibility of synthesis (to be reviewed in Sect. 3.4) relies on an essentially unconstrained environment dynamics. We argue that through exploitation of general constraints on environment dynamics, synthesis methods can be exposed even for some undecidable metric real-time logics, and exemplify this on synthesis procedures for timed controllers from dense-time Duration Calculus in Sect. 3.5.

3.1 Duration Calculus

Duration Calculus (abbreviated DC in the remainder) is a real-time logic that is specially tailored towards reasoning about durational constraints on time-dependent Boolean-valued states. The syntax of the subsets of DC formulae that we will study is

$$\begin{aligned}
\langle formula \rangle &::= \langle atomicform \rangle \mid \neg \langle formula \rangle \mid \\
&\quad\; (\langle formula \rangle \wedge \langle formula \rangle) \mid (\langle formula \rangle ; \langle formula \rangle) \\
k &::\in \mathbb{N} \\
\langle state \rangle &::= \langle variable \rangle \mid \neg \langle state \rangle \mid (\langle state \rangle \wedge \langle state \rangle) \\
\langle variable \rangle &::\in \mathit{Varname}
\end{aligned}$$

Concerning the available atomic formulae, we distinguish two different subsets of DC. In the first subset, often called the $\{\lceil P \rceil, \ell = k\}$ fragment, atomic formulae take the form

$$\langle atomicform \rangle ::= \lceil \langle state \rangle \rceil \mid \ell = k \;\; ,$$

while in the so-called $\{\int P = k\}$ fragment,

$$\langle atomicform \rangle ::= \int \langle state \rangle = k \;\; .$$

Duration Calculus is interpreted over trajectories

$$tr \in \mathit{Traj} \stackrel{\text{def}}{=} \{tr : \mathit{Time} \rightarrow \mathit{Varname} \rightarrow \mathbb{B} \mid \mathit{Time} = \mathbb{R}_{\geq 0}, tr \text{ finitely variable}\}$$

that provide a finitely variable, time-dependent, Boolean-valued valuation of variables. Finite variability — sometimes also called non-Zenoness — means that only finitely many state changes may occur within any finite time interval. The definition of satisfaction of a formula ϕ by a trajectory tr, denoted $tr \models \phi$ is given in Table 2. The set of models of ϕ, i.e. trajectories satisfying ϕ, is denoted $\mathcal{M}[\![\phi]\!]$.

As usual, we say that ϕ *is valid* iff $\mathcal{M}[\![\phi]\!] = \mathit{Traj}$. According to [48], validity is undecidable for the fragment $\{\lceil P \rceil, \ell = k\}$. Consequently, the same applies for the fragment $\{\int P = k\}$, as $\lceil P \rceil$ can be encoded as $\int \neg P = 0 \wedge \neg(\int \mathtt{true} = 0)$ and $\ell = k$ as $\int \mathtt{true} = k$, resp.

Table 2. Semantics of Duration Calculus

$tr, [a,b] \models \phi$ denotes that trajectory tr satisfies a formula ϕ within a time interval $[a,b] \subset \mathbb{R}_{\geq 0}$. It is defined by

$$\begin{array}{lll}
tr,[a,b] \models \int P = k & \text{iff} & \int_a^b \chi_{P,tr}(t)\,\mathrm{d}t = k \ , \\
tr,[a,b] \models \lceil P \rceil & \text{iff} & b > a \wedge \int_a^b \chi_{P,tr}(t)\,\mathrm{d}t = b - a \ , \\
tr,[a,b] \models \ell = k & \text{iff} & b - a = k \ , \\
tr,[a,b] \models \neg\phi & \text{iff} & tr,[a,b] \not\models \phi \ , \\
tr,[a,b] \models (\phi \wedge \psi) & \text{iff} & tr,[a,b] \models \phi \quad \text{and} \quad tr,[a,b] \models \psi \ , \\
tr,[a,b] \models (\phi;\psi) & \text{iff} & \exists\, m \in [a,b] \,.\, \begin{pmatrix} tr,[a,m] \models \phi \quad \text{and} \\ tr,[m,b] \models \psi \end{pmatrix} \ ,
\end{array}$$

where $\chi_{P,tr}(t) = 1$ iff the state formula P evaluates to true in time instant t over trajectory tr, and $\chi_{P,tr}(t) = 0$ otherwise.
A trajectory tr *satisfies a formula* ϕ, denoted $tr \models \phi$, iff all finite prefixes of tr satisfy ϕ — formally, $tr \models \phi$ iff $tr, [0,t] \models \phi$ for each $t \in \mathbb{R}_{\geq 0}$. Accordingly, a trajectory is a counterexample of ϕ, i.e. does not satisfy ϕ, iff some of its finite prefixes satisfies $\neg\phi$.

3.2 Timed controllers

We do now turn to the implementation "technology" we are aiming at. For the purpose of this overview, it is timed transition tables in the sense of Alur and Dill [1] extended by a notion of environment input and transition-table output. Thus, it is basically an untimed, Mealy-type transition table (Σ, σ_0, T), where

- Σ is a finite, nonempty set of states,
- $\sigma_0 \in \Sigma$ is the initial state,
- $T \subseteq \Sigma \times \alpha \times \Sigma$ is the transition relation, where the alphabet α is of the form $(I \uplus O) \rightarrow \mathbb{B}$ with I being the finite set of (Boolean-valued) input ports and O being the finite set of (Boolean-valued) output ports (i.e. α assigns binary values to input and output ports),
- T is *input-open*, which means that from any state $\sigma \in \Sigma$ there is a transition for each input $i \in I \rightarrow \mathbb{B}$.

However, in timed transition tables these basic untimed transition tables are extended with a finite number of real-valued clocks that can be reset upon transitions and can be compared against constants in transition guards. Thus, transitions are of the extended form $(\sigma, a, \sigma', \mathit{guard}, \mathit{reset})$, with $a \in \alpha$, $\sigma, \sigma' \in \Sigma$, *guard* an integer-bounded interval constraint on the clocks Cl, and $\mathit{reset} \subseteq Cl$.

Intuitively, a timed transition table may take transition $(\sigma, a, \sigma', \mathit{guard}, \mathit{reset})$ when it is in internal state σ, the (internal) clock reading satisfies *guard*, and the current input is $a\big|_I$. It then produces output valuation $a\big|_O$ at its output ports, which persists until the next transition, moves to internal state σ', and synchronously resets the clocks in *reset* to 0. The timed transition table thus produces a time-dependent valuation of its input and output ports, i.e. a trajectory. For any timed transition table C, the set of its trajectories is denoted $\mathcal{C}[\![C]\!]$.

Finally, if timed transition tables are to be used as embedded controllers, we must require that they be input open. Due to the clock-dependent behaviour, this involves a slight extension to above notion of being input-open: a timed transition table is *input-open* iff from any state σ under any possible clock valuation there is a transition for each $i \in I \to \mathbb{B}$. A timed transition table with that property is called *timed controller (with input I and outputs O)* in the remainder.

Based on the definition of trajectories of timed controllers, it is straightforward to define when a timed controller satisfies a DC formula. The criterion is trajectory inclusion.

Definition 8 (Satisfaction). *Let C be a timed controller and ϕ a Duration Calculus formula. We say that C* satisfies *ϕ iff $\mathcal{C}[\![C]\!] \subseteq \mathcal{M}[\![\phi]\!]$.*

3.3 Control Problems and Controller Synthesis

Duration Calculus was designed for bridging the gap between requirements capture and controller implementation in an embedded-controller design activity. Consequently, we are interested in the satisfaction problem between controllers from the design space and *control problems* expressed in Duration Calculus.

Definition 9 (Control problem). *A* control problem *is a pair (Req, I), where Req is a formula of Duration Calculus specifying the admissible controller behaviours and $I \subseteq Varname$ is a set of variable names specifying the inputs to the controller, i.e. those variables that the controller cannot control and hence is not allowed to constrain in their evolution over time. We say that a control problem (Req, I) is a $\{\lceil P \rceil, \ell = k\}$ control problem (or a $\{\int P = k\}$ control problem) iff Req is a $\{\lceil P \rceil, \ell = k\}$ formula (a $\{\int P = k\}$ formula, resp.).*

A controller C is said to solve the control problem *(Req, I) iff C has inputs I and satisfies Req.*

Aiming at solutions to control problems we are interested in synthesis methods that derive controllers from control problems. Such a synthesis method can be understood as a partial mapping *synt* from control problems to controllers. Let *synt* be a (for the moment not necessarily effective) partial function from the set of control problems to controllers. We will now define when *synt* provides a sound and complete synthesis method for control problems.

Definition 10 (Soundness and completeness of a synthesis procedure). *We say*

1. *that synt is* sound *iff it maps control problems to solutions thereof, and*
2. *that synt is $\{\lceil P \rceil, \ell = k\}$*-complete *(or $\{\int P = k\}$*-complete*) iff its domain contains all $\{\lceil P \rceil, \ell = k\}$ control problems ($\{\int P = k\}$ control problems, resp.) that are solvable by timed controllers.*

The interesting question is whether there is a sound and complete *mechanic* synthesis method for timed controllers from DC-based control problems. Unfortunately, the answer is negative, as is shown in the next section.

3.4 Synthesis under Unconstrained Environment Dynamics

We start with analyzing the synthesis problem with respect to unconstrained environment dynamics. I.e., inputs may change arbitrarily — but of course finitely variable — over time. Then, a controller cannot contribute to requirements that are in terms of inputs only:

Lemma 11. *Let ϕ be a duration formula and let C be an arbitrary controller with inputs $I \supseteq \mathit{free}(\phi)$. Then C satisfies ϕ iff ϕ is valid.*

Proof. It is obvious that validity of ϕ implies that C satisfies ϕ, as the trajectories of C are necessarily a subset of the universe *Traj* of trajectories. For the converse implication observe that satisfaction of ϕ by a trajectory *tr* does only depend on the valuation that *tr* assigns to variables in $\mathit{free}(\phi)$. Assume that ϕ is invalid and let $tr \not\models \phi$. As C has inputs I, C has a trajectory tr' that coincides with *tr* on all variables in $I \supseteq \mathit{free}(\phi)$. As satisfaction of ϕ depends only on the valuation of its free variables, $tr' \not\models \phi$ follows. I.e., C does not satisfy ϕ. ◻

This lemma has direct consequences for the feasibility of automatic synthesis.

Theorem 12. *Any synthesis procedure is necessarily unsound or $\{\lceil P \rceil, \ell = k\}$-incomplete (and hence also $\{\int P\}$-incomplete) or ineffective. I.e., there is no effective procedure that generates solutions for any solvable $\{\lceil P \rceil, \ell = k\}$ control problem.*

Proof. Let *synt* be a sound and complete mapping from $\{\lceil P \rceil, \ell = k\}$ control problems to timed controllers. Let ϕ be a $\{\lceil P \rceil, \ell = k\}$ formula.

According to Lemma 11, the control problem $(\phi, \mathit{free}(\phi))$ has a solution iff ϕ is valid. As *synt* is sound and complete it follows that $(\phi, \mathit{free}(\phi)) \in \mathrm{dom}\,(\mathit{synt})$ iff ϕ is valid. Thus, an effective mapping *synt* would provide a decision procedure for $\{\lceil P \rceil, \ell = k\}$ formulae, in contrast to the undecidability result of [48]. This implies that *synt* cannot be effective. ◻

Thus, automatic synthesis of controllers is impossible even for the restricted class of $\{\lceil P \rceil, \ell = k\}$ control problems, unless one is willing to sacrifice completeness.

However, it is enlightening to observe that the reduction of the decision problem to the synthesis problem performed in above proof is based on the reduction of the validity problem to a satisfaction problem in Lemma 11, which in turn crucially relies on the unconstrained input dynamics of timed controllers. The latter allows inputs to exhibit arbitrary finitely variable dynamics, which permits above reductions. However, embedded real-time controllers are embedded into an environment which may not be able to provide arbitrary finitely variable stimuli. Hence, it is questionable whether the full range of finitely variable trajectories should be regarded as crucial to the satisfaction problem between controllers and control problems. In most (if not all) application domains, more restrictive constraints on the temporal evolution of trajectories can be justified from physical properties of the systems. Therefore, it makes sense to investigate the synthesis problem for Duration Calculus under suitable restrictions of the possible input and output behaviour.

3.5 Constrained Environment Dynamics

In the remainder, we will investigate the synthesis problem under some reasonable constraints on the possible input and output dynamics. Therefore, we say that $\mathit{Traj}_I \subseteq \mathit{Traj}$ is a *constraint on input behaviour* for the inputs $I \subset \mathit{Varname}$ iff $tr \in \mathit{Traj}_I$ implies $tr' \in \mathit{Traj}_I$ for any $tr' \in \mathit{Traj}$ that differs from tr only on non-input variables (i.e., any tr' that satisfies $tr(t)(x) \neq tr'(t)(x) \Rightarrow x \notin I$). Similarly, $\mathit{Traj}_O \subseteq \mathit{Traj}$ is a *constraint on output behaviour* iff $tr \in \mathit{Traj}_O$ implies $tr' \in \mathit{Traj}_O$ for any $tr' \in \mathit{Traj}$ that differs from tr only on non-output variables.

Definition 13 (Satisfaction under behavioural constraints). *We say that a* controller C satisfies Req under input constraint Traj_I and output constraint Traj_O *iff* $\mathit{Traj}_I \cap \mathcal{C}[\![C]\!] \subseteq \mathcal{M}[\![\mathit{Req}]\!] \cap \mathit{Traj}_O$. *I.e., if C is used in a context where the environment guarantees the constraint on input behaviour then C guarantees both Req and the constraint on output behaviour.*

Now, it is straightforward to define when a controller solves a control problem under input and output constraints and when synthesis is sound and complete with respect to given input and output behavioural constraints.

With this machinery, we are now prepared for investigating the synthesis problem under behavioural constraints.

Bounded Variability of Input and Output Behaviour. Considering the fact that any physically realizable reactive system is subject to band-limitedness, an easily justifiable assumption on *realistic* device models is that state changes can only come arbitrarily close in time if they originate from different subsystems. As the number of subsystems in a given technical system is finite this implies that the number of state changes observable at the controller's interface within a time unit is bounded by a system-dependent natural number. An appropriate behavioural model is that of *n-bounded trajectories*, which are those trajectories that exhibit at most n state changes over any unit-length interval of time, where $n \in \mathbb{N}$ is a system-dependent parameter. The set of n-bounded trajectories is denoted Traj_n. Given inputs I and outputs O, we furthermore denote by $\mathit{Traj}_{I,n}$ ($\mathit{Traj}_{O,m}$) the sets of trajectories which after projection to the inputs I (outputs O, resp.) are n-bounded (m-bounded, resp.).

$\mathit{Traj}_{I,n}$ and $\mathit{Traj}_{O,m}$ represent constraints on input and output behaviour. With respect to synthesis it is interesting to see that these easily justifiable constraints suffice to facilitate synthesizing timed controllers from $\{\lceil P \rceil, \ell = k\}$ control problems:

Theorem 14. *There is an effective, sound, and $\{\lceil P \rceil, \ell = k\}$-complete synthesis procedure for timed controllers when input dynamics is constrained to be n-bounded and output dynamics is constrained to be m-bounded for given $n, m \in \mathbb{N}$.*

Proof. Using standard techniques, an effective mapping of $\{\lceil P \rceil, \ell = k\}$ formulae that do contain *exactly one, outermost, negation* to timed automata that recognize their counterexamples of *finite* variability can be defined. Using the timed regular expression notation proposed by Asarin, Caspi, and Maler in [2], such an automata-theoretic representation of the counterexamples can be achieved through the mapping

$$Counterexamples(\neg\phi) \stackrel{\text{def}}{=} FiniteModels(\phi) \cdot \alpha^{\omega}$$

of $\{\lceil P \rceil, \ell = k\}$-formulae to timed regular expressions, where α is the set of min-terms over the free state variables of ϕ and

$$\begin{aligned}
FiniteModels(\lceil P \rceil) &\stackrel{\text{def}}{=} \left(\left(\bigvee\nolimits_{a \in \alpha, a \models P} a\right)^{*}\right)_{(0,\infty)} \quad , \\
FiniteModels(\ell = k) &\stackrel{\text{def}}{=} (\alpha^{*})_{[k,k]} \quad , \\
FiniteModels(\phi \wedge \psi) &\stackrel{\text{def}}{=} FiniteModels(\phi) \wedge FiniteModels(\psi) \quad , \\
FiniteModels(\phi \,;\, \psi) &\stackrel{\text{def}}{=} FiniteModels(\phi) \cdot FiniteModels(\psi) \quad .
\end{aligned}$$

As the timed regular languages of variability $n + m$ are furthermore effectively closed under complementation (relative to the set of $(n + m)$-bounded trajectories) [45], this procedure can be extended to deal with inner negation also if only the $(n + m)$-bounded counterexamples are of interest. Therefore, any formula ϕ of the $\{\lceil P \rceil, \ell = k\}$ fragment of DC can be effectively assigned a timed automaton $A_{Traj_{n+m} \setminus \phi}$ recognizing its $(n + m)$-bounded counterexamples. While the deterministically recognizable timed regular languages are in general a proper subclass of the non-deterministically recognizable ones, these two classes coincide for the timed regular languages of variability $n + m$. In particular, $A_{Traj_{n+m} \setminus \phi}$ can be made deterministic.

Now, $A_{Traj_{n+m} \setminus \phi}$ can be easily extended to an (again deterministic) timed automaton $A_{Traj_{n+m} \setminus (\phi \cup Traj_{O,m}) \cap Traj_{I,n}}$ that, besides recognizing any $(n + m)$-bounded counterexample of ϕ, also recognizes all trajectories violating the m-boundedness constraints on outputs, yet excludes trajectories violating the n-boundedness constraint on inputs. This effectively reduces the controller synthesis problem to a strategy construction problem in a timed regular game, where $A_{Traj_{n+m} \setminus (\phi \cup Traj_{O,m}) \cap Traj_{I,n}}$ is the game graph. As effective synthesis procedures for timed regular games are known from the literature (cf. [3, 26]), this solves the controller synthesis problem. $\square$

Unfortunately, Theorem 14 does not generalize to $\{\int P = k\}$ control problems, as their requirements formulae feature accumulated durations and thus are considerably more expressive than the $\{\lceil P \rceil, \ell = k\}$ control problems: As was shown in [14], page 35ff., by means of a real-time pumping lemma, it is in general undecidable whether $Traj_n \subseteq \mathcal{M}[\![\phi]\!]$ for $\{\int P = k\}$ formulae ϕ. Thus, by an argument akin to that used in the proof of Theorem 12, it follows that there is no effective, sound, and $\{\int P = k\}$-complete synthesis procedure even when interface dynamics is restricted to n-bounded inputs and m-bounded outputs.

Time-wise Discrete Input and Output Behaviour. In order to obtain automatic synthesis procedures for $\{\int P = k\}$ control problems, we may try to reduce the model class that is regarded crucial to satisfaction still further. If the devices to be designed are embedded into a synchronously clocked environment it makes sense to consider trajectories that are changing state only at evenly spaced time instants. This is captured by studying *time-wise discrete trajectories*, where a trajectory $tr \in \mathit{Traj}$ is called time-wise discrete iff it has discontinuities only in time instants which are multiples of the time unit. The set of time-wise discrete trajectories is denoted DTraj. Accordingly, the input constraint that restricts inputs to change only in time instants which are multiples of the time unit is denoted DTraj_I, while the corresponding behavioural output constraint is denoted DTraj_O.

The restriction to such time-wise discrete interface behaviour allows automatic synthesis even for $\{\int P = k\}$ control problems:

Theorem 15. *There is an effective, sound, and $\{\int P = k\}$-complete (and thus also $\{\lceil P \rceil, \ell = k\}$-complete) synthesis procedure for timed controllers when input and output behaviour is constrained to be time-wise discrete.*

Proof. It is a tedious, yet mostly straightforward exercise to show that for any formula ϕ of the $\{\int P = k\}$ fragment, $\mathcal{M}[\![\phi]\!] \cap \mathit{DTraj}$ is an unrolling of an ω-regular language to real-time based on the convention that one letter per time unit is traversed. A corresponding ω-automaton $\mathit{Aut}_\phi^{\mathrm{twd}}$ can be effectively constructed (the construction is fully pursued in [14, chapter 6.3]).

Now, a similar construction as in the proof of the previous theorem can be applied to $\mathit{Aut}_\phi^{\mathrm{twd}}$ to obtain an appropriate game graph representing the behaviorally constrained synthesis problem. However, this time the game graph obtained is ω-regular as $\mathit{Aut}_\phi^{\mathrm{twd}}$ is an untimed finite automaton and the behavioural constraints can also be formalised with untimed automata using the convention that one letter per time unit is traversed. This effectively reduces the synthesis problem to strategy construction in ω-regular games. As algorithms for the latter are well-known (cf. e.g. [44]), this yields the desired synthesis method. Details of the construction can be found in [13, 14]. □

As, for example, the ProCoS gas-burner requirements specification [41] can be expressed in the $\{\int P = k\}$ subset of DC, this allows automatic synthesis of a timed controller directly from the requirements specification of the gas-burner, which is shown in the next section.

3.6 A Case Study: Synthesizing a Synchronous Controller for the ProCoS Gas-Burner

The ProCoS gas-burner [41, **?**] is a simple model of a computer-controlled gas-burner, depicted in Fig. 1. Its embedded controller has just three binary control lines connected to the environment: *hr* signals heat requests from a thermostat, *fl* signals whether the flame is burning, and *gas* controls the gas valve. The gas valve is the only actuator in the system, and gas is expected to usually ignite spontaneously once the gas valve is opened.[6] However, gas may sometimes fail to ignite, leading to an increasing concentration of flammable gas in the environment, which is an obvious risk. The task of the controller is to prevent unsafe gas concentration in the environment through detection of ignition failures and appropriate actions, and to deliver service as required by *hr* if ignition works as expected.

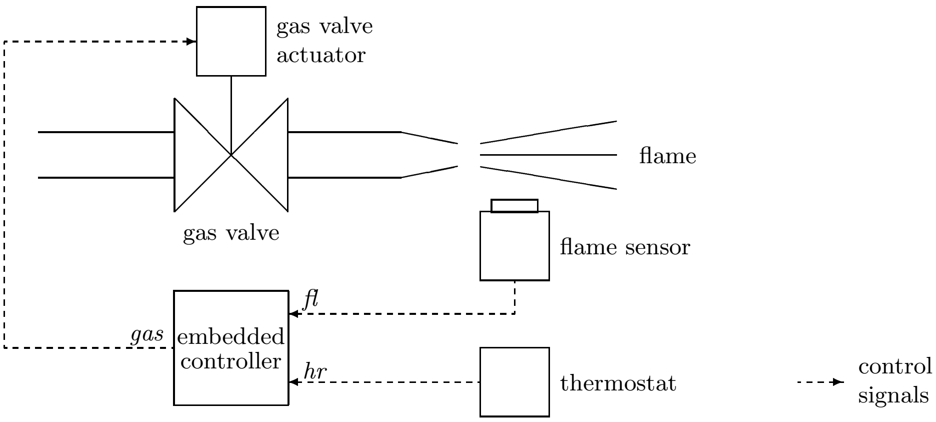


Fig. 1. The ProCoS gas-burner

The corresponding requirements can be easily formalized using $\{\int P = k\}$ formulae. Indeed, the patterns occurring have been the key motivation for development of the Duration Calculus. In the following, we stick to the original requirements given in [41, 9], but sometimes reduce time constants.

The foremost requirement the controller has to ensure is that the gas concentration in the environment is kept below flammable level. As sensors for directly detecting the gas concentration are expensive, the gas concentration has to be safely estimated from the length and temporal distance of periods of leakage of unignited gas to the environment. We assume that safety engineers have shown that the system is safe if unignited gas may not leak from the burner for more than 3 seconds within any 6 seconds of operation.[7] Using DC, this can be for-

[6] The reader reluctant to the idea of spontaneous ignition may equally well think of an ignition device being coupled to the gas valve such that both can be simultaneously controlled by the single control signal *gas*.

[7] In the original formulation of the problem, the corresponding figures where a maximum of 4 seconds leak time within 30 seconds, but these have been reduced in order to make the synthesized controller fit on a page.

malized as

$$\mathit{safe} \stackrel{\text{def}}{=} \Box\left(\ell < 6 \Rightarrow \textstyle\int \mathit{leak} < 3\right) \quad ,$$

where $\Box\phi \stackrel{\text{def}}{=} \neg(\texttt{true}; (\neg\phi); \texttt{true})$ and $\int P < k \stackrel{\text{def}}{=} \neg(\int P = k\,; \texttt{true})$. Using the available sensors, leakage — or indeed a sufficient approximation of leakage — is detected through observing the flame sensor when the gas valve is open: gas is deemed to be leaking iff the flame sensor senses that the flame is not burning while the gas valve is open.

$$\mathit{leak} \stackrel{\text{def}}{=} \mathit{gas} \wedge \neg \mathit{fl} \quad .$$

Requirement *safe* alone is easily satisfied: as leaks can only occur when the gas valve is open, a controller permanently setting control line *gas* to false and thus keeping the gas valve closed will satisfy *safe*. However, a customer will not be satisfied with a gas-burner never delivering service. Therefore, some requirements concerning controllability of the system through *hr* are added. First, we require that $\neg hr$ will shut the gas supply within one second:

$$\mathit{stop} \stackrel{\text{def}}{=} (\lceil \neg hr \rceil \wedge \ell = 1) \rightsquigarrow \lceil \neg gas \rceil \quad ,$$

where $\phi \rightsquigarrow \lceil P \rceil \stackrel{\text{def}}{=} \neg(\texttt{true}; \phi; \lceil \neg P \rceil; \texttt{true})$. Furthermore, we would like to require that *hr* leads to heat supply within a reasonable time span. However, this demand can only be realized if gas does not fail to ignite after opening the valve. Therefore, the startup requirement is relative to an environment assumption which formalizes the normal ignition behaviour. The normal ignition behaviour is that gas ignites soon after opening the valve such that the flame sensor reports a burning flame within 2 seconds. Whenever this is the case, heat should be supplied after at most 8 seconds of continuous heat request:

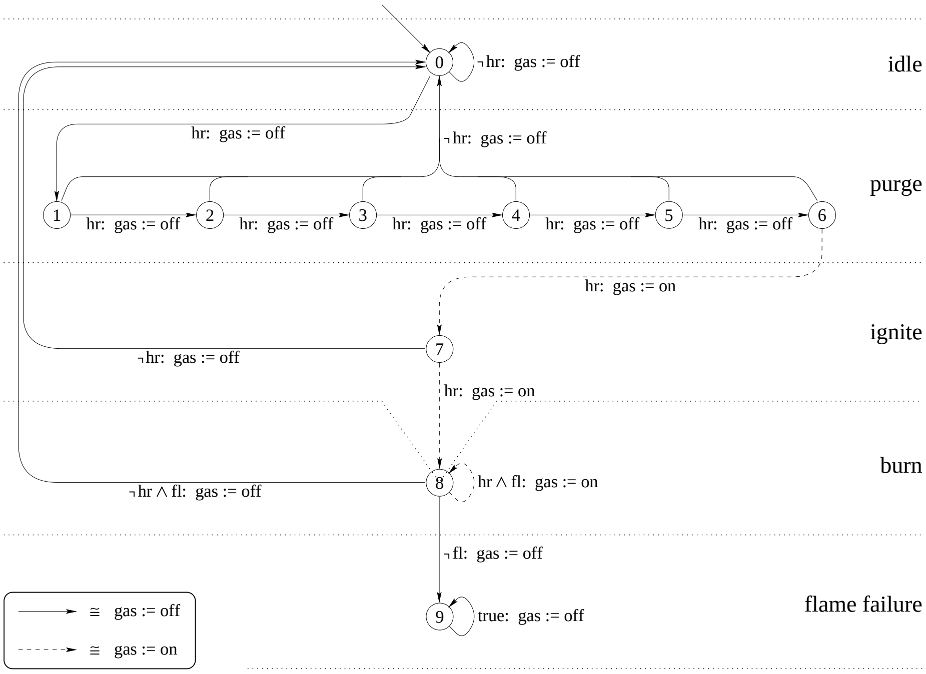
$$\begin{aligned} \mathit{start} &\stackrel{\text{def}}{=} \mathit{flame\ ok} \Rightarrow ((\lceil hr \rceil \wedge \ell = 8) \rightsquigarrow \lceil \mathit{fl} \rceil) \quad , \\ \mathit{flame\ ok} &\stackrel{\text{def}}{=} (\lceil gas \rceil \wedge \ell = 2) \rightsquigarrow \lceil \mathit{fl} \rceil \quad . \end{aligned}$$

The requirement to be guaranteed by the embedded controller is the conjunction of above three requirements:

$$\mathit{GBReq} = \mathit{safe} \wedge \mathit{stop} \wedge \mathit{start} \quad .$$

Furthermore, the design has to respect the signature imposed by the application, namely that *hr* and *fl* are inputs to the controller and that *gas* is an output. The control problem to be solved thus is $(\mathit{GBReq}, \{hr, \mathit{fl}\})$.

The synthesis procedure outlined in Theorem 15 has successfully been applied to this control problem. As the control problem is underconstrained (sometimes it allows free choice between switching *gas* on or off in a certain time instant), this yields a non-deterministic control strategy with respect to the controlled output *gas*. Adding a simple heuristics for resolving this nondeterminism, namely



The automatically synthesized timed transition table for the gas burner control problem. As the timed transition table takes one transition every time unit, the clock constraints and resets have been omitted from the transition diagram. For completeness, a single clock c has to be added, and each transition has to be decorated with guarding condition $c = 1$ and reset function $c := 0$.
Incidentally, the synthesized controller resembles the phase design of the manually developed gas burners of [41, 9]. The corresponding phase names used in those designs are indicated on the right.

Fig. 2. The synthesized gas-burner controller

switching *gas* off whenever possible, and finally applying automaton minimization, we obtained a control automaton with only 10 states, which is depicted in Fig. 2.

We were surprised to see that the resulting controller, although generated by a fully effective procedure, even resembles the phase structure of the manually developed controllers of A. P. Ravn, H. Rischel, and K. M. Hansen [41]. Ravn, Rischel, and Hansen developed their control skeleton around the idea of an *idle phase*, where the controller waits for the next heat request, a *purge phase*, where the gas concentration in the environment is reduced through keeping gas shut off for a while before doing an ignition attempt, an *ignite phase* opening the valve long enough to get the flame burning if *flame ok* holds, and a *burn phase* being entered once the flame is stably burning and being left if it either extinguishes or heat request *hr* is withdrawn. See Fig. 2 for more details of the phase design of [41] and how these phases can be identified in the automatically generated gas-burner controller.

3.7 Complexity of Synthesis

While the gas burner case study shows that engineering-quality controllers can in principle be obtained from DC-based automatic synthesis, it is the complexity of the automata-theoretic constructions involved that impairs practicality of Duration Calculus as a source language for embedded systems synthesis. Even for the simplest subsets of DC that feature chop and negation — not even metric time is necessary — the worst case complexity of the synthesis problem is non-elementary in the size of the specification, irrespective of the particular trajectory class used. The reason is that chop and negation are akin to concatenation and complement of languages s.t. the non-elementary emptiness problem of extended regular languages can be linearly encoded in DC (see [14, Lemma 6.25] for the details of such an encoding). Thus, for practical applications it may be better to seriously restrict the use of chop and/or negation, as is done in the work of H. Dierks [11], where synthesis from a subset of the so-called DC implementables [40] is explored. DC implementables contain exactly one, outermost negation and are restricted to certain patterns of using chop. Beyond circumventing the non-elementariness problem by essentially forbidding negation, the gains of the extra restrictions adopted by Dierks are that timing constants can be dealt with essentially syntactically.

However, our focus has less been on exhibiting practical controller synthesis procedures than on demonstrating the fundamental impact that observational constraints of the environment have on the synthesis process. To this end, we have been able to show that by suitable restriction of the model class used in behavioural descriptions of system dynamics, automatic synthesis for large and even undecidable subsets of Duration Calculus becomes theoretically possible. Thereby, the particular behavioural restrictions adopted are motivated by physical properties of practical control problems, namely band-limitedness of reactive systems and synchronicity of clocked systems.

4 Conclusion

We have summarized two approaches to the provably correct and automatic implementation of abstractly described hard real-time controllers. The more conservative of the two is an extension of an imperative programming language by hard real-time commands that allows one to specify upper bounds for the execution time of basic blocks. This extension allows one to specify absolute timing requirements in the imperative source code, thereby obliging the compiler to generate corresponding machine code. The other source language investigated is

Duration Calculus, a metric-time temporal logic designed for reasoning about real-time systems at a high level of abstraction.

Both approaches exploit in an essential way that the observational power of the environment is limited: Firstly, the majority of the state-space of the embedded controller is hidden from it due to the clear-cut interface between the two. Secondly, protocol restrictions or even physical limitations, like band-limitedness, apply to this interface. These observational limitations can be exploited for gaining implementation freedom, thus facilitating correct implementation of idealized behavioral models. While in the compilation approach this is used to justify the idealization of immediate execution for internal statements it is exploited in the synthesis approach for overcoming undecidability of the synthesis problem.

A key difference between the approaches is the complexity of the resulting procedures. In the case of synthesis from Duration Calculus it is in general non-elementary (ways of improving on this have been discussed at the end of Sect. 3). The complexity of the compilation procedure on the other hand is linear. The other side of the coin is of course the power of the formalisms. The compilation work uses an imperative programming language. It requires one to specify exactly how the desirable behavior is achieved and timing requirements have to be specified rather locally, although this is defused a bit by the immediate execution idealization together with time bounds. In contrast, Duration Calculus supports, by being a full-fledged metric-time temporal logic, extremely advanced programming techniques when used as a source language for automatic code generation. A prominent example, which builds upon the availability of logical negation, is the paradigm of *programming by counterexample*, i.e. specifying what should never happen rather than saying how exactly to achieve this. Furthermore, global timing requirements may be easily specified.

Acknowledgements. The research reported in this article has mainly been performed while the authors were with the Computer Science Department of the Christian-Albrechts-Universität Kiel, Germany, working in the ProCoS project and related projects under the supervision of Hans Langmaack. Over many years he strongly influenced the direction of our research through constant encouragement, insistence, and support. It is a great pleasure to present a summary of the results in a volume dedicated to him on the occasion of his retirement.

We thank the members of the ProCoS project for many fruitful discussions as well as J Moore and Bernhard Steffen for many valuable comments on a draft version of this article. We acknowledge the support of the European Union under grants ESPRIT BRA 3104 and 7071 and of the German Research Council DFG under contract DFG La 426/13-1,2.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

2. E. Asarin, P. Caspi, and O. Maler. A Kleene theorem for timed automata. In G. Winskel, editor, *12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*. IEEE Computer Society Press, 1997.
3. E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems II*, LNCS 999, Springer-Verlag, 1995.
4. R. J. R. Back and J. von Wright. Refinement calculus, Part I: Sequential nondeterministic programs. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems — Models, Formalisms, Correctness. REX Workshop*, LNCS 430, pages 42–66, Springer-Verlag, 1989.
5. R. J. R. Back and J. von Wright. Duality in specification languages: A lattice theoretic approach. *Acta Informatica*, 27(7):583–625, 1990.
6. G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT-Press, to appear.
7. W. R. Bevier, W. A. Hunt, J S. Moore, and W. D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4), 1989.
8. J. P. Bowen, C. A. R. Hoare, H. Langmaack, E.-R. Olderog, and A. P. Ravn. A ProCoS II project final report: ESPRIT Basic Research project 7071. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 59, 1996.
9. J. P. Bowen, M. Fränzle, E.-R. Olderog, and A. P. Ravn. Developing correct systems. In *Proc. 5th Euromicro Workshop on Real-Time Systems, Oulu, Finland*, pages 176–189. IEEE Computer Society Press, 1993.
10. G. M. Brown. Towards truly delay-insensitive circuit realizations of process algebras. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*, Workshops in Computing, pages 120–131. Springer-Verlag, 1991.
11. H. Dierks. Synthesizing controllers from real-time specifications. In *Tenth International Symposium on System Synthesis (ISSS '97)*, pages 126–133. IEEE Computer Society Press, 1997.
12. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
13. M. Fränzle. Synthesizing controllers from duration calculus. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '96)*, LNCS 1135, pages 168–187, Springer-Verlag, 1996.
14. M. Fränzle. *Controller Design from Temporal Logic: Undecidability need not matter*. Dissertation, Technische Fakultät der Christian-Albrechts-Universität Kiel, Germany, 1997. Available as Bericht Nr. 9710, Institut für Informatik und Prakt. Mathematik der Christian-Albrechts-Universität Kiel, Germany, and via WWW under URL `http://ca.informatik.uni-oldenburg.de/~fraenzle/diss.html`.
15. M. Fränzle and M. Müller-Olm. Towards provably correct code generation for a hard real-time programming language. In P. A. Fritzson, editor, *Compiler Construction '94, 5th International Conference Edinburgh U.K.*, LNCS 786, pages 294–308, Springer-Verlag, 1994.
16. P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87, 1991. Also in [32].
17. J. D. Guttman, J. D. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.
18. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.
19. He Jifeng, I. Page, and J. P. Bowen. Towards a provably correct hardware implementation of Occam. In G. J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods*, LNCS 683, pages 214–225. Springer-Verlag, 1993.

20. C. A. R. Hoare. Refinement algebra proves correctness of compiling specifications. In C. C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, Workshops in Computer Science, pages 33–48. Springer-Verlag, 1991.
21. C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorenson, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–687, 1987.
22. C. A. R. Hoare, He Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.
23. inmos limited. *Transputer Instruction Set – A Compiler Writer's Guide*. Prentice Hall International, first edition, 1988.
24. J. J. Joyce. Totally verified systems: Linking verified software to verified hardware. In Leeser and Brown [25], pages 177–201.
25. M. Leeser and G. Brown, editors. *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, LNCS 408, Springer-Verlag, 1990.
26. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In Meyer and Puech [29], pages 229–242.
27. A. J. Martin. The design of a delay-insensitive microprocessor: An example of circuit synthesis by program transformation. In Leeser and Brown [25], pages 244–259.
28. J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J. Schwarz, editor, *Proc. Symp. Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.
29. E. W. Meyer and C. Puech, editors. *Symposium on Theoretical Aspects of Computer Science (STACS 95)*, LNCS 900, Springer-Verlag, 1995.
30. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.
31. J S. Moore. *Piton, A Mechanically Verified Assembly-Level Language*. Kluwer Academic Publishers, 1996.
32. C. Morgan and T. Vickers (Eds.). *On the Refinement Calculus*. Springer-Verlag, 1994.
33. F. L. Morris. Advice on structuring compilers and proving them correct. In *Proceedings ACM Symposium on Principles of Programming Languages (PoPL'93)*, pages 144–152, 1973.
34. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
35. J. M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.
36. M. Müller-Olm. A short description of the prototype compiler. ProCoS Technical Report Kiel MMO 14/1, Christian-Albrechts-Universität Kiel, Germany, August 1995.
37. M. Müller-Olm. *Modular Compiler Verification: A Process-Algebraic Approach Advocating Stepwise Abstraction*, LNCS 1283, Springer-Verlag, 1997.
38. T. S. Norvell. Machine code programs are predicates too. In D. Till, editor, *6th Refinement Workshop*, Workshops in Computing. Springer-Verlag and British Computer Society, 1994.
39. D. J. Pavey and L. A. Winsborrow. Demonstrating equivalence of source code and PROM contents. *The Computer Journal*, 36(7):654–667, 1993.
40. A. P. Ravn. *Design of Embedded Real-Time Computing Systems*. Doctoral dissertation, Department of Computer Science, Danish Technical University, Lyngby, DK, 1995. Available as technical report ID-TR: 1995-170.

41. A. P. Ravn, H. Rischel, and K. M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Transactions on Software Engineering*, 19(1):41–55, 1993.
42. A. P. Ravn and H. Rischel. Real-time constraints in the ProCoS layers. In E. R. Olderog and B. Steffen, editors, *Correct System Design*, this volume.
43. J. W. Thatcher, E. G. Wagner, and J. B. Wright. More on advice on structuring compilers and proving them correct. *Theoretical Computer Science*, 15:223–249, 1981.
44. W. Thomas. On the synthesis of strategies in infinite games. In Meyer and Puech [29], pages 1–13.
45. T. Wilke. *Automaten und Logiken zur Beschreibung zeitabhängiger Systeme*. Dissertation, Technische Fakultät der Christian-Albrechts-Universität Kiel, Germany, 1994.
46. T. Wilke. Specifying timed state sequences in powerful decidable logics and timed automata. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '94)*, LNCS 863, pages 694–715, Springer-Verlag, 1994.
47. M. W. Wilkes and J. B. Stringer. Micro-programming and the design of the control circuits in an electronic digital computer. *Proc. Cambridge Phil. Soc.*, 49:230–238, 1953. also *Annals of Hist. Comp.* **8**, 2 (1986) 121–126.
48. Zhou Chaochen, M. R. Hansen, and P. Sestoft. Decidability and undecidability results for duration calculus. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *Symposium on Theoretical Aspects of Computer Science (STACS 93)*, LNCS 665, pages 58–68, Springer-Verlag, 1993.
49. Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.