Towards Provably Correct Code Generation for a Hard Real-Time Programming Language

Martin Fränzle and Markus Müller-Olm *

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Preußerstr. 1–9, D-24105 Kiel, Germany
E-Mail: {mf|mmo}@informatik.uni-kiel.d400.de

Abstract. This paper sketches a hard real-time programming language featuring operators for expressing timeliness requirements in an abstract, implementation-independent way and presents parts of the design and verification of a provably correct code generator for that language. The notion of implementation correctness used as an implicit specification of the code generator pays attention to timeliness requirements. Hence, formal verification of the code generator design is a guarantee of meeting all deadlines when executing generated code.

1 Introduction

For an increasing number of applications, software failures may be very costly in terms of economic loss or even human suffering. This is particularly true for hard real-time control programs, where correctness does not only depend on logical correctness of results, but also on timely delivery of services.

Testability of such software is poor, as timing constraints add an additional dimension to the behaviour to be examined and, furthermore, dictate the speed of the testing process. Traditionally, this problem has lead to a purely pragmatic separation of concerns: Algorithmic correctness is dealt with by using programming notation, which — whenever reliability is required — is subject to thorough or sometimes even formal investigation, whereas timing properties are dealt with a posteriori by inspection of the machine code generated by a compiler.

Unfortunately, this approach leaves two different problems unresolved: On one hand, investigation of the algorithmic properties stops too early in the development process, as due to the absence of verified development software, particularly compilers, improper target code may still result. On the other hand, inspection of timing properties starts too late in the development process, possibly leading to expensive iterations in development.

Both problems could be resolved by a programming language supplying means to express the relevant patterns of timeliness, together with a highly

* This report reflects work that has been partially funded by the Commission of the European Communities under ESPRIT Basic Research Action 7071 "ProCoS II" (Provably Correct Systems II) and by the Deutsche Forschungsgemeinschaft under contract DFG La 426/13-1.

dependable, i.e. correctness-preserving, compiler. Firstly, a dependable compiler would give certainty about the correctness of machine code whenever the same is true for the source program, making backcompilation, machine code inspection, and similar costly target code analysis techniques superfluous. Secondly, implementation-independent means of expressing timing constraints at the source level could make timing subject to the same paradigm of stepwise development currently successfully applied to algorithmic development.

The traditional approach to achieving reliability of compilers is validation by running test suites, i.e. by compiling a number of test programs and testing their executables. It is questionable whether this can give sufficient confidence to replace target code inspection in safety-critical software development, as test programs normally exhibit rather simple behaviour to allow detection of errors. As stated earlier, this is especially true with respect to timing. Hence, highly dependable compilers for real-time programming languages cannot be constructed without formal verification of their vital parts, particularly code generators.

The aim of the ESPRIT project "Provably Correct Systems II" (abbreviated ProCoS II) is to contribute to the state of the art in development of correct software-hardware systems for embedded, safety-critical real-time control by elaborating an experimental framework for the stepwise, correctness-preserving development of such systems. In this framework, the programming language plays the role of an interface between high levels of abstraction, where system development requires human ingenuity, and low levels of abstraction, where correctness preserving transformations can be applied fully automatic by compilers.

Thus, one of the immediate goals of ProCoS II is to provide a prototype of a real-time programming language designed to solve the interfacing problem between system specification and system implementation and to develop a verified prototype compiler for this language. On one hand, the programming language has to provide sufficiently expressive timing operators to make program correctness arguments without recurrence to a particular implementation possible. On the other hand, it must be implementable on realistic hardware by avoiding overly idealized timing properties.

A prototype compiler might be taken as a pragmatic proof of the latter claim. But far beyond this, its existence will be a demonstration of the feasibility of high-dependability compiler development, opening the perspective for banning target-level work from safety-critical system development.

This paper gives an overview on the work undertaken thus far in ProCoS II towards this goal. Section 2 gives an introduction to ProCoS's real-time programming language TimedPL, and section 3 shows how to extend it to a larger real-time process language used to embedd TimedPL and the target machine language into a common framework in section 6. In section 4, a general notion of one process implementing another is sketched, which is exploited in section 7 to define the detailed correctness predicates for code to be generated for the different syntactic categories of TimedPL. In section 8, these correctness predicates are put to use by stating concrete target code patterns for some source code patterns and by exploiting the algebra of the process language to show implementation correctness.

2 The Real-Time Programming Language TimedPL

ProCoS's real-time programming language TimedPL [FvK93] features concepts to describe both the desired logical and temporal behaviour of programs. Using this basic distinction, TimedPL may be understood as being composed of

1. a simple imperative kernel for describing logical behaviour via imperative sequential algorithms,
2. timing operators for decoration of sequential algorithms, assigning execution times to their logical behaviour, and
3. parallel composition of timed sequential algorithms, introducing concurrency for the sake of both expressivity and efficiency.

TimedPL is closely related to `occam` [inm88a], albeit dropping some features from `occam` for scientific treatability, but seriously adding to its expressivity in ProCoS's central field of research, namely real-time software development.

Like in `occam`, a program is a set of sequential processes executing in parallel and communicating with each other and with the environment solely via unidirectional, synchronous channels. Synchronicity means that whenever a process wants to communicate on a channel it has to wait until the partner residing at the other end of that channel is also willing to communicate. By disallowing shared variables, there is a clear syntactic distinction between effects that might affect parallel partners, namely communications on channels, and those that are completely encapsulated in a sequential process and may be optimized by a compiler, namely state transformations.

Table 1: Outline of TimedPL's syntax

$$
\begin{array}{rcl}
\langle \mathit{program} \rangle & ::= & \langle \mathit{chandecl} \rangle^{*}\ \texttt{par}\ \big(\langle \mathit{parallel\ component} \rangle^{+}\big) \\
\langle \mathit{chandecl} \rangle & ::= & \texttt{chan}\ \langle \mathit{channel} \rangle\ \texttt{latency}\ \langle \mathit{time} \rangle : \\
\langle \mathit{time} \rangle & ::= & \langle \mathit{non-negative\ real} \rangle \\
\langle \mathit{parallel\ component} \rangle & ::= & \langle \mathit{inaccuray\ spec} \rangle \langle \mathit{sequential\ program} \rangle \\
\langle \mathit{inaccuray\ spec} \rangle & ::= & \texttt{drift}\ \langle \mathit{real\ larger\ 1} \rangle : \texttt{granularity}\ \langle \mathit{time} \rangle : \\
\langle \mathit{sequential\ program} \rangle & ::= & \langle \mathit{variable\ declaration} \rangle^{*}\ \langle \mathit{sequential\ process} \rangle \\
\langle \mathit{variable\ declaration} \rangle & ::= & \texttt{var}\ \langle \mathit{variable} \rangle^{*} : \\
\langle \mathit{sequential\ process} \rangle & ::= & \texttt{skip} \mid \texttt{stop} \mid \langle \mathit{variable} \rangle := \langle \mathit{expression} \rangle \mid \\
 & & \langle \mathit{channel} \rangle ? \langle \mathit{variable} \rangle \mid \langle \mathit{channel} \rangle ! \langle \mathit{variable} \rangle \mid \\
 & & \langle \mathit{sequential\ process} \rangle ; \langle \mathit{sequential\ process} \rangle \mid \\
 & & \texttt{if}\ \langle \mathit{bool\ exp.} \rangle\ \texttt{then}\ \langle \mathit{seq.\ proc.} \rangle\ \texttt{else}\ \langle \mathit{seq.\ proc.} \rangle \mid \\
 & & \texttt{while}\ \langle \mathit{bool\ expression} \rangle\ \texttt{do}\ \langle \mathit{sequential\ process} \rangle \mid \\
 & & \langle \mathit{upper\ bound} \rangle \mid \langle \mathit{alternative\ statement} \rangle \\
\langle \mathit{upper\ bound} \rangle & ::= & [\langle \mathit{sequential\ process} \rangle] \leq \langle \mathit{time} \rangle \\
\langle \mathit{alternative\ statement} \rangle & ::= & \texttt{alt}\ \big(\langle \mathit{timed\ alternative} \rangle^{*}\big) \\
\langle \mathit{timed\ alternative} \rangle & ::= & \texttt{wait}\ \langle \mathit{time} \rangle : \langle \mathit{channel} \rangle ? \langle \mathit{variable} \rangle \rightarrow \langle \mathit{seq.\ proc.} \rangle
\end{array}
$$

2.1 The Imperative Kernel

The imperative kernel of TimedPL is essentially the language of while programs, extended by input and output commands to unidirectional, synchronous channels, and by input-guarded alternatives providing a means to conditionally react upon different external stimuli arriving via channels.

More specificly, untimed sequential processes are composed from the atomic processes `skip` (representing the identical state transformation), `stop` (meaning deadlock, i.e. idling forever), assignments, input commands $c?x$ (waiting for communication on channel c and, if communication proceeds, assigning the communicated value to variable x), and output commands $c!e$ (waiting for communication on channel c and, if communication proceeds, sending the value of the expression e along the channel). Atomic processes can be composed by sequential composition, conditionals, while-loops, and input guarded alternatives to form more complicated sequential algorithms.

The imperative kernel of TimedPL is intended for describing the logical behaviour of sequential algorithms. Consequently, unless restricted by addition of timing operators (as described in the next section), execution speed of sequential processes is completely unspecified. This means that any implementation yielding correct logical behaviour, i.e. input-output sequences, is acceptable regardless of its speed.

More specificly, only runtime of TimedPL's *atomic* processes is unspecified, whereas composition operators, even those involving evaluation of control constructs, do *not* take extra execution time beyond that taken by the component processes. The latter convention significantly enhances the expressive power of composition operators when dealing with timed component processes.

2.2 Timing Operators

The intended field of application of TimedPL is hard real-time programming, i.e. construction of programs where correctness does not only depend on algorithmic well-behavedness, but also on never missing any deadline, be it a lower or an upper bound for the time of delivering a certain service. Therefore, we need mechanisms to constrain the runtime consumption of parts of sequential algorithms. TimedPL offers three such mechanisms:

1. upper bounds on the time spent by sequential processes for state transformations,
2. delayed readiness of guarding communications in alternatives, and
3. upper bounds on the time-to-communication taken by a communication command when its communication partner is ready, called the communication's *latency*.

Upper bound timing. Upper bounds can be placed anywhere inside a sequential process for bounding execution time of the enclosed part, which is itself a sequential process. The upper bound timing operator

$$\langle \textit{upper bound} \rangle ::= [\langle \textit{sequential process} \rangle] \leq \langle \textit{time} \rangle$$

confines the enclosed sequential process to spend at most the amount allowed by the upper bound of time controlled by itself, i.e. not spent waiting for communication partners, until termination. If this contradicts with timing conditions imposed by inner timed alternatives, the semantics is miraculous, meaning that the process is not implementable and has to be rejected by the compiler.

Timed alternatives. Lower bounds on reactivity can be achieved by using the delayed readiness of guarding communications offered by timed alternatives

$$\langle \textit{timed alternative} \rangle ::= \texttt{wait}\ \langle \textit{time} \rangle : \langle \textit{channel} \rangle ? \langle \textit{variable} \rangle \rightarrow \langle \textit{sequential process} \rangle\ .$$

An alternative statement consists of processes guarded by input guards, where the readiness of the guarding communications is established when the delay time stated in the guard has elapsed. In contrast to occam, timing is hard, i.e. readiness is established *at* the moment the delay elapses, not arbitrarily thereafter. The latter condition might seem unimplementable due to the impossibility of *absolutely* exact hardware clocks, but TimedPL's semantics features a distinction between local clocks (which may be inaccurate) and global time. This distinction is part of the definition of TimedPL's parallel composition operator, cf. section 2.3.

Communication latencies. Often systems communicating through synchronous channels are understood under the *maximum progress assumption*, where a communication has to take place as soon as both communication partners are simultaneously ready for a communication. This gives nice timing properties, but is unrealistic even for systems with an own processor for each parallel component, as hardware and protocol delays in detecting readiness of a communication partner do not only limit reactivity, but may even blurr the temporal order of events.

In TimedPL programs, a *lower bound on progress* of each communication is explicitely stated. I.e. if communication partners are simultaneously ready for a communication, communication of that or another competing event need not happen unless both communication partners have been simultaneously ready for the event for a specified amount of time, called the *latency* associated to that event. Communication latencies are assigned at a per-channel basis by making latency specifications part of the channel declaration.

2.3 Parallel Composition

A real-time programming language not offering parallelism would be incomplete with respect to both efficiency of progams and expressibility of control algorithms requiring concurrent actions. Hence, TimedPL programs are systems of timed communicating sequential processes combined by outermost parallelism.

In any implementation, timing properties can be guaranteed only with respect to hardware timers, which, unfortunately, do not accurately reflect real-world time due to clock drift and discretization. The consequences of identification of both timing regimes on the behaviour of synchronous systems can be drastic, as independent subsystems may loose synchronicity due to the imperfection of technical clock devices. Thus, we have to model the effects of timer inaccuracies in the semantics of TimedPL to achieve reliable designs.

The concept of coping with these implementation dependencies offered by TimedPL is straightforward: The programmer has to state acceptable clock tolerances for the individual parallel components. Each parallel component is prefixed by an inaccuracy specification stating the maximum allowed drift and discreteness of its local clock. Semantically, these inaccuracy specifications can be taken as a guarantee by the programmer that the correctness of his control algorithm will not be affected by local clocks being imprecise in the stated range. Similarly to the mathematical treatment of component tolerances by the calculus of accidental error in, e.g., electrical engineering, a calculus of nondeterminism caused by clock inaccuracy and of its propagation through different programming operators can be elaborated, providing a mathematically sound basis for mostly separating the problem of dealing with implementation tolerances from the design of a real-time control system as such. Work undertaken in ProCoS II on the semantic basis of such a calculus of timing inaccuracy can be found in [FMO93].

3 Extending TimedPL Towards a Process Calculus

TimedPL's syntax, as outlined in table 1, defines different syntactic layers, namely programs, parallel components, sequential programs, and sequential processes, representing implementation concepts and thus making sense for a compilable language. But when reasoning about real-time control processes, that syntactic variety can be a burden. A more homogeneous process language TimedProc used in the remainder of this article as a framework for reasoning can be derived from TimedPL by dropping syntactic restrictions. In TimedProc, all the syntactic productions defining TimedPL's syntactic classes $\langle \textit{program} \rangle$, $\langle \textit{parallel component} \rangle$, $\langle \textit{sequential program} \rangle$, and $\langle \textit{sequential process} \rangle$ are put together to define the single syntactic class $\langle \textit{process} \rangle$, allowing for parallelism, inaccuracy specifications, and variable declarations inside subprocesses. Furthermore, a generalized bound construct

$$\langle \textit{general bound} \rangle ::= [\langle \textit{process} \rangle] \in \langle \textit{set of times} \rangle$$

and an assertion statement

$$\langle \textit{assertion} \rangle ::= \texttt{assert}\ \langle \textit{bool expression} \rangle$$

are added to the process constructions.

A general bound $[\pi] \in T$ confines the enclosed process π to spend a runtime in the bounding set T of times until termination, where — as with upper bound timing — time spent waiting for a communication partner does not count. In the remainder of this article, the notation $\texttt{wait}\ T$, where T is a set of times, will be used as a convenient abbreviation for the process $[\texttt{skip}] \in T$ that idles for a time in T.

An assertion $\texttt{assert}\ b$ does nothing (not even consume time) whenever the Boolean expression b is true in the current state. But whenever b evaluates to false, $\texttt{assert}\ b$ behaves completely unreliable, implying that *any* implementation is correct under these circumstances. Thus, prefixing a process by an assertion $\texttt{assert}\ b$ means that an implementation need only be well-behaved whenever b evaluates to true.

We obtain a common framework for reasoning about source and target programs by defining the semantics of the machine language by an interpreter expressed in TimedProc in section 6, since TimedProc is a superset of our source language TimedPL.

4 Implementation Correctness

When dealing with correctness of code generation we need a rigorous notion of whether one process implements another one or not. As we are dealing with embedded systems, there is a very natural notion directly at hand:

> *A process ψ* implements *or* refines *a process π, denoted $\psi \sqsupseteq \pi$, iff π can be safely replaced by ψ in any context.*

I.e., ψ *may* only engage in interactions with its environment that π may also engage in, and ψ *must* engage in any interaction π must engage in. This has to apply with respect to both the logical behaviour and timing, i.e. an implementing process will in particular respect all the deadlines that the implemented process meets.

Using this kind of reasoning, which can be formalized by associating each process with the set of trajectories over its state space it may engage in [vK93], algebraic laws of refinement between processes (or implementation correctness, respectively) can be established, equipping TimedProc with a calculus of process refinement. Table 2 gives examples of TimedProc's refinement rules.

Table 2: Some refinement laws of TimedProc

$x := e\,; x := f \doteq x := f[e/x]$	{Assignment merge}
$[\pi] \leq t \sqsupseteq [\pi] \leq t', \text{ if } t \leq t'$	{Tightening Bounds}
$[x := e] \in T \doteq [x := e] \in \{0\}\,; \texttt{wait}\ T$	{Assignment-Bound}
$\texttt{wait}\ T\,; [x := e] \in T' \doteq [x := e] \in T'\,; \texttt{wait}\ T$	{Assignment-Wait}
$\texttt{wait}\ T \sqsupseteq \texttt{wait}\ T', \text{ if } T \subseteq T'$	{Wait-Refinement}

where $\doteq$ is semantic equivalence, i.e. $\pi \doteq \psi$ iff $\pi \sqsupseteq \psi$ and $\psi \sqsupseteq \pi$, and $f[e/x]$ denotes expression f with every occurence of variable x being replaced by expression e.

5 Conceptual Framework of Code Generator Verification

In software engineering it is largely accepted that the formulation of specifications must preceede the construction of programs. Often even a derivation of programs out of specifications by formal or informal transformations is recommended instead of a-posteriori verification. Similarly the design of provably correct machine code to be generated by a compiler should be preceeded by the formulation of the appropriate correctness predicate. The construction of a verified code generator described in this paper is inspired by some ideas of [Hoa91]. One of them is to base correctness of code on the notion of source language refinement consistently extending the chain of refinement steps that led to the source program down to the level of the machine program. A definition in terms of refinement formulae is enabled by defining the machine language semantics by an interpreter $\mathcal{I}$ in source language-like notation.

After the correctness relation has been fixed, correct code can be described by means of theorems about this relation. Typically for compound constructs $op(\pi_1, \ldots, \pi_n)$ these theorems take the form of an implication that under certain syntactic conditions on the surrounding code establishes the correctness of code for the compound construct provided correct code for the components $\pi_1, \ldots, \pi_n$ is supplied. The collection of these theorems allows to define a compiling relation syntactically in a compositional way that is guaranteed to be a subrelation of the correctness relation, i.e. a (syntactic) specification of a correct code generator. In this way the collection of theorems specifies the code generator.

In simple situations (and in particular for entire programs) the correctness predicate can be defined directly as refinement between the source program π and the interpretation of an implementing machine program m:

$$\pi \sqsubseteq \mathcal{I}\, m \ .$$

But in more sophisticated situations further parameters are necessary. If for example the data spaces of the source and the target program are different then a retrieve-mapping Ψ that describes their relationship could be used as an additional parameter of the correctness predicate:[2]

$$\Psi \ ; \ \pi \ \sqsubseteq \ \mathcal{I}\, m \ ; \ \Psi \ .$$

6 Machine Language

The work reported in this article aims at the design of a provably correct code generator that translates TimedPL to transputer code [inm88b]. Of main interest here are the timing aspects, in particular the guarantee that time bounds requested in source programs are met by the generated machine code. The assumption of TimedPL that control structures do not consume time by themselves largely simplifies the reasoning about programs. But clearly, code running on a conventional processor needs time for the evaluation of the Boolean guards that steer the control flow and for the execution of the jump instructions that move the program counter to appropriate places.

The solution to this problem is based on the observation that only the preservation of the timing of external communications and of the communicated values is important for correctness. Internal computation can be moved arbitrarily as long as this does not affect communications. Therefore a compiler for TimedPL can shift the computation time overhead for the implementation of control structures to sequentially neighboured processes.

This section introduces a simple abstract machine language. It has been designed in order to allow a treatment of the timing aspects in isolation and to illustrate how the timing of machine instructions can be formally captured in a process algebraic setting. A number of other translation tasks for a compiler to transputer code and for code for conventional processors in general are not discussed in this paper as solutions are well-known. Examples are the assignment of storage locations to variables, the translation of mnemonic assembler

[2] We assume that Ψ can be written inside the language of processes into which the source language is embedded.

instructions to sequences of bytes, and the generation of code for the evaluation of expressions.

The model machine has the following components: a program counter $\mathtt{P}$, two accumulators $\mathtt{A}$ and $\mathtt{B}$ for integer resp. Boolean values, a storage that is addressed directly by variable names (avoiding the compiler task of assigning integer addressed storage locations to program variable names), and channels that are addressed directly by channel names (avoiding the need to assign links to channels when translating $\mathsf{TimedPL}$ to the machine language). Its instruction set is given by the following grammar, where i ranges over instructions.[3]

$$i ::= \mathtt{stopp} \mid \mathtt{eval}(e) \mid \mathtt{eval}(b) \mid \mathtt{stl}(x) \mid \mathtt{out}(c) \mid \mathtt{in}(c) \mid \mathtt{j}(l) \mid \mathtt{cj}(l) \ .$$

A *machine program* m is a sequence of instructions. We use the notation $\#m$ for the length of m and $m_1{}^\frown m_2$ for the concatenation of m_1 and m_2.

Informally, the logical behaviour of the instructions is as follows: $\mathtt{stopp}$ stops the machine, $\mathtt{eval}(e)$ evaluates the integer expression e leaving the result in register $\mathtt{A}$, $\mathtt{eval}(b)$ evaluates the boolean expression b leaving the result in register $\mathtt{B}$, $\mathtt{stl}(x)$ stores the contents of register $\mathtt{A}$ to the memory location x, $\mathtt{out}(c)$ communicates the contents of register $\mathtt{A}$ synchronously on channel c, $\mathtt{in}(c)$ reads an integer value synchronously from channel c and writes it to register $\mathtt{A}$, $\mathtt{j}(l)$ performs a relative jump by l, and $\mathtt{cj}(l)$ acts like $\mathtt{j}(l)$ if the register $\mathtt{B}$ contains the value false, otherwise it transfers control to the following instruction.

One of the ideas of the chosen approach to code generator verification is to define the machine language semantics via an interpreter written in a process language into which the source language is embedded. Typically such an interpreter consists of one loop essentially. The loop's body consists of a conditional that branches to appropriate actions for each of the machine instructions. The action describing the *untimed* meaning of the instruction $\mathtt{eval}(e)$, for example, is given by the process $\mathtt{A}, \mathtt{P} := e, \mathtt{P} + 1$.

An interpreter defining the *instruction timing* in addition to the logical behaviour can be obtained by using time bounds at appropriate places in the interpreter. The property of the process language that the composition operators do not take time themselves simplifies this. Therefore time is spent only by the actions that describe the logical behaviour of the single instructions. The idea is to define for each instruction i a set $\mathcal{T}(i)$ of possible execution times and to use $\mathcal{T}(i)$ as time bound for the corresponding action. The process defining the *timed* meaning of $\mathtt{eval}(e)$, for example, is $[\,\mathtt{A}, \mathtt{P} := e, \mathtt{P} + 1\,] \in \mathcal{T}(\mathtt{eval}(e))$.

Processor manuals state the number of machine cycles $n(i)$ that are necessary for evaluation of an instruction i. On a machine with clock rate r, execution time of i therefore is $\frac{n(i)}{r}$. But this calculation is oversimplified since the clock generator of the machine cannot be assumed to be accurate. We assume that the imprecision of the clock can be quantified by a *drift constant* $d_M \geq 1$ in the following way: If the machine clock advances by t then the time t' that has actually passed satisfies $\frac{t}{d_M} \leq t' \leq d_M * t$. There are two possibilities for incorporation of clock

[3] At the time being the model machine does not contain instructions for the implementation of alternatives. We are not yet able to handle them appropriately in a way that generalizes to transputer code.

drift into the semantics description by an interpreter: It can either be specified locally for each instruction by using $\mathcal{T}_{\text{drift}}(i) = \{t \mid \frac{1}{d_M} * \frac{n(i)}{r} \leq t \leq d_M * \frac{n(i)}{r}\}$ instead of $\mathcal{T}(i)$ or globally for the entire machine by applying the drift operator **drift** d_M at the outermost level. Local drift specification more directly captures the intuition that the execution time of single instructions is not accurately determined but that only certain intervals can be guaranteed. Global drift specification on the other hand can be more conveniently used in compiler proofs. A compiler must only check that the globally specified drift d_M of the machine is smaller than or equal to the drift allowed by the source program. Fortunately, one can show that it is immaterial whether drift is specified locally or globally, because the drift operator distributes over all sequential operators and only weakens the time bounds of a sequential process if time bounds are not nested.

Table 3 contains the timed semantics of the machine language. $\mathcal{M}\,m$ describes the possible timed behaviours arising from interpreting machine program m. The assertion **assert** $P = \#m + 1$ at the end of the definition of $\mathcal{I}$ ensures that every terminating execution actually ends at address $\#m + 1$. Otherwise the interpreter behaves arbitrary and the machine program can not be a refinement of a reasonable program. In this way the obligation is posed on the compiler constructor to use only code sequences of this kind.

Table 3: Machine Language Semantics

$$\mathcal{M}\,m \overset{\text{def}}{=} \texttt{drift}\ d_M : \mathcal{I}\,m$$

$$\begin{aligned}\mathcal{I}\,m \overset{\text{def}}{=}\ & \texttt{var P,A,B}: \\ & [\texttt{P} := 1] \in \{0\}\ ;\ \texttt{while}\ 1 \leq \texttt{P}\ \wedge\ \texttt{P} \leq \#m\ \texttt{do}\ \mathit{Step}\ ;\ \texttt{assert}\ P = \#m + 1\end{aligned}$$

$$\begin{aligned}\mathit{Step} \overset{\text{def}}{=}\ & \texttt{if}\ m[\texttt{P}] = \texttt{stopp then stop} \\ & \texttt{else if}\ m[\texttt{P}] = \texttt{eval}(e)\ \texttt{then}\ [\texttt{A},\texttt{P} := e, \texttt{P}+1] \in \mathcal{T}(\texttt{eval}(e)) \\ & \texttt{else if}\ m[\texttt{P}] = \texttt{eval}(b)\ \texttt{then}\ [\texttt{B},\texttt{P} := b, \texttt{P}+1] \in \mathcal{T}(\texttt{eval}(b)) \\ & \texttt{else if}\ m[\texttt{P}] = \texttt{stl}(x)\ \texttt{then}\ [x,\texttt{P} := \texttt{A}, \texttt{P}+1] \in \mathcal{T}(\texttt{stl}(x)) \\ & \texttt{else if}\ m[\texttt{P}] = \texttt{j}(l)\ \texttt{then}\ [\texttt{P} := \texttt{P}+1+l] \in \mathcal{T}(\texttt{j}(l)) \\ & \texttt{else if}\ m[\texttt{P}] = \texttt{cj}(l)\ \texttt{then}\ [\texttt{P} := \texttt{if B then P}+1\ \texttt{else P}+1+l] \in \mathcal{T}(\texttt{cj}(l)) \\ & \texttt{else if}\ m[\texttt{P}] = \texttt{out}(c)\ \texttt{then}\ [c\,!\,\texttt{A}\ ;\ \texttt{P} := \texttt{P}+1] \in \mathcal{T}(\texttt{out}(c)) \\ & \texttt{else if}\ m[\texttt{P}] = \texttt{in}(c)\ \texttt{then}\ [c\,?\,\texttt{A}\ ;\ \texttt{P} := \texttt{P}+1] \in \mathcal{T}(\texttt{in}(c)) \\ & \texttt{else chaos}\end{aligned}$$

The generation of correct code employs some properties of the interpreter I that can be proved by application of refinement laws. For example an empty code sequence does not change anything and needs no time for execution:

$$\mathcal{I} <> \doteq \texttt{wait}\ 0\ .$$

This indicates one way of implementing **skip**. A somewhat more elaborate property shows how to implement an assignment statement:

$$\mathcal{I} < \texttt{eval}(e), \texttt{stl}(x) > \ \doteq\ [x := e] \in \mathcal{T}(\texttt{eval}(e)) + \mathcal{T}(\texttt{stl}(x))\ . \qquad (1)$$

Evaluating an expression e first and storing the result to x afterwards, behaves like the assignment $x := e$. It terminates in a time in $\mathcal{T}(\texttt{eval}(e)) + \mathcal{T}(stl(x))$. Note that the additional assignment of the value of e to the register $\texttt{A}$ is not observable since $\texttt{A}$ is a local variable of $\mathcal{I}$.

7 Correctness Predicates

This section describes the correctness predicates for code to be generated for the different syntactic categories of TimedPL. We start with the correctness predicate for sequential processes.

According to the translation theorem about parallel components sketched in the next section only for entire parallel components it must be checked whether the drift of the machine clock is tolerable. Sequential processes can be implemented with the idealized assumption that the clock is accurate. Therefore the machine language interpreter $\mathcal{I}$ with idealized instruction timing can be used in the correctness predicate rather than the drifting one $\mathcal{M}$. Thus an obvious candidate for a correctness predicate for implementation of a sequential process sp by a machine program m is the predicate defined by the formula

$$sp \sqsubseteq \mathcal{I}\, m \ .$$

Although this is a nice predicate for passing implementation correctness from sequential processes to parallel components, it is not well-suited as a predicate for inheriting it from sub-processes since a number of phenomena must be handled.

(i) It must be decided whether time bounds are satisfied by the machine code. Since we are heading for a compositional code generator specification the correctness predicate must give information about the execution time of the code or – what turns out to be more convenient – about bounds that can be guaranteed for the source process.

(ii) The time needed for evaluation of Boolean guards and jumping to appropriate parts of code when evaluating conditionals or loops must be transferred to sub-processes or sequential predecessor or successors due to the assumption that evaluation of control structures does not take extra time. Therefore the correctness predicate must also give information about spare time of the code.

(iii) Execution of code cannot be arbitrarily moved in time if it contains communication instructions, since the communications are visible to the environment. There is a rather complex dependency of inner bounds and shift of spare time. Consider for example the processes

$$\pi_1 = [\texttt{skip} \ ; \ c\,!\,1 \ ; \ \texttt{skip}] \in [0,3] \ \text{ and}$$
$$\pi_2 = [\texttt{skip}] \in [0,1] \ ; \ [c\,!\,1] \in \{0\} \ ; \ [\texttt{skip}] \in [0,2] \ .$$

Both of them can be implemented by $\pi' = \texttt{wait}\ 1 \ ; \ [c\,!\,1] \in \{0\} \ ; \ \texttt{wait}\ 1$. Execution time of π' is 2 and spare time in both cases is 1. But in case of π_2 the spare time may only be used for initial (internal) actions of the sequential successor and must not be used for executing final (internal) actions of the sequential predecessor. On the other hand when implementing π_1 by π' the spare time can be transferred to either the predecessor or the successor or even split between them.

(iv) In contrast, internal computation can be arbitrarily moved in time. If e.g. $\pi_3 = [x := 7] \in [0,3]$ is implemented by $\pi'_3 = [x := 7] \in \{2\}$, then an arbitrary amount of spare time t is available to the sequential predecessor if $t-1$ time units are transferred to the sequential successor and vice versa.

(v) Sometimes a bound requested for the source process is more narrow than the bound immediately guaranteed for the target code, for example if $\pi_4 = [c\,!\;1] \in \{0\}$ is implemented by $\pi'_4 = [c\,!\;1] \in [0,1]$. Then the uncertainty about the termination time of the target program must be transferred to the sequential successor.

To handle these phenomena we use a triple of time sets as additional parameters of the correctness predicate. This triple describes one possible use of the code in a sequential environment. It consists of

- a lower bounded non-empty set $u \subseteq \mathbb{R}$, describing a starting time shift and a starting uncertainty accepted by the code,
- a lower bounded non-empty set $u' \subseteq \mathbb{R}$ describing a resulting termination time shift and termination time uncertainty that must be transferred to the sequential successor,
- a time bound $T \subseteq \mathbb{R}_{\geq 0} \cup \{\infty\}$ that can be guaranteed for the source process.

The predicate $\mathcal{S}$ defined below holds if a sequential process π is implemented by the machine program m accepting a start shift and uncertainty u that results in a termination shift and uncertainty u' such that the bound T can be guaranteed for π.

$$\mathcal{S}\,\pi\,m\,u\,u'\,T \quad \text{iff for all } \tau, \tau' \in \mathbb{R}_{\geq 0} \text{ such that } \tau + u \geq 0 \text{ and } \tau' + u' \geq 0\text{:}$$
$$\texttt{wait}\ \tau\ ;\ [\pi] \in T\ ;\ \texttt{wait}\ \tau' + u' \sqsubseteq \texttt{wait}\ \tau + u\ ;\ \mathcal{I}\,m\ ;\ \texttt{wait}\ \tau'\ ,$$

where $\tau + u$ is the set $\{\tau + t \mid t \in u\}$ and $\tau + u \geq 0$ is written instead of $t \geq 0$ for all $t \in \tau + u$, and similarly for $\tau' + u'$.

Note that for a fixed source process π and implementing machine code m different values for u, u', T are possible. In particular often a narrower bound T can be guaranteed by transferring a wider termination time uncertainty to the sequential successor via u'. Consider, for example, $\pi = \texttt{skip}\ ;\ c\,!\,x\ ;\ \texttt{skip}$ and m such that $\mathcal{I}\,m = \texttt{wait}\ 1\ ;\ [c\,!\,x] \in \{0\}\ ;\ \texttt{wait}\ [0,1]$. We can choose $u = u' = \{0\}$ and $T = [1,2]$. Alternatively we could choose $u = \{0\}$, $u' = [0,1]$, $T = \{1\}$. Many other values for the triple u, u', T are also acceptable.

Translation of the remaining syntactic categories programs, parallel components, and sequential programs employs the assumption that each of the parallel components of a program has its own processor for execution and that the parallel interaction of processors is correctly described by the parallel operator of the process language. A further assumption is that the maximal latency $\delta_M(c)$ of a channel c on a network of processors can be determined. Then the correctness predicates for the translation of these categories are given by straightforward refinement formulae. A formal statement is omitted due to lack of space.

8 Translation Theorems

This section presents a few of the theorems about the code correctness predicates that form the specification of a code generator. The proofs for these theorems are based on the laws of TimedProc. Due to lack of space we do not give the complete collection of theorems here. Furthermore we will only give one of the proofs. The theorems that allow to infer implementation correctness of programs and parallel components from implementation correctness of their constituent parts are quoted here in a textual form only. For some of the constructs building sequential processes formal statements of the correctness theorems are given. A more complete collection of theorems together with their proofs can be found in [MO93].

A program pr is correctly implemented if each of its constituent parallel components is correctly implemented and if $l \geq \delta_M(c)$ for each channel c that is declared in pr with latency l. This follows immediately from monotonicity of parallel composition with respect to refinement.

A parallel component pc is correctly implemented if its constituent sequential process sp is correctly implemented (more precisely if $\mathcal{S}\, sp\, m\, \{0\}\, \{0\}\, T$ for an arbitrarily chosen $T \subseteq Time_\infty$) and the maximum allowed drift specified in pc is greater than or equal to the drift d_M of the machine clock.

Theorem 1 (Translation of assignments). *An assignment statement $x := e$ can be implemented by evaluating the expression e first and then storing the result value to the location x with an appropriate timing condition:*

$$\textit{If } m = <\texttt{eval}(e), \texttt{stl}(x)> \textit{ and } u + \mathcal{T}(\texttt{eval}(e)) + \mathcal{T}(\texttt{stl}(x)) \subseteq u' + T$$
$$\textit{then } \mathcal{S}\,(x := e)\, m\, u\, u'\, T \quad .$$

Proof.

$$
\begin{aligned}
& \texttt{wait}\ \tau + u\ ;\ \mathcal{I}\, m\ ;\ \texttt{wait}\ \tau' \\
\sqsupseteq\ & \{\text{Formula (1)}\} \\
& \texttt{wait}\ \tau + u\ ;\ [x := e] \in \mathcal{T}(\texttt{eval}(e)) + \mathcal{T}(\texttt{stl}(x))\ ;\ \texttt{wait}\ \tau' \\
\sqsupseteq\ & \{\text{Assignment-Bound law, Assignment-Wait law, Wait-Additivity}\} \\
& \texttt{wait}\ \tau\ ;\ [x := e] \in \{0\}\ ;\ \texttt{wait}\ \tau' + u + \mathcal{T}(\texttt{eval}(e)) + \mathcal{T}(\texttt{stl}(x)) \\
\sqsupseteq\ & \{\text{Wait-Refinement law, } u + \mathcal{T}(\texttt{eval}(e)) + \mathcal{T}(\texttt{stl}(x)) \subseteq u' + T\} \\
& \texttt{wait}\ \tau\ ;\ [x := e] \in \{0\}\ ;\ \texttt{wait}\ \tau' + u' + T \\
\sqsupseteq\ & \{\text{Wait-Additivity, Assignment-Bound law}\} \\
& \texttt{wait}\ \tau\ ;\ [x := e] \in T\ ;\ \texttt{wait}\ \tau' + u'
\end{aligned}
$$

□

Theorem 2 (Translation of inputs). *An input statement $c\,?\,x$ can be implemented by first reading a value from channel c and then storing it to location x, with an appropriate timing condition:*

$$\textit{If } m = <\texttt{in}(c), \texttt{stl}(x)>,\ u = \{0\} \textit{ and } \mathcal{T}(\texttt{in}(c)) + \mathcal{T}(\texttt{stl}(x)) \subseteq u' + T$$
$$\textit{then } \mathcal{S}\,(c\,?\,x)\, m\, u\, u'\, T \quad .$$

The difference in the timing conditions of the above two theorems mirrors that internal actions can be shifted arbitrarily in contrast to externally visible communications. For the implementing code of an assignment statement any starting shift set u is acceptable as long as it is compensated by the termination shift set u'. Possible starting and termination shift sets for the code of an input statement however are more precisely determined.

Theorem 3 (Translation of sequential composition). *Concatenating code for two processes π_1 and π_2 yields code for their sequential composition $\pi_1 \,;\, \pi_2$. The sum of bounds of the components provide a guaranteeable bound. The termination uncertainty of the first component must be acceptable as a starting uncertainty for the second component:*

$$\textit{If } \mathcal{S}\,\pi_1\,m_1\,u_1\,u_1'\,T_1,\ \mathcal{S}\,\pi_2\,m_2\,u_2\,u_2'\,T_2,\ \textit{and } u_1' \subseteq u_2$$
$$\textit{then } \mathcal{S}\,(\pi_1\,;\,\pi_2)\,(m_1{}^\frown m_2)\,u_1\,u_2'\,(T_1+T_2) \quad .$$

Theorem 4 (Translation of upper-bounds). *An upper-bound t can be asserted for a source program π if a subset of $[0,t]$ is guaranteeable:*

$$\textit{If } \mathcal{S}\,\pi\,m\,u\,u'\,T \ \textit{and } T \subseteq [0,t] \ \textit{then } \mathcal{S}\,([\pi] \leq t)\,m\,u\,u'\,T \quad .$$

These theorems together with theorems for the remaining constructs induce syntactically defined subrelations of the correctness predicates. The remaining task of compiler construction is to implement these relations. We intend to build a prototype implementation in a functional language like Miranda or ML [Tur86, Wik87]. Thus we must construct functions corresponding to the induced relations. The problem is that the timing parameters u, u' and T can be neither parameters nor parts of the result as in both cases there is a large freedom of choice for them. But the choice is not arbitrary. Only some of the possible values can succesfully be used. Our idea is to use a finite characterization of the set of all possible triples (u, u', T) or of a useful subset.

9 Discussion

This paper has given an overview on current work done in the ProCoS II project concerning the construction of a provably correct compiler for a hard real-time language. The construction has been split into a number of tasks:

(i) A precise definition of the source language has been given. In particular its semantics has been formalized. Work towards this goal is documented in [FMO93, FvK93, vK93]. Furthermore to allow algebraic reasoning about programs, refinement laws have been established. Section 2 to 4 gave an informal account on this work.

(ii) Similarly, a precise definition of the target language is required. Up-to-now a model machine language has been considered (see section 6 and [MO93]). Clearly, to obtain a compiler that translates to machine code of an actual processor its machine language must be formalized. This has been done in the predecessor project ProCoS I for the transputer [inm88b], but without considering timing [Pro93]. We plan to extend this work towards timing.

(iii) The code to be generated by the compiler has been specified (see sections 7 and 8 and [MO93]).

(iv) This code generator specification will be transformed to a fully constructive version.

(v) The compiler will be implemented in a functional language. This comprises construction of a frontend and the implementation of the code generator.

(vi) For a dependable compiler, also a reliable execution mechanism for the implementation language of the compiler is necessary. [BBF92] shows how this can be achieved by application of bootstrapping. A more detailed account can be found in [Pro93].

References

[BBF92] Bettina Buth, Karl-Heinz Buth, Martin Fränzle, Burghard v. Karger, Yassine Lakhneche, Hans Langmaack, and Markus Müller-Olm. Provably correct compiler development and implementation. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, pages 141–155. Springer, 1992. LNCS 641.

[FMO93] Martin Fränzle and Markus Müller-Olm. *Drift and Granularity of Time in Real-Time System Implementation.* ProCoS II project document [Kiel MF 10/2], Christian-Albrechts-Universität Kiel, Germany, August 1993.

[FvK93] Martin Fränzle and Burghard von Karger. *Proposal for a Programming Language Core for ProCoS II.* ProCoS II project document [Kiel MF 11/3], Christian-Albrechts-Universität Kiel, Germany, August 1993.

[vK93] Burghard von Karger. *A simple wide-spectrum model for real time systems.* ProCoS II project document [OU BvK 9/6], Oxford University Programming Research Group, UK, August 1993.

[MO93] Markus Müller-Olm. *On Translation of TimedPL and Capture of Machine Instruction Timing.* ProCoS II project document [Kiel MMO 6/2], Christian-Albrechts-Universität Kiel, Germany, August 1993.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[Hoa91] 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 Computing, pages 33–48. Springer-Verlag, 1991.

[inm88a] INMOS ltd. *occam 2 Reference Manual.* Prentice Hall International, 1988.

[inm88b] INMOS ltd. *Transputer Instruction Set – A Compiler Writer's Guide.* Prentice Hall International, 1988.

[Pro93] Dines Bjørner, C.A.R. Hoare, Hans Langmaack (Eds.). *Provably correct systems.* ProCoS I final deliverable, 1993. Available from the Department of Computer Science, Technical University of Denmark, Building 3440, DK-2800 Lyngby.

[Tur86] David Turner. An overview of miranda. *SIGPLAN Notices*, 1986.

[Wik87] Åke Wikström. *Functional Programming Using Standard ML.* Series in Computer Science. Prentice-Hall, 1987.