C++ Hard-Real-Time Active Library: Syntax, Semantics, and Compilation of Tice Programs

Tadeus Prastowo
DISI, University of Trento
Trento, Italy
tadeus.prastowo@unitn.it

Luigi Palopoli
DISI, University of Trento
Trento, Italy
luigi.palopoli@unitn.it

Luca Abeni
Scuola Superiore S. Anna
Pisa, Italy
luca.abeni@santannapisa.it

ABSTRACT

General-purpose programming languages C and C++ address only the functional aspect of programs. In contrast, real-time programming languages address not only the functional aspect but also the nonfunctional real-time aspect to automatically ensure the consistency of both aspects in the resulting code. Despite many real-time programming languages proposed in the literature, embedded and real-time programs for economical reasons have continued to be written in C/C++ with their real-time aspect being dealt separately using modeling tools (e.g., MATLAB/Simulink) and the consistency of both aspects being checked manually. As embedded and real-time systems permeate through people's lives, it becomes increasingly imperative that both aspects be kept consistent automatically to improve the reliability of the systems. To that end, we propose a real-time programming language called Tice with three novel features: (1) Tice programs are written as modern standard C++ programs. (2) Tice programs can be compiled using any modern off-the-shelf standard C++ compiler. (3) Tice programs compose with other C/C++ programs as C++ libraries. These novel features make Tice significantly more economical than other real-time programming languages proposed in the literature.

KEYWORDS

real-time programming language, hard real-time (HRT), embedded domain-specific language (EDSL), C++ programming, C++ active library, C++ template meta-programming (TMP), static program analysis, software engineering

1 INTRODUCTION

Since its inception, IEEE Spectrum Top Programming Languages annual rankings have C++ in the top four along with C, Java, and Python [2][1]. ROS (Robot Operating System) codebase also reflects the rankings by being written in C++ (64%), Python (14%), C (7%), XML (7%), LISP (4%), Java (3%), and others (1%) [5]. C++, however, is not an RTPL (real-time programming language) despite being prevalent in embedded and real-time system development [11, 14, 15]. On the other hand, RTPLs with different MoCCs (models of real-time computation and communication) have been proposed, such as Esterel [1], TCEL [9], CRL [17], Giotto [8], and others surveyed in [16]. They, however, find little use in the industry where C and C++ are prevalent as reflected in the statement made by one of ROS founders, "any real-time requirements would be met in a special-purpose manner" [7].

While meeting real-time requirements in an ad hoc manner may benefit from a separation of concerns by using different tools for different concerns (e.g., MATLAB/Simulink to address the nonfunctional real-time aspect and C++ to address the functional aspect of an embedded program), doing so limits program development and maintenance speed because extra time is needed to manually verify that the functional and nonfunctional real-time aspects of the programs continue to be consistent. Moreover, any manual effort to ensure the consistency of both aspects is prone to error whose impact could be grave as embedded and real-time systems permeate through people's lives. A systematic solution to the consistency problem is the use of an RTPL, which automatically keeps both program aspects consistent in the resulting code, in particular an RTPL with a high-level real-time abstraction, which affords a separation of concerns between the real-time and functional aspects. With C and C++ being prevalent, however, the cost of manually ensuring the consistency of both program aspects is commonly far less than the cost of using an RTPL, in particular the cost of integration with other existing programs (e.g., operating systems, device drivers, software libraries, and legacy codebases) and supporting tools (e.g., profiler and debugger) as well as the cost of losing the separation of concerns in case of using an RTPL with a low-level real-time abstraction. We therefore propose an *economical* RTPL, called Tice, with a high-level real-time abstraction, which is synthesized from [8, 13]'s time-triggered LET (Logical Execution Time) and from [6, 9]'s specifications of not only a DAG (directed acyclic graph) of periodic asynchronous computation nodes (the asynchrony naturally allows the DAG to be executed concurrently in a multiprocessor system) but also ETE (end-to-end) delay and temporal correlation constraints on the DAG. Tice is an economical RTPL because it requires no change in the *existing* modern standard C++ programming toolchain, in particular the following three novel features: (1) a Tice program is written as a modern standard C++ program, requiring no change to the IDE (integrated development environment), (2) a Tice program can be compiled using *any* modern off-the-shelf standard C++ compiler, requiring no change to the compiler, and (3) a Tice program composes seamlessly with other C++ programs as a C++ library, requiring no change to the linker. To that end, this paper makes two contributions. First, this paper advances the state of the art of RTPLs by introducing an economical RTPL in the form of a C++ active library, which to our knowledge is absent in the literature. Second, this paper presents the details of the syntax and semantics of the proposed language as well as how its C++ active library, which henceforth is called Tice compiler, compiles Tice programs.

[1]We selected *all* language types to avoid language stereotyping bias, and by using *Edit Ranking*, we took the average of applying every weighting scheme, except *Custom*, to all years, except 2014 whose data for C and C++ were missing in the 2018 edition.

EWiLi'18, 4 October 2018, Torino, Italy

```
1 #include <iostream>
2 #include "al.hpp"
3 int main() { std::cout << al::sqrt<-2>(); }
```

Figure 1: Using C++ active library `al.h` by calling its function `al::sqrt` in a C++ program.

```
1 $ g++ -std=c++14 -O2 using-al-1.cpp
8 al.hpp:17:3: error: static assertion failed: sqrt
9 undefined for negative number
```

Figure 2: The metaprograms in `al.h` ensure the domain-specific safety of the program in Figure 1 at compile time.

```
1 $ g++ -S -std=c++14 -O2 -o- using-al-2.cpp
15        movl    $_ZSt4cout, %edi
16        movsd   .LC0(%rip), %xmm0
17        call    _ZNSo9_M_insertIdEERSoT_
62 .LC0:
63        .long   1719614412
64        .long   1073127582
```

Figure 3: Using `sqrt<2>` in Figure 1, the compiler produces an assembly that uses $\sqrt{2}$ value (line 16) computed at compile time by `al.h` (lines 63–64 form hexadecimal `3FF6A09E667F3BCC`, about 1.41421 by IEEE 754), not computing $\sqrt{2}$ at runtime.

We first give an overview of our proposal in Section 2 and follow that with the details of Tice MoCC in Section 3. Tice syntax and semantics are then presented in Section 4 followed by the description of the compilation process in Section 5. Lastly, we outline the related work in Section 6 and our conclusions in Section 7.

2 OVERVIEW

Tice affords its three novel features using C++ template metaprogramming. Specifically, Tice compiler is a collection of template metaprograms packaged in a C++ active library. An active library [3] presents to its users the experience of using a traditional software library in a program as illustrated in Figure 1. But under the hood when a C++ compiler compiles the program, the metaprograms in the active library, which are illustrated in Figure 4, perform activities once exclusive to compilers, such as ensuring program safety as shown in Figure 2 and performing program optimization as shown in Figure 3. Tice whose codebase is publicly accessible at https://savannah.nongnu.org/projects/tice is written in C++14 [12][2] and has dependency only on C++14 standard libraries [12, Ch. 17–30]. While C++17 is the latest standard, C++14 has more mature support than C++17 in popular C++ compilers.

On the other hand, Tice affords its high-level real-time abstraction by adopting time-triggered LET [13] and time-constrained event [9]. A Tice program organizes its functional aspect as a simple DAG, which is a directed graph whose underlying graph has neither multi-edges nor loops and whose arcs form no cycle. The graph nodes model C++ functions to be executed periodically, while the graph arcs model data buffers writable and readable in zero

[2]The draft of [12] is freely accessible at http://isocpp.org/files/papers/N3797.pdf.

```
1 #define sc(T) static constexpr T
2 namespace al {
3  template<bool, const double &S, const double &z = S>
4  struct NRM_core { sc(double) v = z; };
5  template<const double &S, const double &prev_z>
6  struct NRM_core<false, S, prev_z> {
7   sc(double) z {prev_z / 2.0 + S / 2.0 / prev_z};
8   sc(bool  ) below_threshold {z * z - S < 1E-15};
9   sc(double) v {NRM_core<below_threshold, S, z>::v}; };
10 template<int arg, bool arg_valid = arg >= 0>
11 struct NRM {
12  sc(double) x {arg};
13  sc(double) v {NRM_core<x == 0 || x == 1, x>::v}; };
14 template<int arg>
15 struct NRM<arg, false> {
16  sc(double) v {-1.0};
17  static_assert(arg >= 0,
18   "sqrt undefined for negative number"); };
19 template<int arg>
20 inline double sqrt() { return NRM<arg>::v; } };
```

Figure 4: Metaprograms in `al.h` implement `al::sqrt` using Newton-Raphson method (NRM).

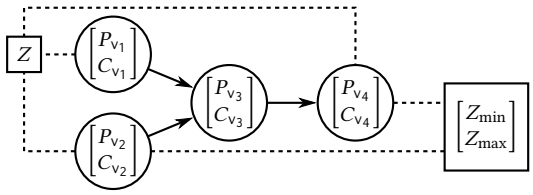

Figure 5: Example of a Tice graph.

time only by the producer and consumer nodes[3], respectively. ETE delay and correlation constraints can then be specified on the graph to ensure that the program is time safe with respect to externally observable events. Such a graph is henceforth called Tice graph, which is illustrated in Figure 5 that depicts nodes as circles where P_v and C_v are node v's period and WCET (worst-case execution time), respectively, arcs as arrows, an ETE delay constraint as a larger square that is connected with dashed lines to the constrained nodes where $Z_{\min}$ and $Z_{\max}$ are the permitted minimum and maximum ETE delays, respectively, and a correlation constraint as a smaller square that is also connected with dashed lines to the constrained nodes where Z is the correlation threshold. A Tice graph, however, does not show all details that are fully and formally captured by Tice formal model. Therefore, a Tice program uses the API (Application Programming Interface) of Tice active library to express Tice formal model. Consequently, Tice syntax uses C++ syntax to state the valid API usage, which serves as Tice language constructs, while Tice semantics is denoted by Tice formal model.

3 FORMAL MODEL

Tice formal model $\mathcal{M}$ is defined as $\mathcal{M} = (\texttt{MoCC}, \texttt{ETE}, \texttt{Corr})$ whose elements give the details of a Tice graph, its ETE delay constraints, and its correlation constraints, respectively. The details of the formal model along with the proofs of its soundness and decidability, including the outline of its mapping to real-time tasks, can be found

[3]A node being the subject of an action or the possessor of a property means that the action/property is performed/possessed by the function modeled by the node.

at https://goo.gl/XLLgsn. In this paper, $\mathbb{N}^+$, $\mathbb{N}$, $\mathbb{Q}$, $\mathbb{Q}^+$, and $\mathbb{Q}^{\geq 0}$ denote the sets of positive integers, nonnegative integers, all rationals, positive rationals, and nonnegative rationals, respectively.

3.1 Model of Computation & Communication

In giving the details of a Tice graph, the first element of $\mathcal{M}$ is defined as $\texttt{MoCC} = (\texttt{MoComp}, \texttt{MoComm})$ whose elements give the details of the graph's nodes and arcs, respectively, formalizing Tice MoCC.

3.1.1 Model of Computation. Tice's model of computation is formalized as $\texttt{MoComp} = (\mathbb{V}, f_{\texttt{P}} : \mathbb{V} \to \mathbb{Q}^+, f_{\texttt{C}} : \mathbb{V} \to \mathbb{Q}^+)$. The nonempty finite set $\mathbb{V}$ contains exactly all of the nodes in a Tice graph. A node v has its period P_v and WCET C_v specified by defining functions $f_{\texttt{P}}$ and $f_{\texttt{C}}$, respectively, in such a way so that $C_v \leq P_v$ $(C_v, P_v \in \mathbb{Q}^+)$.

Each node v in its k-th period ($k \in \mathbb{N}^+$) starts at some time $t_{\texttt{S}} \geq r_{v,k} = (k-1)P_v$ and finishes at some time $t_{\texttt{F}} < kP_v$ (i.e., v has its relative deadline equal to its period). If v had executed on a single processor core without interference since time $t_{\texttt{S}}$ until time $t_{\texttt{F}}$, then $C_v \geq t_{\texttt{F}} - t_{\texttt{S}}$. As a result, if v manipulates its internal states (e.g., by reading/writing some static variables), the manipulation completes by time $t_{\texttt{F}}$. On the other hand, if v has at least one outgoing arc, the output of the computation started at time $t_{\texttt{S}}$ will be readable by the computation(s) modeled by the adjacent node(s) not before or at time $t_{\texttt{F}}$ but at the beginning of the next period (at time $r_{v,k+1}$). This is the semantics of time-triggered LET [13].

Based on the presence of an incoming or an outgoing arc, a node is classified as either a sensor, an actuator, or an intermediary node for the purpose of specifying a time safety constraint on a Tice graph. A sensor node has either no incoming arc and no outgoing arc or no incoming arc but at least one outgoing arc. A sensor node models a computation that obtains events from the environment by manipulating its internal states, for example, by reading some hardware I/O ports. In contrast, an actuator node has at least one incoming arc but no outgoing arc. An actuator node models a computation that delivers events to the environment by manipulating its internal states, for example, by writing some hardware I/O ports. Since internal state manipulations complete by time $t_{\texttt{F}}$, if an event is to be delivered, it will be delivered to the environment before time $t_{\texttt{F}}$, not at the next release time of the actuator node. In other words, Tice relaxes the time-triggered LET semantics on the actuator nodes of a Tice graph. Lastly, an intermediary node has at least one incoming and at least one outgoing arcs.

3.1.2 Model of Communication. Tice's model of communication is formalized as $\texttt{MoComm} = (\mathbb{E}, f_{\texttt{T}} : \mathbb{E} \to \Theta_{\texttt{M}}, f_{\texttt{I}} : \mathbb{E} \to \bigcup_{\theta \in \Theta_{\texttt{M}}} A_\theta)$. The possibly-empty finite set $\mathbb{E}$ contains exactly all of the arcs in a Tice graph. Each of the arcs models a typed unidirectional communication channel, which henceforth is simply called channel, from a producer node to a consumer node. The channel's type is specified by defining function $f_{\texttt{T}}$. A channel consists of two data buffers: (1) a source buffer that is only written by the producer node, and (2) a sink buffer that is only read by the consumer node. Every sink buffer is initialized with data that are specified by defining function $f_{\texttt{I}}$ such that for any arc (v, v') in $\mathbb{E}$, $f_{\texttt{I}}((v, v')) \in A_{f_{\texttt{T}}((v,v'))}$ with A_θ being the set of all possible data whose type is θ.

Writing to a source buffer and reading from a sink buffer are asynchronous and, as a result, nonblocking. If a node v has at least one incoming arc, then whenever v is released, it reads all of the sink buffers synchronously (i.e., all of the reads take zero time). Since every sink buffer is initialized, v has a well-defined behavior. On the other hand, if a node v has at least one outgoing arc, then v has only one source buffer to write its output to because two or more outgoing arcs share a single source buffer[4]. The node v writes to the source buffer whenever v is released, and the write is complete by time $t_{\texttt{F}}$. Then, at the beginning of the next period (when the node is released again) the data in the source buffer is copied to the corresponding sink buffer(s) synchronously (i.e., the copying takes zero time). Hence, the data output by the node v in its k-th release is available for reading by the consumer node(s) only during the period between the k-th and $(k+1)$-th releases. Lastly, when copying to and reading from a sink buffer happen simultaneously, copying is ordered before reading.

3.2 Time Safety Constraints

In describing time safety constraints, (1) $\mathbb{V}_{\texttt{S}}$ denotes the nonempty finite set of sensor nodes, (2) $\mathbb{V}_{\texttt{A}}$ denotes the possibly-empty finite set of actuator nodes, and (3) $v \rightsquigarrow v'$ (v is connected to v') denotes the existence of at least one directed path from node v to node v'.

3.2.1 End-To-End Delay Constraints. The ETE delay constraints specified on a Tice graph are formalized by the second element of $\mathcal{M}$ defined as $\texttt{ETE} = \left(\mathbb{T}_{\texttt{E2E}}, f_{\texttt{E2E}} : \mathbb{T}_{\texttt{E2E}} \to \left(\mathbb{Q}^{\geq 0} \times \mathbb{Q}^+\right)\right)$ with the first element giving the constrained nodes and the second element giving the corresponding lower and upper bounds on the permitted delays. Each of the constraints can only be specified on a connected sensor-actuator node pair, and therefore, $\mathbb{T}_{\texttt{E2E}} \subseteq \left\{(v_{\texttt{S}}, v_{\texttt{A}}) \in (\mathbb{V}_{\texttt{S}} \times \mathbb{V}_{\texttt{A}}) \,\middle|\, v_{\texttt{S}} \rightsquigarrow v_{\texttt{A}}\right\}$. A constraint $(v_{\texttt{S}}, v_{\texttt{A}}) \in \mathbb{T}_{\texttt{E2E}}$ is respected if any event flowing from the sensor node $v_{\texttt{S}}$ to the actuator node $v_{\texttt{A}}$ has an ETE delay D such that $Z_{\min} \leq D \leq Z_{\max}$ with $Z_{\min}$ and $Z_{\max}$ being the permitted minimum and maximum ETE delays, that is, $f_{\texttt{E2E}}((v_{\texttt{S}}, v_{\texttt{A}})) = (Z_{\min}, Z_{\max})$ with $Z_{\min} < Z_{\max}$.

3.2.2 Correlation Constraints. The third element of $\mathcal{M}$ formalizes the correlation constraints specified on a Tice graph and is defined as $\texttt{Corr} = \left(\mathbb{T}_{\texttt{C}}, f_{\texttt{T}} : \mathbb{T}_{\texttt{C}} \to \mathbb{Q}^{\geq 0}\right)$ with the first element giving the constrained nodes and the second element giving the corresponding correlation thresholds. Each of the constraints can only be specified on a nonempty set of sensor nodes that is paired with one actuator node where each of the sensor nodes is connected to the actuator node. Hence, $\mathbb{T}_{\texttt{C}} \subseteq \bigcup_{v_{\texttt{A}} \in \mathbb{V}_{\texttt{A}}} \left\{(\mathbb{V}'_{\texttt{S}}, v_{\texttt{A}}) \,\middle|\, \mathbb{V}'_{\texttt{S}} \in \left(\wp\left(\left\{v_{\texttt{S}} \in \mathbb{V}_{\texttt{S}} \,\middle|\, v_{\texttt{S}} \rightsquigarrow v_{\texttt{A}}\right\}\right) \setminus \{\emptyset\}\right)\right\}$. Each constraint $(\mathbb{V}'_{\texttt{S}}, v_{\texttt{A}}) \in \mathbb{T}_{\texttt{C}}$ is respected if for any consumer node $v_{\texttt{C}}$ with at least two incoming arcs and for any event flowing from the sensor nodes in $\mathbb{V}'_{\texttt{S}}$ that $v_{\texttt{C}}$ reads synchronously at time t, the times when the events were read by the sensor nodes have absolute differences that are not greater than Z with Z being the correlation threshold, that is, $f_{\texttt{T}}((\mathbb{V}'_{\texttt{S}}, v_{\texttt{A}})) = Z$.

4 SYNTAX AND SEMANTICS

As already mentioned, Tice is not implemented as a standalone language, but as an active library, which is also known as an embedded domain-specific language (EDSL), in the C++ language. This

[4] An arc being the subject of an action or the possessor of a property means that the action/property is performed/possessed by the channel modeled by the arc.

means that software developers can rely on the usual C++ features in writing a program, complementing them with some Tice constructs to describe the structure of the program (as a Tice graph) and the hardware properties. The C++ compiler will take care of transforming Tice annotations in a C++ program at compile time. As a result, Tice is based on the C++ syntax (with particular references to templates). In the rest of this section, unless stated otherwise, every line-number reference refers to Figure 6.

Figure 6 presents a description of the Tice syntax, showing the canonical C++ syntax in bold face enclosed within a pair of single quotes. Canonical C++ syntax means that the parts in bold face can be written differently as long as the result is valid C++ syntax and has the same semantics as the replaced parts. For instance, instead of using a long comma-separated typedef list to follow the syntax rule in line 1, multiple typedefs can be employed to separate the hardware-dependent part in one file as in Figure 7(b) from the hardware-independent part in another file as in Figure 7(d).

The Tice syntax in Figure 6 uses the following nonterminal (symbol) definitions. Every nonterminal with suffix "_ident" represents a (valid) sequence of C++ terminals that expresses a C++ identifier. The nonterminals "nonnegative_int" and "positive_int" represent sequences of C++ terminals that express values in $\mathbb{N}$ and $\mathbb{N}^+$, respectively. In line 10, "fn_ptr" represents a sequence of C++ terminals that expresses a function pointer. In lines 26 and 27, "type" represents a sequence of C++ terminals expressing a C++ *object type*, which is any type other than `void`, a reference type, and a function type [12, §3.9¶8]. In line 26, "init_val" represents a sequence of C++ terminals that expresses a value whose type is compatible with the type expressed by the nonterminal "type" in the same line. Lastly, in line 27, "init_val_ptr" represents a sequence of C++ terminals that expresses a pointer to an object whose type is compatible with the type expressed by the nonterminal "type" in the same line.

While every nonterminal with suffix "_ident" can formally be replaced with a single nonterminal, such as "identifier", we refrain from doing so for the following reading aid. In line 6, the C++ identifier represented by the nonterminal "HW_desc_ident" can be referred to in line 14 when the line's "HW_desc_ident" represents the same identifier. In line 10, "comp_ident" can be referred to in line 20 when the line's "comp_ident" represents the same identifier. In line 20, "node_ident" can be referred to in line 15/24/25/36/37 when the respective line's "node_ident" represents the same identifier. Lastly, in line 18, "Tice_program_ident" can be referred to in a place where it is valid to instantiate a class with a nondefault constructor, which usually is in function `main` (an alternative is in the declaration of a global variable). With this reading aid in place, it should be clear that, although it is valid by the EBNF grammar, using the identifier represented by the nonterminal "comp_ident" in line 10 as the identifier represented by the nonterminal "HW_desc_ident" in line 14 will result in Tice compiler raising a compilation error.

The syntax rule in line 1 shows that a Tice program consists of two parts: hardware-dependent part ("HW_dependent_part") and hardware-independent part ("HW_independent_part"). The rules to specify the hardware-dependent part are found in lines 4–11, while the rules to specify the hardware-independent part are found in lines 13–37. The rules to specify both parts make use of the auxiliary rules in lines 39–42. The auxiliary rule "positive_rational" specifies a positive rational p/q by specifying a pair of positive integers, the first and second of which being p and q, respectively. Similarly, "nonnegative_rational" specifies a value in $\mathbb{Q}^{\geq 0}$ by specifying a pair of nonnegative and positive integers. Each auxiliary rule assumes one as the pair's second element if it is omitted.

Rule "HW_dependent_part" in line 4 shows that the hardware-dependent part of a Tice program consists of a target hardware description ("HW_desc") and one or more computations ("computation") that constitute the program functional aspect. In line 6, rule "HW_desc" shows that currently a target hardware description specifies only the set of available processor core IDs ("core_ids") (e.g., Figure 7(b, c) line 5). When the core ID is omitted, a Tice program will be compiled to a single non-real-time task that immediately terminates normally. This is useful to validate the design of the program real-time aspect by letting Tice compiler perform program safety checks only without mapping the Tice graph to real-time tasks. Lastly, rule "computation" in line 9 shows that a computation must specify its WCET on the target hardware ("WCET"), which constructs function f_{C} (cf. Section 3.1.1) piecewise, and a pointer to the computation itself ("fn_ptr") (e.g., Figure 7(b, c) lines 6–9).

Rule "HW_independent_part" in line 13 shows that the hardware-independent part of a Tice program consists of one or more nodes ("node") that form the set $\mathbb{V}$ (cf. Section 3.1.1) and a Tice graph that is constructed by referring to the nodes ("node_ident"). Zero or more arcs can then be specified on the Tice graph ("feeder"), and if at least one arc is specified, it is possible to specify zero or more ETE delay constraints ("ETE_delay") (cf. Section 3.2.1) followed by zero or more correlation constraints ("correlation") (cf. Section 3.2.2). Rule "node" in line 19 shows that a node in $\mathbb{V}$ models a computation ("comp_ident") that will be executed periodically at the specified rate ("period"), which constructs function f_{P} (cf. Section 3.1.1) piecewise, (e.g., Figure 7(d) lines 9–10). Rule "feeder" in line 22 shows that the set $\mathbb{E}$ (cf. Section 3.1.2) is constructed in terms of its disjoint subsets where each subset (e.g., Figure 7(d) lines 12–13) specifies all of the incoming arcs of a consumer node ("consumer") (e.g., `v3`). Each of the incoming arcs is then specified in terms of the producer node ("producer") (e.g., `v1` and `v2`) and the channel modeled by the arc ("channel"). Rule "channel" in line 26 gives two different ways to specify the initial value of a channel. When the value can be specified as a nontype template parameter (e.g., values of integral types like `int`), the first alternative (`Chan_inlit`) can be used to specify the value literally ("init_val") (e.g., Figure 7(d) line 12). Else, the value has to be specified externally (e.g., as a global variable as in Figure 7(d) line 7) and the second alternative (`Chan`) is used to specify the channel's initial value by means of a pointer ("init_val_ptr") (e.g., Figure 7(d) line 14). Either way, the specification of a channel's initial value constructs function f_{I} (cf. Section 3.1.2) piecewise. Both alternatives also specify the channel's type ("type") in the same way, which constructs function f_{t} piecewise. Rule "ETE_delay" in line 28 imposes an ETE delay constraint on a sensor ("sensor") and an actuator ("actuator") nodes, the pair of which forms the set $\mathbb{T}_{\mathrm{E2E}}$, with some permitted minimum and maximum delays ("min_delay" and "max_delay"), the pair of which constructs function f_{E2E} piecewise, (e.g., Figure 7(d) line 15). Lastly, rule "correlation" in line 33 imposes a correlation constraint on an actuator ("actuator") and a nonempty set of sensor ("sensor") nodes, the pair of which forms the set $\mathbb{T}_{\mathrm{C}}$, with a particular threshold ("threshold"), which constructs function f_{T} piecewise, (e.g., Figure 7(d) line 16).

```
Tice_program = 'typedef ',
  HW_dependent_part, ',', HW_independent_part, ';';

HW_dependent_part = HW_desc, ',',
  computation, {',', computation};
HW_desc = 'HW<', core_ids, '>', HW_desc_ident;
core_ids = 'Core_ids<',
  [ nonnegative_int, {',', nonnegative_int} ], '>';
computation = 'Comp(',
  fn_ptr, ',', WCET, ')', comp_ident;
WCET = positive_rational;

HW_independent_part = node, {',', node}, ',',
  'Program<', HW_desc_ident, ',',
  node_ident, {',', node_ident},
  [ ',', feeder, {',', feeder},
    {',', ETE_delay}, {',', correlation} ],
  '>', Tice_program_ident;
node = 'Node<',
  comp_ident, ',', period, '>', node_ident;
period = positive_rational;
feeder = 'Feeder<', producer, ',', channel,
  {',', producer, ',', channel}, ',', consumer, '>';
producer = node_ident;
consumer = node_ident;
channel = 'Chan_inlit<', type, ',', init_val, '>'
        | 'Chan<', type, ',', init_val_ptr, '>';
ETE_delay = 'ETE_delay<',
  sensor, ',', actuator, ',',
  min_delay, ',', max_delay, '>';
min_delay = nonnegative_rational;
max_delay = positive_rational;
correlation = 'Correlation<', actuator, ',',
  threshold, ',', sensor, {',', sensor}, '>';
threshold = nonnegative_rational;
sensor = node_ident;
actuator = node_ident;

positive_rational = 'Ratio<',
  positive_int, [',', positive_int], '>';
nonnegative_rational = 'Ratio<',
  nonnegative_int, [',', positive_int], '>';
```

Figure 6: Tice syntax expressed in the ISO/IEC standard EBNF (Extended Backus-Naur Form) [10].

5 COMPILATION

A Tice program is compiled in two stages as shown in Figure 8. The first stage is Tice compiler front-end that performs program safety checks, while the second stage is Tice compiler back-end that maps a Tice program to a set of real-time tasks and their schedule on available processor cores (the schedule states, among other things, the task partitioning policy). The mapping performed by the back-end only needs to ensure that the result behaves in accordance with the Tice formal model expressed in the program, and hence, the mapping is not necessarily injective (e.g., all functions modeled by the graph nodes can be mapped to one real-time task scheduled using cyclic executive on one core). To supply further information needed by the back-end (e.g., core IDs), a Tice program uses `HW`, the Tice hardware description class template. The two-stage compilation makes Tice not only extensible but also portable because without changing the program real-time aspect, which is processed by the front-end, the back-end can not only incorporate new results from real-time scheduling research but also retarget the program for different hardware described by different `HW` instantiation.

A Tice program is compiled when the C++ compiler encounters the instantiation of the identifier that is represented by "Tice_program_ident" in Figure 6 line 18 (e.g., `P(argc, argv)` in Figure 7(d) line 21), which corresponds to the "Yes" branch of the sole decision process in Figure 8. The metaprograms of Tice compiler front-end then perform program safety checks according to the constraints presented in Section 3. For example, `g++ -std=c++14`

```
#include <array>
typedef std::array<double, 5> X;
int fn1(); double fn2(); X fn3(int, double); void fn4(X);
```

(a) File `subprograms.hpp`.

```
#include "subprograms.hpp"
#include "tice/v1.hpp"
using namespace tice::v1;
typedef
HW<Core_ids<4>> hw_desc,
Comp(&fn1,Ratio<1,10>) cp1,
Comp(&fn2,Ratio<3,10>) cp2,
Comp(&fn3,Ratio<2>)    cp3,
Comp(&fn4,Ratio<2,10>) cp4;
```

(b) File `hw-1.hpp`.

```
#include "subprograms.hpp"
#include "tice/v1.hpp"
using namespace tice::v1;
typedef
HW<Core_ids<2, 4>> hw_desc,
Comp(&fn1,Ratio<3,10>) cp1,
Comp(&fn2,Ratio<9,10>) cp2,
Comp(&fn3,Ratio<6>)    cp3,
Comp(&fn4,Ratio<6,10>) cp4;
```

(c) File `hw-2.hpp`.

```
#if   defined USE_HW_1
#include "hw-1.hpp"
#elif defined USE_HW_2
#include "hw-2.hpp"
#endif
#include <cstdlib>
namespace { X i3_4; double i2_3(0); };
using namespace tice::v1; typedef
Node<cp1, Ratio<6>> v1,     Node<cp2, Ratio<5>> v2,
Node<cp3, Ratio<2>> v3,     Node<cp4, Ratio<3>> v4,
Program<hw_desc, v1, v2, v3, v4,
        Feeder<v1, Chan_inlit<int, -1>,
               v2, Chan<double, &i2_3>, v3>,
        Feeder<v3, Chan<X, &i3_4>, v4>,
        ETE_delay<v2, v4, Ratio<0>, Ratio<12>>,
        Correlation<v4, Ratio<5>, v1, v2>      > P;
int main(int argc, char *argv[]) {
  for (int i = 1; i <= i3_4.size(); ++i)
    i3_4[i-1] = (argv[i] ? std::strtod(argv[i], nullptr)
                         : 0);
  P(argc, argv); }
```

(d) File `main.cpp`.

Figure 7: Ordinary C++ functions implementing the program functional aspect (a) are annotated using Tice to execute on two different multiprocessor systems (b) and (c) whose real-time aspect is implemented by a Tice program (d) whose lines 8–16 express the Tice graph in Figure 5.

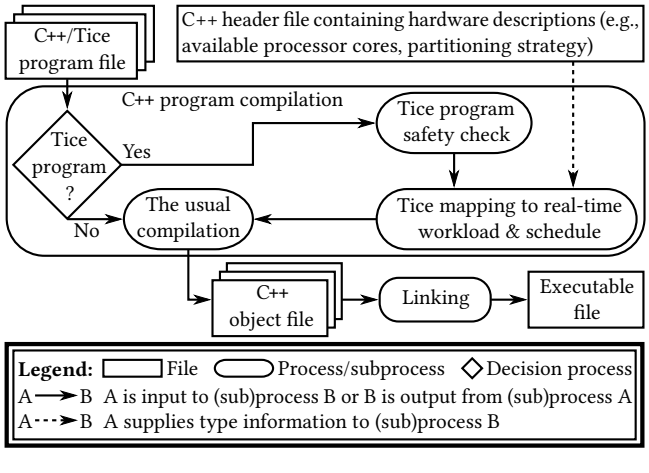

Figure 8: The compilation of a C++/Tice program.

`-Wfatal-errors` prints the following error message as the first meaningful line when a node's period is less than its WCET, which may happen in the evolution of an embedded program as it is ported to another hardware that increases the WCET or as the revision of its HRT specifications decreases the period: `In instantiation of struct tice::v1::error::node::period_is_greater_than_or_equal_to_wcet<false>`, while the third last meaningful line gives the error location at line 28 column 45 of the compiled file: `v1/test-v1_internals_node-fail_1.cpp:28:45: required from here`. To compare, `clang++ -std=c++14 -Wfatal-errors` prints similar error messages with the most important ones being found near the earliest and latest lines. That is, using `Wfatal-errors` switch coupled with our design of raising an error using individual metaprograms with meaningful names within nested namespaces results in informative error messages whose reasons and locations are easily located near the earliest and latest lines printed by the C++ compiler. The back-end metaprograms would then run to map the program to real-time tasks, raising any error in the same manner.

6 RELATED WORK

Tuchscherer, et al. [18] propose for the control functions (functional aspect) of an embedded automated driving system to be expressed directly in C++, instead of separately in MATLAB/Simulink, to have an integral model representation (i.e., an automatic consistency) with the dynamic object management (nonfunctional aspect) of the system. Tice similarly expresses the functional and the real-time (nonfunctional) aspects of a program directly in C++.

Deters, et al. [4] propose a C++ HRT active library that organizes a program's functional aspect in terms of a Liu-Layland taskset whose tasks are C++ classes to be scheduled on a single processor core using rate monotonic. The active library ensures that the program is time safe by performing a schedulability test. Tice raises the real-time abstraction level of [4] by effectively hiding the active library of [4] within Tice compiler back-end as one possible mapping target.

Tice formal model $\mathcal{M}$ has its first element `MoCC` simplify the MoCC of Giotto [8], mainly by allowing only a single mode and by not strictly keeping the time-triggered LET on the actuator tasks. On the other hand, the remaining elements of $\mathcal{M}$ (`ETE` and `Corr`), which formalize the ETE delay and correlation constraints specified on a Tice graph, are inspired by [6].

Lastly, unlike other RTPLs (e.g., [1, 8, 9]) that require the use of a new compiler if not also a new programming language, Tice being embedded in C++ imposes no such requirements and no extra cost beyond the usual cost of using a new C++ library.

7 CONCLUSION AND FUTURE WORK

We proposed Tice, an *economical* RTPL in the form of a C++ active library, whose high-level real-time abstraction is synthesized from [8, 9]. Tice is economical because Figure 7 shows that Tice syntax is a straightforward C++ that many embedded and real-time system developers are accustomed to. Tice syntax also directly expresses a Tice graph, which represents Tice's MoCC and its associated semantics (cf. Section 3), allowing for easier maintenance and evolution. Furthermore, Tice programs can be edited, compiled, linked, debugged, and profiled using *existing* modern standard C++ programming toolchains, which are prevalent in the development of embedded and real-time systems. Aside from that, as an RTPL, Tice key advantage over the general-purpose language C++ is its ability to keep consistent the program functional and real-time (nonfunctional) aspects automatically. The key advantage is demonstrated in Section 5 where a mistake in the program evolution is automatically caught by Tice metaprograms that can be engineered to raise informative error messages, giving not only the locations but also the domain-specific reasons of the errors.

Tice development sufficiently progressed to answer questions related to programming experience as given in the preceding paragraph. Our short-term goal is to complete Tice compiler back-end to answer questions related to the real-time quality of the generated code. Our long-term goal would be on answering questions on interoperability with external tools, such as a WCET analyzer to automatically supply the computation WCETs, and questions on Tice industrial impacts, especially to see whether "any real-time requirements would [still] be met in a special-purpose manner" [7].

REFERENCES

[1] G. Berry, S. Moisan, and J.-P. Rigault. 1983. Esterel: Towards a synchronous and semantically sound high-level language for real-time applications. In *Proc. RTSS*. IEEE, Silver Spring, MD, USA, 30–37.

[2] S. Cass and P. Bulusu. 2018. The Top Programming Languages 2018. http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018 Accessed: 2018-09-21.

[3] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. 2000. Generative Programming and Active Libraries. In *Generic Programming*, M. Jazayeri, R. G. K. Loos, and D. R. Musser (Eds.). Springer, Berlin, 25–39. https://doi.org/10.1007/3-540-39953-4_3

[4] M. Deters, C. Gill, and R. Cytron. 2003. Rate-Monotonic Analysis in the C++ Type System. In *Proc. RTAS 2003 Workshop on Model-Driven Embedded Systems*. IEEE, Washington, DC. http://www.cse.wustl.edu/~cdgill/RTAS03/mdes_program.html

[5] T. Foote. 2017. Celebrating 9 Years of ROS. https://spectrum.ieee.org/automaton/robotics/robotics-software/celebrating-9-years-of-ros Accessed: 2018-09-21.

[6] R. Gerber, S. Hong, and M. Saksena. 1994. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In *Proc. RTSS*. IEEE, Washington, DC, USA, 192–203. https://doi.org/10.1109/REAL.1994.342716

[7] B. Gerkey. 2014. Why ROS 2.0? http://design.ros2.org/articles/why_ros2.html Accessed: 2018-09-21.

[8] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. 2001. Giotto: A Time-Triggered Language for Embedded Programming. In *EMSOFT*, T. Henzinger and C. Kirsch (Eds.). Springer, Berlin, 166–184. https://doi.org/10.1007/3-540-45449-7_12

[9] S. Hong and R. Gerber. 1993. Scheduling with Compiler Transformations: The TCEL Approach. In *Proc. RTOSS*. IEEE, Washington, DC, USA, 80–84. http://dl.acm.org/citation.cfm?id=192806.185857

[10] ISO and IEC. 1996. *ISO/IEC 14977:1996 Extended BNF* (1st ed.). ISO, Geneva, Switzerland. https://www.iso.org/standard/26153.html

[11] ISO and IEC. 2006. *ISO/IEC TR 18015:2006 Technical Report on C++ Performance* (1st ed.). ISO, Geneva, Switzerland. https://www.iso.org/standard/43351.html

[12] ISO and IEC. 2014. *ISO/IEC 14882:2014 C++* (4th ed.). ISO, Geneva, Switzerland. https://www.iso.org/standard/64029.html

[13] C. M. Kirsch and A. Sokolova. 2012. The Logical Execution Time Paradigm. In *Advances in Real-Time Systems*, S. Chakraborty and J. Eberspächer (Eds.). Springer, Berlin, 103–120. https://doi.org/10.1007/978-3-642-24349-3_5

[14] C. Kormanyos. 2013. *Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming*. Springer, Berlin. https://doi.org/10.1007/978-3-642-34688-0

[15] Lockheed Martin. 2005. *Joint Strike Fighter Air Vehicle C++ Coding Standards, Document Number 2RDU00001 Rev C*. Joint Strike Fighter Program Office, Arlington, VA, USA. http://web.archive.org/web/201507/jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards_Rev_C.doc

[16] A. D. Stoyenko. 1992. The Evolution and State-of-the-art of Real-time Languages. *J. Syst. Softw.* 18, 1 (April 1992), 61–84. https://doi.org/10.1016/0164-1212(92)90046-M

[17] A. D. Stoyenko, T. J. Marlowe, and M. F. Younis. 1995. A Language for Complex Real-Time Systems. *Comput. J.* 38, 4 (1995), 319–338. https://doi.org/10.1093/comjnl/38.4.319

[18] D. Tuchscherer, A. Weibert, and F. Tränkle. 2016. Modern C++ As a Modeling Language for Automated Driving and Human-robot Collaboration. In *Proc. MODELS*. ACM, New York, NY, USA, 136–142. https://doi.org/10.1145/2976767.2976772