Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

Tice: a Real-Time Language Compilable Using C++ Compilers

Tadeus Prastowo[1] | Luigi Palopoli*[1] | Luca Abeni[2]

[1]DISI, University of Trento, Trento, Italy
[2]Scuola Superiore S. Anna, Pisa, Italy

Correspondence
*Luigi Palopoli (Povo-2, Room 130), Via Sommarive, 9 Trento, 38123 Italy. Email: luigi.palopoli@unitn.it

Abstract

Model-based development (MBD) holds the promise to capture potential timing problems in embedded software during the early phases of the development, securing the production of bug-free embedded software. For most MBD approaches, the source code is just an intermediate artifact that can be generated automatically from the models. This assumption clashes with an undeniable fact: a large share of the commercial embedded software exploits existing libraries or is developed using C/C++ natively.

A way to reconcile the ambitions of MBD with the use of a programming language is by offering new language constructs and an innovative compilation tool-chain that prevents model error and timing problems "by construction." However, the persistent popularity of C/C++ among embedded programmers and the limited availability of tools have severely limited the uptake of alternative programming languages for embedded software.

Therefore, we propose an original route. Our language proposal, named Tice, has been shaped as a C++ active library. Tice retains full compatibility with existing C++ code, which can be integrated easily into new Tice-based projects. The enforcement of Tice syntax and semantics can be made by a standard C++ compiler, forgoing the need for new tools. In this paper, we describe Tice's syntax, semantics, and model of computation and communication. We demonstrate Tice's practical applicability on an industrial scale use-case and give ample evidence for Tice's efficient compilation using off-the-shelf C++ compilers. Lastly, we show Tice's code generation process.

KEYWORDS:
real-time programming language, logical execution time (LET), template meta-programming (TMP), C++ active library, C++ EDSL (embedded domain-specific language), embedded software engineering

1 | INTRODUCTION

Embedded software is the main stay of modern information and communication technology industry: it lies at the heart of a whole lot of innovative applications, such as automated driving, robotics, and Internet of Things. The very peculiar nature of embedded software adds much to the complexity of its development. A very important paradigm of these complexities is the enforcement of a correct real-time behavior in the face of strong constraints on the cost of the hardware. For many embedded systems, uncontrolled delays or timing errors can result in a malfunctioning system that can cause substantial material losses. Such problems can be very hard to track down during the programming phase because they can be the result of unfortunate input and workload patterns, which are very difficult to anticipate and replicate. When a timing violation materializes in a late stage of the development or, even worse, after the system delivery, it can be extremely expensive to repair.

1.1 | Landscape

Model-based development (MBD)[1] is a methodology first proposed a couple of decades ago and has gained traction in some crucial applications in the embedded systems landscape, such as the automotive industry. In an MBD cycle, a system is modeled as a set of communicating blocks, each one associated with a mathematical transformation. Blocks are concurrent entities whose communication has formally specified semantics. The path to the implementation of the model consists of a number of refinement steps, in which the model becomes increasingly detailed (i.e., less abstract) until it finally takes the form of executable code. The key requirement of MBD is that a number of relevant properties are preserved across each refinement step so that the correctness of the system is eventually guaranteed by construction. In this framework, the source code expressed in high-level programming languages is no different from any intermediate format and is, by and large, automatically generated from the models.[2] This philosophy needs to come to terms with an undeniable reality: most embedded software developers still prefer direct coding in a programming language for developing embedded software or at least need the freedom to reuse libraries and code from previous projects. More often than not, the source code is not for them an intermediate format but a native way for expressing the system design. In this setting, the only way to follow an MBD cycle faithfully is for the models to percolate down into the programming language to become language constructs and for the compilation tool-chain to become capable of verifying and enforcing the properties of the models.

While real-time language research has proposed many real-time languages over several decades, which we will report about in Section 7, none of them has been widely adopted by embedded developers. In contrast, C++ has maintained a dominant position, being the standard choice to program embedded systems,[3,4,5,6] including high-profile projects such as Mars rovers,[7] F-35 jet fighters,[8] and the Robot Operating System.[9] While Python[3,9] and Java[10] have gradually conquered important niches (especially in data processing and human-machine interfaces), they are not considered as a realistic choice for system programming. Therefore, even the embedded software that is nominally developed in Python/Java ends up relying on C/C++ components and libraries for low-level control of the devices, not to mention that C/C++ also serves as the basis for implementing the Python interpreters and the Java virtual machines themselves. Lastly, Ada[10] is still used in a relatively small niche of safety-critical systems requiring certification, but we could easily argue that the presence of Ada programs in a lot of embedded systems produced every year has become marginal as demonstrated by the adoption of C++ in the F-35 jet fighters project.[8]

Since C++ is not a programming language originally thought for real-time programming, embedded software engineering still deals with the real-time aspect of embedded software on an ad hoc basis, quoting one of the founders of the Robot Operating System, "any real-time requirements would be met in a special-purpose manner."[11] In other words, the respect of the real-time constraints could emerge from the application of "self-discipline" or an appropriate methodology without being exposed in the programs. In particular, the adoption of C++ in the aforementioned high-profile projects shows that practical methodologies exist to deal with the aspects of the C++ language that are considered unsuitable for use in the real-time context, such as exception handling, dynamic memory allocation/deallocation, and the use of virtual functions in the C++ standard library whose instructions, virtual tables, and data may be stored in different locations resulting in non-deterministic cache usage patterns. However, the recent accident involving a Tesla car suggests that even a single accident in the face of thousands of hours of flawless operations could undermine people's trust in a new technology,[12] while a reckless adoption of design short-cuts, solely dominated by economic concerns, could lead to catastrophic outcomes.[13] The difference between high and low quality software is often a few lines of code, which can be the consequence of "seemingly inconsequential choices". The good choices are those based on "codified scientific knowledge rather than on the intuition and on the experience of the programmer".[14] Programming discipline and the adoption of guidelines such as MISRA C/MISRA C++[15,16] can be helpful but do not solve the problem. On the contrary, a well-designed set of language primitives can be a game changer inasmuch as we capitalize on the lessons learned from the unsuccessful (or partially successful) attempts of the past. We believe we can condense such lessons in three paragraphs.

1.2 | Expected Features

First, an embedded real-time application has a functional aspect and an architectural aspect. At the functional level, we can see the application as a network of functional blocks (*runnables* in the AUTOSAR terminology) that interact according to well-defined semantics (often called *model of computation and communication*). At the architectural level, the different blocks are mapped into a set of container tasks, which in turn are associated with their scheduling parameters and, possibly, allocated to different processor cores. Timing constraints are associated with the functional model, but their enforcement heavily depends on the architectural choices. This being said, a language for real-time applications should enable the developer to design the functional model, to decorate the model with timing constraints, to define the mapping of the functional model into some architectural model (or to synthesize the mapping itself), and to propagate the execution constraints to the architectural entities while securing their enforcement.

Second, the language should allow the developers to easily integrate their existing codebases into their new projects. These codebases are in large part composed of legacy C/C++ code and libraries. More generally, embedded software developers' fondness for C/C++ is rooted in a number of objective reasons: "easy access to hardware, low memory requirements, and efficient run-time performance being foremost among them."[17] Additionally, the mastery of the known limitations of the C/C++ language, such as the absence of some run-time checks, the error-prone syntax, and the undefined and implementation-defined areas, is believed to require very experienced programmers. Therefore, adopting a new language should not entail the programmers abandoning their existing programming tools (compilers and possibly also editors, debuggers, and program analyzers) if not also their mastery of the C/C++ language. Nevertheless, many of the programmers have been exposed to the novelty of MBD; within many of their projects, they have to integrate significant parts that have been developed using modeling tools (e.g., MATLAB/SIMULINK) by exploiting their code-generation ability. While the use of such modeling tools gives a sounder basis for engineering the control performance of embedded software, it stops short of providing a sounder basis for the integration of the generated C source programs. The integration with external C/C++ programs, often manual or semi-manual, is an error-prone procedure that can easily invalidate the properties of the model-based code. For this reason, we believe that the advantages of a small set of model-based primitives built on top of their favorite languages could make the novelty palatable even to the most traditionalist and performance-obsessed embedded developers.

Third, in order to make any proposal acceptable, it has to come along with a set of efficient and up-to-date compilation tools able to capture all the inconsistencies of the project and to report them with clear and usable error messages. More importantly, the build process should not be burdened by very long compilation times, which the programmer could find difficult to accept for the frequency of the builds.

1.3 | Contributions

In order to meet the challenging requirements, we have developed a novel real-time programming language called Tice with the main objective of seamlessly integrating within an MBD cycle the system's real-time model and existing C/C++ components and libraries. To achieve the main objective, Tice has been developed with a number of important features:

1. Tice is shaped as an active C++ library; it therefore retains full compatibility with the use of a C/C++ codebase.

2. Tice allows the programmer to define the application as a DAG (directed acyclic graph) of functional blocks and to specify their mapping to architectural entities (concurrent tasks). The programmer can define different types of timing constraints along the different paths taken by the data through the DAG. The adoption of time-triggered LET (logical execution time), which is introduced in the real-time language Giotto,[18] as the model of computation and communication allows us to decouple the timing analysis of the functional behavior from the enforcement of the constraints that make the architectural mapping respect the assumptions of the LET model.

3. Tice is implemented using the technique known as template meta-programming (TMP). Therefore, the syntax and the semantic checks are made directly by the C++ compiler compliant with the C++ standard. What is more, the use of TMP can be combined with other compile-time checks on the data types (e.g., it is possible to associate each piece of data with its measurement unit and demand that the unit stays consistent between the producer and consumer of the data).

4. Tice as much as possible uses TMP data structures with constant access time. Therefore, the compilation process is extremely efficient.

5. Tice reports a comprehensible error message when the compilation fails (the copious and incomprehensible error messages are a known problem incurred by the advanced use of templates in C++) as shown in Figure 10, Figure 11, and Figure 12.

6. Tice generates executables that as much as possible use no techniques (e.g., loop unrolling) and C++ features (e.g., exception) that are considered unsuitable for use in the real-time context. And, if the resulting executables are still deemed to

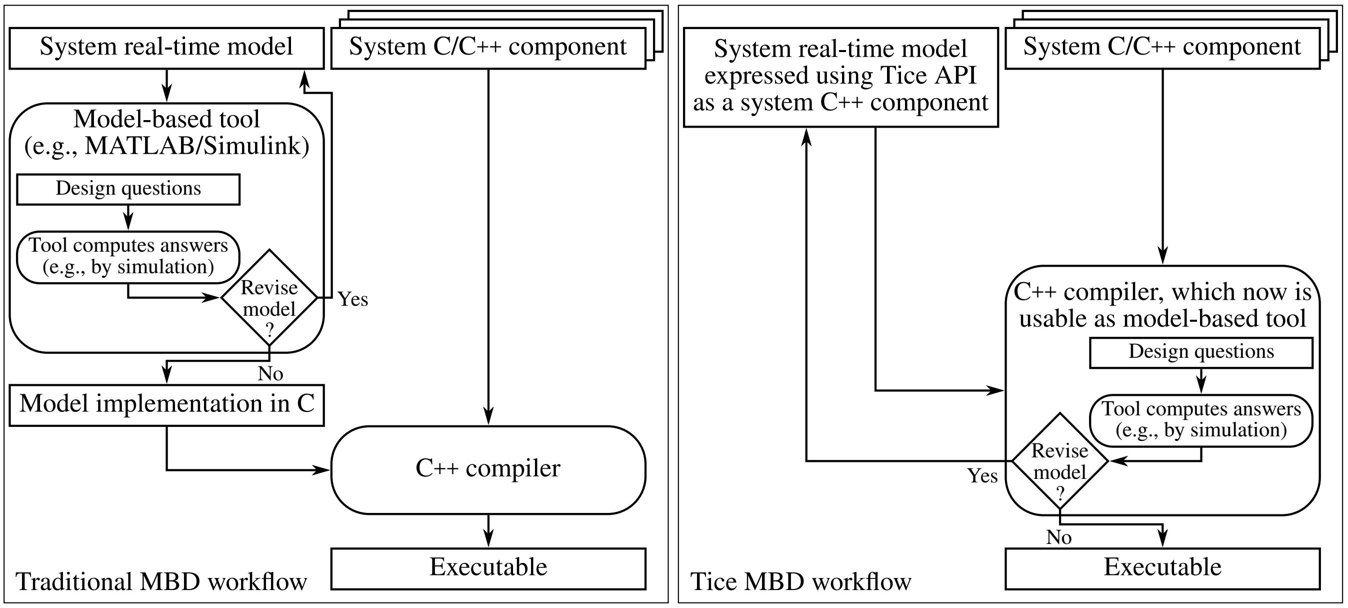

System real-time model
System C/C++ component
Model-based tool (e.g., MATLAB/Simulink)
Design questions
Tool computes answers (e.g., by simulation)
Revise model ?
Yes
No
Model implementation in C
C++ compiler
Executable
Traditional MBD workflow
System real-time model expressed using Tice API as a system C++ component
System C/C++ component
C++ compiler, which now is usable as model-based tool
Design questions
Tool computes answers (e.g., by simulation)
Revise model ?
Yes
No
Executable
Tice MBD workflow

FIGURE 1 The difference between the traditional and Tice MBD workflows.

be using unsuitable techniques and C++ features (or deemed to be missing some techniques and C++ features), Tice can be extended easily to generate executables without undesirable and with desirable techniques and features as explained in Section 6.3.

7. Tice requires none of its users to have any knowledge of TMP. The users of Tice are only required to be capable of using an ordinary C++ library whose application programming interface (API) enables the construction of an object whose type expresses some system's real-time model that will be implemented by the constructed object. We refer to the expression of such an object in a C++ program as a Tice program.

Tice therefore enables a new seamlessly integrated workflow within an MBD cycle as shown in Figure 1. Specifically, while the system real-time model and C/C++ components in the traditional MBD workflow are not integrated seamlessly because the separate model-based tool cannot see the C/C++ components, the system real-time model and C/C++ components are seamlessly integrated in the Tice MBD workflow because the model-based tool is none other than the C++ compiler itself.

In this paper, we make four main contributions:

1. In Section 2, we first explain why Tice is capable of seamlessly integrating a system's real-time model with the system's C/C++ components by making the real-time model compilable using off-the-shelf C++ compilers. Then, in Section 3 we define Tice syntax and semantics to show the kind of real-time models that are expressible in Tice. Note that Tice's main novelty is not the kind of real-time models that are expressible in Tice because we synthesized Tice from existing real-time languages as explained in Section 7; in contrast to other real-time languages that follow the traditional MBD workflow shown in Figure 1, Tice's main novelty is its capability to seamlessly integrate a system's real-time model with the system's C/C++ components to enable the novel Tice MBD workflow shown in Figure 1.

2. In Section 4, we first present an engineering case study called ROSACE (Research Open-Source Avionics and Control Engineering).[19] As the case study calls for the development of embedded software, we then program the embedded software in C++ with the software's real-time aspect being programmed in Tice in Section 4.1. Therefore, we demonstrate Tice's prowess as a real-time language that is not only integrable seamlessly with other C/C++ software components but also compilable using off-the-shelf C++ compilers.

3. In Section 4.2, we use the ROSACE case study to demonstrate the possibility to use Tice and an off-the-shelf C++ compiler as a modeling tool, much like MATLAB/SIMULINK. Furthermore, we show in Section 5 that the possibility is practically feasible by showing the times that popular off-the-shelf C++ compilers, namely GCC and Clang, took to analyze (i.e., compile) some Tice models that approximate cases represented by the ROSACE case study. While other off-the-shelf

C++ compilers exist, GCC and Clang are representative of the others to show the practical feasibility of the proposed Tice MBD workflow because GCC and Clang embody the state of the art in the compilation of standard C++ programs[20,21] and GCC especially has a large user base in embedded software development,[22,23,24] including the high-profile Mars rovers project.[7] Therefore, we show that Tice and an off-the-shelf C++ compiler are altogether usable as a modeling tool that can integrate its generated programs automatically with the other C/C++ software components that make up the complete embedded software.

4. In Section 6, we show how the C++ active library that implements Tice generates code that implements the compiled Tice model. In particular, we show that the library is easily extensible to generate code for various different architecture in Section 6.3.

Lastly, we give the rationale that we use to define the language of Tice and its semantics in Section 7, in which we also outline related work, and draw our conclusions in Section 8, in which we also outline some possible future work.

2 | C++ ACTIVE LIBRARY AND ITS USE OF TEMPLATES IN EMBEDDED SOFTWARE

Tice is capable of seamlessly integrating a system's real-time model with the system's C/C++ components because the real-time model is expressed in Tice and Tice compilers are none other than off-the-shelf C++ compilers. Specifically, the constructs of the Tice language are none other than the API of a C++ library called Tice library. Tice library, however, would not implement the language of Tice were it not capable of analyzing Tice programs expressed using the library API at compile time. Consequently, Tice library has been programmed as a collection of template metaprograms, which will direct off-the-shelf C++ compilers to perform the needed analyses at compile time. Having been programmed as a collection of template metaprograms, Tice library is called as a C++ *active* library, and the implemented language is called as being embedded in C++.[25,26] Tice therefore is a C++ embedded domain-specific language whose embedding is done through TMP (template metaprogramming). While TMP was not designed into C++[27] but discovered by Erwin Unruh in 1994,[28] C++ has evolved to support TMP better. This is because one of the C++ design goals is to provide good support for library development that in turn will provide good support for application development.[6,28]

2.1 | C++ Active Library — Ordinary C++ Library for Its Users but Active for Its Engineers

Figure 2 shows the building blocks (i.e., the constructs) of TMP. The left part of Figure 2 shows a mathematical function that is implemented as a template metaprogram shown on the right part of Figure 2. As shown on the left part, the factorial function is defined only for n between 0 and 12, inclusive. Consequently, the template metaprogram must check at compile time that it is indeed the case. As shown on the right part, the main building block of TMP is a class template (lines 1–2).[29] The class template can be specialized to handle different cases (lines 4–5 specialize the class template for $n = 0$, defining it in lines 6–7), which is analogous to the conditional construct in an imperative language, and the class template's definition can instantiate itself (lines 9–13 define the class template by instantiating itself in line 12), which is analogous to the repetition construct in an imperative language. At this point, it is easy to see that the building block is a Turing-complete programming construct.[29] Another important building block, which did not exist back in 1994, is a static assertion to ensure that some property holds at compile time. For example, line 11 ensures that the template metaprogram will raise a compile-time error if it is used to compute some undefined value at compile time.

To show how the template metaprogram `fact` shown in Figure 2 works at compile time, we will consider the compilation of file `fact.cpp` whose content is the right part of Figure 2 that is appended with "`int main() {return fact<5>::value;}`". The compilation will produce an executable that upon execution terminates with an exit code of 120 — the value of 5! computed by the template metaprogram `fact`. The computation of 5! to produce the value 120, however, happens not when the executable is executed but at compile time when the executable is being produced by a C++ compiler. Specifically, when the C++ compiler processes the expression `fact<5>::value`, the compiler looks for the template definition of `fact` and finds two candidates, the specialized definition for $\texttt{n} = 0$ and the non-specialized definition for any other value of `n`. Since `fact<5>` matches the non-specialized definition, the compiler processes the definition shown in Figure 2 lines 9–13 by first checking that $\texttt{n} \leq 12$, which indeed is the case and therefore raises no compile-time error, and then continues by evaluating the value of class data member `value`. During the evaluation, the compiler encounters the need to evaluate `fact<4>::value` and therefore

$$\text{fact} : \{n \in \mathbb{N} \mid n \leq 12\} \to \mathbb{N}$$

$$\text{fact}(n) = \begin{cases} 1 & \text{, if } n = 0 \\ n \,\text{fact}(n-1) & \text{, otherwise} \end{cases}$$

```
 1 template<unsigned n>
 2 struct fact;
 3
 4 template<>
 5 struct fact<0> {
 6   static constexpr unsigned value = 1;
 7 };
 8
 9 template<unsigned n>
10 struct fact {
11   static_assert(n <= 12, "Arg 1 (n) is out of range");
12   static constexpr unsigned value = n * fact<n - 1>::value;
13 };
```

FIGURE 2 Implementing a factorial function (left) as a C++ template metaprogram (right).

```
 1 $ g++-9 -S -fno-dwarf2-cfi-asm -o - fact.cpp
 2         .file "fact.cpp"
 3         .text
 4         .globl main
 5         .type main, @function
```

```
 6 main:
 7 .LFB0:
 8         pushq %rbp
 9 .LCFI0:
10         movq %rsp, %rbp
```

```
11         movl $120, %eax
12         popq %rbp
13 .LCFI2:
14         ret
15 .LFE0:
```

FIGURE 3 Compilation of Figure 2 (right) appended with "`int main() {return fact<5>::value;}`" (`fact.cpp`).

repeats the look-up process all over again, successively encountering the needs to evaluate `fact<3>::value`, `fact<2>::value`, `fact<1>::value`, and eventually `fact<0>::value`. Upon encountering the need to evaluate `fact<0>::value`, the compiler finds that it matches the specialized definition for `n` = 0 shown in Figure 2 in lines 4–7 and processes it accordingly, evaluating `fact<0>::value` to 1. The compiler then recursively backs up to evaluate `fact<1>::value` to 1, `fact<2>::value` to 2, and so on, eventually evaluating `fact<5>::value` to 120. At that point, the statement "`return fact<5>::value;`" is compiled into the intermediate representation of the statement "return 120;" that eventually is compiled into the assembly instruction shown in Figure 3 lines 11–14. As a result, the executable that is obtained from the assembly instructions by an assembler will not compute 5! at runtime but simply return the value 120 already computed at compile time. It is important to understand that the expression `fact<5>::value` is not optimized by the compiler itself because the compilation command shown in Figure 3 line 1 requests no optimization to be performed (i.e., the compilation command specifies no optimization option, such as `-O2`).

2.2 | Executable Size and Debugging Concerns over Templates Used by a C++ Active Library

The use of class templates in TMP does not imply code bloat, which is the usual reason to ban the use of templates in C++ embedded software.[7] The compilation's assembly listing shown in Figure 3 shows that instead of the code bloat that results from the template instantiations `fact<5>`, `fact<4>`, ..., and `fact<0>`, the template metaprogram `fact` optimizes the expression `fact<5>::value` by directing the C++ compiler to compute its final value and incorporate only the final value (120) in the resulting code in line 11 of Figure 3. In the same way, the C++ active library that implements Tice generates code only as needed, resulting in no code bloat despite the library API being a set of C++ templates. Consequently, the usual reason to ban the use of C++ templates in embedded software development should not hinder the wide-spread adoption of Tice as a real-time language.

The use of class templates in TMP also does not imply executables that are harder to debug owing to their having C++ objects whose types have very long names obtained by recursive template instantiations. As shown in Figure 3, despite the recursive instantiations of template `fact` explained in the previous section, the assembly listing shows no object with a long type name other than the single typeless value 120 in line 11. Furthermore, even if the use of class templates in TMP indeed results in executables whose C++ objects have very long type names, strategies exist in many cases to shorten the object names, for example, by not triggering the C++ one-definition rule (ODR), not to mention that C++ debugging tools themselves may already have means to deal with long type names.

Indeed, the use of C++ templates in embedded software is usually encouraged.[5,8,16] First of all, the judicious use of templates guarantees data consistency and integrity, such as ensuring data consistency and integrity across different C++ functions when the data have physical units of measurement. Secondly, the judicious use of templates lowers runtime overhead by performing domain-specific optimization as exemplified in Figure 3, which in turn lowers energy consumption. These two well-known reasons for using templates in embedded software when coupled with the fact that templates are Turing-complete programming construct allow us to perform even greater variety of static analyses and domain-specific optimization as embodied in the C++ active library that implements Tice.

3 | A REAL-TIME LANGUAGE EMBEDDED IN C++

Since Section 2 has explained why and how Tice is capable of seamlessly integrating a system's real-time model with the system's C/C++ components as well as addressing the usual concern about using C++ templates in embedded software, we will now show the kind of real-time models that are expressible in Tice by describing its syntax and semantics.

3.1 | The Syntax of Tice

The syntax of Tice is none other than the C++ syntax to use the following API members of Tice library, which are enclosed within the C++ namespace `tice::v1`:[30]

- Template `Ratio`. This template is used to express a rational number p/q, and therefore, takes two integer parameters p and q. The second parameter can be omitted, in which case it defaults to one. For example, the integer 1 is expressed as `Ratio<1>`, while the rational number 1/1000 is expressed as `Ratio<1,1000>`. Note that C++ has various safer ways to specify a rational number with a large denominator, for example, 1 μs can be expressed in seconds as `Ratio<1,1000000>` or in a safer way as `Ratio_us(1)` by defining the appropriate macro or as `1_us` by defining the appropriate user-defined literal. In this paper, however, the safer ways are not used to show the canonical syntax of Tice.

- Template `Core_ids`. This template is used to express a set of processor IDs that can be used to execute the set of real-time tasks that Tice library will generate at compile time. Consequently, this template takes a sequence of integers, which can be empty. For example, `Core_ids<0,2>` expresses the fact that the set of real-time tasks must execute on processor cores whose IDs are 0 and 2, while `Core_ids<>` expresses the fact that no real-time tasks shall be generated. If an empty sequence is specified, Tice library will not generate any real-time task but will still analyze the expressed Tice program, which is useful to turn an off-the-shelf C++ compiler into a modeling tool as shown in Section 4.2.

- Template `HW`. This template is used to express the target hardware for which Tice library shall generate the real-time executable. Currently, this template takes `Core_ids` as its sole parameter because Tice library currently generates real-time executables only for hardware with a homogeneous multicore architecture. For example, `HW<Core_ids<0,2>>` expresses target hardware with at least two processor cores, while `HW<Core_ids<>>` expresses the special target hardware for which Tice library generates no code, which is useful when using an off-the-shelf C++ compiler as a modeling tool as shown in Section 4.2. It is not hard to see, however, that in the future, more parameters could be supplied so that Tice library could generate real-time executables also for other hardware architectures (e.g., heterogeneous processors).

- Macro `Comp` and template `comp::Unit`. The macro `Comp` expands to template `comp::Unit` to express the WCET (worst-case execution time) of a C++ function. This macro therefore takes two parameters, the first one being the identifier of the C++ function that is prefixed with `&` (the address-of operator) and the second one being the WCET expressed using `Ratio`. For example, the assignment of a WCET of 250 μs to the C/C++ function `fn` is expressed as `Comp(&fn, Ratio<250, 1000000>)`. It is, however, useful to use `comp::Unit` directly to resolve the ambiguity that results from taking the address of an overloaded function. In this case, while the second parameter of `comp::Unit` is also the second parameter of `Comp`, the first parameter of `comp::Unit` is template `Value`, which takes two parameters. While the second parameter of `Value` is also the first parameter of `Comp`, to resolve the ambiguity, the type of the pointer to the desired overloaded function is specified as the first parameter of `Value`. For example, if `fno` identifies two overloaded C++ functions whose prototypes are "`float fno(float)`" and "`double fno(double)`", respectively, then the assignment of a WCET of 250 μs to the latter `fno` is expressed as `comp::Unit<Value<double(*)(double), &fn>, Ratio<250, 1000000>>`.

- Template `Node`. This template is used to express a real-time periodic computation whose relative deadline is none other than the period itself. This template therefore takes two parameters, the first one being `Comp` or `comp::Unit` to express the computation and the second one being `Ratio` to express the period. For example, after the statement "`typedef Comp(&fn, Ratio<250, 1000000>) comp1;`" is used to identify `comp1` with the computation carried out by the C/C++

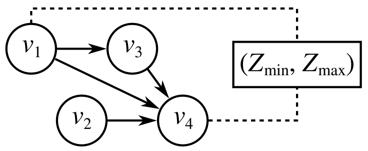


FIGURE 4 A DAG decorated with an end-to-end delay constraint.

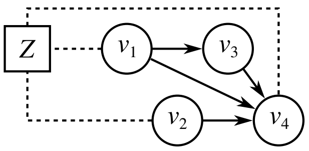


FIGURE 5 A DAG decorated with a correlation constraint.

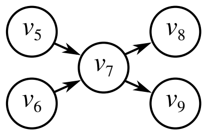


FIGURE 6 Paths $\vec{\pi}_1 = \{(v_5, v_7), (v_7, v_8)\}$ and $\vec{\pi}_2 = \{(v_6, v_7), (v_7, v_9)\}$ meet at node v_7.

function `fn` whose WCET is 250 μs, the periodic repetition of `comp1` every 1 ms is expressed as `Node<comp1, Ratio<1, 1000>>`. The period of an instance of template `Node` is retrievable through the class data member `period`. For example, after the statement "`typedef Node<comp1, Ratio<1, 1000>> v1;`" is used to identify the Tice node with `v1`, the expression `v1::period` retrieves the node's period, which is `Ratio<1, 1000>`.

- Templates `Chan` and `Chan_inlit`. These templates are used to express a communication buffer (i.e., channel). The buffer is a register buffer, which buffers only a single value, that facilitates the communication between some pair of the expressed real-time periodic computations. Both templates therefore take two parameters, the first one being the data type of the buffered value (e.g., `double`) and the second one being the initial value of the buffer. If the initial value has an integral data type (e.g., `bool` and `int`), the initial value can be specified directly as the second parameter of `Chan_inlit`. For example, if `v1` and `v2` identify two distinct Tice nodes, then the buffer that is needed for the communication of `int` data between `v1` and `v2` can be expressed as `Chan_inlit<int, -1>` with -1 being the initial value of the buffer. Otherwise, the initial value has to be placed in a global variable whose identifier prefixed with `&` is then specified as the second parameter of `Chan`. For example, if `v3` and `v4` identify two distinct Tice nodes, then the buffer that is needed for the communication of a 3-element `double` array data between `v3` and `v4` can be expressed as `Chan<SpatialPos, &spatialPosInit>` with `SpatialPos` being the user-defined type "`struct SpatialPos { double data[3]; };`" and `spatialPosInit` being a global variable defined by the statement "`SpatialPos spatialPosInit = {};`".

- Template `Feeder`. This template is used to express the unidirectional communications that take place among the expressed real-time periodic computations. As the communications are unidirectional, `Feeder` expresses the communications that take place from a number of data producers (channel writers) to a single data consumer (channel reader). Consequently, `Feeder` takes a variable number of parameters with three parameters being the minimum. While the last parameter always specifies the consumer using `Node`, the preceding parameters specify producer-channel pairs where the producers are specified using `Node` and the channels are specified using either `Chan` or `Chan_inlit`. The pairs are specified such that every producer is immediately followed by the paired channel. For example, if `v5`, `v6`, `v7`, `v8`, and `v9` identify five distinct Tice nodes whose communication buffers are identified with `ch_5_7`, `ch_6_7`, `ch_7_8`, and `ch_7_9`, then the unidirectional internode communication shown in Figure 6 is expressed as three `Feeder` instances: all arcs going to `v7` are expressed as `Feeder<v5, ch_5_7, v6, ch_6_7, v7>`, the arc going to `v8` is expressed as `Feeder<v7, v_7_8, v8>`, and the arc going to `v9` is expressed as `Feeder<v7, v_7_9, v9>`. Furthermore, for every producer-channel pair, the return type of the producer C/C++ function is required to be assignable to the channel's data type, while the data type of the k-th channel is required to be capable of initializing the k-th parameter of the consumer's C/C++ function.

- Template `ETE_delay`. This template is used to express a last-to-first end-to-end delay constraint by taking four parameters, the first two being the constrained producer and consumer, which are specified using `Node`, while the latter two being the lower and upper bounds of the delay, which are specified using `Ratio`. For example, if `v1` and `v4` identify two distinct Tice nodes such that `v1` is a source node, `v4` is a sink node, and some end-to-end path exists from `v1` to `v4` as shown in Figure 4, then the last-to-first end-to-end delay constraint depicted in Figure 4 is expressed as `ETE_delay<v1, v4, Z_min, Z_max>` with `Z_min` and `Z_max` being `Ratio` instances such that the value of `Z_min` is less than the value of `Z_max`.

- Template `Correlation`. This template is used to express a correlation constraint by taking a variable number of parameters with three being the minimum. The first parameter is the constrained consumer, which is specified using `Node`, and the second parameter is the correlation threshold, which is specified using `Ratio`. The remaining parameters are the constrained producers, which are specified using `Node`. For example, if `v1`, `v2`, and `v4` identify three distinct Tice nodes such that `v1` and `v2` are source nodes, `v4` is a sink node, and some end-to-end path exists from `v1` to `v4` as well as from `v2` to `v4` as shown in Figure 5, then the correlation constraint depicted in Figure 5 is expressed as `Correlation<v4, Z, v1, v2>` with `Z` being a `Ratio` instance.

- Template `Program`. This template is used to express a single complete Tice program. Hence, this template takes a variable number of parameters that are partitioned into five logical sequences, the first two of which cannot be empty:

Seq-A) This sequence has an `HW` instance as its sole element.

Seq-B) This sequence has at least one `Node` instance.

Seq-C) This sequence has zero or more `Feeder` instances.

Seq-D) This sequence has zero or more `ETE_delay` instances.

Seq-E) This sequence has zero or more `Correlation` instances.

For example, if `Z_min`, `Z_max`, and `Z` are some `Ratio` instances and `v1`, `v2`, and so on up to `v9` identify nine distinct Tice nodes whose communication buffers are identified with `ch_1_3`, `ch_1_4`, `ch_2_4`, `ch_3_4`, `ch_5_7`, `ch_6_7`, `ch_7_8`, and `ch_7_9`, then Listing 1, Listing 2, and Listing 3 show complete Tice programs expressing the Tice models shown in Figure 4, Figure 5, and Figure 6, respectively, such that, if the respective C++ variable-declaration statement can be compiled successfully, then `p` is an object whose member function `run` when called at runtime implements using four processor cores the Tice model expressed in the object's type, which is none other than an instance of template `Program`.

Listing 1: C++ variable-declaration statement embedding a Tice program expressing the Tice model shown in Figure 4.

```
Program</*Seq-A*/ HW<Core_ids<0, 1, 2, 3>>,
        /*Seq-B*/ v1, v2, v3, v4,
        /*Seq-C*/ Feeder<v1, ch_1_3, v3>, Feeder<v1, ch_1_4,
                                                v2, ch_2_4,
                                                v3, ch_3_4, v4>,
        /*Seq-D*/ ETE_delay<v1, v4, Z_min, Z_max>> p;
```

Listing 2: C++ variable-declaration statement embedding a Tice program expressing the Tice model shown in Figure 5.

```
Program</*Seq-A*/ HW<Core_ids<0, 1, 2, 3>>,
        /*Seq-B*/ v1, v2, v3, v4,
        /*Seq-C*/ Feeder<v1, ch_1_3, v3>, Feeder<v1, ch_1_4,
                                                v2, ch_2_4,
                                                v3, ch_3_4, v4>,
        /*Seq-E*/ Correlation<v4, Z, v1, v2>> p;
```

Listing 3: C++ variable-declaration statement embedding a Tice program expressing the Tice model shown in Figure 6.

```
Program</*Seq-A*/ HW<Core_ids<0, 1, 2, 3>>,
        /*Seq-B*/ v5, v6, v7, v8, v9,
        /*Seq-C*/ Feeder<v5, ch_5_7,
                         v6, ch_6_7, v7>,
                  Feeder<v7, ch_7_8, v8>, Feeder<v7, ch_7_9, v9>> p;
```

3.2 | The Semantics of Tice

The single complete Tice program expressed using `Program` specifies a Tice model. A Tice model $\mathcal{M}$ is a graph that superimposes three graphs sharing some common nodes and is formally defined in (1) as a 10-tuple with $\mathbb{Q}^{\geq 0}$ being the set of all nonnegative rationals, $\mathbb{Q}^{+}$ being the set of all positive rationals, Θ_{M} being the set of all C++ object types (i.e., C++ data types other than `void`, reference types, and function types), A_θ being the set of all data items whose type is $\theta \in \Theta_{\text{M}}$, and the rest being defined in the following paragraphs.

$$\mathcal{M} = \Big(\mathbb{V}, \mathbb{E}, \mathbb{T}_{\text{E2E}}, \mathbb{T}_{\text{Cor}}, f_{\text{P}} : \mathbb{V} \to \mathbb{Q}^+, f_{\text{C}} : \mathbb{V} \to \mathbb{Q}^+, f_{\text{t}} : \mathbb{E} \to \Theta_{\text{M}}, f_{\text{I}} : \mathbb{E} \to \bigcup_{\theta\in\Theta_{\text{M}}} A_\theta, \\ f_{\text{E2E}} : \mathbb{T}_{\text{E2E}} \to \left(\mathbb{Q}^{\geq 0} \times \mathbb{Q}^+\right), f_{\text{Cor}} : \mathbb{T}_{\text{Cor}} \to \mathbb{Q}^{\geq 0}\Big) \tag{1}$$

The next paragraphs and the rest of this paper use the following common notations and definitions:

- The possibly-empty finite set $\mathbb{V}_{\text{s}}$ is the set of all source nodes in $\mathbb{V}$ as defined in (2).

$$\mathbb{V}_{\text{s}} = \{\, v \in \mathbb{V} \mid (v, \cdot) \in \mathbb{E}, (\cdot, v) \notin \mathbb{E} \,\} \tag{2}$$

- The possibly-empty finite set $\mathbb{V}_{\text{a}}$ is the set of all sink nodes in $\mathbb{V}$ as defined in (3).

$$\mathbb{V}_{\text{a}} = \{\, v \in \mathbb{V} \mid (v, \cdot) \notin \mathbb{E}, (\cdot, v) \in \mathbb{E} \,\} \tag{3}$$

- For any two distinct nodes $v, v' \in \mathbb{V}$, a path from v to v' is denoted $\vec{\pi}_{v,v'}$ and exists if $\emptyset \subset \vec{\pi}_{v,v'} \subseteq \mathbb{E}$ and either $\vec{\pi}_{v,v'} = \left\{\left(v, v'\right)\right\}$, which can also be written as $v \to v'$, or $\vec{\pi}_{v,v'} = \bigcup_{1\leq j\leq i} \left\{\left(v_{j-1}, v_j\right), \left(v_j, v_{j+1}\right)\right\}$ for some positive integer i such that $v_0 = v$ and $v_{i+1} = v'$, which can also be written as $v_0 \to \ldots \to v_j \to \ldots \to v_{i+1}$.

- For any two distinct nodes $v, v' \in \mathbb{V}$, the proposition that $\vec{\pi}_{v,v'}$ exists is true is denoted $v \rightsquigarrow v'$.

- An end-to-end path is some path $\vec{\pi}_{v,v'} \subseteq \mathbb{E}$ such that $v \in \mathbb{V}_{\text{s}}$ and $v' \in \mathbb{V}_{\text{a}}$.

- Whenever it is more important to highlight the members of a path than the source and sink nodes of the path, the path is written as $\vec{\pi}_k$ for some positive integer k. For example, to highlight the two different possible paths from v_1 to v_4 in Figure 4, namely $\left\{\left(v_1, v_4\right)\right\}$ and $\left\{\left(v_1, v_3\right), \left(v_3, v_4\right)\right\}$, the paths are written as $\vec{\pi}_1$ and $\vec{\pi}_2$ instead of as $\vec{\pi}_{v_1,v_4}$ and $\vec{\pi}'_{v_1,v_4}$.

- For any two distinct paths $\vec{\pi}_1, \vec{\pi}_2 \subseteq \mathbb{E}$, the pair is denoted $\vec{\pi}_1$-$\vec{\pi}_2$ and is formally defined as $\vec{\pi}_1\text{-}\vec{\pi}_2 = \left\{\vec{\pi}_1, \vec{\pi}_2\right\}$. Note that the pair is not defined as a 2-tuple $\left(\vec{\pi}_1, \vec{\pi}_2\right)$ but a set $\left\{\vec{\pi}_1, \vec{\pi}_2\right\}$ to abstract from the ordering of the two elements in the pair so that $\vec{\pi}_1\text{-}\vec{\pi}_2 = \vec{\pi}_2\text{-}\vec{\pi}_1$.

The first of the three superimposed graphs is a DAG $(\mathbb{V}, \mathbb{E})$ whose nodes v are function blocks (computational nodes) to be executed periodically every $f_{\text{P}}(v)$ time units with an implicit deadline and a WCET of $f_{\text{C}}(v)$ such that $f_{\text{C}}(v) < f_{\text{P}}(v)$ ($f_{\text{C}}(v) = f_{\text{P}}(v)$ is disallowed to model a computation that does not terminate and begin at the same time) and whose arcs $(v, v'') \in \mathbb{E}$ are directed communication channels from producers v to consumers v''. (The DAG source and sink nodes can be used to model sensors and actuators, respectively.) The set $\mathbb{V}$ is finite and nonempty, while the set $\mathbb{E}$ is finite and possibly empty. Each channel (v, v'') is a data buffer with the following properties:

- The buffer is a register buffer. A register buffer is defined by Feiertag, et al.[31] to be a buffer that holds a single data item at any given time.

- The buffer is typed and holds only data items whose type is $f_{\text{t}}((v, v''))$, which is the buffer's type.

- The buffer initially holds some initial data item $f_{\text{I}}((v, v''))$.

- The buffer has a non-consuming read (i.e., the buffer is a sampling port), and hence, each time the consumer v'' reads the channel, the consumer will always read either some initial data item if the producer v has not written anything to the channel or the latest data item written by the producer (i.e., some unread data item can be lost being overwritten).

- A write and a read that take place at the same time (i.e., synchronously) are sequenced so that the write already completes before the read starts.

For example, the Tice model shown in Figure 6 has $\mathbb{V} = \left\{v_5, v_6, v_7, v_8, v_9\right\}$ and $\mathbb{E} = \left\{\left(v_5, v_7\right), \left(v_6, v_7\right), \left(v_7, v_8\right), \left(v_7, v_9\right)\right\}$.

The second of the three superimposed graphs is a possibly empty undirected graph $\left(\mathbb{V}_{\text{E2E}}, \mathbb{T}_{\text{E2E}}\right)$ with no isolated nodes whose nodes are some connected pairs of the source and sink nodes in $\mathbb{V}$ (possibly all of such pairs) as defined in (4) and whose edges $\left(v_{\text{s}}, v_{\text{a}}\right)$ as defined in (5) are each a last-to-first end-to-end delay constraint applied on one source node v_{s} and one sink node v_{a}. Each constraint's lower and upper bounds are given by the pair $\left(Z_{min}, Z_{max}\right) = f_{\text{E2E}}\left(\left(v_{\text{s}}, v_{\text{a}}\right)\right)$. The second superimposed graph, therefore, specifies a possibly-empty set of last-to-first end-to-end delay constraints $\mathbb{T}_{\text{E2E}}$ on the nodes of the first superimposed graph. For example, the Tice model shown in Figure 4 has $\mathbb{V}_{\text{E2E}} = \{v_1, v_4\}$ and $\mathbb{T}_{\text{E2E}} = \{(v_1, v_4)\}$ with $f_{\text{E2E}}((v_1, v_4)) = (Z_{min}, Z_{max})$.

$$\mathbb{V}_{\text{E2E}} \subseteq \left\{ v_{\text{s}}, v_{\text{a}} \mid v_{\text{s}} \in \mathbb{V}_{\text{s}}, v_{\text{a}} \in \mathbb{V}_{\text{a}}, v_{\text{s}} \rightsquigarrow v_{\text{a}} \right\} \tag{4}$$

$$\mathbb{T}_{\text{E2E}} \subseteq \left\{ (v_{\text{s}}, v_{\text{a}}) \in \mathbb{V}_{\text{E2E}} \times \mathbb{V}_{\text{E2E}} \;\middle|\; v_{\text{s}} \in \mathbb{V}_{\text{s}}, v_{\text{a}} \in \mathbb{V}_{\text{a}} \right\} \tag{5}$$

Similarly, the third of the three superimposed graphs is a possibly empty undirected graph $(\mathbb{V}_{\text{Cor}}, \mathbb{E}_{\text{Cor}})$ with no isolated nodes whose nodes are also some connected pairs of the source and sink nodes in $\mathbb{V}$ (possibly all of such pairs) as defined in (6) but whose edges as defined in (7) are not constraints. Instead, a set of the edges $\mathbb{E}_{\text{C},v_{\text{a}}}$ with a common sink node v_{a} as defined in (8) specifies one correlation constraint with threshold $f_{\text{Cor}}(\mathbb{E}_{\text{C},v_{\text{a}}})$ that is applied on the sink node v_{a} and one or more source nodes v_{s} such that $(v_{\text{s}}, v_{\text{a}}) \in \mathbb{E}_{\text{Cor}}$. The third graph, therefore, specifies a possibly-empty set of correlation constraints $\mathbb{T}_{\text{Cor}}$ defined in (9) on the nodes of the first superimposed graph. For example, the Tice model shown in Figure 5 has $\mathbb{V}_{\text{Cor}} = \{v_1, v_2, v_4\}$, $\mathbb{E}_{\text{Cor}} = \{(v_1, v_4), (v_2, v_4)\}$, and $\mathbb{T}_{\text{Cor}} = \{\{(v_1, v_4), (v_2, v_4)\}\}$ with $f_{\text{Cor}}(\{(v_1, v_4), (v_2, v_4)\}) = Z$.

$$\mathbb{V}_{\text{Cor}} \subseteq \left\{ v_{\text{s}}, v_{\text{a}} \mid v_{\text{s}} \in \mathbb{V}_{\text{s}}, v_{\text{a}} \in \mathbb{V}_{\text{a}}, v_{\text{s}} \rightsquigarrow v_{\text{a}} \right\} \tag{6}$$

$$\mathbb{E}_{\text{Cor}} \subseteq \left\{ (v_{\text{s}}, v_{\text{a}}) \in \mathbb{V}_{\text{Cor}} \times \mathbb{V}_{\text{Cor}} \;\middle|\; v_{\text{s}} \in \mathbb{V}_{\text{s}}, v_{\text{a}} \in \mathbb{V}_{\text{a}} \right\} \tag{7}$$

$$\mathbb{E}_{\text{C},v_{\text{a}}} \in \left(2^{\{(v_{\text{s}}, v_{\text{a}}) \in \mathbb{E}_{\text{Cor}}\}} \setminus \emptyset \right) \tag{8}$$

$$\mathbb{T}_{\text{Cor}} \subseteq \bigcup_{v_{\text{a}} \in (\mathbb{V}_{\text{Cor}} \cap \mathbb{V}_{\text{a}})} \left(2^{\{(v_{\text{s}}, v_{\text{a}}) \in \mathbb{E}_{\text{Cor}}\}} \setminus \emptyset \right) \tag{9}$$

At this point, every element in the 10-tuple that formally defines a Tice model $\mathcal{M}$ in (1) has already been defined by the preceding three paragraphs. However, since the context of the formalism is still on graphs, the notion of confluent nodes will be defined now to ease the presentation of the formal semantics of correlation constraints in Section 3.2.2. For any two distinct end-to-end paths $\vec{\pi}_{v_1,v'}, \vec{\pi}_{v_2,v'} \subseteq \mathbb{E}$ that share a common sink node v' with the source nodes v_1 and v_2 being possibly the same node, the path pair $\vec{\pi}_{v_1,v'}$-$\vec{\pi}_{v_2,v'}$ has a nonempty finite set of confluent nodes $\mathbb{V}^{\Diamond}_{\vec{\pi}_{v_1,v'},\vec{\pi}_{v_2,v'}}$ defined formally in (10) and intuitively are consumer nodes where the two paths meet either for the very first time or after branching.

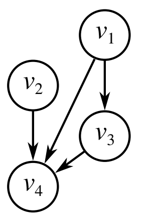
$$\mathbb{V}^{\Diamond}_{\vec{\pi}_1,\vec{\pi}_2} = \left\{ v_1 \;\middle|\; (\cdot, v_1) \in (\vec{\pi}_1 \setminus \vec{\pi}_2) \right\} \cap \left\{ v_2 \;\middle|\; (\cdot, v_2) \in (\vec{\pi}_2 \setminus \vec{\pi}_1) \right\} \tag{10}$$

For example, in Figure 6, four end-to-end paths exist: $\vec{\pi}_1 = \{(v_5, v_7), (v_7, v_8)\}$, $\vec{\pi}_2 = \{(v_6, v_7), (v_7, v_9)\}$, $\vec{\pi}_3 = \{(v_5, v_7), (v_7, v_9)\}$ and $\vec{\pi}_4 = \{(v_6, v_7), (v_7, v_8)\}$. And, while there are six path pairs $\vec{\pi}_1$-$\vec{\pi}_2$, $\vec{\pi}_1$-$\vec{\pi}_3$, $\vec{\pi}_1$-$\vec{\pi}_4$, $\vec{\pi}_2$-$\vec{\pi}_3$, $\vec{\pi}_2$-$\vec{\pi}_4$, and $\vec{\pi}_3$-$\vec{\pi}_4$, only $\vec{\pi}_1$-$\vec{\pi}_4$ and $\vec{\pi}_2$-$\vec{\pi}_3$ have common sink nodes, which are v_8 and v_9, respectively, and therefore have one or more confluent nodes, which in this case is just v_7 at which the respective two paths meet for the very first time. Formally, $\mathbb{V}^{\Diamond}_{\vec{\pi}_1,\vec{\pi}_4} = \left\{ v \;\middle|\; (\cdot, v) \in \{(v_5, v_7)\} \right\} \cap \left\{ v \;\middle|\; (\cdot, v) \in \{(v_6, v_7)\} \right\} = \{v_7\} \cap \{v_7\} = \{v_7\}$ and similarly $\mathbb{V}^{\Diamond}_{\vec{\pi}_2,\vec{\pi}_3} = \{v_7\}$.

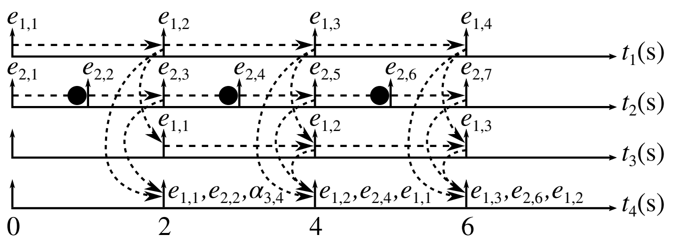
3.2.1 | Model of Computation and Communication (MoCC)

For the semantics of Tice models, we adopt time-triggered LET (logical execution time)[32] with a sink-node relaxation. A sink-node relaxation means that, if a system's response is produced (e.g., actuated) by reading from or writing to some hardware register, then the computation of every sink node $v' \in \mathbb{V}_{\text{a}}$ is assumed to read from or write to the hardware register at any time point throughout the computation. In contrast, the original time-triggered LET requires that the read from or write to the hardware register happens only at certain deterministic time points. The rationale for the relaxation is two Tice's design decisions:

- The time-triggered LET semantics is implemented at compile time only on the values returned by the C/C++ functions that are used as the computations of a Tice model's nodes. Note that this also means that Tice does not check at compile time whether the C/C++ functions used as the computations of a Tice model's nodes implement the time-triggered LET semantics correctly.

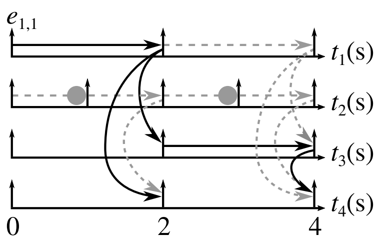


(a) The model.

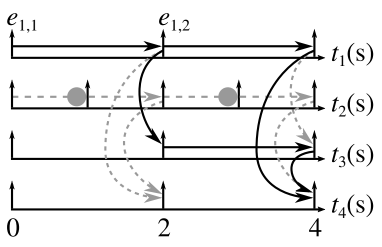


$e_{1,1}$ $e_{1,2}$ $e_{1,3}$ $e_{1,4}$
t_1(s)
$e_{2,1}$ $e_{2,2}$ $e_{2,3}$ $e_{2,4}$ $e_{2,5}$ $e_{2,6}$ $e_{2,7}$
t_2(s)
$e_{1,1}$ $e_{1,2}$ $e_{1,3}$
t_3(s)
$e_{1,1}, e_{2,2}, \alpha_{3,4}$
$e_{1,2}, e_{2,4}, e_{1,1}$
$e_{1,3}, e_{2,6}, e_{1,2}$
t_4(s)
0 2 4 6

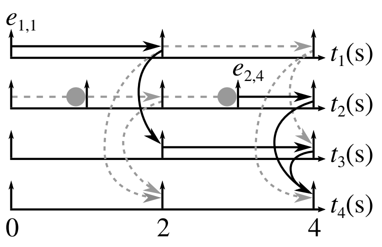
(b) Execution timelines for $P_1 = P_3 = P_4 = 2$ s, $P_2 = 1$ s, and $t_1, t_2, t_3, t_4 \in [0, 6]$.



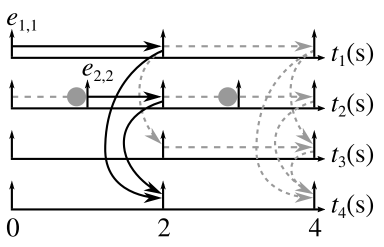
(c) End-to-end flows from v_1 to v_4.



(d) Confluent flows along $\vec{\pi}_2 = \{(v_1, v_4)\}$ and $\vec{\pi}_1 = \{(v_1, v_3), (v_3, v_4)\}$.



(e) Confluent flows along $\vec{\pi}_3 = \{(v_2, v_4)\}$ and $\vec{\pi}_1 = \{(v_1, v_3), (v_3, v_4)\}$.



(f) Confluent flows along $\vec{\pi}_2 = \{(v_1, v_4)\}$ and $\vec{\pi}_3 = \{(v_2, v_4)\}$.

FIGURE 7 Executing a Tice model using time-triggered LET MoCC with a sink-node relaxation.

- A Tice model's sink nodes are identified automatically at compile time based on the nodes whose computations are C/C++ functions whose return type is `void` (i.e., returning no value). Hence, it follows from the previous design decision that the time-triggered LET semantics is not implemented for the computation of every Tice model's sink node. However, nothing prevents the C/C++ functions from implementing the time-triggered LET semantics on their own.

Aside from that, in this and the next sections, Figure 7a will be used as a running example. The DAG of the Tice model shown in Figure 7a is the DAG that is shown in Figure 4 and Figure 5 but rotated 90-degree clockwise to aid reading its internode communication on its execution timelines shown on its right where every timeline would cross their corresponding nodes had the timelines been extended to the left.

Using time-triggered LET as the MoCC of Tice models, every node $v \in \mathbb{V}$ and every arc $\left(v, v''\right) \in \mathbb{E}$ in a Tice model must be assigned some period $f_{\text{P}}(v)$ and some initial data $f_{\text{I}}\left(\left(v, v''\right)\right)$, respectively. All nodes in a Tice model then release (i.e., make available for execution) their computations synchronously at time $t = 0$ (i.e., no offset), and subsequently each node releases its computation exactly once at the start of every $f_{\text{P}}(v)$ time units. Formally, the set of all time points $\mathbb{A}_v$ at which the computation of node v is periodically released is defined in (11). For example, the Tice model shown in Figure 7a has its nodes v_1, v_2, v_3, and v_4 assigned in Figure 7b the periods $f_{\text{P}}(v_1) = 2$ s, $f_{\text{P}}(v_2) = 1$ s, $f_{\text{P}}(v_3) = 2$ s, and $f_{\text{P}}(v_4) = 2$ s, respectively, and therefore as depicted in Figure 7b using the upward arrows on every timeline, $\mathbb{A}_{v_1} = \mathbb{A}_{v_3} = \mathbb{A}_{v_4} = \{0, 2, 4, 6, \dots\}$ and $\mathbb{A}_{v_2} = \{0, 1, 2, 3, \dots\}$.

$$\mathbb{A}_v = \left\{ n f_{\text{P}}(v) \mid n \in \mathbb{N} \right\} \tag{11}$$

Once released at time $t \in \mathbb{A}_v$, a computation will be started at some time $t_s \geq t$ and take at most $f_{\text{C}}(v)$ time units to complete at some time $t_f > t_s$. Note that $t_f - t_s \leq f_{\text{C}}(v)$ if the computation experiences no interference that has not been accounted by $f_{\text{C}}(v)$ (e.g., unexpectedly being preempted by a higher priority computation or unexpectedly waiting to access a memory shared by multiple processor cores). In Tice, a computation is started and completes when the C/C++ function used as a Tice node's computation is called and returns, respectively. A Tice model then is implementable on some target hardware if it is possible to schedule the computation of every node in such a way so that every computation released at time $t \in \mathbb{A}_v$ always completes before its deadline at the next computation's release time $(t + f_{\text{P}}(v)) \in \mathbb{A}_v$ (i.e., $t_f < t + f_{\text{P}}(v)$).

In time-triggered LET, internode communication takes place only at deterministic time points:

- The set of all time points $\mathbb{A}_v^+$ at which a producer node $v \in \mathbb{V}$ writes to every outgoing arc's channel are those given by (12). Based on (12), every producer node v does not write to their outgoing channels at time $t = 0$. For example, referring to Figure 7a, node v_1 writes to arcs (v_1, v_3) and (v_1, v_4) at every time point in the set $\mathbb{A}_{v_1}^+ = \{2, 4, 6, \dots\}$ as shown in Figure 7b with curving dashed arrows that at time 2, 4, and 6, respectively, go from the upward arrow on the timeline t_1

to the upward arrows on the timelines t_3 and t_4.

$$\mathbb{A}_v^+ = \mathbb{A}_v \setminus \{0\} \tag{12}$$

Furthermore, since a channel initially holds some initial value and can only hold one data item at any given time, a write by the channel's producer v at time $t = \min \mathbb{A}_v^+$ overwrites the channel's initial value, while a write at any time $t' \in \left(\mathbb{A}_v^+ \setminus \left\{\min \mathbb{A}_v^+\right\}\right)$ overwrites the data item that was written to the channel at time $t' - f_{\mathrm{P}}(v)$. In Figure 7b, a dashed horizontal line whose right endpoint is a solid circle shows that, due to being overwritten by the data item that is written at the line's right endpoint, the data item that is written at the line's left endpoint is lost without ever being read. On the other hand, a dashed horizontal arrow shows that, due to being overwritten by the data item that is written at the arrow's head, the data item that is written at the arrow's origin is lost but after being read at least once.

- The set of all time points $\mathbb{A}^*_{v,v'',t}$ at which a consumer node $v'' \in \mathbb{V}$ reads from the channel of some arc $(v, v'') \in \mathbb{E}$ when the producer node v writes at time $t \in \mathbb{A}_v^+$ are those given by (13).

$$\mathbb{A}^*_{v,v'',t} = \left\{ f_{\mathrm{P}}(v'')\left\lceil \frac{t}{f_{\mathrm{P}}(v'')} \right\rceil + k f_{\mathrm{P}}(v'') \;\middle|\; f_{\mathrm{P}}(v'')\left\lceil \frac{t}{f_{\mathrm{P}}(v'')} \right\rceil + k f_{\mathrm{P}}(v'') < t + f_{\mathrm{P}}(v), k \in \mathbb{N} \right\} \tag{13}$$

  For example, referring to Figure 7a, the data item that is found in the channel of arc (v_1, v_4) when the producer node v_1 is released at time $t = 2$ is read by the consumer node v_4 at every time point in the set $\mathbb{A}^*_{v_1,v_4,2} = \{2\}$ as shown in Figure 7b with a curving dashed arrow that at time 2 goes from the upward arrow on the timeline t_1 to the upward arrow on the timeline t_4. Furthermore, the set $\mathbb{A}^*_{v_1,v_4,0}$ is undefined due to $0 \notin \mathbb{A}^+_{v_1}$ because at time $t = 0$ every channel holds some initial data item, not some data item written by the channel's producer.

On the other hand, as long as internode communication has not taken place between a pair of producer and consumer nodes $(v, v'') \in \mathbb{E}$ because the producer v has not written any data item to the channel of arc (v, v''), the consumer v'' always reads some initial value $f_{\mathrm{I}}\left(\left(v, v''\right)\right)$ at any time point in the set $\mathbb{A}^-_{v,v''}$ defined in (14). Note that the initial release time is always in the set, that is, $0 \in \mathbb{A}^-_{v,v''}$, and for every node v'' shown in Figure 7a, every upward arrow of the timeline of v'' shown in Figure 7b that is not marked with $e_{i,j}$ for some positive integers i and j indicates every release time of v'' that is in $\mathbb{A}^-_{v,v''}$ for *all* of its producers v.

$$\mathbb{A}^-_{v,v''} = \left\{ t'' \in \mathbb{A}_{v''} \;\middle|\; t'' < \min \mathbb{A}^*_{v,v'',t}, t = \min\left\{ t' \in \mathbb{A}_v^+ \;\middle|\; \mathbb{A}^*_{v,v'',t'} \neq \emptyset \right\} \right\} \tag{14}$$

Lastly, as to the relationship between the periodic computation of every node in a Tice graph and their internode communication in time-triggered LET, at every release time $t \in \mathbb{A}_v$, a Tice model's node v reads the channels of all incoming arcs for the data items that are the input of the computation that will start at $t_s \geq t$ and complete at $t_f < t + f_{\mathrm{P}}(v)$ and, if $t \in \mathbb{A}_v^+$, the node v also writes the output of the computation that completes at $t'_f < t$ to the channels of all outgoing arcs, all of which occur synchronously (i.e., all reads and writes take zero time to complete also at time t). To implement the synchronous read and write, some buffering strategy can be used as detailed in Section 6.2. If a producer v and its consumer v'' happen to be released at time $t \in \mathbb{A}_v$ synchronously (i.e., $t = \min \mathbb{A}^*_{v,v'',t}$), the producer's write on the channel is sequenced before the consumer's read, which again can be implemented by some buffering strategy as detailed in Section 6.2. Therefore, given a Tice model's pair of producer and consumer nodes $\left(v, v''\right) \in \mathbb{E}$, the data dependency that exists between the computations of v and the computations of v'' are defined implicitly by the internode communication of time-triggered LET described in the preceding paragraph.

3.2.2 | Temporal Constraints

The notion of a last-to-first end-to-end delay is defined on any member of the set of all end-to-end paths $\overline{\mathbb{E}}_{v_{\mathrm{s}},v_{\mathrm{a}}} \subseteq \left(2^{\mathbb{E}} \setminus \emptyset\right)$ that exist between some pair of connected source node v_{s} and sink node v_{a} in a Tice model as defined in (15). Specifically, in time-triggered LET, the last-to-first end-to-end delay of every end-to-end path $\vec{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}} \in \overline{\mathbb{E}}_{v_{\mathrm{s}},v_{\mathrm{a}}}$ is computed at any source node's release time $t \in \mathbb{A}_{v_{\mathrm{s}}}$ by the function $g_{\vec{\pi}_{v,v'}} : \mathbb{A}_v \to \{\infty\} \cup \mathbb{Q}^+$ defined in (16) where v'' is any node such that $(v, v'') \in \vec{\pi}_{v,v'}$ and $\vec{\pi}_{v'',v'}$ is $\vec{\pi}_{v,v'} \setminus \{(v, v'')\}$. Intuitively, $g_{\vec{\pi}_{v,v'}}(t) = \infty$ for some $t \in \mathbb{A}_v$ means that the sample obtained by v at time t is never consumed by v', although when $\left|\vec{\pi}_{v,v'}\right| > 1$, the sample may still be consumed by other consumers v'' where $(\cdot, v'') \in \vec{\pi}_{v,v'}$.

$$\overline{\mathbb{E}}_{v_{\mathrm{s}},v_{\mathrm{a}}} = \left\{ \vec{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}} \subseteq \mathbb{E} \;\middle|\; v_{\mathrm{s}} \in \mathbb{V}_{\mathrm{s}}, v_{\mathrm{a}} \in \mathbb{V}_{\mathrm{a}} \right\} \tag{15}$$

$$g_{\vec{\pi}_{v,v'}}(t) = \begin{cases} \min\left(\left\{\infty\right\} \cup \mathbb{A}^*_{v,v',t+f_{\mathrm{P}}(v)}\right) + f_{\mathrm{P}}\left(v'\right) - t & \text{, if } \left|\vec{\pi}_{v,v'}\right| = 1 \\ \min\left(\left\{\infty\right\} \cup \left\{ t^* + g_{\vec{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t+f_{\mathrm{P}}(v)} \right\}\right) - t & \text{, otherwise} \end{cases} \tag{16}$$

For example, Figure 7c illustrates last-to-first end-to-end delay computations on the end-to-end paths $\vec{\pi}_2 = \{(v_1, v_4)\}$ and $\vec{\pi}_1 = \{(v_1, v_3), (v_3, v_4)\}$ shown in Figure 7a when the source node v_1 is released at time $t = 0$. To do so, Figure 7c highlights the relevant computation-communication patterns in Figure 7b by solidifying the dashes of relevant patterns, drawing the irrelevant patterns in gray, and removing any irrelevant markers. For the path $\vec{\pi}_2$ whose length is one arc, the last-to-first end-to-end delay computed using (16) at time $t = 0$ results in four time units because $f_{\mathrm{P}}(v_1) = f_{\mathrm{P}}(v_4) = 2$ and $\mathbb{A}^*_{v_1,v_4,2} = \{2\}$. The computation can be understood intuitively by considering Figure 7c for the scenario where a hardware sensor that is controlled by v_1 samples at time 0 some cause $e_{1,1}$ from the system's environment for which the system has to produce an effect by processing and communicating the cause to a hardware actuator that is controlled by v_4. As shown in Figure 7c, after being sampled at time 0, the cause is processed by v_1 and is written to the channel of arc (v_1, v_4) at time 2 as indicated by the solid horizontal arrow that goes from the upward arrow at time 0 to the upward arrow at time 2. Synchronously at time 2, the processed cause is read from the channel by v_4 as indicated by the solid curving arrow from the upward arrow at time 2 on timeline t_1 to the upward arrow at time 2 on timeline t_4. Since the deadline of the computation of v_4 is at time 4, at the latest the cause will produce an effect by time 4, and therefore, the last-to-first end-to-end delay of the cause sampled at time 0 is four time units for the cause-effect chain $\vec{\pi}_2$. On the other hand, for the cause-effect chain $\vec{\pi}_1$, the last-to-first end-to-end delay of the cause is six time units as computed using (16) because $\left|\vec{\pi}_1\right| = 2$, $f_{\mathrm{P}}(v_3) = 2$, $\mathbb{A}^*_{v_1,v_3,2} = \{2\}$, and $g_{\{(v_3,v_4)\}}(2) = 4$ due to $\mathbb{A}^*_{v_3,v_4,4} = \{4\}$. Therefore, a last-to-first end-to-end delay constraint $\left(v_{\mathrm{s}}, v_{\mathrm{a}}\right) \in \mathbb{T}_{\mathrm{E2E}}$ specified on a Tice model with a pair of lower and upper bounds $\left(Z_{min}, Z_{max}\right) = f_{\mathrm{E2E}}\left(\left(v_{\mathrm{s}}, v_{\mathrm{a}}\right)\right)$ is respected if and only if (17) holds where $\mathbb{G}_{\vec{\pi}_{v,v'}} = \left\{ g_{\vec{\pi}_{v,v'}}(t) \;\middle|\; t \in \mathbb{A}_v \right\}$. Note that the subtraction by $f_{\mathrm{P}}\left(v_{\mathrm{a}}\right)$ in (17) accounts for the time-triggered LET relaxation for every sink node explained at the start of Section 3.2.1, and more importantly, (17) has been shown to be decidable at compile time by Prastowo in Proposition 14.[33]

$$Z_{min} \le \min\left\{ \min \mathbb{G}_{\vec{\pi}_{v,v'}} \;\middle|\; \vec{\pi}_{v,v'} \in \overline{\mathbb{E}}_{v_{\mathrm{s}},v_{\mathrm{a}}} \right\} - f_{\mathrm{P}}\left(v_{\mathrm{a}}\right) < \max\left\{ \max \mathbb{G}_{\vec{\pi}_{v,v'}} \;\middle|\; \vec{\pi}_{v,v'} \in \overline{\mathbb{E}}_{v_{\mathrm{s}},v_{\mathrm{a}}} \right\} \le Z_{max} \tag{17}$$

On the other hand, the notion of a correlation is defined on every pair of confluent end-to-end paths $\vec{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}}, \vec{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}} \in \left(\bigcup_{v''_{\mathrm{s}} \in \mathbb{V}_{\mathrm{s}}} \overline{\mathbb{E}}_{v''_{\mathrm{s}},v_{\mathrm{a}}}\right)$ such that $\mathbb{V}^{\Diamond}_{\vec{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}},\vec{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}} \neq \emptyset$ with v_{s} and v'_{s} being not necessarily distinct (note that when $v_{\mathrm{s}} = v'_{\mathrm{s}}$, the requirement that $\mathbb{V}^{\Diamond}_{\vec{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}},\vec{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}} \neq \emptyset$ ensures that $\vec{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}} \neq \vec{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}$ (e.g., the paths $\{(v_1, v_4)\}$ and $\{(v_1, v_3), (v_3, v_4)\}$ shown in Figure 7a) where the samples consumed by v_{a} from the two different paths at time $t \in \mathbb{A}_{v_{\mathrm{a}}}$ may be obtained by v_{s} at different times $t_1, t_2 \in \mathbb{A}_{v_{\mathrm{s}}}$). Specifically, in time-triggered LET, the correlation of every such path pair $\vec{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}}$-$\vec{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}$ is computed with respect to every node $v^{\Diamond} \in \mathbb{V}^{\Diamond}_{\vec{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}},\vec{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}}$ at any of $v^{\Diamond}$'s release time $t \in \left(\mathbb{D}_{\vec{\pi}_{v_{\mathrm{s}},v^{\Diamond}}} \cap \mathbb{D}_{\vec{\pi}_{v'_{\mathrm{s}},v^{\Diamond}}}\right)$ where $\vec{\pi}_{v_{\mathrm{s}},v^{\Diamond}} \subseteq \vec{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}}$, $\vec{\pi}_{v'_{\mathrm{s}},v^{\Diamond}} \subseteq \vec{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}$, and $\mathbb{D}_{\vec{\pi}_{v,v'}}$ is defined in (18) with $t^{\mathtt{min}}_{\vec{\pi}_{v,v'}} = \min\left\{ t \in \mathbb{A}_v \;\middle|\; g_{\vec{\pi}_{v,v'}}(t) \neq \infty \right\}$. Such computation is done by evaluating $\left|h_{\vec{\pi}_{v_{\mathrm{s}},v^{\Diamond}}}(t) - h_{\vec{\pi}_{v'_{\mathrm{s}},v^{\Diamond}}}(t)\right|$ with the function $h_{\vec{\pi}_{v,v'}} : \mathbb{D}_{\vec{\pi}_{v,v'}} \to \mathbb{A}_v$ being defined in (19) where v'' is any node such that $(v, v'') \in \vec{\pi}_{v,v'}$ and $\vec{\pi}_{v'',v'}$ is $\vec{\pi}_{v,v'} \setminus \{(v, v'')\}$.

$$\mathbb{D}_{\vec{\pi}_{v,v'}} = \left\{ t' \in \mathbb{A}_{v'} \;\middle|\; t' \ge t^{\mathtt{min}}_{\vec{\pi}_{v,v'}} + g_{\vec{\pi}_{v,v'}}\left(t^{\mathtt{min}}_{\vec{\pi}_{v,v'}}\right) - P_{v'} \right\} \tag{18}$$

$$h_{\vec{\pi}_{v,v'}}(t) = \begin{cases} f_{\mathrm{P}}(v)\left(\left\lfloor \frac{t}{f_{\mathrm{P}}(v)} \right\rfloor - 1\right) & \text{, if } \left|\vec{\pi}_{v,v'}\right| = 1 \\ f_{\mathrm{P}}(v)\left(\left\lfloor \frac{h_{\vec{\pi}_{v'',v'}}(t)}{f_{\mathrm{P}}(v)} \right\rfloor - 1\right) & \text{, otherwise} \end{cases} \tag{19}$$

For example, Figures 7(d)–(f) illustrate how correlations are computed for the path pairs $\vec{\pi}_1$-$\vec{\pi}_2$, $\vec{\pi}_1$-$\vec{\pi}_3$, and $\vec{\pi}_2$-$\vec{\pi}_3$, respectively, where $\vec{\pi}_1 = \{(v_1, v_3), (v_3, v_4)\}$, $\vec{\pi}_2 = \{(v_1, v_4)\}$, $\vec{\pi}_3 = \{(v_2, v_4)\}$, and each of the pairs has only one confluent node, namely v_4, because $\mathbb{V}^{\Diamond}_{\vec{\pi}_1,\vec{\pi}_2} = \mathbb{V}^{\Diamond}_{\vec{\pi}_1,\vec{\pi}_3} = \mathbb{V}^{\Diamond}_{\vec{\pi}_2,\vec{\pi}_3} = \{v_4\}$. The figures show that the correlation at v_4 is undefined at $t = 0$ because when v_4 reads any of its incoming channels synchronously at $t = 0$, only initial data items are read, that is, $0 \notin \left(\left(\mathbb{A}_{v_4} \setminus \mathbb{A}^-_{v_1,v_4}\right) \setminus \mathbb{A}^-_{v_3,v_4}\right)$ in Figure 7d, $0 \notin \left(\left(\mathbb{A}_{v_4} \setminus \mathbb{A}^-_{v_2,v_4}\right) \setminus \mathbb{A}^-_{v_3,v_4}\right)$ in Figure 7e, and $0 \notin \left(\left(\mathbb{A}_{v_4} \setminus \mathbb{A}^-_{v_1,v_4}\right) \setminus \mathbb{A}^-_{v_2,v_4}\right)$ in Figure 7f, or equivalently, $0 \notin \left(\mathbb{D}_{\vec{\pi}_1} \cap \mathbb{D}_{\vec{\pi}_2}\right)$ in Figure 7d, $0 \notin \left(\mathbb{D}_{\vec{\pi}_1} \cap \mathbb{D}_{\vec{\pi}_3}\right)$ in Figure 7e, and $0 \notin \left(\mathbb{D}_{\vec{\pi}_2} \cap \mathbb{D}_{\vec{\pi}_3}\right)$ in Figure 7f. On the other hand, Figures 7(d)–(e) show that the correlation at v_4 is undefined at $t = 2$ because, even though v_4 no longer reads only initial data items due to

$2 \in \left(\left(\mathbb{A}_{v_4} \setminus \mathbb{A}^{-}_{v_1,v_4}\right) \setminus \mathbb{A}^{-}_{v_3,v_4}\right)$ in Figure 7d and $2 \in \left(\left(\mathbb{A}_{v_4} \setminus \mathbb{A}^{-}_{v_2,v_4}\right) \setminus \mathbb{A}^{-}_{v_3,v_4}\right)$ in Figure 7e, v_4 still has not read from both of the two meeting paths the data items that are initially read by the source nodes of the respective paths, specifically $2 \notin \left(\mathbb{D}_{\vec{\pi}_1} \cap \mathbb{D}_{\vec{\pi}_2}\right)$ in Figure 7d and $2 \notin \left(\mathbb{D}_{\vec{\pi}_1} \cap \mathbb{D}_{\vec{\pi}_3}\right)$ in Figure 7e. In contrast, Figure 7f shows that the correlation at v_4 is defined at $t = 2$ because v_4 reads no initial data from the pair $\vec{\pi}_2$-$\vec{\pi}_3$ due to $2 \in \left(\left(\mathbb{A}_{v_4} \setminus \mathbb{A}^{-}_{v_1,v_4}\right) \setminus \mathbb{A}^{-}_{v_2,v_4}\right)$ and has read from both of the two meeting paths the data items that are initially read by the source nodes of the respective paths due to $2 \in \left(\mathbb{D}_{\vec{\pi}_2} \cap \mathbb{D}_{\vec{\pi}_3}\right)$, and therefore, the correlation at v_4 can be determined to be $\left|h_{\{(v_1,v_4)\}}(2) - h_{\{(v_2,v_4)\}}(2)\right| = \left|f_{\mathtt{P}}(v_1)(1-1) - f_{\mathtt{P}}(v_2)(2-1)\right| = 1$ time unit. The correlation can be understood intuitively by considering Figure 7f for the scenario where hardware sensors that are controlled by v_1 and v_2 sample at time 0 and 1 some causes $e_{1,1}$ and $e_{2,2}$, respectively, from the system's environment for which the system has to produce an effect by processing and communicating the cause to a hardware actuator that is controlled by v_4. As shown in Figure 7f, after being sampled at time 0, cause $e_{1,1}$ is processed by v_1 and is written to the channel of arc (v_1, v_4) at time 2 as indicated by the solid horizontal arrow that goes from the upward arrow at time 0 to the upward arrow at time 2. Similarly, after being sampled at time 1, cause $e_{2,2}$ is processed by v_2 and is written to the channel of arc (v_2, v_4) at time 2 as indicated by the solid horizontal arrow that goes from the upward arrow at time 1 to the upward arrow at time 2. Synchronously at time 2, the processed causes are read from the channels by v_4 as indicated by the solid curving arrows from the upward arrows at time 2 on timelines t_1 and t_2, respectively, to the upward arrow at time 2 on timeline t_4. Consequently, v_4 will produce an effect based on causes that are sampled by one time-unit apart. For the same reason, Figures 7(d)–(e) show that the correlations at v_4 is defined at $t = 4$ where the correlations can be determined to be $\left|h_{\{(v_1,v_4)\}}(4) - h_{\{(v_1,v_3),(v_3,v_4)\}}(4)\right| = \left|f_{\mathtt{P}}(v_1)(2-1) - f_{\mathtt{P}}(v_1)\left(\left\lfloor \frac{h_{\{(v_3,v_4)\}}(4)}{f_{\mathtt{P}}(v_1)} \right\rfloor - 1\right)\right| = \left|2 - 2\left(\left\lfloor \frac{2}{2} \right\rfloor - 1\right)\right| = 2$ time units and $\left|h_{\{(v_2,v_4)\}}(4) - h_{\{(v_1,v_3),(v_3,v_4)\}}(4)\right| = \left|f_{\mathtt{P}}(v_2)(4-1) - f_{\mathtt{P}}(v_1)\left(\left\lfloor \frac{h_{\{(v_3,v_4)\}}(4)}{f_{\mathtt{P}}(v_1)} \right\rfloor - 1\right)\right| = \left|3 - 2\left(\left\lfloor \frac{2}{2} \right\rfloor - 1\right)\right| = 3$ time units, respectively. Therefore, a correlation constraint $\mathbb{E}_{\mathtt{C},v_\mathtt{a}} \in \mathbb{T}_{\mathtt{Cor}}$ specified on a Tice model with correlation threshold $Z = f_{\mathtt{Cor}}\left(\mathbb{E}_{\mathtt{C},v_\mathtt{a}}\right)$ is respected if and only if (20) holds where $\mathbb{H}^{\vec{\pi}_{v_\mathtt{s},v^\Diamond}}_{\vec{\pi}_{v'_\mathtt{s},v^\Diamond}} = \left\{ \left|h_{\vec{\pi}_{v_\mathtt{s},v^\Diamond}}(t') - h_{\vec{\pi}_{v'_\mathtt{s},v^\Diamond}}(t')\right| \;\middle|\; t' \in \left(\mathbb{D}_{\vec{\pi}_{v_\mathtt{s},v^\Diamond}} \cap \mathbb{D}_{\vec{\pi}_{v'_\mathtt{s},v^\Diamond}}\right) \right\}$. Note that (20) has been shown to be decidable at compile time by Prastowo in Proposition 16.[33]

$$\max\left(\left\{ \max \mathbb{H}^{\vec{\pi}_{v_\mathtt{s},v^\Diamond}}_{\vec{\pi}_{v'_\mathtt{s},v^\Diamond}} \;\middle|\; \begin{array}{l} \vec{\pi}_{v_\mathtt{s},v_\mathtt{a}}, \vec{\pi}_{v'_\mathtt{s},v_\mathtt{a}} \in \bigcup_{v''_\mathtt{s} \in \{ v_\mathtt{s} \mid (v_\mathtt{s},v_\mathtt{a}) \in \mathbb{E}_{\mathtt{C},v_\mathtt{a}} \}} \overline{\mathbb{E}}_{v''_\mathtt{s},v_\mathtt{a}}, \quad \vec{\pi}_{v_\mathtt{s},v_\mathtt{a}} \neq \vec{\pi}_{v'_\mathtt{s},v_\mathtt{a}}, \\ v^\Diamond \in \mathbb{V}^\Diamond_{\vec{\pi}_{v_\mathtt{s},v_\mathtt{a}},\vec{\pi}_{v'_\mathtt{s},v_\mathtt{a}}}, \quad \vec{\pi}_{v_\mathtt{s},v^\Diamond} \subseteq \vec{\pi}_{v_\mathtt{s},v_\mathtt{a}}, \quad \vec{\pi}_{v'_\mathtt{s},v^\Diamond} \subseteq \vec{\pi}_{v'_\mathtt{s},v_\mathtt{a}} \end{array} \right\} \cup \left\{0\right\}\right) \leq Z \tag{20}$$

4 | THE ROSACE CASE STUDY

The ROSACE case study is a complete case study in engineering a longitudinal flight controller, which controls the altitude and air speed of an airborne aircraft. The case study champions a design methodology with three stages to facilitate the communication between the control and software engineers in the second stage by means of plotted graphs.

In the case study's first stage, the control engineers design the controller using MATLAB/SIMULINK. Once done, the controlled plant (i.e., aircraft) is simulated in MATLAB/SIMULINK with its altitude, air speed, and some other outputs traced and plotted as reference graphs for the case where, starting at the flight condition whose altitude and air speed are 10 km and 230 m/s, respectively, the controller's reference input for the altitude is step changed to 11 km at the start of the simulation (i.e., at $t = 0$, which in the case study's resulting graphs is offset by 50 s to ease reading). In the case study's second stage, the software engineers implement the MATLAB/SIMULINK model as embedded software and execute the resulting software on their computers while tracing some output data produced during execution (possibly using some software simulator that simulates the software's real-time execution). The trace data can then be plotted to be compared with the reference graphs as the basis for discussions with the control engineers to reach an agreement on the controller's design, which, if not yet reached, can be done either by modifying the software (repeating stage two) or the MATLAB/SIMULINK model (repeating the first two stages). Upon reaching an agreement, the software engineers in the last stage embed the software on the target hardware to validate the resulting system in the same manner (i.e., by plotting the target hardware's outputs and comparing them to the reference graphs).

For our purpose, only the case study's second stage is used in the rest of this section where we express the ROSACE's MATLAB/SIMULINK model as a Tice model that is then programmed as a Tice program by using the Tice library API. In doing so, we have used the work of Pierre-Emmanuel Hladik found in the ROSACE repository[34] because Hladik has implemented the MATLAB/SIMULINK model in the real-time language GIOTTO whose MoCC is time-triggered LET,[32] easing our work.

$$\begin{tikzcd}
h_s \arrow[r] & h_f \arrow[dr] & & \\
a_{z_s} \arrow[r] & a_{z_f} \arrow[dr] & h_h \arrow[d] & \\
q_s \arrow[r] & q_f \arrow[r] \arrow[dr] & V_z \arrow[r] & L \\
V_{z_s} \arrow[r] & V_{z_f} \arrow[ur] \arrow[r] & V_a \arrow[r] & E \\
V_{a_s} \arrow[r] & V_{a_f} \arrow[ur] & &
\end{tikzcd}$$

```
 1 #include <tice/v1.hpp>
 2 #include "tice_model_implementation.hpp"
 3
 4 using namespace tice::v1;
 5
 6 typedef Node<f_h_s  , Ratio<5 , 1000>> h_s  ; // Altitude sensor
 7 typedef Node<f_a_z_s, Ratio<5 , 1000>> a_z_s; // Vertical acceleration sensor
 8 typedef Node<f_q_s  , Ratio<5 , 1000>> q_s  ; // Pitch rate sensor
 9 typedef Node<f_V_z_s, Ratio<5 , 1000>> V_z_s; // Vertical speed sensor
10 typedef Node<f_V_a_s, Ratio<5 , 1000>> V_a_s; // True airspeed sensor
11 typedef Node<f_h_f  , Ratio<10, 1000>> h_f  ; // Altitude filter
12 typedef Node<f_a_z_f, Ratio<10, 1000>> a_z_f; // Vertical acceleration filter
13 typedef Node<f_q_f  , Ratio<10, 1000>> q_f  ; // Pitch rate filter
14 typedef Node<f_V_z_f, Ratio<10, 1000>> V_z_f; // Vertical speed filter
15 typedef Node<f_V_a_f, Ratio<10, 1000>> V_a_f; // Airspeed filter
16 typedef Node<f_h_h  , Ratio<20, 1000>> h_h  ; // Altitude hold
17 typedef Node<f_V_z  , Ratio<20, 1000>> V_z  ; // Altitude control
18 typedef Node<f_V_a  , Ratio<20, 1000>> V_a  ; // Airspeed control
19 typedef Node<f_L    , Ratio<5 , 1000>> L    ; // Elevator actuator
20 typedef Node<f_E    , Ratio<5 , 1000>> E    ; // Engine thrust actuator
21 typedef Program</* 1*/target_hw,
22                 /* 2*/h_s, /* 3*/a_z_s, /* 4*/q_s, /* 5*/V_z_s, /* 6*/V_a_s,
23                 /* 7*/h_f, /* 8*/a_z_f, /* 9*/q_f, /*10*/V_z_f, /*11*/V_a_f,
24                 /*12*/h_h, /*13*/V_z  , /*14*/V_a, /*15*/L    , /*16*/E,
25                 /*17*/Feeder<h_s  ,   h_s_to_h_f  , h_f>,
26                 /*18*/Feeder<a_z_s, a_z_s_to_a_z_f, a_z_f>,
27                 /*19*/Feeder<q_s  ,   q_s_to_q_f  , q_f>,
28                 /*20*/Feeder<V_z_s, V_z_s_to_V_z_f, V_z_f>,
29                 /*21*/Feeder<V_a_s, V_a_s_to_V_a_f, V_a_f>,
30                 /*22*/Feeder<h_f  ,   h_f_to_h_h  , h_h>,
31                 /*23*/Feeder<h_h  ,   h_h_to_V_z  ,
32                              a_z_f, a_z_f_to_V_z  ,
33                              q_f  ,   q_f_to_V_z  ,
34                              V_z_f, V_z_f_to_V_z  , V_z>,
35                 /*24*/Feeder<q_f  ,   q_f_to_V_a  ,
36                              V_z_f, V_z_f_to_V_a  ,
37                              V_a_f, V_a_f_to_V_a  , V_a>,
38                 /*25*/Feeder<V_z  ,   V_z_to_L    , L>,
39                 /*26*/Feeder<V_a  ,   V_a_to_E    , E>
40                 /*27*/TEMPORAL_CONSTRAINTS> Prog;
```

FIGURE 8 ROSACE's MATLAB/SIMULINK model expressed as Tice model (left) and its expression in Tice language (right).

4.1 | The Embedded Software Programmed in Tice

Figure 8 shows the ROSACE's MATLAB/SIMULINK model as a Tice model on the left, and on the right, the Tice model is programmed as a Tice program using the Tice library API, which is included in line 1 and is already described in Section 3.1. The Tice model's nodes shown on the left are then assigned their computations and periods in lines 6–20. The assigned computations are declared in the C++ header file included in line 2. Once every node has been declared, the Tice model is expressed in lines 21–40. To aid reading, every parameter has been preceded by a comment specifying the parameter's position. While parameter 1 identifies the target hardware that the Tice model will be executed on, which is declared in the file included in line 2, parameters 2–16 draw all of the nodes shown on the figure. Parameters 17–26, on the other hand, draw all of the arcs where all arcs pointing to the same consumer node is drawn altogether by a single `Feeder`. For example, parameter 24 draws all of the arcs that point to V_a by specifying as the last parameter `V_a`, as the first producer-channel pair `q_f` and `q_f_to_V_a`, which draws the arc whose tail is at q_f and whose body represents the channel `q_f_to_V_a`, as the second producer-channel pair `V_z_f` and `V_z_f_to_V_a` to draw the arc that originates from V_{z_f} with `V_z_f_to_V_a` as its channel, and as the third producer-channel pair `V_a_f` and `V_a_f_to_V_a` to draw the last arc. All of the channels are declared in the file included in line 2. Lastly, starting with parameter 27, the rest of the parameters, if any, specify a number of last-to-first end-to-end delay and correlation constraints. As the case study expresses no such constraint on the MATLAB/SIMULINK model, the preprocessing macro `TEMPORAL_CONSTRAINTS` will expand to nothing at compile time (see Section 4.2 for the case where the macro should expand to some `ETE_delay` and `Correlation` instances). Once the Tice model has been programmed, it can be implemented with some specific set of function blocks, which determines the kinds of data and the ways the data are to be processed, and for some specific target hardware, which determines the WCETs of the function blocks as well as their mappings to execution threads and the scheduling of the threads on the available processor cores.

Figure 9 shows the implementation of the Tice model with the set of function blocks derived from Pierre-Emmanuel Hladik's work (the data-dependent part in lines 4–39) and for certain target hardware with four homogeneous processor cores (the

```
 1 #include <tice/v1.hpp>
 2 using namespace tice::v1;
 3
 4 // Data-dependent part
 5 extern const double
 6   init_h_s, init_a_z_s, init_q_s, init_V_z_s, init_V_a_s,
 7   init_h_f, init_a_z_f, init_q_f, init_V_z_f, init_V_a_f,
 8   init_h_h, init_V_z  , init_V_a;
 9 typedef Chan<double, &init_h_s  >   h_s_to_h_f  ;
10 typedef Chan<double, &init_a_z_s> a_z_s_to_a_z_f;
11 typedef Chan<double, &init_q_s  >   q_s_to_q_f  ;
12 typedef Chan<double, &init_V_z_s> V_z_s_to_V_z_f;
13 typedef Chan<double, &init_V_a_s> V_a_s_to_V_a_f;
14 typedef Chan<double, &init_h_f  >   h_f_to_h_h  ;
15 typedef Chan<double, &init_h_h  >   h_h_to_V_z  ;
16 typedef Chan<double, &init_a_z_f> a_z_f_to_V_z  ;
17 typedef Chan<double, &init_q_f  >   q_f_to_V_z  ;
18 typedef Chan<double, &init_V_z_f> V_z_f_to_V_z  ;
19 typedef Chan<double, &init_q_f  >   q_f_to_V_a  ;
20 typedef Chan<double, &init_V_z_f> V_z_f_to_V_a  ;
21 typedef Chan<double, &init_V_a_f> V_a_f_to_V_a  ;
22 typedef Chan<double, &init_V_z  >   V_z_to_L    ;
23 typedef Chan<double, &init_V_a  >   V_a_to_E    ;
24
25 double f_a_z_s_impl();  double f_h_s_impl();
26 double f_V_z_s_impl();  double f_q_s_impl();
27 double f_V_a_s_impl();
28 double f_h_f_impl(const double &h_s);
29 double f_a_z_f_impl(const double &a_z_s);
30 double f_q_f_impl(const double &q_s);
31 double f_V_z_f_impl(const double &V_z_s);
32 double f_V_a_f_impl(const double &V_a_s);
33 double f_h_h_impl(const double &h_f);
34 double f_V_z_impl(const double &h_h, const double &a_z_f,
35                   const double &q_f, const double &V_z_f);
36 double f_V_a_impl(const double &q_f, const double &V_z_f,
37                   const double &V_a_f);
38 void f_L_impl(const double &V_z);
39 void f_E_impl(const double &V_a);
```

```
40 // Hardware-dependent part
41 #ifdef TICE_V1_NOGEN
42 typedef HW<Core_ids<>> target_hw;
43 #else
44 typedef HW<Core_ids<0, 1,
45                     2, 3>> target_hw;
46 #endif
47
48 //// Function block WCETs on target_hw
49 typedef Comp(&f_h_s_impl,
50              Ratio<1, 1000>) f_h_s;
51 typedef Comp(&f_a_z_s_impl,
52              Ratio<1, 1000>) f_a_z_s;
53 typedef Comp(&f_q_s_impl,
54              Ratio<1, 1000>) f_q_s;
55 typedef Comp(&f_V_z_s_impl,
56              Ratio<1, 1000>) f_V_z_s;
57 typedef Comp(&f_V_a_s_impl,
58              Ratio<1, 1000>) f_V_a_s;
59 typedef Comp(&f_h_f_impl,
60              Ratio<1, 1000>) f_h_f;
61 typedef Comp(&f_a_z_f_impl,
62              Ratio<1, 1000>) f_a_z_f;
63 typedef Comp(&f_q_f_impl,
64              Ratio<1, 1000>) f_q_f;
65 typedef Comp(&f_V_z_f_impl,
66              Ratio<1, 1000>) f_V_z_f;
67 typedef Comp(&f_V_a_f_impl,
68              Ratio<1, 1000>) f_V_a_f;
69 typedef Comp(&f_h_h_impl,
70              Ratio<1, 1000>) f_h_h;
71 typedef Comp(&f_V_z_impl,
72              Ratio<1, 1000>) f_V_z;
73 typedef Comp(&f_V_a_impl,
74              Ratio<1, 1000>) f_V_a;
75 typedef Comp(&f_L_impl,
76              Ratio<1, 1000>) f_L;
77 typedef Comp(&f_E_impl,
78              Ratio<1, 1000>) f_E;
```

FIGURE 9 The content of the C++ header file `tice_model_implementation.hpp`.

hardware-dependent part in lines 40–78). In lines 9–23, the data-dependent part declares every channel that implements the arcs of the Tice model. The channels are typed by the kinds of data to be communicated, which are specified as their first parameters. The channel types will ensure the compatibility between the output ports of the producers and the connected input ports of the consumers. Consequently, in lines 25–39, the data-dependent part also declares the function block implementations whose return types are the types of the producer output ports and whose parameters specify the types of the consumer input ports. This is because whenever a producer changes its output port type or a consumer changes one of its input port types, the connected channel *may* need to change its type as well, which in turn *may* necessitate the other endpoint's input/output port type to change as well. The change, however, is not always necessary because some types are and can be made compatible in C++. For example, changing all output and input port types in lines 25–39 to `float` requires no change to the channel declarations as far as the language is concerned. Aside from that, the initial data of every channel are stored in constant variables typed `double` declared in lines 5–8. The types of the initial data *may* need to change when the types of the initialized channels change, and vice versa. As before, the change is not always necessary. For example, changing the variable types to `float` requires no change to the channel declarations.

On the other hand, the hardware-dependent part in lines 49–78 declares the WCETs of the function blocks on the target hardware. For our purpose, there is no need to derive the tightest WCETs, and therefore, we have assigned the same WCET to every function block. Lastly in lines 41–46, the hardware-dependent part declares the target hardware itself. A special target hardware specifying no ID will be used as the target hardware if the preprocessing macro `TICE_V1_NOGEN`, which is also a Tice API member, is defined for the case described in Section 4.2. Otherwise, the target hardware makes available four cores to execute the function blocks.

At this point, it should be easy to see that, as a real-time language whose semantics is already described in Section 3.2, Tice is integrable seamlessly with other C/C++ software components. For example, the data-dependent part of the Tice model's implementation can easily use other kinds of data defined by other C/C++ software components, such as replacing the unitless `double` with some unit-aware C++ types to avoid the kind of mismatch that destroyed the maiden flight of Ariane 5 rocket.[35] Lastly, as

```
1 $ make -B -s CXX=/usr/bin/g++-9 CXXFLAGS='-D TICE_V1_NOGEN -D TEMPORAL_CONSTRAINTS=", \
2 ETE_delay<h_s, L, Ratio<0>, h_s::period>                                             \
3 "' 2>&1 | head -n 3
4 In file included from tice_model.hpp:26,
5                  from main.cpp:22:
6 ../../../tice/v1.hpp: In instantiation of ‘struct
7 tice::v1::error::program::end_to_end_delay_is_between_min_and_max_delays<false, 27, std::ratio<0>,
8 std::ratio<60001, 1000000>, std::ratio<5, 1000>, tice::v1::error::Path_forming_node_pos<2, 7, 12,
9 13, 15>, 2>’:
```

(a) GCC version 9.1.0.

```
 1 $ make -B -s CXX=/usr/bin/clang++-9 CXXFLAGS='-D TICE_V1_NOGEN -D TEMPORAL_CONSTRAINTS=", \
 2 ETE_delay<h_s, L, Ratio<0>, h_s::period>                                                 \
 3 "' 2>&1 | head -n 6
 4 In file included from main.cpp:22:
 5 In file included from ./tice_model.hpp:26:
 6 ../../../tice/v1.hpp:2354:11: fatal error: static_assert failed "Min and max end-to-end delays are
 7 not respected"
 8           static_assert(arg__is_between_min_and_max_delays,
 9           ^          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
10 ../../../tice/internals/v1/v1_internals_program.hpp:7036:11: note: in instantiation of template
11 class 'tice::v1::error::program::end_to_end_delay_is_between_min_and_max_delays<false, 27,
12 std::ratio<0, 1>, std::ratio<60001, 1000000>, std::ratio<5, 1000>,
13 tice::v1::error::Path_forming_node_pos<2, 7, 12, 13, 15>, 2>' requested here
```

(b) Clang version 9.0.0.

FIGURE 10 Tice and off-the-shelf C++ compilers as modeling tools to figure out the last-to-first end-to-end delay from h_s to L.

a real-time language that is compilable using off-the-shelf C++ compilers, the Tice program shown in Figure 8 and its particular implementation shown in Figure 9 along with other C++ source program files in the directory `experiments/v1-rosace` in the Tice repository[36] can be compiled by GCC and Clang to produce embedded software whose execution on the target hardware behaves according to the ROSACE's reference graphs.

4.2 | Using Tice and Off-The-Shelf C++ Compilers as a Modeling Tool

As a model-based real-time language, Tice together with an off-the-shelf compiler can also be used as a modeling tool. For example, an engineer can request an off-the-shelf C++ compiler to analyze the Tice model in Figure 8 for its last-to-first end-to-end delay and correlation properties. For example, referring to the left part of Figure 8, a question can be asked about the model, "what is the last-to-first end-to-end delay from h_s to L?" To answer the design question, an end-to-end delay constraint can be attached to nodes h_s and L with lower and upper bounds set to zero and some positive value, respectively, as shown in line 2 of both Figure 10a and Figure 10b. In both figures, lines 1–3 compile the Tice program in the same way it is compiled to obtain the ROSACE embedded software described in Section 4.1 except that the `CXXFLAGS` is set to specify the compiler option `-D` twice. The first `-D` option defines the macro `TICE_V1_NOGEN` to save compilation time by preventing the compilers from mapping the Tice model to a set of real-time tasks due to line 42 of Figure 9 defining target hardware with no processor core and from generating any executable code implementing the Tice model due to Tice library's code generator being turned off when the macro is defined. On the other hand, the second `-D` option replaces the macro `TEMPORAL_CONSTRAINTS` in line 40 of Figure 8 with an `ETE_delay` that expresses the desired end-to-end delay constraint. By setting the `CXXFLAGS` in that manner, as shown in line 4 onwards of both Figure 10a and Figure 10b, the compilers correctly fail to compile as compilers but correctly succeed to answer the design question as modeling tools.

As shown in the last three lines of both figures, the compilers tell by an instance of the class template `end_to_end_delay_-is_between_min_and_max_delays` that the last-to-first end-to-end delay from h_s to L is greater than the period assigned to h_s, which is `std::ratio<5, 1000>` (i.e., 5 ms), because, along the path that goes through parameters 2 (h_s), 7 (h_f), 12 (h_h), 13 (V_z), and 15 (L) shown in Figure 8 (i.e., `tice::v1::error::Path_forming_node_pos<2, 7, 12, 13, 15>`), the data sampled by the source node (i.e., h_s) when the source node is released for the second time (i.e., the 2 in the last line) experiences a last-to-first end-to-end delay of about 60 ms (i.e., `std::ratio<60001, 1000000>`). Note that based on the time-triggered LET semantics given in Section 3.2.1, it is not hard to see that the delay is exactly 60 ms due to being a multiple of the period of the path's sink node L; the reason why Tice library reports `std::ratio<60001, 1000000>` instead of `std::ratio<60, 1000>` or `std::ratio<6, 100>` or `std::ratio<3, 50>` is to indicate that the violated bound is the upper

```
$ make -B -s CXX=/usr/bin/g++-9 CXXFLAGS='-D TICE_V1_NOGEN -D TEMPORAL_CONSTRAINTS=", \
Correlation<L, Ratio<0>, h_s, a_z_s, q_s, V_z_s>                                   \
"' 2>&1 | head -n 3
In file included from tice_model.hpp:26,
                 from main.cpp:22:
../../../tice/v1.hpp: In instantiation of ‘struct
tice::v1::error::program::correlation_is_within_threshold<false, 27, std::ratio<20000, 1000000>,
std::ratio<0>, tice::v1::error::Path_forming_node_pos<2, 7, 12, 13>,
tice::v1::error::Path_forming_node_pos<3, 8, 13>, 3>’:
```

(a) GCC version 9.1.0.

```
$ make -B -s CXX=/usr/bin/clang++-9 CXXFLAGS='-D TICE_V1_NOGEN -D TEMPORAL_CONSTRAINTS=", \
Correlation<L, Ratio<0>, h_s, a_z_s, q_s, V_z_s>                                       \
"' 2>&1 | head -n 6
In file included from main.cpp:22:
In file included from ./tice_model.hpp:26:
../../../tice/v1.hpp:2505:11: fatal error: static_assert failed "Correlation threshold is not
respected"
          static_assert(arg__is_within_threshold,
          ^             ~~~~~~~~~~~~~~~~~~~~~~~~
../../../tice/internals/v1/v1_internals_program.hpp:7744:11: note: in instantiation of template
class 'tice::v1::error::program::correlation_is_within_threshold<false, 27, std::ratio<20000,
1000000>, std::ratio<0, 1>, tice::v1::error::Path_forming_node_pos<2, 7, 12, 13>,
tice::v1::error::Path_forming_node_pos<3, 8, 13>, 3>' requested here
```

(b) Clang version 9.0.0.

FIGURE 11 Tice and off-the-shelf C++ compilers as modeling tools to figure out the correlation of all sensed data flowing to L.

bound instead of the lower bound. An instance of `end_to_end_delay_is_between_min_and_max_delays` always has as its first parameter the value `false`, as its second parameter the position of the violated constraint on the parameter list of `Program` (27 in this case), as its third parameter the constraint's lower bound (`std::ratio<0>` in Figure 10a and `std::ratio<0, 1>` in Figure 10b), and as the remaining parameters the violating end-to-end delay, the constraint's upper bound, the violating path, and the source node's violating release ordinal as already described. Based on the answer, the engineer can easily revise the upper bound of the constraint given in the `CXXFLAGS` to 60 ms (by replacing `h_s::period` by `Ratio<60, 1000>`, which while possible is not written as `Ratio<6, 100>` or `Ratio<3, 50>` to highlight by the second parameter the fact that the time unit is 1/1000 s) and recompile the software to iteratively find out whether the end-to-end delay is still within the desired range. If the compilation succeeds, it means that the last-to-first end-to-end delay along any possible end-to-end paths from h_s to L is between the constraint's lower and upper bounds.

Similarly, to answer design questions about data correlations, the engineer can constraint the relevant source and sink nodes with correlation constraints whose thresholds are zero. For example, to find out the correlation among all of the source data of the sink node L shown in Figure 8, a correlation constraint can be specified on all of the source nodes by setting the value of `TEMPORAL_CONSTRAINTS` in the `CXXFLAGS` to "`, Correlation<L, Ratio<0>, h_s, a_z_s, q_s, V_z_s>`" as shown in line 2 of both Figure 11a and Figure 11b. As before, the compilations correctly fail to compile but correctly succeed to answer the design question by an instance of the class template `correlation_is_within_threshold` as shown in the last three lines of both figures. Every instance of the template comes with seven parameters: (1) the value `false`, (2) the position of the violated constraint on the parameter list of template `Program` (27 in this example), (3) the violating correlation (20 ms in this example), (4) the constraint's threshold (0 ms in this example), (5) the first of the two meeting paths ($h_s \rightarrow h_f \rightarrow h_h \rightarrow V_z$ in this example), (6) the second of the two meeting paths ($a_{z_s} \rightarrow a_{z_f} \rightarrow V_z$ in this example), and (7) the release ordinal of the confluent node where the violation is witnessed (the third release of V_z in this example). Based on the answer, the engineer can easily revise the constraint's threshold to 20 ms and recompile the software to iteratively find out whether the data correlation is as desired. If the compilation succeeds, it means that L can process data whose ages differ by at most the constraint's correlation threshold.

While the interactive question-and-answer sessions described in the preceding two paragraphs may seem too tedious to find out some end-to-end delay or correlation whose actual value can indeed be just computed and reported right away by the off-the-shelf C++ compilers, the interactive question-and-answer sessions reflect the interactive sessions that in the traditional MBD workflow shown in Figure 1 take place not when using an off-the-shelf C++ compiler but when using a separate model-based tool (e.g., MATLAB/SIMULINK). Therefore, the main objective of the preceding two paragraphs in describing the interactive question-and-answer sessions is to show how an off-the-shelf C++ compiler can indeed be usable as a model-based tool in the proposed Tice MBD workflow shown in Figure 1. Furthermore, since C++ active libraries are Turing-complete and seamlessly

```
../../../tice/v1.hpp: In instantiation of ‘struct
tice::v1::error::program::gedf_schedulability_test_is_successful<false, std::ratio<2500001,
1000000>, std::ratio<2285715, 1000000>, std::ratio<500000, 1000000>, std::ratio<571429, 1000000>
>’:
../../../tice/v1.hpp:2518:25: error: static assertion failed: gEDF (global earliest-deadline
first) backend cannot realize the expressed Tice model because the total processor utilization is
greater than its bound and/or the processor utilization of some task exceeds the maximum bound
(Try reducing the WCETs of the function blocks first before redesigning the Tice model with
greater periods and/or less nodes)
```

(a) GCC version 9.1.0.

```
../../../tice/v1.hpp:2518:11: fatal error: static_assert failed "gEDF (global earliest-deadline
first) backend cannot realize the expressed Tice model because the total processor utilization is
greater than its bound and/or the processor utilization of some task exceeds the maximum bound
(Try reducing the WCETs of the function blocks first before redesigning the Tice model with
greater periods and/or less nodes)"
          static_assert(arg__is_successful,
          ^             ~~~~~~~~~~~~~~~~~~
../../../tice/internals/v1/v1_internals_program.hpp:7911:11: note: in instantiation of template
class 'tice::v1::error::program::gedf_schedulability_test_is_successful<false,
std::ratio<2500001, 1000000>, std::ratio<2285715, 1000000>,
std::ratio<500000, 1000000>, std::ratio<571429, 1000000> >' requested here
```

(b) Clang version 9.0.0.

FIGURE 12 Tice and off-the-shelf C++ compilers as modeling tools catch an inconsistency when integrating the generated code.

integrable with one another, it is conceivable to develop another C++ active library that extends Tice by bringing the experience of using MATLAB/SIMULINK into the proposed Tice MBD workflow.[37]

Lastly, any of the temporal constraints produced during the modeling process can be incorporated into the model by taking the constraint out of the `CXXFLAGS` and placing it just before the `TEMPORAL_CONSTRAINTS` found on the parameter list of the template `Program` shown in Figure 8. Every incorporated constraint then becomes one of the program safety properties that the compiler will enforce at compile time automatically. Coupled with C and C++ being the de facto languages in embedded software engineering, this means that Tice and an off-the-shelf C++ compiler as a modeling tool is capable of not only generating the code implementing the model but also integrating the generated code with the rest of the embedded software automatically. In this way, if some information about the other C/C++ programs is incorporated into the Tice model through their header files, the information will be synchronized and checked by the compiler automatically at compile time, ensuring the integrity of the resulting embedded software. For example, suppose the WCETs of the function blocks shown on the hardware-dependent part of Figure 9 are replaced by macros so that the WCETs can be supplied automatically by some external tool that defines the macros at compile time after analyzing the definitions of the function blocks. Then, Tice and an off-the-shelf C++ compiler as a modeling tool can raise an error when the generated code cannot be integrated with the function blocks due to their WCETs as illustrated in Figure 12.

5 | THE COMPILATION TIMES OF TICE PROGRAMS

The main objective of this section is to show the practical feasibility of the proposed Tice MBD workflow shown in Figure 1. Specifically, considering the design session described in Section 4.2 concerning the Tice model shown in Figure 8, this section shows how long it would take in the worst case to answer one design question (i.e., one compilation that ends either successfully or with an error as illustrated in Figure 10a). Therefore, this section presents in Section 5.1 the constructions of Tice models that approximate cases represented by the ROSACE case study in the aviation domain and, to some extent, the WATERS 2017 Industrial Challenge in the automotive domain. This section then presents in Section 5.2 the compilation times of GCC and Clang as popular off-the-shelf C++ compilers when they compile the constructed Tice models. The compilation times analyzed in Section 5.2.1 up to Section 5.2.8 show that the proposed Tice MBD workflow is practically feasible for Tice programs that are represented by the ROSACE case study and, to some extent, the WATERS 2017 Industrial Challenge.

5.1 | Evaluation Setup

In constructing the Tice models to be evaluated, we use the following six criteria illustrated using the ROSACE case study shown in Figure 8:

Criterion 1 The number of nodes. ROSACE has 15 nodes, while the Tice models used in the evaluations have up to 27 nodes.

Criterion 2 The number of arcs. ROSACE has 15 arcs, while the Tice models used in the evaluations involving the use of arcs have between 21 and 44 arcs, inclusive.

Criterion 3 The number of distinct end-to-end paths. ROSACE has seven distinct end-to-end paths:

$$\vec{\pi}_1 = h_s \rightarrow h_f \rightarrow h_h \rightarrow V_z \rightarrow L \qquad \vec{\pi}_5 = q_s \rightarrow q_f \rightarrow V_a \rightarrow E$$
$$\vec{\pi}_2 = a_{z_s} \rightarrow a_{z_f} \rightarrow V_z \rightarrow L \qquad \vec{\pi}_6 = V_{z_s} \rightarrow V_{z_f} \rightarrow V_a \rightarrow E$$
$$\vec{\pi}_3 = q_s \rightarrow q_f \rightarrow V_z \rightarrow L \qquad \vec{\pi}_7 = V_{a_s} \rightarrow V_{a_f} \rightarrow V_a \rightarrow E$$
$$\vec{\pi}_4 = V_{z_s} \rightarrow V_{z_f} \rightarrow V_z \rightarrow L$$

On the other hand, the Tice models used in the evaluations involving the use of arcs, except those in Section 5.2.7 and Section 5.2.8, have 32 distinct end-to-end paths, which is the maximum number of end-to-end paths attainable without forming a cycle with the least value of Criterion 2, which is 21 arcs.

Criterion 4 The length of the shortest end-to-end path. ROSACE's shortest end-to-end paths (e.g., $\vec{\pi}_2$ and $\vec{\pi}_3$) have a length of three arcs, while the Tice models used in the evaluations involving the use of arcs have up to 21 arcs.

Criterion 5 The number of distinct end-to-end path pairs that have confluent nodes. ROSACE has nine distinct end-to-end path pairs that have confluent nodes:

- Six pairs whose confluent node is V_z: (1) $\vec{\pi}_1$-$\vec{\pi}_2$ (2) $\vec{\pi}_1$-$\vec{\pi}_3$ (3) $\vec{\pi}_1$-$\vec{\pi}_4$ (4) $\vec{\pi}_2$-$\vec{\pi}_3$ (5) $\vec{\pi}_2$-$\vec{\pi}_4$ (6) $\vec{\pi}_3$-$\vec{\pi}_4$
- Three pairs whose confluent node is V_a: (1) $\vec{\pi}_5$-$\vec{\pi}_6$ (2) $\vec{\pi}_5$-$\vec{\pi}_7$ (3) $\vec{\pi}_6$-$\vec{\pi}_7$

On the other hand, the Tice models used in the evaluations involving the use of arcs have 496 distinct path pairs, except for those in Section 5.2.8 that have 861 distinct path pairs.

Criterion 6 The number of times the non-isolated node(s) with the shortest period is released in the hyperperiod (i.e., least-common multiple of the periods) of all non-isolated nodes. In ROSACE, the shortest period of the non-isolated nodes is 5 ms and the hyperperiod of all non-isolated nodes is $\text{lcm}\{5 \text{ ms}, 10 \text{ ms}, 20 \text{ ms}\} = 20$ ms so that the nodes with the shortest period is released $20 \text{ ms}/5 \text{ ms} = 4$ times in the hyperperiod. On the other hand, the Tice models used in the evaluations involving the use of distinct periods release the nodes with the shortest period up to a thousand times.

Then, with the objective of figuring out the compilation times of cases represented by the ROSACE case study and, to some extent, the WATERS 2017 Industrial Challenge, the six criteria are used to define eight sets of Tice models shown in Table 1 where:

- Every node in the compiled Tice models is assigned a WCET of one millisecond (100 μs for the models measured in Section 5.2.7 and Section 5.2.8) because assigning different WCET values to different nodes practically has no effect on the compilation times.
- Every model involving the use of arcs (i.e., $k_2 \neq 0$) is constructed in such a way so that it has exactly one source node and one sink node. Since every model involving the use of arcs, except those in Section 5.2.7 and Section 5.2.8, always has 32 distinct end-to-end paths as shown in Table 1, having exactly one source node and one sink node allows us to measure the worst-case compilation times of Tice models without having to measure every possible combination of temporal constraints that can be applied. To make our point clear, we will consider the design session described in Section 4.2 concerning the Tice model shown in Figure 8. Since the Tice model has 5 source and 2 sink nodes but not every source node is connected to every sink node, there are at most $5 \times 2 = 10$ distinct end-to-end delay constraints and $2^5 \times 2 = 64$ distinct correlation constraints that can be applied on the model. In the worst case, all of them will be tried one after

TABLE 1 Sets of Tice models to be evaluated (down a column, a dotted cell uses the text of the last non-dotted cell above it).

Evaluator	**Criterion 1 (k_1)**	**Criterion 2 (k_2)**	**Criterion 3 (k_3)**	**Criterion 4 (k_4)**	**Criterion 5 (k_5)**	**Criterion 6 (k_6)**
Section 5.2.1	$k_1 \in \{6, 9, \ldots, 27\}$	0	0	0	0	1
Section 5.2.2	$k_1 \in \mathbf{A}$	$k_2 \in \mathbf{B}$	32	$k_4 \in \mathbf{C}$	$\binom{k_3}{2} = 496$	●
Section 5.2.3	●	●	●	●	●	●
Section 5.2.4	●	●	●	●	●	●
Section 5.2.5	27	41	●	21	●	$k_6 \in \mathbf{D}$
Section 5.2.6	●	●	●	●	●	●
Section 5.2.7	$15 + 2 = 17$	$15 + 7 = 22$	7	$3 + 2 = 5$	$\binom{k_3}{2} = 21$	4
Section 5.2.8	$21 + 2 = 23$	$28 + 16 = 44$	42	$3 + 2 = 5$	$\binom{k_3}{2} = 861$	200

$\mathbf{A} = \{7\} \cup \{9, 12, \ldots, 27\}$ $\mathbf{B} = \{21\} \cup \{23, 26, \ldots, 41\}$ $\mathbf{C} = \{1\} \cup \{3, 6, \ldots, 21\}$ $\mathbf{D} = \{10, 15, 20, 40, 45, 75, 100, 1000\}$

another, and as already shown in Section 4.2, each try will result in several iterations involving the same constraint until the constraint is respected. Since this section's objective is to find out how long it would take in the worst-case to perform one iteration, this section needs to consider only the last of the iterations when the compilation is successful due to its having the longest compilation time. And, rather than having to determine which of the 10 end-to-end delay and 64 correlation constraints has the longest compilation time, this section considers the applications of either all of the 10 end-to-end delay constraints, which are evaluated in Section 5.2.3 and Section 5.2.5, or all of the 64 correlation constraints, which are evaluated in Section 5.2.4 and Section 5.2.6, by adding two nodes A and B to the Tice model so that A and B become the only source and sink nodes, respectively, by A's having outgoing arcs to all of the five existing source nodes and B's having incoming arcs from all of the two existing sink nodes. By introducing A and B, applying one end-to-end delay or one correlation constraint to A and B already overmeasures the compilation time of every individual constraint because the individual constraint applies to some path that is now the subpath of some path from A to B. The overmeasurement, however, does not invalidate this section's objective because every respected constraint is likely to be left in place while another constraint is added to answer the next design question; this at worst results in the compilation of either all of the 10 end-to-end delay or all of the 64 correlation constraints, resulting in no overmeasurement and achieving this section's objective.

- Every model involving the use of multiple periods (i.e., $k_6 \neq 1$), except those in Section 5.2.7 and Section 5.2.8, are assigned periods from some set of usable periods. There are eight sets of usable periods, three of which have members that are different in a logarithmic fashion and five of which have 2^i, 3^j, and 5^k as their members for some positive integers i, j, and k. Specifically, the three sets are $\{1, 10\}$, $\{1, 10, 100\}$, and $\{1, 10, 100, 1000\}$, resulting in $k_6 = 10$, $k_6 = 100$, and $k_6 = 1000$, respectively, and the five sets are $\{2, 3, 5\}$, $\{4, 3, 5\}$, $\{2, 9, 5\}$, $\{8, 3, 5\}$, and $\{2, 3, 25\}$, resulting in $k_6 = 15$, $k_6 = 20$, $k_6 = 45$, $k_6 = 40$, and $k_6 = 75$, respectively. A Tice model is then assigned multiple periods from one of the usable-period sets by first ordering the model's nodes in some order and then visiting the ordered nodes one after another while assigning any period that either has never been chosen or, if all has ever been chosen, is the least recently chosen from the usable-period set.

Since the Tice models used in the evaluations have nodes up to 27 nodes, which is almost twice that of ROSACE, 41 arcs at the maximum, which is almost three times greater than that of ROSACE, 32 distinct end-to-end paths, which is almost five times greater than that of ROSACE, 496 distinct path pairs that have confluent nodes, which is almost two orders of magnitude larger than that of ROSACE, and releasing the nodes with the shortest period up to thousand times, which is almost three orders of magnitude larger than that of ROSACE, the Tice models used in the evaluations should be representative to assess the practical feasibility of Tice MBD workflow involving Tice programs that are represented by the ROSACE case study and, to some extent, the WATERS 2017 Industrial Challenge, which has slightly more arcs at 44, a third more distinct end-to-end paths at 42, and two times more distinct path pairs that have confluent nodes at 861.

The evaluations reported in this section were carried out on a Lenovo E40-80 laptop that came with an Intel Core i3-5010U processor, which has two physical 64-bit 2.1-GHz cores with two threads per core to have four logical processor cores in total, and 2 GiB of DDR3 RAM that had subsequently been expanded to 10 GiB by installing an additional 8-GiB memory module.

The computer's operating system was the desktop version of Ubuntu 16.04.6 whose package manager was used to install the latest GCC from http://ppa.launchpad.net/ubuntu-toolchain-r/test/ubuntu and the latest Clang from http://apt.llvm.org/xenial. The evaluations measure compilation times by executing GCC/Clang using the Bash shell's built-in command `time` so that the sum of the reported user and system times can be taken as the measured compilation times. To report the compilation times, every evaluation is repeated ten times so that the average and standard deviation of the measured compilation times can be reported. The reported compilation times are successful compilation times where the test programs raised no compile-time error. We do not report the compilation times where the test programs raised compile-time errors because the compilation times were less than those of the successful ones. Additionally, no swapping of data between the main memory and disk was observed during measurements except in the first out of ten executions of GCC for the evaluation that is reported in Figure 18 at the right-most tick, which swapped out about 350 MiB from the main memory to disk without swapping them back in (before any evaluation commenced, the main memory held about 900 MiB, while the disk held no swapped-out memory). The complete evaluation software is available in directory `experiments/v1-performance` in the Tice repository.[36] The software is to be checked out at commit `c795b273dd` to run the second sub-experiment using the latest commit of the test files whose IDs are between 100 and 199, inclusive (i.e., after cloning the Tice repository and entering the cloned directory tree at its root, the whole tree at commit `c795b273dd` is first checked out before then checking out only the files that match the pattern `experiments/v1-performance/*.txt` at commit `master`, after which the second sub-experiment can be run on all of the files by the Bash shell command: `./run --back-end=0,1,2,3:1[0-5]?=1/1000,1[67]?=1/10000 /usr/bin/g++-9 /usr/bin/clang++-9`).

5.2 | Evaluations

For every evaluation described below, the compilation times are measured on two different cases. In the first case, target hardware that has no processor core is used as in line 42 of Figure 9 so that the compilers only perform model analyses, neither generating the code implementing the analyzed models nor integrating the generated code. The compilation times measured in the first case will therefore show the practical feasibility of using Tice and an off-the-shelf C++ compiler as a modeling tool. In the second case, target hardware that has four processor cores is used as in lines 44–45 of Figure 9 so that the compilers not only perform model analyses but also generate the code implementing the analyzed models and integrate the generated code with the other C++ software components to produce the complete executables. The compilation times measured in the second case will therefore show the compilation times of Tice as a real-time language. To make the compilation times measured in the two cases comparable, the macro `TICE_V1_NOGEN` described in Section 4.2 is not defined so that the compilers process the same amount of source code as no part is excised by the preprocessors. Beside that, no optimization option (e.g., `-O2`) is used as it may incur extra compilation time that has nothing to do with the analysis and the code generation and integration of Tice models. Additionally, the special file `/dev/null` is used as the output file because using an actual file will cause the measured compilation times to include factors that deal with the output process itself that can be improved or made worse by decisions external to Tice library and off-the-shelf compilers, such as using a fast solid-state drive or slow hard disk drive.

Lastly, to ease presentation, we will call the part of Tice library that analyzes Tice models as Tice front-end and the part of Tice library that generates code implementing the Tice models and integrates the generated code with other C/C++ software components as Tice back-end. While the first case of compilations, whose target hardware has no processor core, uses Tice front-end exclusively, the second case of compilations, whose target hardware has four processor cores, uses both Tice front-end and Tice back-end because Tice back-end can only work on valid Tice models, requiring Tice front-end to validate the models first. The evaluation results can then be categorized based on whether they are of interest from the perspective of using an off-the-shelf C++ compiler as a modeling tool as already demonstrated in Section 4.2. If the evaluation results are of interest, then only Tice front-end compilation times matter. Otherwise, both Tice front-end and back-end compilation times matter. The categorization as well as the compilation use-case, the relevant temporal constraint, and the varied criterion of every evaluation result are given in Table 2 to help locate the evaluation results that are needed to estimate the compilation time of a given Tice model. For example, considering the ROSACE program shown in Figure 8 whose Tice model has $k_1 = 15$, $k_2 = 15$, $k_3 = 7$, $k_4 = 3$, $k_5 = 9$, and $k_6 = 4$ with $k_1, k_2, \ldots$, and k_6 being already defined in Table 1, the model's compilation shown in Figure 10a, which applies one end-to-end delay constraint and uses GCC 9.1.0 to answer a design question (i.e., only Tice front-end compilation time is of interest), can be estimated to take at most 3.884 s because, according to Table 2, Figure 19 is the most relevant to estimate the compilation time needed to answer a design question involving the use of *any number* of end-to-end delay constraints for the given criteria $k_1, k_2, \cdots, k_6$. A similar procedure can be applied on the compilation shown in Figure 11a to estimate that its

TABLE 2 Evaluation result characteristics (down a column, a dotted cell uses the text of the last non-dotted cell above it).

Evaluation Result	Relevant if Using C++ Compiler as Modeling Tool?	Compilation Use-Case	Temporal Constraint	Varied Criterion (see Table 1)
Figure 13	No	Independent periodic real-time tasks	None	Criterion 1
Figure 14	●	LET computation & communication	●	Criterion 2
Figure 15	Yes	●	End-to-end delay	Criterion 4
Figure 16	●	●	Correlation	●
Figure 17	●	●	End-to-end delay	Criterion 6
Figure 18	●	●	Correlation	●
Figure 19	●	●	None/either/both	Temporal constraints
Figure 21	●	●	●	●

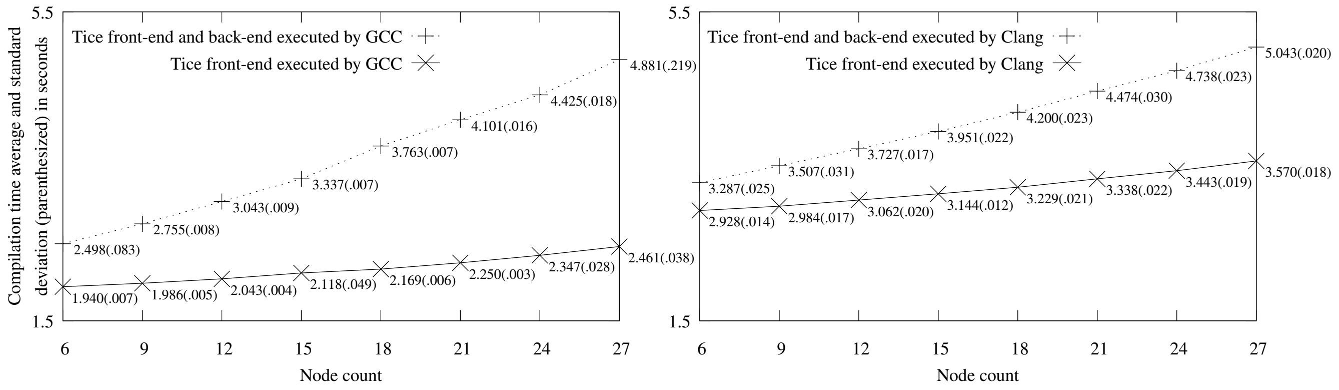


FIGURE 13 Compilation times of Tice models that have only isolated nodes.

compilation time is at most 4.312 s. If the Tice model uses both end-to-end delay and correlation constraints in *any number*, then Figure 19 estimates that the compilation time is at most 4.439 s.

5.2.1 | Isolated Nodes Only

In this section, we report the compilation times of Tice models whose DAGs have only isolated nodes (i.e., no arcs and no temporal constraints). While Tice programs expressing only isolated nodes might not be interesting from the perspective of a modeling tool, they are useful from the perspective of a real-time language to implement periodic real-time tasks. As shown in Figure 13, we evaluated the compilation times of GCC and Clang when they compiled Tice models whose numbers of isolated nodes are between 6 and 27, inclusive, in an increment of 3 nodes, the number of which is chosen to have eight data points to make their plotted graph legible. As Tice programs expressing only isolated nodes are likely more useful to produce executables than to analyze Tice models, the focus in Figure 13 should be on the compilation times measured in the second case, which executes both Tice front-end and Tice back-end.

It can be concluded from Figure 13 that the compilation times of Tice programs expressing independent periodic real-time tasks would be practically reasonable as the largest Tice model with 27 of such tasks could be compiled into an executable virtually under 5 seconds. Other than that, GCC produced executables faster than Clang for smaller Tice programs. Hence, it seems that GCC is specially tuned to compile small C++ programs in general.

5.2.2 | Data Flows without Temporal Constraints

In this section, we report the compilation times of Tice models whose DAGs have arcs. As in the preceding section, while Tice programs expressing data flows are indeed interesting from the perspective of a real-time language, it might not be the case from

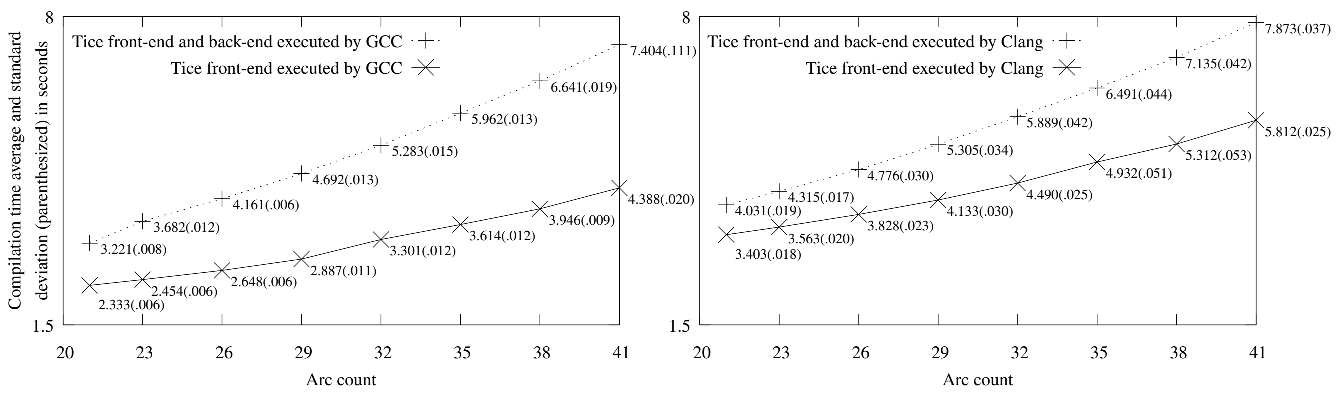


FIGURE 14 Compilation times of Tice models that have arcs but no temporal constraint.

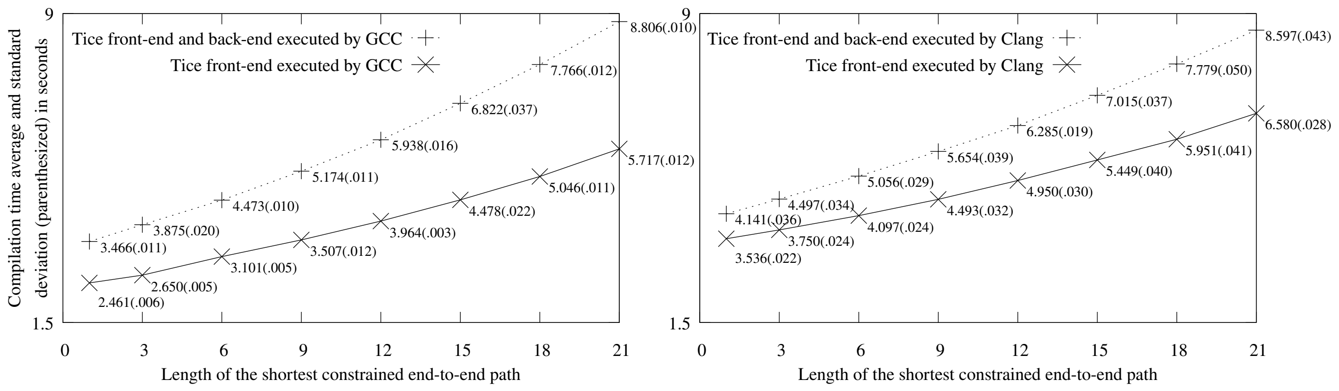


FIGURE 15 Compilation times of Tice models that have exactly one end-to-end delay constraint.

the perspective of a modeling tool because nothing is asked about the data flows. Hence, as before, the focus in Figure 14 should be on the compilation times measured in the second case, which executes both Tice front-end and Tice back-end.

The Tice models evaluated in this section were constructed in such a way so that every Tice model has exactly one source node, one sink node, 32 distinct end-to-end paths, and 496 path pairs having confluent nodes but a distinct shortest end-to-end path length that is varied from one model to the next starting from 1 and then continuing with 3 up to 21, inclusive, in a 3-arc increment, which also varies the number of nodes starting from 7 and then continuing with 9 up to 27, inclusive, in a 3-node increment. Therefore, the Tice models have between 21 and 41 arcs, inclusive, specifically one Tice model with 21 arcs and seven Tice models with 23 up to 41 arcs in a 3-arc increment, respectively.

It can be concluded from Figure 14 that the compilation times of Tice programs expressing real-time data-flow LET-based processing would be practically reasonable as the largest Tice model with 41 arcs could be compiled into an executable under 8 seconds. As before, GCC produced executables faster than Clang for smaller Tice programs.

5.2.3 | One Last-to-First End-to-End Delay Constraint

In this section, we report the compilation times of Tice models that have exactly one last-to-first end-to-end delay constraint. In this report, the Tice models constructed for the previous section's report were reused by attaching one end-to-end delay constraint to the only available source and sink nodes. In contrast to the two preceding sections, without losing their appeal from the perspective of a real-time language, the Tice programs are now interesting from the perspective of a modeling tool as demonstrated in Section 4.2. Hence, the focus in Figure 15 should be on the compilation times measured in the first case, which exclusively executes Tice front-end.

As explained in the previous section, the Tice models have 21 arcs at the minimum and a fixed number of end-to-end paths, namely 32 paths. While the number of end-to-end paths is almost five times greater than that of the ROSACE case study, the number is the maximum number of end-to-end paths attainable with 21 arcs without forming a cycle so that different data-flow

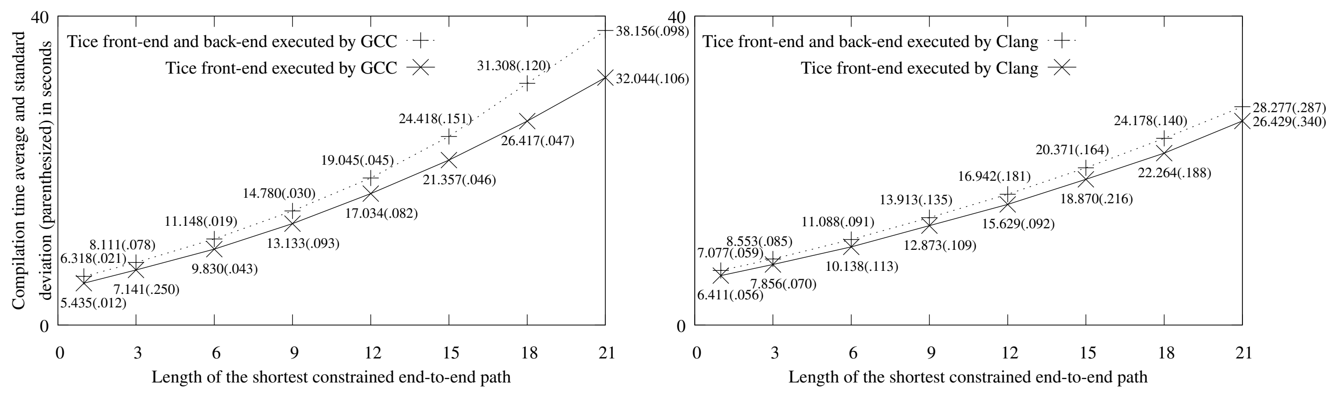


FIGURE 16 Compilation times of Tice models that have exactly one correlation constraint.

configurations could be tried iteratively, the worst of which we have anticipated to have 32 end-to-end paths. In Figure 15, the length of the shortest end-to-end path is varied because, based on Figure 7 and (16), the analysis of an end-to-end delay constraint should grow proportional to the length of a constrained end-to-end path.

It can be concluded from Figure 15 that the analyses of this kind of Tice models would be practically reasonable as the largest Tice model with 32 end-to-end paths of average length 21 arcs could be analyzed under 7 seconds. While 7 seconds might seem a bit too long for MATLAB/SIMULINK users, recall that it is the anticipated worst-case time needed to analyze a Tice model during modeling; the usual analysis time of the cases that are represented by the ROSACE case study could be around 4 seconds as shown in Figure 19. Furthermore, as the demand of using off-the-shelf C++ compilers as modeling tools grows,[37,38] the usual analysis time would get even shorter, matching the usual time experienced by MATLAB/SIMULINK users. Additionally, the code generation and integration of Tice models would also be practically reasonable as the largest Tice model could be compiled into an executable under 9 seconds. As before, GCC produced executables faster than Clang for smaller Tice programs. However, GCC was consistently faster than Clang when the compilers were used as modeling tools.

5.2.4 | One Correlation Constraint

In this section, we report the compilation times of Tice models that have exactly one correlation constraint. In this report, the Tice models constructed for the previous section's report were reused by replacing the end-to-end delay constraint with a correlation constraint. As in the previous section, the Tice programs are interesting from both the perspectives of a modeling tool and a real-time language. As before, the focus in Figure 16 should be on the compilation times measured in the first case, which exclusively executes Tice front-end.

As explained in the previous section, the Tice models have 21 arcs at the minimum, 32 possible end-to-end paths, and as explained in Section 5.2.2, 496 path pairs that have confluent nodes. While the number of path pairs is more than an order of magnitude greater than that of the ROSACE case study, which has 9 path pairs, as in the previous section, the use-case of Tice and an off-the-shelf C++ compiler as a modeling tool requires us to anticipate the worst possible data-flow configuration asked.

As in the preceding section, it can be concluded from Figure 16 that the analyses of this kind of Tice models would still be practically reasonable as a modeling tool because Figure 16 reports the anticipated worst cases. The usual cases represented by the ROSACE case study could be an order of magnitude faster around 4 seconds as shown in Figure 19. Furthermore, Figure 16 confirms the observations made in the preceding sections: GCC is optimized to compile small C++ programs. Additionally, Figure 16 shows that Clang is faster than GCC at analyzing large Tice models. Lastly, the times needed to produce the executables are still practically reasonable as most of the compilation times were spent at analyzing Tice models. The difference between the upper and lower curves shows that the time needed to generate the code implementing the models and to integrate the generated code to produce the executables were under 9 seconds.

5.2.5 | One Last-to-First End-to-End Delay Constraint and Multiple Periods

In this section, eight Tice models were constructed in the same way the largest Tice model in Section 5.2.3 was constructed. The nodes of the Tice models, however, were assigned different periods as described in Section 5.1. Specifically, each of the eight Tice models has its nodes assigned periods from exactly one of the eight sets of usable periods in the manner already described

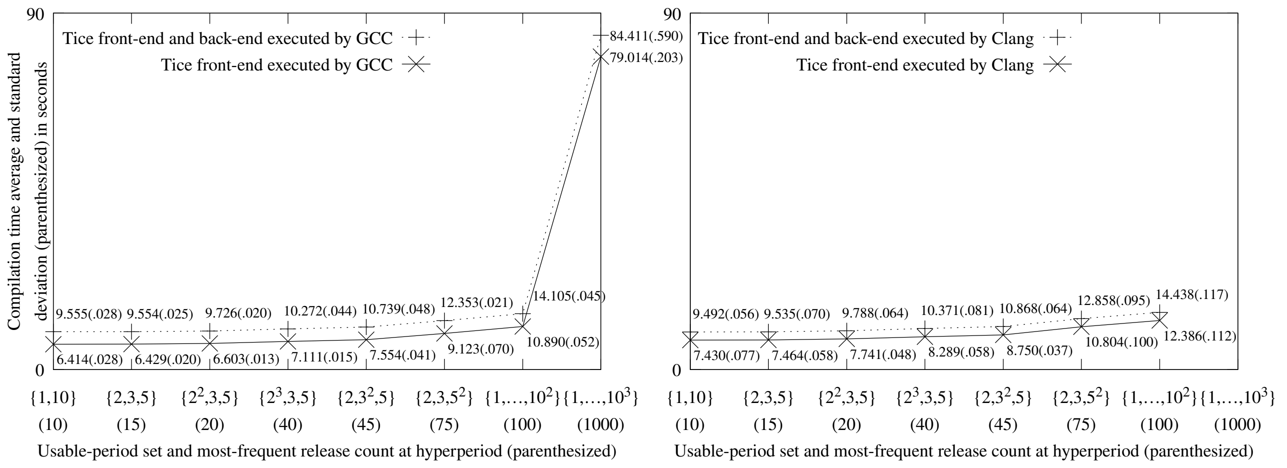


FIGURE 17 Compilation times of multi-periodic Tice models with exactly one end-to-end delay constraint.

in Section 5.1. While the Tice programs are interesting from both the perspectives of a modeling tool and a real-time language, contrary to the previous sections, the focus in Figure 17 should be on the compilation times measured in both the first and the second cases because the node periods affect both the analysis effort and the code generation and integration effort.

It can be concluded from Figure 17 that the analyses of multi-periodic Tice models would still be practically reasonable because Figure 17 reports the anticipated worst cases. This means that for the cases represented by the ROSACE case study, the required analysis time could be around 4 seconds as shown in Figure 19 because Figure 17 reports the largest Tice model evaluated in Figure 15. Beside that, Figure 17 shows that the analysis times grow proportional to the most-frequent release count at hyperperiod. Furthermore, Figure 17 shows that assigning different periods to Tice nodes has no visible impact on the time needed to produce the executables as the difference between the upper and lower curves stays virtually constant. Lastly, in this multi-periodic evaluations, GCC performed better than Clang. First, GCC compiled faster than Clang as shown in Figure 17. Secondly, GCC could compile the largest multi-periodic Tice model successfully, while Clang crashed with a segmentation fault error. Consequently, the right part of Figure 17 shows no data for the right-most tick.

5.2.6 | One Correlation Constraint and Multiple Periods

This section repeats the evaluations reported in the previous section but constructing the eight Tice models in the same way the largest Tice model in Section 5.2.4 was constructed as well as replacing the end-to-end delay constraints with correlation constraints. As before, the Tice programs are interesting from both the perspectives of a modeling tool and a real-time language, and the focus in Figure 18 should be on the compilation times measured in both the first and the second cases because the node periods affect both the analysis effort and the code generation and integration effort.

It can be concluded from Figure 18 that the analyses of this kind of multi-periodic Tice models would be practically reasonable despite the large compilation times reported in Figure 18. As already explained in Section 5.2.5 and Section 5.2.4, based on the ROSACE case study and keeping in mind that Figure 18 reports the largest Tice model evaluated in Figure 16, the compilation times of the cases represented by the ROSACE case study could be around 4 seconds as shown in Figure 19. As concluded in the preceding section, Figure 18 shows that the most-frequent release count at hyperperiod is strongly correlated with the analysis times but not with the executable-producing times. However, while Clang crashed with a segmentation fault error when analyzing the largest multi-periodic Tice model reported in the preceding section, Clang successfully compiled the model when the end-to-end delay constraint was replaced with a correlation constraint. Furthermore, Clang compiled significantly faster than GCC. Hence, Clang is better suited than GCC to analyze multi-periodic Tice models with correlation constraints that involve a large number of confluent path pairs. On the other hand, based on the previous section's conclusion, GCC is better suited than Clang to analyze multi-periodic Tice models with end-to-end delay constraints that involve a large number of long end-to-end paths.

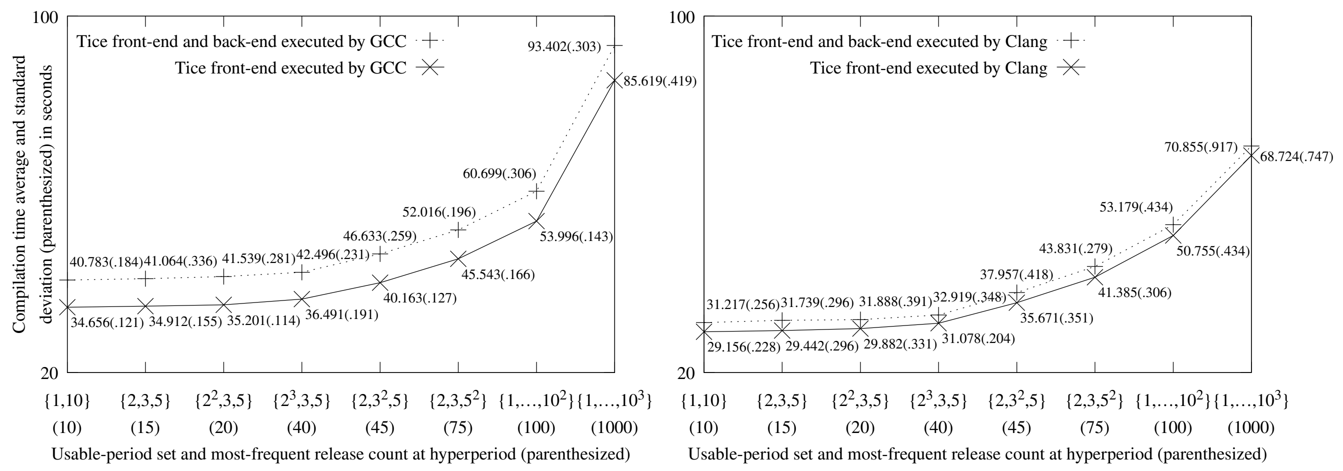


FIGURE 18 Compilation times of multi-periodic Tice models with exactly one correlation constraint.

5.2.7 | ROSACE (Multiple Periods) with None/Either/Both Constraints

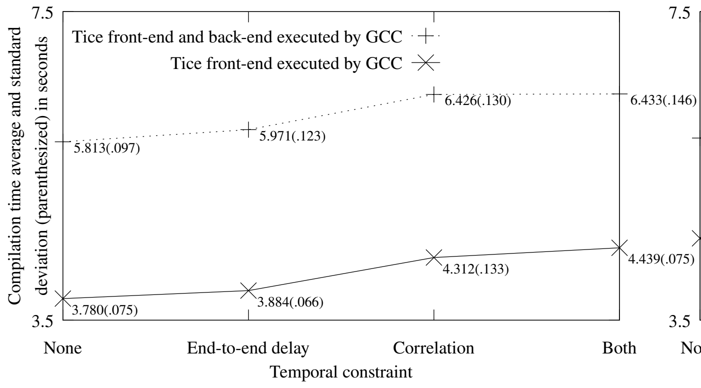
While the preceding evaluations have focused on evaluating the worst-case compilation scenarios for the cases that are represented by the ROSACE case study, this section focuses on evaluating the worst-case compilation scenarios of the ROSACE case study itself, specifically the compilations of the Tice program shown in Figure 8. To do so, as already explained in Section 5.1, the Tice models used in this evaluation is the Tice model shown in Figure 8 that is enlarged by adding two new nodes that become the only source and sink nodes on which the temporal constraints are applied. Consequently, beside adding two nodes to the Tice model shown in Figure 8, the evaluated Tice models also add seven arcs, lengthen the shortest end-to-end path by two arcs, and enlarge the number of distinct end-to-end path pairs that have confluent nodes by twelve path pairs. However, by assigning a period of 5 ms to the two new nodes, the evaluated Tice models keep the same number of times the non-isolated nodes with the shortest period is released in the hyperperiod of all non-isolated nodes.

It can be concluded from Figure 19 that the analyses of Tice programs whose Criterion 1 up to 6 as shown in Table 1 are very close to the criteria of the ROSACE Tice program shown in Figure 8 would be very reasonable in practice. Furthermore, the analyses of last-to-first end-to-end delay constraints is significantly less costly than the analyses of correlation constraints ($3.884 - 3.780 = 0.104$ s vs. $4.312 - 3.780 = 0.532$ s for GCC and $4.667 - 4.561 = 0.106$ s vs. $5.113 - 4.561 = 0.552$ s for Clang), which is not surprising because as described in Section 3.2.2 the analyses of end-to-end delay constraints consider only one end-to-end path at a time while the analyses of correlation constraints consider not only two end-to-end paths but also each of their confluent nodes at a time. Furthermore, using both end-to-end delay and correlation constraints result in compilation times that are longer by the sum of the time taken to analyze each kind of the constraints ($3.780 + 0.104 + 0.532 = 4.416$ s, which is almost 4.439 s reported for GCC, and $4.561 + 0.106 + 0.552 = 5.219$ s, which is slightly more than 5.204 s reported for Clang). That is, the two kinds of the constraints are computed by Tice library independently. Additionally, as the gap between the upper and lower curves stays virtually the same but shrinks when both kinds of constraints are used together, it shows that Tice library indeed consists of two independent parts: Tice front-end, which analyzes a given Tice model, and Tice back-end, which implements a valid Tice model. Lastly, since the upper curves of both GCC and Clang plots are virtually the same in their shapes and locations but GCC's lower curve is even lower than Clang's lower curve, it means that GCC's C++ front-end is likely about 20% faster than Clang's in answering various design questions asked on the Tice program shown in Figure 8.

5.2.8 | WATERS 2017 (Multiple Periods) with None/Either/Both Constraints

Considering that the authors of the ROSACE case study in proposing a framework for automatic multicore code generation[39] measured the processing time of the framework on not only the ROSACE case study in the aviation domain but also the WATERS 2017 Industrial Challenge in the automotive domain,[40] this section also evaluates the worst-case compilation scenarios of the WATERS 2017 Industrial Challenge.

Unlike the ROSACE case study that comes complete with the source of the C functions, the WATERS 2017 Industrial Challenge does not come with the source of the AUTOSAR runnables, which are analogous to C/C++ functions. Instead, the challenge



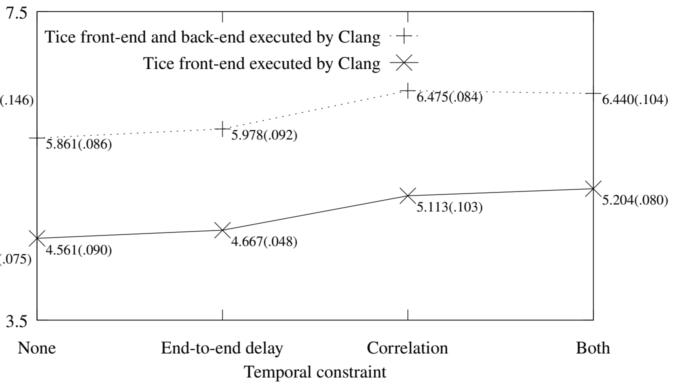


FIGURE 19 Compilation times of multi-periodic ROSACE Tice models with different temporal constraints.

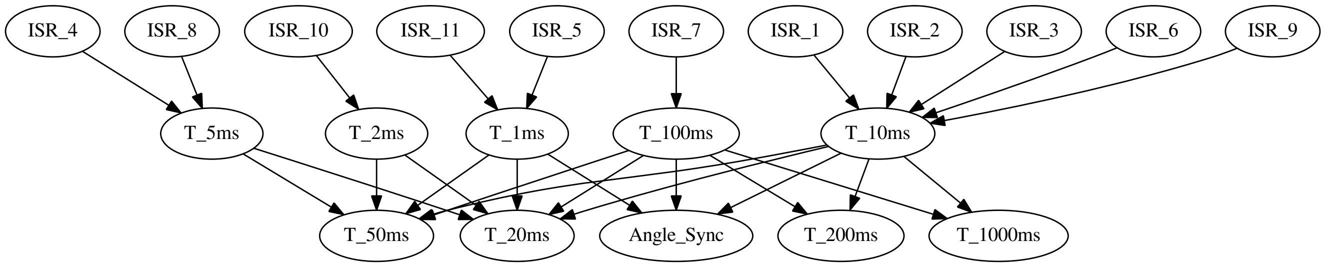


FIGURE 20 The result of running Algorithm 1 on the cyclic graph presented by Table 3.

only describes the software architecture, namely the runnables, their directed communication, and more importantly their grouping into tasks that are required to communicate using time-triggered LET and scheduled using a fixed-priority scheduling policy on a multicore processor. The architecture description is given in a zipped XML file that can be downloaded by first going to http://waters2017.inria.fr/program/ and then selecting the link titled "Presentation of the 2017 Industrial Challenge" before lastly downloading the file by the link titled "Bosch_ChallengeModel_w_LabelStats_fixedLabelMapping_App4mc_v072.zip". Since the communication among runnables in the same task is required to use the semantics of AUTOSAR implicit communication, which is different from the semantics of time-triggered LET communication, while the communication among runnables in two different tasks is required to use the semantics of time-triggered LET,[40] the Tice models evaluated in this section use only the tasks of the runnables as their Tice nodes.

Analysis of the XML file shows that there are 21 tasks, which are taken as distinct Tice nodes. While 11 out of the 21 tasks are source nodes because their runnables do not read data from the runnables of other tasks, the communication among the remaining 10 nodes is not acyclic. This means that, while there are 81 distinct arcs that include the 11 arcs associated with the 11 source nodes as shown in Table 3, some of the 70 arcs need to be removed to have a DAG and consequently a valid Tice model. To do so, Algorithm 1 is run on the cyclic graph to obtain a DAG with 28 arcs, 5 sink nodes, the longest end-to-end paths being 3-arc long, and 42 distinct end-to-end paths as shown in Figure 20. Therefore, in comparison with the DAG of ROSACE shown in Figure 8, the DAG of ROSACE is taller by one arc in terms of the longest end-to-end path, but the DAG of WATERS 2017 has more branches and confluent nodes. Furthermore, WATERS 2017 has a greater number of distinct periods assigned to the DAG nodes.

Following the construction strategy described in Section 5.1 to evaluate the worst-case compilation scenarios, as shown in Table 1, with respect to the DAG shown in Figure 20, the Tice models evaluated in this section has 2 more nodes (the single source and sink nodes) with a period of 1 ms, 16 more arcs (the arcs from the new single source node to the 11 former source nodes and the arcs from the 5 former sink nodes to the new single sink node), the longest end-to-end paths being longer by 2 arcs, and the number of times the non-isolated nodes with the shortest period is released in the hyperperiod of all non-isolated nodes is capped at 200 times so as to avoid any swap to disk. To cap Criterion 6 shown in Table 1 at 200 times, the periods of the first five tasks shown in Table 3 are set to 1 ms, the sixth and eighth to 5 ms, the ninth up to the eleventh to 10 ms, `Angle_Sync`

TABLE 3 The 81 arcs found in the WATERS 2017 Industrial Challenge (tasks are ordered by their priorities, and task periods are either stated in the names or in parentheses with the 11 source nodes being given their minimum interarrival times).

From \ To	T_1ms	Angle_Sync	T_2ms	T_5ms	T_10ms	T_20ms	T_50ms	T_100ms	T_200ms	T_1000ms	Out-Degree
1) ISR_10 (0.7 ms)			●								1
2) ISR_5 (0.9 ms)	●										1
3) ISR_6 (1.1 ms)					●						1
4) ISR_4 (1.5 ms)				●							1
5) ISR_8 (1.7 ms)				●							1
6) ISR_7 (4.9 ms)								●			1
7) ISR_11 (5 ms)	●										1
8) ISR_9 (6 ms)					●						1
9) ISR_1 (9.5 ms)					●						1
10) ISR_2 (9.5 ms)					●						1
11) ISR_3 (9.5 ms)					●						1
12) T_1ms		●			●	●	●	●			5
13) Angle_Sync (6.66 ms)	●		●	●	●	●	●	●	●	●	9
14) T_2ms	●				●	●	●	●			5
15) T_5ms	●				●	●	●	●			5
16) T_10ms	●	●	●	●		●	●	●	●	●	9
17) T_20ms	●	●			●		●	●	●	●	7
18) T_50ms	●	●			●	●		●	●	●	7
19) T_100ms	●	●	●	●	●	●	●		●	●	9
20) T_200ms	●	●			●	●	●	●		●	7
21) T_1000ms	●	●			●	●	●	●	●		7
In-Degree	11	7	4	5	14	9	9	10	6	6	**Total: 81 arcs**

Algorithm 1 Turning the cyclic graph of WATERS 2017 into a DAG to make for a valid Tice model.

```
$\mathbb{E} \leftarrow \emptyset, \mathbb{V}_c \leftarrow \emptyset$
**for** $v \in \mathbb{V}$ **do**
    **if** $v.parents = \emptyset$ **then**
        $\mathbb{V}_c \leftarrow \mathbb{V}_c \cup \{v\}$
    **end if**
**end for**
**while** $\mathbb{V}_c \neq \emptyset$ **do**
    $\mathbb{V}_n \leftarrow \emptyset$
    **for** $v_c \in \mathbb{V}_c$ **do**
        **for** $v'_c \in v_c\,.children$ **do**
            $\mathbb{E} \leftarrow \mathbb{E} \cup \left\{\left(v_c, v'_c\right)\right\}, \mathbb{V}_n \leftarrow \mathbb{V}_n \cup \left\{v'_c\right\}$
        **end for**
    **end for**
    **for** $v_n \in \mathbb{V}_n$ **do**
        **for** $v'_n \in \mathbb{V}_n\,.parents$ **do**
            $v'_n\,.children \leftarrow v'_n\,.children \setminus \left\{v_n\right\}$
        **end for**
    **end for**
    $\mathbb{V}_c \leftarrow \mathbb{V}_n$
**end while**
```

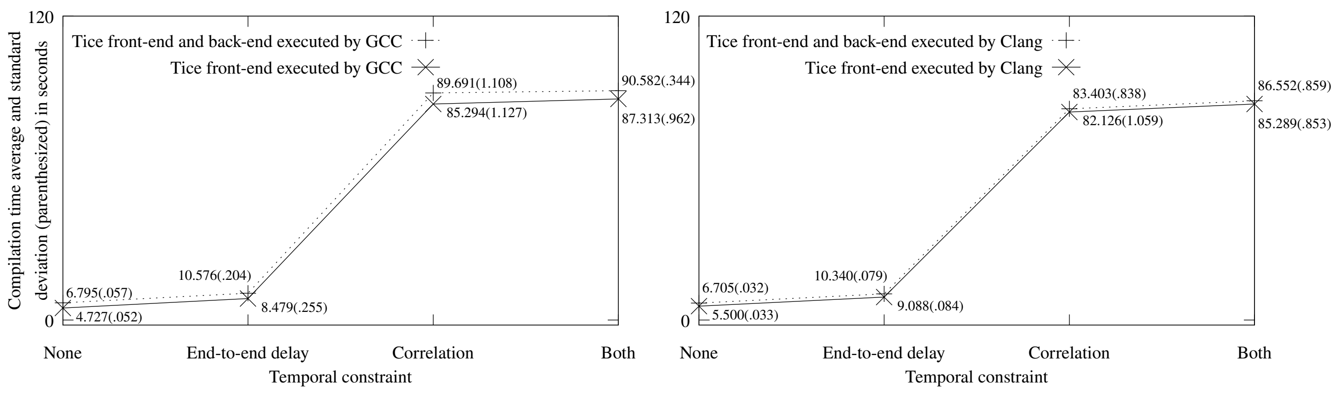


FIGURE 21 Compilation times of multi-periodic WATERS 2017 Tice models with different temporal constraints.

to 5 ms, and `T_1000ms` is set to 200 ms. The measured compilation times are then shown in Figure 21 and analyzed below (note that every subsection in Section 5.2 measures the worst-cast scenarios as explained in Section 5.1):

- When there is no temporal constraint, Figure 21 agrees with Figure 14, which also shows the evaluations of Tice models with arcs but no temporal constraints, and therefore, the same analyses and conclusions apply, mainly that the compilation times of Tice models similar to the ones evaluated in this section would be practically reasonable.

- When there is *any number* of last-to-first end-to-end delay constraints, Figure 21 when compared with Figure 15 shows that the application of any number of last-to-first end-to-end delay constraints in Tice models that are similar to the ones evaluated in this section would still be practically reasonable because, despite Criterion 6 of the Tice models evaluated in this section being 200, their compilation times were just about 3 seconds longer than those shown in Figure 15 and about 3 seconds shorter than those shown in Figure 17.

- When there is *any number* of correlation constraints, Figure 21 agrees with Figure 18, which also shows the evaluations of multi-periodic Tice models with correlation constraints. In particular, Figure 21 shows compilation times that are longer than those predicted by Figure 18 because, despite Criterion 6 being 200 for the evaluations in Figure 21, the number of distinct path pairs (i.e., Criterion 5) evaluated in Figure 21 is about 74% larger than that evaluated in Figure 18. While the compilation times shown in Figure 21 are not practical to use off-the-shelf C++ compilers as modeling tools to analyze the correlation constraints of Tice models that are similar to the ones evaluated in this section, the WATERS 2017 Industrial Challenge gives the hint that in the application domain of the challenge, which is the automotive domain, correlation constraints would not be used as often as last-to-first end-to-end delay constraints in practice. In other words, Figure 21 shows the worst-case compilation scenarios in the automotive domain with respect to the use of correlation constraints in Tice models where the usual cases, which mostly use end-to-end delay constraints, could have practically reasonable compilation times.

- When both last-to-first end-to-end delay and correlation constraints are present in *any number*, Figure 21 agrees with Figure 19 in showing that the analyses of last-to-first end-to-end delay constraints is significantly less costly than the analyses of correlation constraints.

To conclude, while Figure 21 has been analyzed with the conclusions that the compilation times of Tice models that are similar to the ones evaluated in this section would be reasonable in practice, the Tice models evaluated in Figure 21 have their Criterion 6 capped at 200 times by modifying the actual periods and interarrival times actually used in the WATERS 2017 Industrial Challenge, which are shown in Table 3. In other words, the actual periods and interarrival times used in the WATERS 2017 Industrial Challenge have shown that the internals of Tice library[41] should be further improved, in particular by using different techniques to decrease its memory consumption when analyzing temporal constraints, for example, by implementing the analyses using constexpr-function iterations instead of class-template recursions.

6 | THE GENERATION OF CODE TO IMPLEMENT TICE PROGRAMS

Once the Tice model expressed in a Tice program has been validated by an off-the-shelf C++ compiler as directed by Tice library, which is the C++ active library that implements Tice, Tice library will direct the C++ compiler to generate the code that implements the model according to the MoCC in Section 3.2.1 on the specified execution platform. The execution platform as described in Section 3.1 is specified by an instance of template `HW` that is given as the first parameter of the `Program` instance.

In case the specified execution platform is the special platform that has no processor, no implementing code will be generated. Otherwise, the implementing code will be generated by first mapping the expressed set of nodes to a set of RT (real-time) tasksets and their scheduling and partitioning policies. If the mapping fails due to not finding an RT taskset or a scheduling or partitioning policy to realize the expressed model's MoCC on the execution platform, Tice library will raise a compile-time error. Otherwise, Tice library will generate the code that implements the RT taskset and its scheduling and partitioning policies for the execution platform.

While Tice library can in fact generate code for various execution platforms, which can be selected by defining some pre-processing macro, currently Tice library generates code only for an execution platform that provides API for pthreads (i.e., POSIX threads)[10] and their scheduling using the `SCHED_DEADLINE` policy,[42] which uses the gEDF (global earliest-deadline first) scheduling algorithm. Since the communication code generated by Tice already implements the LET communication semantics as explained in Section 6.2, other than the aforementioned platform (e.g., a GNU/Linux operating system), no additional middleware layer is needed to enforce the LET communication semantics.

6.1 | The Generated Computation Code

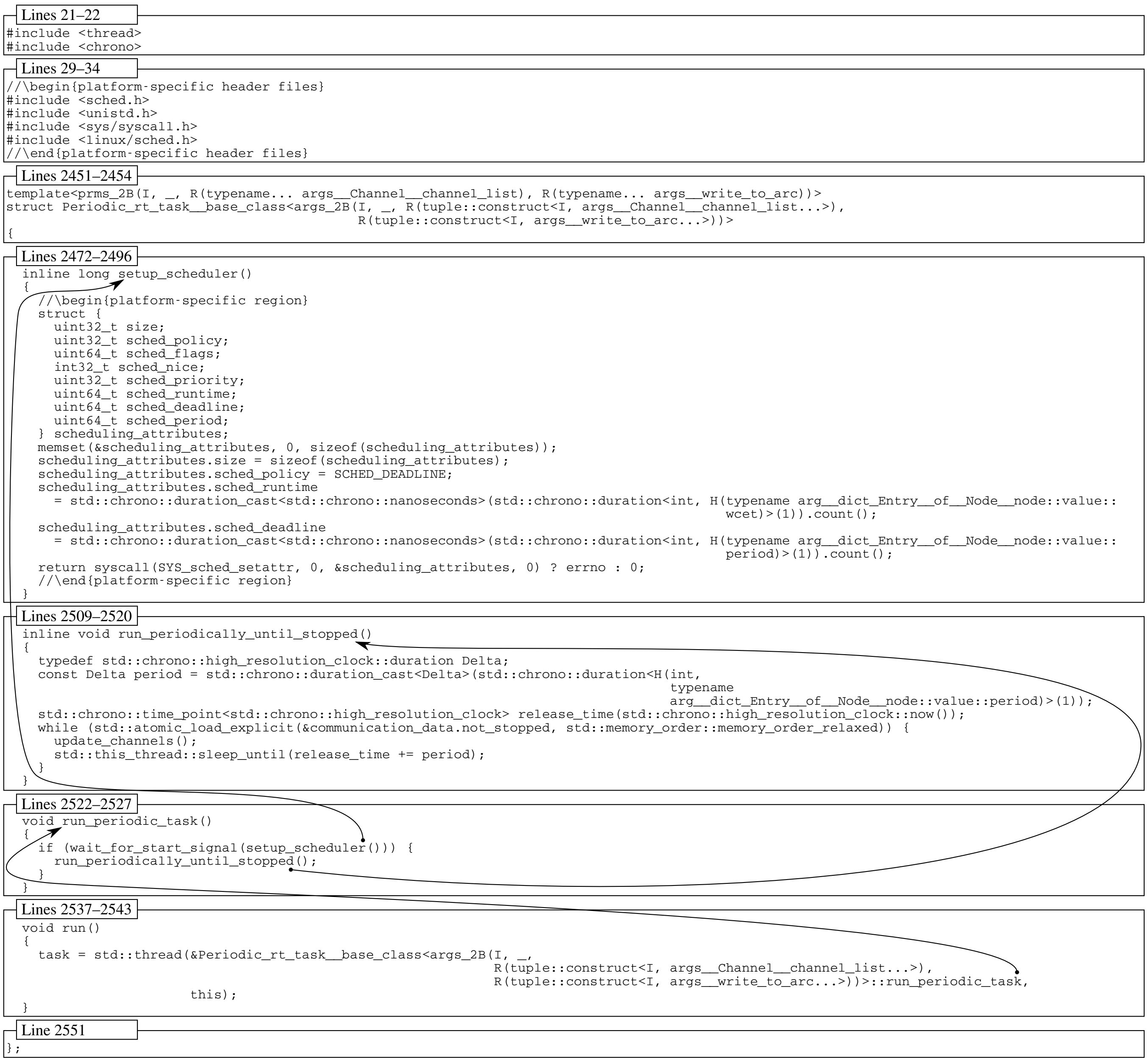
While Tice library can indeed use various strategies to map the expressed nodes to some suitable RT taskset and scheduling and partitioning policies (e.g., by requiring more information to be specified on the parameter list of the `HW` instance beside processor core IDs), currently Tice library maps every node in the expressed Tice model to a unique RT task to be scheduled using gEDF, which does not partition the resulting RT taskset.[43]

The mapping makes every member τ of the constructed RT taskset Γ to have a period P_τ and WCET C_τ that are equal to those of the corresponding node. And, letting k be the number of core IDs specified using template `Core_ids`, the mapping is successful if and only if: (1) the utilization of the RT taskset (i.e., $\sum_{\tau\in\Gamma} C_\tau/P_\tau$) is not greater than $k^2/(2k-1)$, and (2) the utilization of every member of the RT taskset (i.e., C_τ/P_τ) is not greater than $k/(2k-1)$, both of which are the tightest sufficient conditions for gEDF schedulability.[43] A compile-time error will be raised if the mapping fails; otherwise, every RT task is implemented as a C++ thread as shown in lines 2537–2543 of Figure 22.

Referring to Figure 22, the implementing threads are scheduled using the `SCHED_DEADLINE` policy with the pthread's scheduling attributes `sched_runtime`, `sched_deadline`, and `sched_period` being set to the RT task's WCET, the RT task's period, and zero to mean being equal to the preceding attribute,[44] respectively, as shown in lines 2472–2496 (the C++ thread standard library provides no API to configure the C++ thread scheduling policy, and therefore, Tice library directly uses the underlying platform API). Each of the threads performs a loop that invokes the C++ function that is assigned to the node that the thread implements. The thread invokes the C++ function once in every cycle of the loop, which starts only after every thread is ready to start (lines 2522–2527), which can be graciously stopped (the loop condition in lines 2509–2520), and which invokes the C++ function by invoking function `update_channels` (the loop body in lines 2509–2520) whose definition is shown in either lines 2408–2419 of Figure 23a if the implemented node is a producer node or lines 2434–2440 of Figure 23b otherwise (the C++ function is invoked through the pointer `arg__dict_Entry__of__Node__node::value::fn_info::fn_ptr::value`).

6.2 | The Generated Communication Code

Beside generating the computation code, Tice library also generates the code that implements the communication of every producer-consumer pair expressed in the Tice model. Figure 23 shows the communication code that Tice library generates for every node-implementing thread (the generated class `Periodic_rt_task__base_class` shown in Figure 22 is inherited by the generated class `Periodic_rt_task` as shown in lines 2390–2393 of Figure 23a and lines 2431–2433 of Figure 23b). Figure 23a and Figure 23b differ only in their template specializations (template parameter pack `args__write_to_arc` is empty in lines 2385–2389 of Figure 23a but non-empty in lines 2428–2430 of Figure 23b) and the need to write to one or more buffers



Lines 21–22

```
#include <thread>
#include <chrono>
```

Lines 29–34

```
//\begin{platform-specific header files}
#include <sched.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/sched.h>
//\end{platform-specific header files}
```

Lines 2451–2454

```
template<prms_2B(I, _, R(typename... args__Channel__channel_list), R(typename... args__write_to_arc))>
struct Periodic_rt_task__base_class<args_2B(I, _, R(tuple::construct<I, args__Channel__channel_list...>),
                                              R(tuple::construct<I, args__write_to_arc...>))>
{
```

Lines 2472–2496

```
  inline long setup_scheduler()
  {
    //\begin{platform-specific region}
    struct {
      uint32_t size;
      uint32_t sched_policy;
      uint64_t sched_flags;
      int32_t sched_nice;
      uint32_t sched_priority;
      uint64_t sched_runtime;
      uint64_t sched_deadline;
      uint64_t sched_period;
    } scheduling_attributes;
    memset(&scheduling_attributes, 0, sizeof(scheduling_attributes));
    scheduling_attributes.size = sizeof(scheduling_attributes);
    scheduling_attributes.sched_policy = SCHED_DEADLINE;
    scheduling_attributes.sched_runtime
      = std::chrono::duration_cast<std::chrono::nanoseconds>(std::chrono::duration<int, H(typename arg__dict_Entry__of__Node__node::value::
                                                                                            wcet)>(1)).count();
    scheduling_attributes.sched_deadline
      = std::chrono::duration_cast<std::chrono::nanoseconds>(std::chrono::duration<int, H(typename arg__dict_Entry__of__Node__node::value::
                                                                                            period)>(1)).count();
    return syscall(SYS_sched_setattr, 0, &scheduling_attributes, 0) ? errno : 0;
    //\end{platform-specific region}
  }
```

Lines 2509–2520

```
  inline void run_periodically_until_stopped()
  {
    typedef std::chrono::high_resolution_clock::duration Delta;
    const Delta period = std::chrono::duration_cast<Delta>(std::chrono::duration<H(int,
                                                                                   typename
                                                                                   arg__dict_Entry__of__Node__node::value::period)>(1));
    std::chrono::time_point<std::chrono::high_resolution_clock> release_time(std::chrono::high_resolution_clock::now());
    while (std::atomic_load_explicit(&communication_data.not_stopped, std::memory_order::memory_order_relaxed)) {
      update_channels();
      std::this_thread::sleep_until(release_time += period);
    }
  }
```

Lines 2522–2527

```
  void run_periodic_task()
  {
    if (wait_for_start_signal(setup_scheduler())) {
      run_periodically_until_stopped();
    }
  }
```

Lines 2537–2543

```
  void run()
  {
    task = std::thread(&Periodic_rt_task__base_class<args_2B(I, _,
                                                         R(tuple::construct<I, args__Channel__channel_list...>),
                                                         R(tuple::construct<I, args__write_to_arc...>))>::run_periodic_task,
                       this);
  }
```

Line 2551

```
};
```

FIGURE 22 Computation code generator in file `internals/v1/v1_internals_program.hpp` at commit `28fe6ac76a`.[36]

(function `update_channel` is defined and invoked in lines 2394–2406 and lines 2408–2419 of Figure 23a, respectively, but neither defined nor invoked in Figure 23b).

While Tice library can choose from many different options, currently Tice library generates a distinct buffer object for every arc expressed in the model by assuming that the arc's producer and consumer are on distinct threads and associates every buffer object with exactly one producer and one consumer threads. Every buffer object has two public member functions: `get_buffer_to_write`, which we call writing function, to write to the buffer and `get_buffer_to_read`, which we call reading function, to read from the buffer. The writing and reading functions return a pointer and a reference to a buffer whose type is the arc's channel type, respectively. While the writing function is invoked only by the producer thread (the last parameter of the invocation of `update_channel` in lines 2408–2419 of Figure 23a), the reading function is invoked by both the producer and the consumer threads (the first statement in lines 2408–2419 of Figure 23a and in lines 2434–2440 of Figure 23b). In each of their periods, the producer and consumer threads invoke the writing and reading functions only once, respectively, as already shown in lines 2509–2520 of Figure 22.

Lines 2385–2389

```
template<prms_2A(I, _, R(typename... args__Channel__channel_list),
                 R(typename... args__write_to_arc, v1::array::Idx... args__read_from_idx))>
struct Periodic_rt_task<args_2A(I, _, R(tuple::construct<I, args__Channel__channel_list...>),
                                R(pair::construct<H(I, tuple::construct<I, args__write_to_arc...>,
                                                    Array_idx_list<I, args__read_from_idx...>)>))> :
```

Lines 2390–2393

```
#define base_class Periodic_rt_task__base_class<args_2B(I, _, R(tuple::construct<I, args__Channel__channel_list...>),   \
                                                R(tuple::construct<I, args__write_to_arc...>))>
  base_class
{
```

Lines 2394–2406

```
private:
  template<typename arg__return_type_, typename arg__channel_type_>
  inline void update_channel(const arg__return_type_ &output, arg__channel_type_ *channel) {
    if (channel) {
      *channel = output;
    }
  }

  template<typename arg__return_type_, typename arg__channel_type_, typename... args__channel_type_>
  inline void update_channel(const arg__return_type_ &output, arg__channel_type_ *channel, args__channel_type_ *... channels) {
    update_channel(output, channel);
    update_channel(output, channels...);
  }
```

Lines 2408–2419

```
  void update_channels()
  {
    typename arg__dict_Entry__of__Node__node::value::fn_info::return_type producer_output
      = arg__dict_Entry__of__Node__node::value::fn_info::fn_ptr::value(std::get<args__read_from_idx>(base_class::channel_db)
                                                                      .get_buffer_to_read(base_class::release_idx)...);

    update_channel(producer_output,
                   std::get<args__write_to_arc::value>(base_class::channel_db)
                   .get_buffer_to_write(base_class::release_idx, base_class::period, base_class::period_db[args__write_to_arc::key])...);

    ++base_class::release_idx;
  }
```

Lines 2421–2426

```
public:
  Periodic_rt_task(Periodic_rt_task_communication_data<I, args__Channel__channel_list...> &communication_data) :
    base_class(communication_data) {
  }
};
#undef base_class
```

(a) Code generated for every producer node, which includes every consumer node that is also a producer node.

Lines 2428–2430

```
template<prms_2A(I, _, R(typename... args__Channel__channel_list), R(v1::array::Idx... args__read_from_idx))>
struct Periodic_rt_task<args_2A(I, _, R(tuple::construct<I, args__Channel__channel_list...>),
                                R(pair::construct<I, tuple::construct<I>, Array_idx_list<I, args__read_from_idx...>>))> :
```

Lines 2431–2433

```
#define base_class Periodic_rt_task__base_class<args_2B(I, _, R(tuple::construct<I, args__Channel__channel_list...>), R(tuple::construct<I>))>
  base_class
{
```

Lines 2434–2440

```
private:
  void update_channels()
  {
    arg__dict_Entry__of__Node__node::value::fn_info::fn_ptr::value(std::get<args__read_from_idx>(base_class::channel_db)
                                                                  .get_buffer_to_read(base_class::release_idx)...);
    ++base_class::release_idx;
  }
```

Lines 2442–2447

```
public:
  Periodic_rt_task(Periodic_rt_task_communication_data<I, args__Channel__channel_list...> &communication_data) :
    base_class(communication_data) {
  }
};
#undef base_class
```

(b) Code generated for every sink node, which is every consumer node that is not also a producer node, and isolated node, which is neither producer nor consumer node.

FIGURE 23 Communication code generator in file `internals/v1/v1_internals_program.hpp` at commit `28fe6ac76a`.[36]

The writing and reading functions are implemented by an instance of class `Channel__base_class` that is generated by the template in lines 2215–2296 of file `internals/v1/v1_internals_program.hpp` at commit `28fe6ac76a`.[36] The writing and reading functions internally work on a private data member called communication array. The communication array is an array whose element type is the arc's channel type and whose elements are initially the channel's initial data. Since the writing and reading functions share the communication array and other data members, all data members are protected by one thread mutex (mutual exclusion). Currently, Tice library requires no specification of the mutex synchronization cost on the execution platform, and therefore, Tice library does not account for it when checking the gEDF schedulability conditions. As a result, currently the mutex synchronization cost must be included in the WCET assigned to every node.

The communication array's size is three so that one slot can be used exclusively by the producer thread throughout each of its periods, another slot can be used exclusively by the consumer thread throughout each of its period, and the remaining slot can be used to prevent the producer thread from blocking waiting for the consumer thread to finish using its slot when the producer's current period ends not at the start of some consumer's period. The communication protocol that is described next uses another private data member that is an array of size three called usage array. The usage array's elements can be taken as the start times of some periods of the associated consumer. Initially, two of the usage array's elements are zero, which is the start time of the consumer's first period, while the remaining element is the end time of the consumer's period that according to the MoCC is the last to read the channel's initial data.

Lastly, the writing and reading functions of a single buffer use the following communication protocol. An invoked writing function will determine whether according to the MoCC the associated consumer will read the output data. If it is not the case, the writing function will return a null pointer. Otherwise, the writing function will first determine the time t that according to the MoCC the output data should be made visible to the associated consumer. The writing function will then compute a as the start time of the consumer's period at t and b as the end time of the consumer's period that according to the MoCC is the last to read the output data. The usage array is then sought for an element whose value is the smallest that is less than or equal to a. Based on the MoCC, the initial elements of the communication and usage arrays, and the complete protocol description, it can be proven that the element is guaranteed to be found at some index i because such an element means that the consumer thread is currently not using the communication array's slot at i. Once found, b is assigned to the usage array at index i, and a pointer to the communication array at index i is returned. On the other hand, an invoked reading function will compute b as the start time of the current period of the associated consumer and searches its usage array for every element whose value is the smallest that is greater than b. As in the preceding paragraph, it can be proven that *exactly* one element will be found. Once found at some index i, the reading function returns a pointer to the communication array at index i.

6.3 | Generating Code for Other Architecture

The computation and communication code generation described in Section 6.1 and Section 6.2 is just one out of the many ways Tice models can be implemented on various different architecture. Therefore, we will now show how Tice library can be extended to generate various other implementing code.

With regard to generating the computation code, there are two cases to consider: (1) the thread scheduling and partitioning policies and (2) the underlying thread API. Generating different computation code in either case should extend only the header-include list and the definition of function `setup_scheduler` in lines 29–34 and lines 2472–2496 of Figure 22, respectively. The extension can be done in two different ways: using conditional preprocessor directives or using template specializations. Using template specializations entails extending the parameter list of template `HW` described in Section 3.1 to express the desired scheduling and partitioning policies. Either way, template `construct__back_end__generate` found in file `internals/v1/v1_internals_program.hpp`[36] has to be extended in the same way to incorporate the corresponding schedulability test, which currently instantiates template `construct__schedulability_test__gedf` if the first parameter is an instance of `HW` whose first parameter is an instance of `Core_ids` with at least one core ID.

With regard to generating communication code by considering every individual arc and their incident nodes, different code should extend only the header-include list in lines 29–34 of Figure 22 and the definition of class template `Channel__base_-class` in file `internals/v1/v1_internals_program.hpp`.[36] Even if the communication code to be generated does not consider every individual arc, the needed extension could still be limited to the aforementioned two areas by employing some template metaprograms to obtain the desired `Channel__base_class`. If that proves insufficient, the needed extension should be limited to the definition of class template `Channel`, which inherits the class template `Channel__base_class`, and the places in file `internals/v1/v1_internals_program.hpp`[36] that use `Channel`.

7 | RELATED WORK

The development of new language proposals targeting real-time applications has been quite popular some years ago. The first family of languages that attracted a considerable attention and also achieved some industrial impact are *synchronous languages*. In this family, we can mention control-oriented languages, such as ESTEREL (1983),[45] and data-oriented languages, such as LUSTRE (1987)[46] and SIGNAL (1987).[47] The original idea of synchronous languages is to decompose the application into a set of logically parallel blocks, where each block is logically equivalent to a state machine. The fixed-point semantics for block compositions allows the compiler to generate a "big" state machine, which is an iterative and deterministic implementation of the program. Key advantages of the synchronous languages are that any project can be implemented in hardware (VHDL) or software alike and that any project can be statically checked. The context of synchronous languages is quite different than the one considered here. In Tice, an application is defined as a set of nodes that are eventually contained in a set of concurrent tasks. What is more, in synchronous languages, the time variable is not explicitly considered: the hardware is assumed to be fast enough to make the time required by each transition negligible. In Tice, the timing constraints are first-class citizens in the program definition. More recent proposals of synchronous languages, such as PRELUDE (2009),[48] open the possibility for a multi-task implementation of synchronous systems, but the conceptual distance from the programming paradigm of C/C++ is quite wide.

A number of different proposals offer a set of language constructs to define real-time tasks. Some proposals date back to around 1990: REAL-TIME EUCLID (1986),[49] which is based on the Euclid language, and TCEL (1993).[50] Such languages proposed themselves as alternatives to C/C++, but they did not raise much interest in the community of embedded system developers. A number of other proposals are modifications of the C/C++ language. In this class, we can mention REAL-TIME CONCURRENT C (1991),[51] CRL (1995),[52] PSIC (1998),[53] ALERT (1999),[54] FOREC (2013),[55] and TIMED C (2018).[56] Such proposals were essentially dialects of the C language that simplified the definitions of concurrent tasks, real-time constraints, and scheduling policies to be passed on to the scheduler. Most of these proposals came along with their own compilation tool-chains, very frequently in the form of a source-to-source compiler. In contrast, Tice does not need any support other than a C++ compiler compliant with the most recent C++ standard. More importantly, Tice champions a clear separation of concerns between properties that can be checked by looking at the functional model and assuming that it operates according to the MoCC and properties that need to be enforced in the back-end of the compiler to ensure that the architectural mapping of the application respects the MoCC.

Tice borrows its LET MoCC from GIOTTO,[18] which is gaining acceptance, particularly in the automotive industry, to facilitate the transition of embedded software from unicore to multicore processors.[57] The TCEL follow-up paper that addresses the synthesis problem[58] inspires the kinds of temporal constraints that Tice can enforce, namely end-to-end delay and correlation constraints. The semantics of the end-to-end delay that Tice can enforce is called last-to-first according to the nomenclature that is introduced by Feiertag, et al.[31] To our knowledge the semantics of correlation has not been explored to the same extent that permits us to name the semantics of Tice's correlation constraint in the same manner. Other possible semantics of end-to-end delay has been further studied by Forget, et al. in proposing a language to express end-to-end delay constraints with various semantics as well as the framework to verify the expressed constraints on the multi-periodic synchronous MoCC of PRELUDE,[59] while Khatib, et al.[60] and Aba, et al.[61] investigate end-to-end delay in a synchronous dataflow graph and in minimizing a system's energy consumption, respectively. Aside from that, while different strategies exist to implement the LET MoCC depending on the target hardware and the optimality criteria,[57,62,63] currently Tice library uses a simplistic non-optimal implementation strategy described in Section 6.1 and Section 6.2 while showing how to accommodate other implementation strategies in Section 6.3.

Lastly, in using C++ as the platform to integrate different models, Tuscherer, et al. propose that MATLAB/SIMULINK models be expressed directly in C++ programs so that the models can be seamlessly integrated with the C++ object models for sub-microscopic traffic-flow simulations of autonomous-driving vehicles.[38] On the other hand, Deters, et al. propose to integrate application models and Liu-Layland workload models by expressing the Liu-Layland workload models directly in C++ programs using a C++ active library that at compile time checks the schedulability of the expressed model, discarding some tasks automatically to improve the schedulability, and generates the schedule accordingly.[64] Additionally, Gil and Lenz propose a C++ active library for interacting with SQL databases safely by eliminating potential mismatches between application and database models.[65] Veldhuizen also proposes a C++ active library for expressing a number of scientific-computation data models in C++ programs so that the library can generate optimal code for operations on the expressed models.[66] However, none of the models is the MoCC (i.e., the semantics) of a real-time language, which Tice proposes.

8 | CONCLUSIONS

We have proposed a novel real-time language, called Tice, that distinguishes itself from other real-time languages by being compilable using off-the-shelf C++ compilers and workable with existing C++ programming tools (e.g., program analyzers, editors, and debuggers). In Section 3, we have shown Tice's syntax and semantics as well as demonstrating Tice's efficacy in Section 4 on a concrete real-time embedded-system engineering case study called ROSACE. In Section 4.2, we also highlighted the particular use-case of using Tice and an off-the-shelf C++ compiler altogether as a modeling tool, demonstrating it on the ROSACE case study in the aviation domain and to some extent on the WATERS 2017 Industrial Challenge in the automotive domain. Our evaluations of the particular use-case of using Tice and an off-the-shelf C++ compiler altogether as a modeling tool in Section 5 show that the use-case would be practically feasible because Tice models exemplified by the ROSACE case study and to some extent the WATERS 2017 Industrial Challenge would be compilable using an off-the-shelf C++ compiler repetitively, each time with different parameters, in a reasonable time. Lastly, our evaluations in Section 5 show that GCC performed better than Clang in general.

Therefore, future real-time language research could investigate how Tice would improve the current practice of engineering the real-time aspect of embedded software. The result could then be used as a baseline to investigate how the various features and models proposed in existing and future real-time languages would be best applied on which real-time engineering problems. Such a baseline is possible because real-time languages that are compilable using off-the-shelf C++ compilers can be compared empirically with variability only in the language factor as the same off-the-shelf C++ compilers and programming tools can be used on the different real-time languages proposed. Aside from that, Tice itself could be further improved, for example, by using different techniques to analyze temporal constraints with less memory and by incorporating different mapping strategies in generating the set of real-time tasks depending on the target hardware, the expression of which can be further enriched by incorporating information other than processor core IDs. Additionally, Tice could also incorporate further types of timing constraints, such as those presented by Forget, et al.[59] As Tice has been designed to be extensible, we are currently working on another paper to deal with the problem of extending Tice with different mapping strategies, further types of timing constraints, and communication with external tools, such as a WCET analyzer, a constraint solver, and a simulator, all of which without requiring any modification to the off-the-shelf C++ compilers. Lastly, research on real-time languages that are compilable using off-the-shelf C++ compilers could spur research to improve the state of the arts of C++ compilers.

ACKNOWLEDGMENTS

We thank Giuseppe Lipari and Julien Forget at the University of Lille in Villeneuve-d'Ascq, France, for the discussions on real-time languages where Giuseppe Lipari gave us pointers to the ROSACE case study[19] and repository,[34] and Julien Forget gave us pointers to Pierre-Emmanuel Hladik's work in the repository and the paper that studies further semantics of end-to-end delay.[59] Lastly, we would like to thank the anonymous reviewers for providing constructive feedback.

References

1. Baleani M, Ferrari A, Mangeruca L, et al. Correct-by-construction transformations across design environments for model-based embedded software development. In: DATE. IEEE; 2005; Washington, DC, USA: 1044–1049.

2. Sangiovanni-Vincentelli A, Di Natale M. Embedded System Design for Automotive Applications. *Computer* 2007; 40(10): 42–51. doi: 10.1109/MC.2007.344

3. Cass S. Interactive: The Top Programming Languages 2019. Online at https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019; accessed on August 3, 2020.

4. StackOverflow. Developer Survey 2019: How Technologies Are Connected. Online at https://insights.stackoverflow.com/survey/2019#technology-_-how-technologies-are-connected; accessed on August 3, 2020.

5. Kormanyos C. *Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming*. Berlin, Germany: Springer. 1st ed. 2013

6. Meyers S. Why C++ Sails When The Vasa Sank. Online at https://events.yandex.ru/lib/talks/1954/; accessed on August 3, 2020.

7. Maimone M. C++ on Mars. Online at https://www.youtube.com/watch?v=3SdSKZFoUa8; accessed on August 3, 2020.

8. Emshoff B. Using C++ on Mission and Safety Critical Platforms. Online at https://www.youtube.com/watch?v=sRe77Mdna0Y; accessed on August 3, 2020.

9. Foote T. Celebrating 9 Years of ROS. Online at https://spectrum.ieee.org/automaton/robotics/robotics-software/celebrating-9-years-of-ros; accessed on August 3, 2020.

10. Burns A, Wellings A. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Harlow, Essex, England: Pearson Education Limited. 4th ed. 2009.

11. Gerkey B. Why ROS 2.0?. Online at https://design.ros2.org/articles/why_ros2.html; accessed on August 3, 2020.

12. Lin P. Tesla Autopilot Crash. Online at https://spectrum.ieee.org/cars-that-think/transportation/self-driving/tesla-autopilot-crash-why-we-should-worry-about-a-single-death; accessed on August 3, 2020.

13. Travis G. How the Boeing 737 Max Disaster Looks to a Software Developer. Online at https://spectrum.ieee.org/aerospace/aviation/how-the-boeing-737-max-disaster-looks-to-a-software-developer; accessed on August 3, 2020.

14. Barr A. *The Problem with Software: Why Smart Engineers Write Bad Code*. Cambridge, MA, USA: The MIT Press. 1st ed. 2018.

15. Motor Industry Software Reliability Association. *MISRA-C:2012 Guidelines for the use of the C language in critical systems*. Warwickshire, UK: MIRA Limited. 3rd ed. 2013.

16. Motor Industry Software Reliability Association. *MISRA-C++:2008 Guidelines for the use of the C++ language in critical systems*. Warwickshire, UK: MIRA Limited. 1st ed. 2008.

17. Jones N. Introduction to MISRA C. Online at https://www.embedded.com/electronics-blogs/beginner-s-corner/4023981/Introduction-to-MISRA-C; accessed on August 3, 2020.

18. Henzinger TA, Horowitz B, Kirsch CM. Giotto: A Time-Triggered Language for Embedded Programming. In: EMSOFT. Springer; 2001; Berlin, Germany: 166–184

19. Pagetti C, Saussié D, Gratia R, Noulard E, Siron P. The ROSACE case study: From Simulink specification to multi/many-core execution. In: RTAS. IEEE; 2014; Washington, DC, USA: 309–318

20. Free Software Foundation. C++ Standards Support in GCC. Online at https://gcc.gnu.org/projects/cxx-status.html; accessed on August 3, 2020.

21. University of Illinois at Urbana-Champaign. C++ Support in Clang. Online at http://clang.llvm.org/cxx_status.html; accessed on August 3, 2020.

22. Beningo J. *Reusable Firmware Development: A Practical Approach to APIs, HALs, and Drivers*. New York, NY, USA: Apress. 1st ed. 2017.

23. Barr M, Massa A. *Programming Embedded Systems: With C and GNU Development Tools*. Sebastopol, CA, USA: O'Reilly. 2nd ed. 2006.

24. Edwards L. *Embedded System Design on a Shoestring: Achieving High Performance with a Limited Budget*. Kidlington, Oxfordshire, England: Newnes. 1st ed. 2003.

25. Czarnecki K, Eisenecker U, Glück R, Vandevoorde D, Veldhuizen T. Generative Programming and Active Libraries. In: Jazayeri M, Loos RGK, Musser DR., eds. *Generic Programming*. Selected papers of International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27–May 1, 1998. Berlin, Germany: Springer. 2000 (pp. 25–39)

26. Veldhuizen TL. *Active Libraries and Universal Languages*. PhD thesis. Indiana University, Indianapolis, IN, USA; 2004.

27. Sheard T, Jones SP. Template Meta-programming for Haskell. *SIGPLAN Not.* 2002; 37(12): 60–75. doi: 10.1145/636517.636528

28. Stroustrup B. Evolving a Language in and for the Real World: C++ 1991–2006. In: SIGPLAN. ACM; 2007; New York, NY, USA: 4-1–4-59

29. Czarnecki K, Eisenecker UW. *Generative Programming: Methods, Tools, and Applications*. Boston, MA, USA: Addison-Wesley. 1st ed. 2000.

30. Prastowo T, Palopoli L, Abeni L. C++ Hard-real-time Active Library: Syntax, Semantics, and Compilation of Tice Programs. *SIGBED Rev.* 2019; 16(3): 69–74. doi: 10.1145/3373400.3373411

31. Feiertag N, Richter K, Nordlander J, Jonsson J. A Compositional Framework for End-to-End Path Delay Calculation of Automative Systems under Different Path Semantics. In: CRTS. IEEE; 2008; Washington, DC, USA.

32. Kirsch CM, Sokolova A. The Logical Execution Time Paradigm. In: Chakraborty S, Eberspächer J., eds. *Advances in Real-Time Systems*. Festschrift for Georg Färber. Berlin, Germany: Springer. 2012 (pp. 103–120)

33. Prastowo T. *Toward C++ as a Platform for Language-Oriented Programming: On the Embedding of a Model-Based Real-Time Language*. PhD thesis. Università degli Studi di Trento, Trento, Italy; 2020.

34. Pagetti C, Saussié D, Gratia R, et al. The Repository of the ROSACE Case Study. Online at https://forge.onera.fr/projects/rosace-case-study; accessed on August 3, 2020.

35. Agency ES. N° 33–1996: Ariane 501 - Presentation of Inquiry Board report. Online at https://www.esa.int/Newsroom/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report; accessed on August 3, 2020.

36. Prastowo T. The Repository of Tice Library. Online at https://savannah.nongnu.org/projects/tice; accessed on August 3, 2020.

37. Groarke P. Re: [EXTERNAL] Re: Linear algebra library proposal. Online at https://lists.isocpp.org/sg14/2019/06/0149.php; accessed on August 3, 2020.

38. Tuchscherer D, Weibert A, Tränkle F. Modern C++ As a Modeling Language for Automated Driving and Human-robot Collaboration. In: MODELS. ACM; 2016; New York, NY, USA: 136–142

39. Pagetti C, Forget J, Falk H, Oehlert D, Luppold A. Automated Generation of Time-Predictable Executables on Multicore. In: RTNS '18. ACM; 2018; New York, NY, USA: 104–113

40. Forget J, Boniol F, Pagetti C. WATERS Industrial Challenge 2017 with Prelude. In: 8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems. Online; 2017; http://waters2017.inria.fr/program/.

41. Prastowo T, Palopoli L, Abeni L, Lipari G. Analyses of a Model-Based Real-Time Language Embedded in C++. In: SAC. ACM; 2020; New York, NY, USA: 1330–1339

42. Faggioli D, Abeni L, Lelli J, Cucinotta T, Scordino C. Deadline Task Scheduling. Online at https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt; accessed on August 3, 2020.

43. Baruah S, Bertogna M, Buttazzo G. *Multiprocessor Scheduling for Real-Time Systems*. Cham, Switzerland: Springer International Publishing. 1st ed. 2015

44. Kerrisk M, Zijlstra P, Lelli J. sched—overview of CPU scheduling. Online at http://man7.org/linux/man-pages/man7/sched.7.html; accessed on August 3, 2020.

45. Berry G, Moisan S, Rigault JP. Esterel: Towards a synchronous and semantically sound high-level language for real-time applications. In: RTSS. IEEE; 1983; Silver Spring, MD, USA: 30–37.

46. Caspi P, Pilaud D, Halbwachs N, Plaice JA. LUSTRE: A Declarative Language for Real-time Programming. In: POPL. ACM; 1987; New York, NY, USA: 178–188

47. Gautier T, Le Guernic P, Besnard L. SIGNAL: A declarative language for synchronous programming of RT systems. In: FPCA. Springer; 1987; Berlin, Germany: 257–277

48. Forget J. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*. PhD thesis. Institut Supérieur de l'Aéronautique et de l'Espace, Toulouse, France; 2009.

49. Kligerman E, Stoyenko AD. Real-Time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering* 1986; SE-12(9): 941-949. doi: 10.1109/TSE.1986.6313049

50. Hong S, Gerber R. Scheduling with Compiler Transformations: The TCEL Approach. In: RTOSS. IEEE; 1993; Washington, DC, USA: 80–84.

51. Gehani N, Ramamritham K. Real-time Concurrent C: A language for programming dynamic real-time systems. *Real-Time Systems* 1991; 3(4): 377–405. doi: 10.1007/BF00365999

52. Stoyenko AD, Marlowe TJ, Younis MF. A Language for Complex Real-Time Systems. *The Computer Journal* 1995; 38(4): 319–338. doi: 10.1093/comjnl/38.4.319

53. Louise S, David V, Delcoigne J, Aussaguès C. OASIS Project: Deterministic Real-Time for Safety Critical Embedded Systems. In: 10th Workshop on ACM SIGOPS European Workshop. ACM; 2002; New York, NY, USA: 223–226

54. Palopoli L, Buttazzo G, Ancilotti P. A C language extension for programming real-time applications. In: RTCSA. IEEE; 1999; Washington, DC, USA: 103–110

55. Girault A, Hili N, Jenn E, Yip E. A Multi-Rate Precision Timed Programming Language for Multi-Cores. In: 2019 Forum for Specification and Design Languages (FDL). IEEE; 2019; Washington, DC, USA: 1–8

56. Natarajan S, Broman D. Timed C: An Extension to the C Programming Language for Real-Time Systems. In: RTAS. IEEE; 2018; Washington, DC, USA: 227–239

57. Biondi A, Natale MD. Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm. In: RTAS. IEEE; 2018; Washington, DC, USA: 240–250

58. Gerber R, Hong S, Saksena M. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In: RTSS. IEEE; 1994; Washington, DC, USA: 192–203

59. Forget J, Boniol F, Pagetti C. Verifying end-to-end real-time constraints on multi-periodic models. In: ETFA. IEEE; 2017; Washington, DC, USA: 1–8

60. Khatib J, Munier-Kordon A, Klikpo EC, Trabelsi-Colibet K. Computing Latency of a Real-Time System Modeled by Synchronous Dataflow Graph. In: RTNS. ACM; 2016; New York, NY, USA: 87–96

61. Ait Aba M, Zaourar L, Munier A. Approximation Algorithm for Scheduling a Chain of Tasks on Heterogeneous Systems. In: Euro-Par 2017: Parallel Processing Workshops. Springer International Publishing; 2018; Cham, Switzerland: 353–365

62. Henzinger TA, Kirsch CM. The Embedded Machine: Predictable, Portable Real-time Code. In: PLDI. ACM; 2002; New York, NY, USA: 315–326

63. Horowitz B. Single-mode, Single-processor Giotto Scheduling. Tech. Rep. UCB/CSD-03-1238, EECS Department, University of California, Berkeley; Berkeley, CA, USA: 2003.

64. Deters M, Gill C, Cytron R. Rate-Monotonic Analysis in the C++ Type System. In: MDES. IEEE; 2003; Washington, DC.

65. Gil JY, Lenz K. Simple and safe SQL queries with C++ templates. *Science of Computer Programming* 2010; 75(7): 573–595. doi: 10.1016/j.scico.2010.01.004

66. Veldhuizen TL. Blitz++: The Library that Thinks it is a Compiler. In: Langtangen HP, Bruaset AM, Quak E., eds. *Advances in Software Tools for Scientific Computing*. 1st ed. Berlin, Germany: Springer. 2000 (pp. 57–87)

How to cite this article: T. Prastowo, L. Palopoli, and L. Abeni (2020), Tice: a Real-Time Language Compilable Using C++ Compilers, *Software: Practice and Experience*, *2020;00:1–6*.