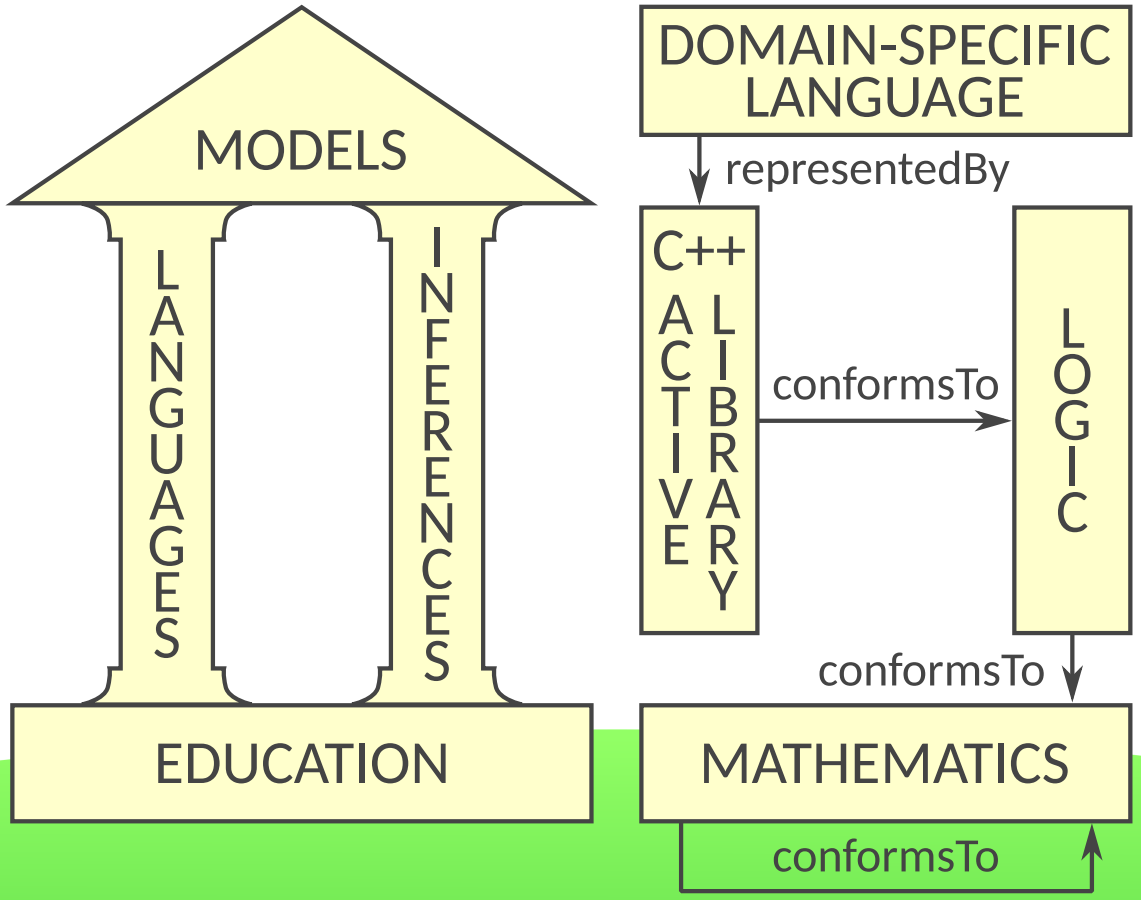
TOWARD C++ AS A PLATFORM FOR LANGUAGE-ORIENTED PROGRAMMING: ON THE EMBEDDING OF A MODEL-BASED REAL-TIME LANGUAGE



TADEUS PRASTOWO

Toward C++ as a Platform for Language-Oriented Programming:

On the Embedding of a Model-Based Real-Time Language

Abstract Cyber-physical systems are dynamic physical systems that are controlled by computers for their safe and sound operations (e.g., cars, satellites, robots, elevators, and many others). Consequently, the programs running cyber-physical systems have *real-time* requirements, which require the programs to compute not only correctly but also timely because dynamic physical systems need to move to correct positions within certain duration to ensure safe and sound operations. To satisfy real-time requirements in better ways, many real-time languages have been proposed in the literature. Nevertheless, the general-purpose non-real-time languages C and C++ have remained the de facto languages to program cyber-physical systems, including Mars rovers and F-35 jet fighters. Given this reality, the better ways to satisfy real-time requirements have been the use of *model-based* tools (e.g., MATLAB®/Simulink®) that allow cyber-physical systems to be designed by modeling and simulating them and the resulting models to be translated automatically to C programs. Model-based tools, however, leave the resulting C programs for manual integration with other C/C++ programs, such as legacy/third-party device drivers and libraries. Since manual integration could slip in some inconsistencies, which proved fatal in the maiden flight of Ariane-5 rocket, this work shows how the *standard* features of C++, which support active libraries, can be used to embed a model-based real-time language, called Tice, as a C++ active library that can be used to declaratively express models of real-time systems that are processable by off-the-shelf *standard* C++ compilers (e.g., GCC and Clang) that automatically not only translate the models into C/C++ programs but also check both the validity of the models and the consistency of the models with other C/C++ programs. Furthermore, being compilable by off-the-shelf standard C++ compilers also sets Tice apart from other real-time languages already proposed in the literature because the other languages require either their own special compilers/interpreters or non-standard C/C++ compilers. Consequently, while Tice itself either uses no C++ features that are unsuitable for cyber-physical systems (e.g., exception) or uses some in judicious manner (e.g., template instantiations to generate programs), Tice prevents no usage that is permitted by standard C++ compilers. Beside that, as C++ active libraries are indeed ordinary C++ libraries, C++ active libraries are seamlessly composable as ordinary C++ libraries, and therefore, as models play an increasingly important role in software engineering, this work shows the potential of C++ as a platform for language-oriented programming where different languages that express different kinds of models and are embedded as C++ active libraries could be composed seamlessly.

Tadeus Prastowo

Trento, Italy, 2020

UNIVERSITÀ DEGLI STUDI
DI TRENTO

ICT
Doctoral School

A DOCTORAL DISSERTATION submitted to the
DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
(DIPARTIMENTO DI INGEGNERIA E SCIENZA DELL'INFORMAZIONE)
in partial fulfillment of the requirements for the degree of
DOTTORE DI RICERCA IN INFORMATICA
that is awarded by the
UNIVERSITY OF TRENTO
(UNIVERSITÀ DEGLI STUDI DI TRENTO)
at Via Sommarive, 9 Trento, 38123 Italy.

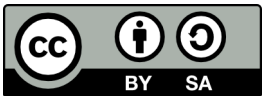
Advisors

- Prof. Luigi Palopoli
  Università degli Studi di Trento
  Trento, Italy
- Prof. Luca Abeni
  Scuola Superiore S. Anna
  Pisa, Italy

Examiners

- Prof. Giuseppe Lipari
  Université de Lille 1
  Lille, France
- Prof. Marco Di Natale
  Scuola Superiore S. Anna
  Pisa, Italy
- Prof. Roberto Passerone
  Università degli Studi di Trento
  Trento, Italy

© 2020 Tadeus Prastowo <0x66726565@gmail.com>. This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.



To my beloved Father in Heaven,

my Lord & Savior, Christ Jesus,

and my Helper, the Holy Spirit.

To my beloved wife, Hana,

and our beloved sons:

Qielo (the heavens),

Velio (the secret),

Klavio (the key).

"Here is what I have seen: It is good and fitting for one to eat and drink, and to enjoy the good of all his labor in which he toils under the sun all the days of his life which God gives him; for it is his heritage. As for every man to whom God has given riches and wealth, and given him power to eat of it, to receive his heritage and rejoice in his labor—this is the gift of God. For he will not dwell unduly on the days of his life, because God keeps him busy with the joy of his heart."

— Ecclesiastes 5:18–20 (NKJV)

"Go, eat your bread with joy,
And drink your wine with a merry heart;
For God has already accepted your works.
Let your garments always be white,
And let your head lack no oil.

Live joyfully with the wife whom you love all the days of your vain life which He has given you under the sun, all your days of vanity;"

— Ecclesiastes 9:7–9 (NKJV)

Contents

List of Figures

Preface

This work is intended to be accessible by everyone interested in the world of computers with a specific focus on *software engineering*, particularly in *computer programming*. Consequently, after completing this paragraph, readers who understand the terms:

- "C++", "embedded systems", and "real-time" as well as how the terms are related can jump to Chapter 1 on page 1,
- "C++" and "real-time programming language" and who only would like to know how the former can be used as the latter can jump to Chapter 4 on page 25,
- "C++" and "language-oriented programming" as well as how the terms are related by the term "embedding of a language" and who only would like to know the techniques to do so in C++ can jump to Chapter 6 on page 57.

The rest and those who prefer not to jump should find this preface useful in some ways. Before closing this paragraph, readers are informed that this work uses numeric citations to refer to other sources listed in the bibliography starting on page 63 (e.g., the sentence "this document is produced using [70], a language that is embedded within another language [66]" has two numeric citations, the first and second of which refer to LaTeX and TeX, respectively). Lastly, this paragraph is closed by noting that parts of this work have been published as one original research article [88] and one original research paper [89].

Programs are written to automate various things so that more can be accomplished in a single day with fewer errors and lower cost. Programs running industrial PCs[1] and PLCs[2] manufacture various products 24 hours a day with minimal supervision. This means that those programs in a single day produce more medicines, more pasta, more smartphones, and more of other products than what would otherwise be possible if various industries still relied on people to control their production machinery. Beside manufacturing more products in a single day, programs also serve more needs by running automated teller machines, payment processing systems, online learning systems, and various other services 24 hours a day with minimal supervision. Programs have also been written to automate harder tasks, such as autonomously driving cars, diagnosing diseases, and translating spoken and written languages. Furthermore, programs have also been written to perform creative things, such as running computers to win games of chess [123], Go [122], and *Jeopardy!*® [124]. Hence, it is not hard to envision a future where programs run computers to architect buildings, direct movies, write novels, build dams, and realize other things, including writing programs [4, 46, 78, 99]. As of year 2020, however, programs have not yet run computers to conceive and write programs in the way new novels are conceived and written and new dams are engineered and built. Instead, programs are conceived, engineered, written, and tested by *software engineers* in an activity called computer programming.

Computer programming uses *software languages* [69] as its primary means and have different *programming domains* that use their own main software languages [37], for example, just to mention a few:

- The domain of web applications uses the *scripting language* JavaScript, the *markup language* HTML, and the *styling language* CSS.
- The domain of enterprise information system uses the *programming language* Java and the *database query language* SQL.
- The domain of mobile applications uses the programming languages Java for Android® devices and Swift for iOS® devices.

The programming domain of this work is the domain of *embedded systems* that uses the programming languages C and C++. Unlike other software languages used in other programming domains, C and C++ are designed to effectively and efficiently manipulate computer hardware because, by definition, embedded systems are mostly hidden from sight due to the computer hardware being embedded within other static/dynamic physical systems, such as cars, bullet trains, satellites, robots, elevators, and similar dynamic physical systems, and smart door locks, refrigerators, washing machines, vending machines, and similar static physical systems (smartphones, smartwatches, game consoles, and similar devices are computer hardware, much like laptops, instead of computer hardware that is embedded within and to control another physical system). Therefore, computer programming using C and C++ is considerably harder because their effectiveness and efficiency in manipulating

[1] `https://en.wikipedia.org/wiki/Industrial_PC`.
[2] `https://en.wikipedia.org/wiki/Programmable_logic_controller`.

computer hardware make it easy for humans to make mistakes. Programming mistakes in the domain of embedded systems, particularly in the subdomain of *cyber-physical systems*, however, may have grave results, including monetary loss as exemplified by the failed maiden flight of Ariane-5 rocket in June 1996 [1] and the loss of lives as exemplified in the crashes of two new Boeing 737 MAX 8 in October 2018 and March 2019 [111].

To make computer programming in the domain of embedded systems easier to get right, there are three complementary approaches that have long been used in practice:

- C and C++ are used from other software languages, such as Java and Python. In this approach, C and C++ are used to engineer the parts of programs that really have to be engineered in C and C++ while leaving the other parts to be engineered in Java and Python. The other parts that are engineered in Java and Python will then use the parts that are engineered in C and C++.

- C and C++ are written by automatically translating other software languages (e.g., MATLAB®/Simulink®). In this approach, instead of using parts that are engineered in C and C++, program parts are engineered in other software languages that are then translated automatically to C/C++.

- C and C++ are used by manually following a set of rules, such as [76, 79, 80].

This work, therefore, proposes C++ as the means to combine the three complementary approaches. This means that C++ is used to define other software languages that are easier to use than C++ while being automatically composable among themselves and with C/C++ in such a way so that the defined software languages are not only translatable to C/C++ but also capable of enforcing a set of rules automatically. To show the proposal's feasibility, this work presents a programming language called Tice that has been engineered in such a way so that it is automatically composable with, translatable to, and checkable as C/C++ programs while being easier to use than C/C++ in designing and implementing the real-time aspect of *embedded programs*, which are the programs that run embedded systems.

Lastly, the term "real-time" used in this work has the specific meaning of computations that are temporally[3] predictable [73]. By being predictable, the term "real-time" does not refer to fast/quick computations whose results are available as soon as possible; instead, it means that the results can be guaranteed to be available within certain duration, which can be as short as one millisecond or even shorter or as long as one second or even longer. Being real-time is the characteristic of embedded programs in cyber-physical systems because cyber-physical systems operate in a space-time continuum and are required to move to correct positions within certain duration to ensure safe and sound operations.

The Significance of This Work

This work is an answer to the problem that I had in my career as a software engineer. Between 2012 and 2013, I worked at an information-technology company that engineered ERP (enterprise resource planning) programs using mainly Python, Java, and SQL. Back in mid-2000s during my undergraduate study, Java was the hype, but the programs that I engineered using Java computed slower than those I engineered using Python given the same computer. Consequently, unless otherwise required, I engineered most of them using Python, which was the next hype after Java. However, it did not take me long to see that business-intelligence programs engineered using Python were difficult to get right and slower than if I engineered them using SQL. But, SQL was not the next hype, and, unlike my experience in switching from using Java to Python to engineer most programs, switching from Python to SQL to engineer most programs would not be right because SQL was a database query language. Hence, I came to the conclusion that different parts of a single program ought to be engineered using the right languages. It turned out that my conclusion was the main idea of *language-oriented programming* [120], which I discovered later in my doctoral study.

However, throughout my undergraduate and graduate studies as well as my software engineering career, I had never heard about the importance of using multiple languages to engineer different parts of a single program. Consequently, I had thought that I had to use as few languages as possible, which ideally was one. This thought led me first to switch from Java to Python and then to realize that a problem existed when I found SQL to be better than Python in engineering business-intelligence programs. The problem was none other than the fact that I had never studied the use of multiple languages to engineer different parts of a single program, and therefore, I had several questions, such as:

[3]The adverb "temporally" is not the adjective "temporary"; the corresponding adjective of the adverb "temporally" is "temporal", which means being related to time.

- How would multiple languages better be used together? Surely, it could be better than the way SQL and Java had been used together, which was by using as little SQL as possible and as much Java as possible.
- How would different parts of a single program that were written in multiple languages better be kept consistent? Surely, it could be better than the way SQL and Java had been kept consistent, which allows SQL to be used by Java but not the other way around.

To answer my questions, I started to gather information from the Internet that led me first to Donald E. Knuth's *literate programming* as revisited by Norman Ramsey in [90], second to Paul Graham's essays on his success in using *domain-specific languages* (DSL) embedded in Lisp to build a successful start-up company worth millions of US dollars [42], third to Jean-Marie Favre's papers on *model-driven engineering* (MDE) [28, 29], fourth to Alan Kay's 5-year NSF-funded research program on the steps toward the reinvention of programming [56–62], and lastly to Charles Simonyi's *intentional programming* [92, 93, 100]. After two years of part-time research, however, I still could not piece them together such that I could confidently use multiple languages in engineering a single program because they all offered their own language-oriented programming platforms, which were far from what I routinely used at work. Nevertheless, they were all in support of the hypothesis that different parts of a single program ought to be engineered using the right languages. Hence, in mid-2015, I held a seminar titled "Ultimate Programming Language Exists Not" in my undergraduate alma mater to solicit ideas but to no avail. Lastly, in the end of 2015, I contacted my graduate advisor, Prof. Luca Abeni, which led to my doctoral study, the making of this work, and an answer: the widely-used C++ language can be used as a language-oriented programming platform where using multiple languages is none other than using multiple C++ libraries, which is very close to what people routinely use at work in the domain of embedded systems.

Acknowledgments

This work would not be possible without Prof. Luca Abeni's knowledge that his colleague, Prof. Luigi Palopoli, and I had a similar research interest into software languages. Furthermore, this work would not be the right answer without Prof. Luigi Palopoli's profound knowledge about the model-driven engineering of cyber-physical systems. Working with both of them throughout my doctoral study has been a great privilege, and I really thank both of them for helping me in so many different ways.

This work would be worse without the external referees, Prof. Giuseppe Lipari and Prof. Marco Di Natale, taking their time and making the effort to review this work. This work has also benefited from Prof. Roberto Passerone's examination and Prof. Luigi Palopoli's and Prof. Luca Abeni's advice. I am really grateful for all of their feedback.

Lastly, this work would not exist without the scholarship that the University of Trento has granted me throughout my doctoral study and the many high-quality services that the University of Trento has provided: the libraries, the welcome office, the mission office, the office of the ICT Doctoral School, the office of the PhD in science and technology, and many others.

Please skip this page.

1 Introduction

In the future, when programs would run computers to conceive and write programs in the way new novels are conceived and written and new dams are engineered and built, people could program computers in English, which can be very vague and ambiguous in specifying *computer behavior* (i.e., the behavior of a set of computers) [4, 46, 78, 99]. In such a future, a *stakeholder*, such as some company's representative, could simply say: "an embedded program to control a car brake that is ASIL-D certifiable." Then, some program running some computer would automatically figure out the kind of brake to control, the kind of control to use, the relevant ISO®[1] standards, and other myriad details based on big data and dialogues with relevant people and systems. Afterwards, the program running the computer would proceed to either write the desired embedded program or to behave as the desired embedded program itself, refining the program or the behavior as necessary based on test results. Currently, however, *software engineers* spend a lot of efforts engineering the programs needed to realize the computer behavior specified by stakeholders in English or other natural language. Moreover, since the stakeholders' need and computers evolve over time, from time-to-time not only the stakeholders will specify changes to the computer behavior but also different computers will need to realize the desired behavior. Consequently, software engineers also spend a lot of efforts maintaining existing programs. Nevertheless, software engineering research and some entrepreneurs and professionals in the software industry have moved toward the future through an effort that is known by various names, including *model-driven engineering* (MDE) [28, 29, 97], *language-oriented programming* [120], and *intentional programming* [2, 92, 93, 100–103, 113]. The hypothesis of the effort is:

> Software engineering complexity is minimized by engineering different parts and aspects of a software product using the most appropriate kinds of models expressed by means of software languages.

This work assumes that the hypothesis is correct, and by the following three key observations of the domain of cyber-physical systems:

- the need shown by real-time language research to have real-time constraints as first-class programming constructs, which would be best obtained without requiring people to change their existing software tools,
- the widespread use of C++ as a programming language, which fortunately allows for the embedding of other languages by implementing them as C++ active libraries, and
- the importance of automatic integration to ensure consistency among programs generated by model-based tools and legacy/third-party device drivers and libraries,

this work proposes the following hypothesis:

> C++ support for active libraries makes for a platform where different software languages, including a model-based real-time language, are implementable as C++ active libraries and are composable and integrable as and with ordinary C++ libraries seamlessly and automatically.

1.1 The Problem

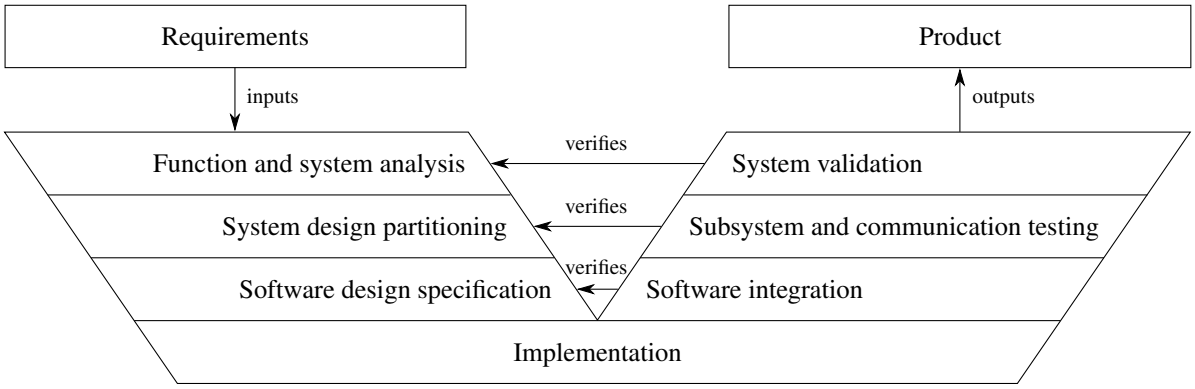


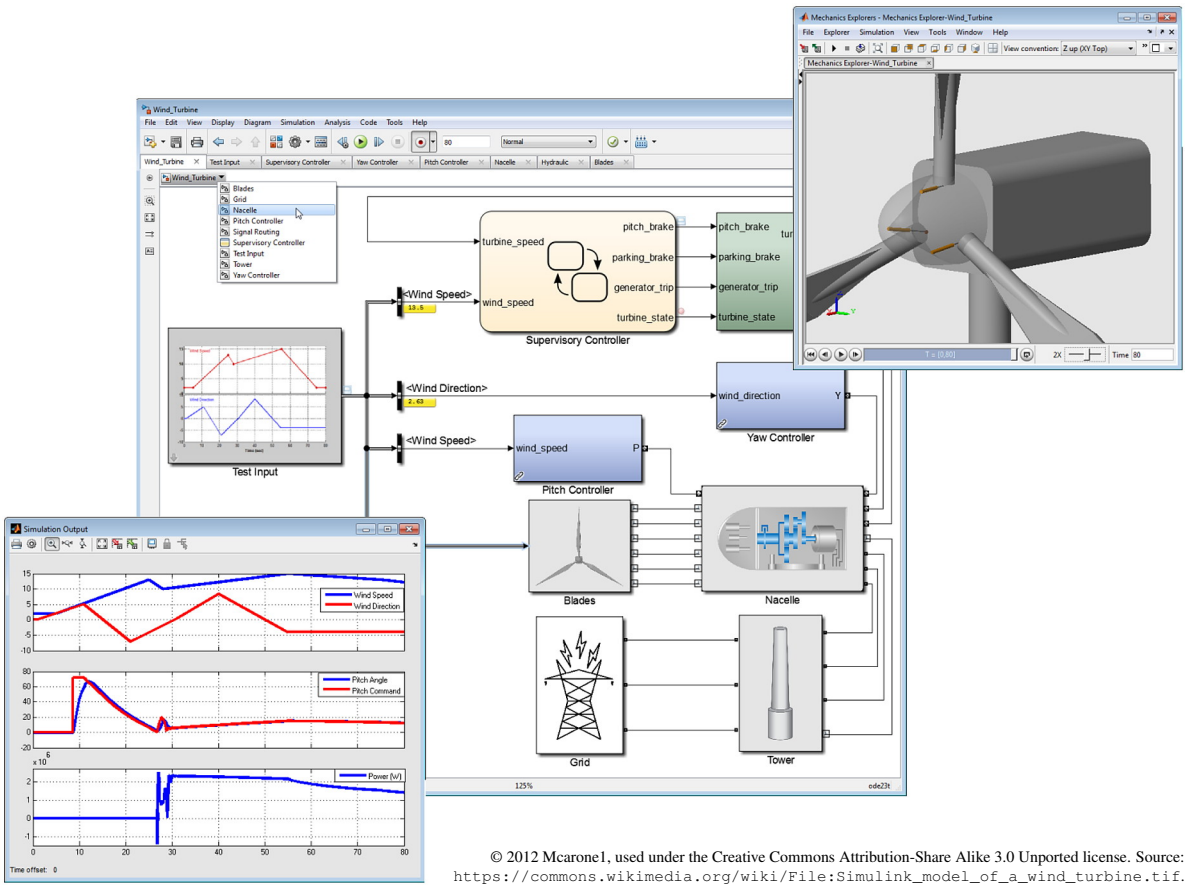
Figure 1.1: The V-model of embedded software engineering [125].

[1] https://en.wikipedia.org/wiki/ISO

The effort taken by MDE, language-oriented programming, and intentional programming can be explained by considering one possible model of software engineering depicted in Figure 1.1. In the V-model, the software engineering process is depicted by a V-shaped diagram whose input is stakeholders' requirements (e.g., an embedded program to control a car brake that is ASIL-D certifiable) and whose output is the required software product. The V-shaped diagram is layered to show that the engineering process involves successive top-down refinements on the diagram's left part through which the given requirements are successively made precise enough to be implemented as embedded programs, while the diagram's right part shows that every refinement layer is responsible for verifying that the embedded programs implement the refinement's analysis/design correctly. Traditionally, computer programming is performed only during implementation, which is depicted as the bottom-most layer of the V-shaped diagram. In contrast, the effort taken by MDE, language-oriented programming, and intentional programming makes computer programming the part of every other refinement layer and even the requirement specification process itself, which provides the input, using the most appropriate *software languages* [69] such that [55, 102]:

- The implementation can be automated completely to produce the required software product by automatic translation and integration of the various software languages.
- Software engineering and maintenance effort is minimized because using the most appropriate software languages to program different aspects of the required software product maximizes *separation of concerns* to minimize complexity during engineering and minimizes the loss of high-level domain-specific information to maximally ease the *concept assignment problem* [11] during maintenance.

As of year 2020, however, the composition and integration of various software languages remain parts of the challenges facing MDE [14].



© 2012 Mcarone1, used under the Creative Commons Attribution-Share Alike 3.0 Unported license. Source: https://commons.wikimedia.org/wiki/File:Simulink_model_of_a_wind_turbine.tif.

Figure 1.2: Using MATLAB®/Simulink® to design the real-time control aspect of a cyber-physical system.

In engineering the real-time control aspect of cyber-physical systems, the use of models is already part of the engineering routine through the use of model-based tools [77, 95], such as MATLAB®/Simulink® [85, 121]. Using MATLAB®/Simulink®, for example, the model of a *plant* (i.e., the physical part of a cyber-physical system)

and the model of a *controller* (i.e., the cyber part of a cyber-physical system) are designed as a *block diagram* (i.e., interconnected planar shapes, such as rectangles) using a graphical user interface (GUI) as shown on the center window depicted in Figure 1.2. A block in the diagram is either made up of a number of more primitive blocks or some primitive block itself (e.g., a constant block or a zero-order hold block or some other primitive block). The primitive blocks, and hence the blocks made up of the primitive blocks, have formal parameters and definitions that can be translated into C programs automatically. Referring to Figure 1.1, the automatic translation of a controller model into a C program means that the bottom-most layer of the V-model (i.e., the implementation) involves no manual work, which is kept to the other refinement layers (e.g., the top-right and center windows depicted in Figure 1.2, which are close to the given requirements). The corresponding verification depicted in Figure 1.1, on the other hand, is performed by first simulating the block diagram and tracing the input and output of the simulated controller model as shown on the bottom-left window depicted in Figure 1.2 (the window's top graph is the trace of the controller's input, while the window's middle graph is the trace of the controller's output). Then, to verify that the C program correctly implements the controller model, the traced output is compared with the trace of the actual output produced by the C program when the C program processes the traced input [83]. The main purpose of simulating the block diagram, however, is to help engineers quickly obtain the right controller that satisfies the given requirements. Therefore, the engineering of the real-time control aspect of cyber-physical systems has shown that the use of the most appropriate kinds of models to engineer a software product not only is practically feasible but also has real and immediate benefits. Nevertheless, the real-time control aspect is just one aspect of a cyber-physical system. Other aspects may not be expressible in model-based tools [95], and the C programs obtained by translating the models automatically may not satisfy either some other implementation constraint (e.g., the computer hardware requires the use of some special algorithm to be fault-tolerant) or some part of the given requirements (e.g., the requirement to use some certified legacy/third-party device driver or library). In other words, the C programs obtained by translating the models automatically are left for manual integration. Manual integration, however, could slip in some inconsistencies, which proved fatal in the maiden flight of Ariane-5 rocket in June 1996 [1].

1.2 The Proposed Solution

To automatically integrate the real-time aspect of cyber-physical systems with the other aspects of the systems, various *real-time languages* have been proposed in the literature as shown in Chapter 3. Nevertheless, the general-purpose non-real-time languages C and C++ have remained the de facto languages to program embedded systems, which include cyber-physical systems [9, 18, 32, 68, 105], including Mars rovers [75] and F-35 jet fighters [25]. Given this reality, C++ indeed presents a unique opportunity to engineer different parts and aspects of a software product using the most appropriate kinds of models because C++ has the potential to be usable for computer programming not only in the bottom-most layer of the V-model (i.e., the implementation) but also in every other refinement layer, possibly including the requirement specification process itself, which provides the input to the software engineering process. The unique opportunity that C++ presents comes in the form of *standard* language features (e.g., the C++ keyword "constexpr" and C++ template specializations and instantiations) that support *active libraries*. A C++ active library is none other than an ordinary C++ library, which comes as a set of C++ header files, that can play an active role when a C++ program using the library is *compiled* by an off-the-shelf *standard* C++ compiler (e.g., GCC or Clang). The active role that a C++ active library can play includes:

- Checking that the C++ library is used correctly.
- Providing intelligent advice on how to use the C++ library correctly.
- Generating better implementation based on the particular way the C++ library is used.
- Answering design questions, possibly by performing simulations.
- Cooperating with external tools in either answering design questions or generating the implementation, or both.

While the first three points are the hallmark of a compiler, the last two points are the hallmark of a model-based tool. In other words, a C++ active library can be used to implement both a software language and a model-based tool with a minimal cost owing to C++ active libraries being processable by off-the-shelf standard C++ compilers and tools (e.g., program analyzers, editors, and debuggers). Furthermore, as C++ active libraries are indeed ordinary C++ libraries, and ordinary C++ libraries are seamlessly composable, C++ active libraries are also seamlessly composable. Therefore, since composition and integration have been highlighted in §1.1 as parts

of the main problems that need to be solved to engineer different parts and aspects of a software product using the most appropriate kinds of models, C++ indeed has the potential to solve them by being a platform for language-oriented programming where different languages that express different kinds of models in every other parts of the V-model other than the implementation layer can be composed seamlessly and integrated automatically by being implemented as C++ active libraries. Additionally, since C++ is one of the de facto languages to program embedded systems, at least in the domain of embedded systems C++ also has the potential to solve the problem of adoption ("if MDE is so good, why its use has not been so widespread?") [101], which in turn would solve the chicken-and-egg dilemma faced by MDE when it comes to efficiency and scalability [55] (in accordance with the economics of optimization [94], MDE tools would be optimized if many users would benefit, but few users would use the MDE tools if they were not optimized).

1.3 The Contributions of This Work

This work has five main contributions with their own respective limitations:

Contribution 1. This work proposes a novel real-time language, called Tice, that differentiates itself from other real-time languages already proposed in the literature by being:

- Processable by off-the-shelf standard C++ tools, including compilers, program analyzers, editors, and debuggers.
- Compilable into programs that are linkable and executable as other programs that are obtained by compiling standard C++ programs.

Consequently, while the active library that implements Tice itself either uses no C++ features that are deemed unsuitable for cyber-physical systems (e.g., exception [25] while noting that recent research has worked toward making exception suitable for use in cyber-physical systems [91]) or uses some in judicious manner (e.g., template instantiations to generate programs), Tice prevents no usage that is permitted by standard C++ compilers, including the use of features that are deemed unsuitable for use in cyber-physical systems. In contrast, other real-time languages are capable of providing only the features that are deemed suitable for use in cyber-physical systems by requiring the use of either their own special compilers/interpreters or non-standard C/C++ compilers. Similarly, while other real-time languages are capable of ensuring that their semantics are followed by the programmers by checking with their own compilers/interpreters that some features (e.g., input/output operations) are not used, Tice can only do so only as far as it is reasonable owing to the fact that the C++ affords its programmers the greatest programming flexibility possible. Aside from that, Tice introduces no novel real-time programming concept but synthesizes the novel concepts that have already been proposed by other real-time languages into a model-based real-time language. Additionally, Tice performs no WCET (worst-case execution time) measurement but expects WCET information to be provided by some other means.

With regard to being processable by off-the-shelf standard C++ tools, Tice has been tested with two well-known off-the-shelf C++ compilers, namely GCC and Clang, both of which starting from version 6, because:

- Both GCC [35] and Clang [82] are compliant with the C++ standard [52].
- GCC is widely-used to program embedded systems [6, 9, 24, 75].
- Clang is used to show not only the portability of Tice and its implementing active library between off-the-shelf standard C++ tools but also how well off-the-shelf C++ compilers process Tice programs.

Other than testing the two compilers systematically, including the resulting programs for their linkability and executability, this work performs no other systematic test on other kinds of off-the-shelf standard C++ tools, such as program analyzers, editors, and debuggers.

Contribution 2. This work shows by Theorem 1 on page 40 that an end-to-end path cannot prevent all data items coming from the path's source from reaching the path's sink if the path's inter-path-segment communication follows the MoCC (model of computation and communication) of either a time-triggered LET (logical execution time) or a BET (bounded execution time) with precedence constraints. While the theorem is considered intuitive and therefore not addressed in [30] for the former MoCC and in [33] for the latter MoCC, the theorem underpins the semantics of a correlation constraint, which is one of the two kinds of real-time constraints expressible in Tice. The theorem, however, is proven for the former MoCC but not the latter MoCC as the former is Tice's MoCC.

Contribution 3. This work shows that the real-time constraints of Tice are decidable based on their semantics because being decidable and not semi-decidable is crucial in a compilation process, which is expected to always terminate within a finite time. This contribution, however, could be limited because it only shows the decidability of Tice.

Contribution 4. This work also implements the Tice language as a C++ active library, analyzes the time complexity of the resulting implementation, and validates the analysis results empirically using GCC and Clang. The empirical validation shows that C++ compilation technology as represented by GCC and Clang has come to the point of performing well in compiling a C++ active library whose algorithms may have an exponential time-complexity. The time-complexity of the C++ active library implementing Tice, however, is not the most optimal one possible because better algorithms could be used to implement Tice better.

Contribution 5. This work shows novel techniques that are needed to implement a model-based real-time language as a C++ active library so that, despite the complexity of the implemented language, the active library keeps being efficient and maintainable, which involves editing and debugging C++ template metaprograms. The techniques, however, might not be directly usable to implement other kinds of software languages. Additionally, the techniques have been tested only with the more recent C++ standards, namely C++14 [51] and C++17 [52].

Lastly, the hypothesis of this work is strengthened by showing that the current C++ compilation technology as represented by GCC and Clang performs well to compile a model-based real-time language whose models can require combinatorial analyses with exponential time-complexity. Additionally, the hypothesis is also strengthened by showing that the C++ active library implementing the model-based real-time language is composable and integrable with ordinary C++ libraries seamlessly and automatically. This work, however, does not address the composability and the integrability of the C++ active library with other C++ active libraries that implement other kinds of software languages.

1.4 The Organization of This Work

As already stated in the preface, this work is intended to be accessible by everyone interested in the world of computers, particularly in software engineering and computer programming. Consequently, Chapter 2 will first present all of the needed material that will make all of the latter chapters accessible. Therefore, after completing this section, readers who already understand:

- The terms “imperative programming”, “declarative programming”, “low-level abstraction”, “high-level abstraction”, “compiler”, “interpreter”, and “operating systems” as well as how the terms are related can jump to §2.2 on page page 11.
- All of the previous terms and the terms “external domain-specific language” (DSL) and “internal/embedded domain-specific language” (EDSL) and how the terms are related can jump further to §2.4 on page page 14.
- All of the previous terms and the terms “C++ template metaprogramming” (C++ TMP) and how the terms are related can jump further to §2.5 on page page 16.
- All of the previous terms and the terms “release time”, “relative deadline”, “WCET”, “period”, “end-to-end delay”, and “real-time scheduling” can skip Chapter 2 entirely.

This work will start to unfold in Chapter 3 by relating itself to other existing work in the literature. This work will then unfold completely by presenting:

- Contribution 1 and Contribution 2 in Chapter 4.
- Contribution 3 and Contribution 4 in Chapter 5.
- Contribution 5 in Chapter 6.

Lastly, Chapter 7 outlines the conclusions and future work.

Please skip this page.

2

Preliminaries

As of year 2020 programs have not yet run computers to conceive and write programs in the way new novels are conceived and written and new dams are engineered and built. Instead, programs run computers to translate or interpret existing programs faithfully, much like faithfully translating or interpreting existing novels to make them accessible to people who understand different languages. The objective of translating or interpreting existing programs, however, is to make them run computers that understand different sets of instructions. In this setting, this work defines the following terms:

Machine language. A set of instructions that a particular machine understands.

Computer. A physical machine that processes input data into output data by following a sequence of instructions in the computer's machine language while facilitating the change of the currently followed sequence with a different sequence of instructions in the computer's machine language (i.e., being programmable).

Program. Information that can run (i.e., be followed by) a computer either directly as given or indirectly after further processing (e.g., being translated or interpreted). Parts of the information that are runnable are called *subprograms* (in contrast, examples of non-runnable parts are English messages to be displayed on the computer's screen and constant values like π). A program is also known as a *code*.

Source program. An existing program to be translated or interpreted.

Interpreter. A program that faithfully interprets for a particular computer by reading source programs on an as-needed basis in the computer's machine language. From the perspective of the users who feed source programs to a computer run by an interpreter, the computer has become another computer with its own machine language, which is none other than the language used to write the source programs, and therefore, an interpreter is also known as a *virtual machine*.

Compiler. A program that faithfully *compiles* (i.e., translates) source programs to new programs called *target programs*. Since a compiler generates new programs, a compiler is also known as a *code generator*.

Source language. A language that is used to write a source program. After the turn of the last millennium, it has become common to write a source program by writing its parts in different source languages as web applications (e.g., Facebook®, Instagram®, and Amazon®) have their source programs written in at least three source languages: HTML, JavaScript, and SQL.

Additionally, the term "X program" where X is the name of a source language is used to mean a source program written in X. Aside from that, the definitions capture the fact that some interpreter may interpret source programs written in some machine language. Similarly, the definitions also capture the fact that some compiler may translate source programs written in some machine language into target programs written in another machine language or some source language. Furthermore, the term "program" is not defined as "a sequence (or a collection or a group or some other synonym) of instructions" because such a definition fails for some kinds of source programs (e.g., Haskell programs, which are written as mathematical expressions, and Prolog programs, which are written as facts and rules).

In the future, when programs would run computers to conceive and write programs in the way new novels are conceived and written and new dams are engineered and built, a source language could be English [4, 46, 78, 99]. Currently, however, a source language is a software language [69], which can be a machine language or a programming language (the source programs are usually translated, for example, C++) or a scripting language (the source programs are usually interpreted, for example, JavaScript). In contrast to English and other natural languages, software languages are engineered to be formal and precise so that they facilitate the faithful translations and interpretations of their source programs by compilers and interpreters. Being formal, dotted and undotted i's mean different things, and therefore, the experience of writing and reading those languages is like that of writing and reading mathematical formulas (e.g., "$1\ \frac{1}{2}$" and "$1 \cdot \frac{1}{2}$" means one-and-a-half and one half, respectively). Being precise, those languages have literal meanings and leave no room for ambiguity, and therefore, a lot of things need to be elaborated and repeated (e.g., "A and B go to X and Y, respectively, by bus" must be written as "A goes to X by bus U; B goes to Y by bus V").

Since software languages are formal and precise, software engineers spend a lot of efforts engineering (i.e., writing) the source programs needed to realize the computer behavior specified by stakeholders in English or

other natural language. Moreover, since the stakeholders' need and computers evolve over time, from time-to-time not only the stakeholders will specify changes to the computer behavior but also different computers will need to realize the desired behavior. Consequently, software engineers also spend a lot of efforts maintaining (i.e., editing) the already written source programs. Therefore, since the dawn of the Computer Age in World War II (i.e., in the 1940s), people have worked on making software engineering easier with at least two important results [97]:

Operating system (OS). A set of programs that manages (i.e., runs) computers and provides a set of software services, such as the service to load and run programs, the service to read input data, and the service to write output data. OSes (e.g., GNU®/Linux® Ubuntu®, Microsoft® Windows®, and Android®) ease software engineering by allowing software engineers to use the provided software services. In doing so, their programs become *portable* across different computers managed by the OSes, such as various smartphones managed by Android®, needing no specific engineering for each specific smartphone (i.e., computer).

To ease software engineering even further, programs can be made portable across different OSes. To that end, work has been done to standardize the software services of different OSes. As a result, programs that use only the software services that are specified by an *OS standard* become portable across different OSes that *implement* the standard. Two prominent OS standards are POSIX®[1], which is well-known in various software industries and implemented by many OSes, and AUTOSAR®[2], which is well-known in the automotive software industry and implemented by a few OSes.

Lastly, the term "executable" is defined as "a program that can be loaded by some OS to run some computer managed by the OS." This definition captures the fact that source programs are executables if some OS can load the source programs to let them run some managed computer after further processing (e.g., being compiled or interpreted).

Source language abstraction. A set of concepts that not only abstracts either computers' machine languages or their sequential manner of executing instructions, or both, but also is engineered to be translatable by a compiler or interpretable by an interpreter faithfully. Source language abstraction eases software engineering by allowing software engineers to write or edit source programs in terms of the concepts that are closer or equal to the concepts used in the requirements (i.e., the concepts that stakeholders use to specify the desired computer behavior), relying on compilers or interpreters to flesh out the details needed by the source programs to run computers.

2.1 Source Language Abstraction

Source language abstraction at the lowest level results in *assembly languages*, and compilers that translate only assembly programs are called *assemblers*. Assembly languages abstract instructions in the machine languages as mnemonic words, numeric and named constants, and operations on the constants. Additionally, assembly languages also provide additional features to ease programming, such as symbolic evaluations. For example, in the x86-32 machine language, which is understood by many computers that use the Intel® and AMD® microprocessors but not by many smartphones that use the ARM® microprocessors, the requirement to evaluate $\sum_{i=1}^{1000} i^2$ (i.e., $1^2 + 2^2 + \ldots + 1000^2$) can be satisfied by the program shown in Figure 2.1(a). When the program shown in the figure is followed by a computer whose machine language is x86-32, the computer executes the program sequentially opcode-by-opcode (i.e., line-by-line) where the semantics of every opcode is shown in Figure 2.1(b). When the computer no longer executes the backward jump in line 11 due to the value in register ECX being already 1000, the result of the evaluation is available in register EAX. If additional requirements were present, the program could be written further to, for example, display the result on the screen. Based on Figure 2.1(a), it should be evident that writing and editing programs in machine languages are laborious and prone to error (i.e., time-consuming and expensive), something that software engineers in the Computer Age's early decades lived through using electric plugs and switches and then with punch cards to make the binary patterns (i.e., patterns of 1s and 0s). Therefore, to ease software engineering, as demonstrated in Figure 2.2, an assembly language abstracts a machine language in the following manner, which is a non-exhaustive list:

- The binary patterns of opcodes, operands, and registers are abstracted as mnemonic words (e.g., `incl` instead of `01000`), numeric constants (e.g., `$0` instead of 32 times of `0`), and named constants (e.g., `%eax` instead of `000`), respectively.

[1] `https://standards.ieee.org/project/1003_1.html`
[2] `https://www.autosar.org`

```
1  10111000 00000000 00000000 00000000 00000000
2  10111001 00000000 00000000 00000000 00000000
3  11101011 00001000
4
5  01000001
6  10001001 11001011
7  00001111 10101111 11011011
8  00000001 11011000
9
10 10000001 11111001 11101000 00000011 00000000 00000000
11 01111100 11110000
```

(a) The program, which is evenly spaced every eight binary digits (i.e., every one byte).

Line	Opcode	Data	Semantics
1	10111	Register: 000; Operand: 00000000 00000000 00000000 00000000	Assign zero to register EAX
2	10111	Register: 001; Operand: 00000000 00000000 00000000 00000000	Assign zero to register ECX
3	11101011	Operand: 00001000	Skip the next eight (1000 in binary is 8 in decimal) bytes (i.e., jump to the start of line 8)
5	01000	Register: 001	Increment register ECX by one
6	10001001	Addressing Mode: 11; Register: 001; Register/Memory: 011	Write ECX's value to register EBX
7	00001111 10101111	Addressing Mode: 11; Register: 011; Register/Memory: 011	Multiply EBX with EBX and store the result in EBX
8	00000001	Addressing Mode: 11; Register: 011; Register/Memory: 000	Sum EAX and EBX and store the result in EAX
10	10000001	Addressing Mode: 11; Opcode: 111; Register/Memory: 001; Operand: 11101000 00000011 00000000 00000000	Compare ECX and one-thousand (00000011 11101000 in binary is 1000 in decimal)
11	01111100	Operand: 11110000	If ECX $<$ 1000, jump backward from the current line's end by skipping 16 bytes, that is, to the start of line 5 (11110000 in binary two's complement is -16 in decimal)

(b) The semantics of the program according to [49] (due to x86-32's little-endian byte-ordering, every space-segmented value in column Operand is to be read segment-by-segment backward but within each segment digit-by-digit forward).

Figure 2.1: A program that is written in the x86-32 machine language to run a computer to evaluate $\sum_{i=1}^{1000} i^2$.

```
1          movl   $0, %eax
2          movl   $0, %ecx
3          jmp    .L2
4  .L1:
5          incl   %ecx
6          movl   %ecx, %ebx
7          imull  %ebx, %ebx
8          addl   %ebx, %eax
9  .L2:
10         cmpl   $1000, %ecx
11         jl     .L1
```

Figure 2.2: Rewriting the program in Figure 2.1(a) in the assembly language of GNU® Assembler, keeping the lines in this figure and in Figure 2.1(a) in a one-to-one correspondence, that is, the compiler will translate line i of this figure into line i of Figure 2.1(a).

```
1  int main() {
2    auto result = 0;
3    for (auto i = 1; i <= 1000; ++i) {
4      result += i * i;
5    }
6    return result;
7  }
```

Figure 2.3: Rewriting the program in Figure 2.2 in C++.

- Variations in operand orders are abstracted as a uniform operand order. For example, while the figure shows that the assembly language's `movl`, `imull`, and `addl` all store their results in their last operands, the corresponding machine instructions do not. Specifically, the corresponding machine instructions of lines 1, 2, and 7 all store their results in column Register (i.e., in their first operands), while those in lines 6 and 8 all store their results in column Register/Memory (i.e., in their second operands).

An assembly language, however, does not abstract a machine language's sequential manner of instruction execution, and therefore, lines 1, 2, ..., 11 of Figure 2.2 correspond to lines 1, 2, ..., 11 of Figure 2.1(a), respectively. Aside from that, as demonstrated in Figure 2.2, an assembly language also facilitates programming by, among other things, evaluating symbols. For example, the symbol `.L2` is used to write `jmp .L2` to make writing and editing easier than if `jmp $8` could be written because the symbol liberates software engineers from the tedious and error-prone task of calculating manually the number of bytes to skip, which may change whenever the assembly program is edited.

While assembly languages improve software engineering experience as already demonstrated, their *abstraction level* can still be raised to ease software engineering even further. For example, the program shown in Figure 2.2 when written in C++ as demonstrated in Figure 2.3 is abstracted further in the following manner, which is a non-exhaustive list:

- The machine registers (e.g., `%eax`, `%ebx`, and `%ecx`) whose numbers are very limited and have fixed names are abstracted as *variables* whose numbers are virtually unlimited and can be named with concepts that are used in the requirements, for example, `i` (it is then the job of a C++ compiler to translate the potentially numerous variables into the few available machine registers).
- The one-dimensional instruction sequence along the vertical, that is, forward line-by-line or jump forward or backward over a number of lines, is abstracted as a two-dimensional text along the vertical and the horizontal. In C++ programs, their vertical dimension consists of *statements*, while their horizontal dimension consists of *expressions*, such as `result += i * i`, and *language constructs*, such as `for (`<a variable declaration or an initializing expression>`;` <an expression to decide whether to loop>`;` <an expression to evaluate after each loop>`)` <a statement>.

While the lines of Figure 2.2 and Figure 2.3 no longer correspond one-to-one due to the dimensionality change, the parts of Figure 2.3 (e.g., `i * i`) still correspond one-to-one with the lines of Figure 2.2. This means that

```
sum [i * i | i <- [1..1000]]
```

Figure 2.4: Rewriting the program in Figure 2.3 in Haskell.

```
int main() {
  return sum(1, 1000, [](auto i) {return i * i;});
}
```

Figure 2.5: Rewriting the program in Figure 2.3 using a C++ library.

the abstraction level of C++ programs is usually still not high enough to write the concepts that are used in a requirement. Indeed, it is the case for the part $\sum_{i=1}^{1000}$ found in the requirement to evaluate $\sum_{i=1}^{1000} i^2$ because the part is written in Figure 2.3 not as a single symbol but as multiple lines with many symbols.

When the abstraction level of a source language is not sufficient to write the concepts that are used in a requirement, the abstraction level can be raised in mainly two different ways [36]:

Engineering another source language. For example, Haskell has been engineered in such a way so that its source programs are written as mathematical expressions as demonstrated in Figure 2.4. The obvious drawback of this approach is the cost that is needed not only to engineer a formal and precise source language with a high abstraction level but also to engineer its compiler or interpreter. The main benefit of this approach, on the other hand, is the possibility to have virtually all kinds of abstractions exactly in the way they are intended to be realized, such as the possibility to write $\sum_{i=1}^{1000}$ exactly as it is, which is one of the goals of *language workbenches* [98] as clearly demonstrated in [54, 110].

Engineering a *library*. For example, a C++ library can be engineered in such a way so that some mathematical expressions can be written in C++ programs in the way they are written in the given requirements as demonstrated in Figure 2.5. The obvious drawback of this approach is the limitations on the kinds of abstractions that are realizable as libraries in a *host language*. For example, C++ as a host language currently does not allow a library's member to assume the symbol $\sum$ as its *identifier*, and hence, the identifier `sum` is used instead in Figure 2.5. Likewise, the expression `i * i` is written with additional symbols in Figure 2.5 to satisfy the host language's requirements (i.e., to make for a valid C++ program). The main benefit, on the other hand, is the zero cost to have a compiler or an interpreter.

2.2 Engineering New Source Languages

From time-to-time, new source languages are engineered to obtain either the desired source language abstractions or the desired compilers or interpreters. Indeed, in a number of cases, the associated cost of engineering and maintaining a compiler or an interpreter is less than the income earned by having a source language abstraction exactly as desired. In control engineering, for example, which engineers programs such as autopilot programs that automatically fly airplanes or run some other vehicles, the source language of MATLAB®/Simulink® is engineered for drawing diagrams of dynamical system models. With its abstraction, the source language allows control engineers to easily model dynamical systems to be controlled along with their controllers and, more importantly, to simulate the systems when being controlled by their controllers using the MATLAB®/Simulink® interpreter. In this way, control engineers can use the simulation results to interactively design the required controllers to quickly obtain the required control performance. Once obtained, control engineers can compile their controller models into executables using a MATLAB®/Simulink® compiler by supplying some information about the computers to run (e.g., the number of processor cores) and some *architectural mapping* (e.g., assigning different parts of the models to run different processor cores). Since the source language abstraction of MATLAB®/Simulink® expedites the work of control engineers who in turn find the price of a MATLAB®/Simulink® license worthwhile to pay, the cost of engineering and maintaining MATLAB®/Simulink® interpreter and compiler is less than the income earned by The MathWorks, Inc., the proprietor of MATLAB®/Simulink®.

On the other hand, in some other cases, the associated cost of engineering and maintaining a compiler or an interpreter is covered not so much by having a high source language abstraction but by some properties of the compiler or interpreter itself. The Java language, for example, is very similar to the C++ language and only slightly more abstract than C++ by removing some C++ features that were deemed unsuitable for most

programmers (e.g., operator overloading and multiple inheritance) who had no need to manipulate computer hardware efficiently (e.g., pointers and the absent of an out-of-bound array access check and garbage collection). Java, however, is engineered to be compiled for an interpreter called JVM (Java virtual machine) that in turn is engineered to be capable of interpreting Java programs for as many different kinds of computers as possible. Therefore, being similar to C++ in the language, the abstraction level, and the use of a compiler to check for programming mistakes but significantly more portable than C++ is the unique selling point of Java that makes it profitable to engineer and maintain both a Java compiler and an interpreter.

In other cases, however, the cost of engineering and maintaining a compiler or an interpreter may not be profitable to obtain the desired source language abstractions. Nevertheless, the cost can be lowered by using a language workbench [7, 31, 48, 67, 74, 96, 104], which then bears much of the cost. A language workbench is a software tool similar to MATLAB®/Simulink® that is used to both model software languages and implement them, for example, by giving their semantics in some software language that the workbench then translates into C programs or interprets using JVM. Therefore, a language workbench is an independent platform for language-oriented programming. Being an independent platform, however, means that a language workbench faces the problem of adoption, which might be exacerbated by the fact that familiar language constructs could mean different things depending on the currently active language in the workbench as demonstrated in [54].

Lastly, since a new source language that is engineered to obtain some desired source language abstraction is usually specific to a certain problem domain (e.g., the domain of control engineering in the case of MATLAB®/Simulink®), such a new source language is known as an *external DSL* (domain-specific language).

2.3 Engineering New Libraries

Much more frequently than engineering new source languages is the engineering of new libraries. While the engineering of a new library can take a bottom-up approach with the goal of keeping programs maintainable by removing duplication, it can also take a top-down approach with the goal of reducing the complexity of programs by defining a stable API (application programming interface). From the perspective of language-oriented programming, defining a stable API amounts to defining a software language whose symbols are the stable API members [109]. And, since a library needs no special compiler/interpreter other than that of the host language and is usually engineered specifically for some problem domain, a library is known as an *internal DSL* or an *embedded DSL* (EDSL).

In contrast to an unstable (internal/private) API, a stable (external/public) API has members that are guaranteed by the library's engineers to have the same semantics despite the evolution of the library. For example, if the library member `sum` shown in Figure 2.5 belongs to a stable API, then despite any change to the implementation of `sum` (Figure 2.3 shows just one out of the many possible implementations of `sum`), the member will keep its semantics to evaluate $\sum_{i=a}^{b} f(i)$ as well as keeping taking the summation start index a as its first formal parameter, the summation end index b as its second formal parameter, and the summation formula $f(i)$ as its third formal parameter. If instead `sum` belongs to an unstable API, then `sum` might change its semantics or formal parameters, for example, by reordering the positions of its formal parameters.

Traditionally, a library can only check that its API member is used correctly at *runtime* (i.e., when the API member already runs some computer). Taking the library member `sum` shown in Figure 2.5 as an example, if there is a requirement to ensure that the first formal parameter (the summation start index) is less than or equal to the second formal parameter (the summation end index), then traditionally the check is performed at runtime and, if the condition is not satisfied, a runtime error can be raised. However, in a number of cases, which include the example shown in Figure 2.5, the check for the correct usage of a library's API member can indeed be done before runtime because the actual parameters of the member is already known before runtime. In Figure 2.5, for example, it is already known before runtime that the actual parameter for the first formal parameter is one while the actual parameter for the second formal parameter is a thousand. Since any check before runtime is traditionally done by a compiler as part of the compilation process, and libraries are not part of a compiler, it follows that traditionally a library cannot check that its members are used correctly before runtime despite the possibility to do so.

Compilers vs. Interpreters

Since a compiler is analogous to a human translator who works with documents over a given period of time while an interpreter is analogous to a human interpreter who help conversing people understand each other throughout their conversation, it is easy to see that:

- A compiler has the advantage over an interpreter by being capable of checking and analyzing its source programs thoroughly so that not only as many errors as possible are caught at *compile-time* (i.e., during compilation before runtime) but also as efficient target programs as possible are produced. On the other hand, a compiler takes more time than an interpreter in processing the source programs to obtain the target programs, and further steps are usually needed after compilation before the target programs can run a computer.
- An interpreter has the advantage over a compiler by being capable of allowing its source programs to immediately run a computer. On the other hand, an interpreter can raise errors at runtime despite the errors being detectable before runtime and usually takes longer to produce computation results than compiled source programs. Nevertheless, raising an error at runtime might be beneficial because the error could be fixed immediately. Furthermore, an interpreter could use runtime information immediately to make its source programs run more efficiently at runtime, while to benefit from the runtime information, a compiler has to recompile the source programs while analyzing the runtime information to produce target programs that are more efficient than what were possible by only analyzing the source programs thoroughly.

In the domain of embedded systems, particularly in the subdomain of cyber-physical systems, a runtime error that can be detected before runtime should indeed be caught and fixed before runtime because embedded systems usually run either autonomously or with minimal supervision, not to mention that a runtime error in cyber-physical systems could have a very grave result. Consequently, software engineers usually employ software languages that have a compiler to program cyber-physical systems. Compilation, however, does not preclude interpretation because nothing prevents a program that is obtained by a compilation to be interpreted (e.g., a Java program must be compiled first before being interpreted by a JVM). Therefore, the objective of employing software languages that have a compiler is to catch as many errors as possible at compile-time, which makes for safer and sounder cyber-physical systems.

Active Libraries

Among all software languages that have language constructs to engineer ordinary libraries and have a compiler, some of them (e.g, Lisp, C++, and Haskell) have language constructs to engineer *active libraries*. Examples of the main language constructs to engineer active libraries are Lisp macros, C++ templates, and Template Haskell, which a Haskell extension that is included by the Glasgow Haskell Compiler (GHC) since version 6. Given a software language that supports an active library, an active library is an ordinary library of the language, which is seamlessly composable and automatically integrable with other ordinary libraries of the language, that at compile-time can check for the correct use of its API members and can generate more efficient implementations of the API members based on their actual parameters [19, 115]. Therefore, C++ libraries, such as the one whose API member is shown in Figure 2.5, can indeed check that their members are used correctly before runtime whenever it is possible to do so (i.e., whenever their actual parameters are known at compile-time).

The Drawbacks of Libraries as Languages

While implementing embedded languages (i.e., libraries as languages) has the advantage over engineering new source languages by the zero cost to have a compiler or an interpreter, an embedded language has the following drawbacks:

- If the host language has no support for active libraries, then the embedded language can only check its usage and makes its programs more efficient only at runtime and, therefore, has the advantages and disadvantages of an interpreter mentioned previously.
- The host language supports active libraries but may either not be capable of supporting certain language features or make it less straightforward to do so. This means that without the appropriate engineering techniques, an embedded language could quickly become unmaintainable. This was indeed the case when Tice was initially engineered and the reason why this work presents the engineering techniques that have made Tice engineering successful in Chapter 6.
- The embedded language must be expressed using the symbols and the syntax of the host language. This, however, can be seen as an advantage because there is no possibility to confuse the semantics of familiar symbols and syntax, which could happen when using a language workbench as demonstrated in [54]. Furthermore, this drawback can be overcome easily by using a sophisticated IDE (integrated development environment), such as Eclipse, as demonstrated in [21].

```
1  template<unsigned n, bool n_is_valid = (n < 13)>
2  struct factorial_of_valid_n;
3
4  template<unsigned n>
5  struct factorial_of_valid_n<n, true> {
6    static const unsigned value = n * factorial_of_valid_n<n - 1>::value;
7  };
8
9  template<>
10 struct factorial_of_valid_n<0, true> {
11   static const unsigned value = 1;
12 };
13
14 template<unsigned n>
15 struct factorial {
16   static const unsigned value = factorial_of_valid_n<n>::value;
17 };
18
19 int main() {
20   return factorial<5>::value;
21 }
```

Figure 2.6: A C++ template metaprogram that implements a factorial function.

2.4 C++ as a Platform For Language-Oriented Programming

Since C++ active libraries are capable of implementing any algorithm (i.e., *Turing-complete*) [20], they can be used to *implement* software languages (i.e., to allow for the expressions of the languages, to validate the expressions of the languages, and to transform the expressions of the languages into programs). Additionally, since C++ active libraries are seamlessly composable and automatically integrable, software languages implemented as C++ active libraries are also seamlessly composable and automatically integrable. Therefore, C++ can be used as a platform for language-oriented programming [120]. To implement a software language as a C++ active library, however, knowledge of *template metaprogramming* (TMP) is needed and will be provided in the rest of this section (a more detailed treatment is available in Chapter 10 of [20]).

To show how a C++ active library is implemented in terms of C++ templates, a C++ active library to evaluate the factorial function $n!$ for any nonnegative integer n less than or equal to 12 is implemented using C++ templates in Figure 2.6 between lines 1 and 17 and is used in line 20. The limit of 12 is imposed because, assuming that the largest value that an `unsigned` variable can store is about four billions ($2^{32} - 1$ to be exact), the value of 13! being over six billions will overflow the variable. Additionally, the active library has two API members. The first API member is the private API member `factorial_of_valid_n` that is implemented between lines 1 and 12, while the second API member is the public API member `factorial` that is implemented between lines 14 and 17. The purpose of the public API member is to give the private API member the chance to check whether the public API member is used correctly (i.e., to check that the actual parameter of the template instantiation, which is five in line 20, is valid, namely not greater than 12). On the other hand, the purpose of the private API member is to compute the factorial of any valid n by the formula that is given in (2.1).

$$\texttt{factorial\_of\_valid\_n} : \{\, n \in \mathbb{N} \mid n \leq 12 \,\} \to \mathbb{N}$$

$$\texttt{factorial\_of\_valid\_n}(n) = \begin{cases} n \cdot \texttt{factorial\_of\_valid\_n}(n-1) & \text{, if } n \neq 0 \\ 1 & \text{, otherwise} \end{cases} \tag{2.1}$$

When the C++ program shown in Figure 2.6 is compiled by a standard C++ compiler, the compiler will follow the standard template instantiation rules upon processing line 20 by first looking up for the primary template `factorial`. Since the primary template is found between lines 14 and 17 with a formal parameter that matches the given actual parameter and without any template specialization, the compiler instantiates the primary template's definition with five as the actual parameter of `n`. Since the instantiation defines the accessed field `value` with `const` qualifier, the compiler proceeds further to obtain the accessed field's constant value by processing the template instantiation in line 16. Following the standard template instantiation rules, the pri-

```
1  $ g++ -fno-exceptions -fno-asynchronous-unwind-tables -S -o- figure_2.6.cpp
2          .file   "figure_2.6.cpp"
3          .text
4          .globl  main
5          .type   main, @function
6  main:
7          pushq   %rbp
8          movq    %rsp, %rbp
9          movl    $120, %eax
10         popq    %rbp
11         ret
12         .size   main, .-main
13         .ident  "GCC: (Ubuntu 9.2.1-17ubuntu1~16.04) 9.2.1 20191102"
14         .section        .note.GNU-stack,"",@progbits
```

Figure 2.7: The compilation of the program shown in Figure 2.6 into an assembly program.

mary template `factorial_of_valid_n` is looked up and found between lines 1 and 2 to have two formal parameters, the latter of which has a default value. Since the template instantiation in line 16 provides only one actual parameter, the compiler automatically resolves the second actual parameter by evaluating the default value's expression, which results in the boolean true. Once the compiler resolves all of the actual parameters of the instantiation in line 16, the compiler sees that the primary template has two template specializations. The first template specialization between lines 4 and 7 is specialized to match any instantiation of the primary template whose second actual parameter is the boolean true, while the second template specialization between lines 9 and 12 is specialized to match any instantiation of the primary template whose first and second actual parameters are the integer zero and the boolean true, respectively. Since the actual parameters of the instantiation in line 16 match the first specialization better (the first actual parameter being five fails to match the second template specialization's first parameter, which is zero), the compiler instantiates the definition of the first specialization. Since the instantiation defines the accessed field `value` with `const` qualifier, the compiler proceeds further to obtain the accessed field's constant value by processing the expression in line 6, which involves the instantiation of template `factorial_of_valid_n` whose first actual parameter is now four instead of five. As a result, the compiler repeats the steps to instantiate template `factorial_of_valid_n` all over again, starting from looking up its primary template until the compiler arrives at this step again to instantiate template `factorial_of_valid_n` with the first actual parameter being three instead of four. The repetition will finally end when the compiler arrives at this step again to instantiate template `factorial_of_valid_n` with the first actual parameter being zero. When the first actual parameter is zero, instead of instantiating the definition of the first specialization as before, the compiler will instantiate the definition of the second specialization, which matches the first actual parameter better than the first specialization. And, since the const-qualified field `value` of the second specialization's definition has an expression that involves no further template instantiation but just the integer one in line 11, the repetition comes to an end. Afterwards, the compiler will start backtracking by first evaluating the instantiation of `factorial_of_valid_n<n - 1>::value` in line 6 to the integer one resolved in line 11 and evaluate its multiplication with another integer one that is given as the actual parameter for `n` in the preceding instantiation of `factorial_of_valid_n<n - 1>::value` in line 6. The multiplication result is then taken as the value of the const-qualified field `value` of the preceding instancing of `factorial_of_valid_n<n - 1>::value` before backtracking further by performing the same evaluation steps all over again. The compiler finally stops backtracking once it arrives at the initial step in line 20 and evaluates `factorial<5>::value` to 120, which is none other than 5!, as shown in Figure 2.7 line 9.

> As the evaluation of 5! takes place at **compile-time** instead of at runtime, Figure 2.7 shows an assembly program that **has no instruction to compute 5!** but a single instruction in line 9 to store the value 120 in register EAX.

Therefore, Figure 2.7 demonstrates the capability of a C++ active library to efficiently implement an API member at compile-time, which is the hallmark of an optimizing compiler (e.g., GCC and Clang). Note that in Figure 2.7 line 1, the optimizing compiler is not requested to perform any optimization because no *optimization option* (e.g., `-O2` or `-Os`) is given. Despite the compiler not being requested to perform an optimization, however, the active library can still perform an optimization by evaluating 5! at compile-time.

On the other hand, the capability of a C++ active library to check for the correctness of its usage can be

```
1 $ g++ -fno-exceptions -fno-asynchronous-unwind-tables -S -o- figure_2.6-err.cpp
2         .file   "figure_2.6-err.cpp"
3 figure_2.6-err.cpp: In instantiation of 'const unsigned int factorial<13>::value':
4 figure_2.6-err.cpp:20:25:   required from here
5 figure_2.6-err.cpp:16:25: error: incomplete type 'factorial_of_valid_n<13, false>'
6  used in nested name specifier
7    16 |   static const unsigned value = factorial_of_valid_n<n>::value;
8       |                         ^~~~~
```

Figure 2.8: The compilation of the program shown in Figure 2.6 when line 20 instantiates `factorial<13>`.

demonstrated by replacing the template instantiation in Figure 2.6 line 20 with `factorial<13>`. Since the public API member `factorial` is incorrectly used, the compilation of the program correctly fails as shown in Figure 2.8. The compilation fails because, after the compiler has followed all of the initial steps described previously to reach Figure 2.6 line 16 for the very first time and has resolved all of the actual parameters for the very first instantiation of `factorial_of_valid_n<n>` in that line, the second actual parameter is resolved to the boolean false due to 13 being greater than 12. Since none of the two template specializations of `factorial_of_valid_n` have their second parameter match the boolean false, the compiler instantiates the definition of the primary template of `factorial_of_valid_n` in accordance with the standard template instantiation rules. However, since the primary template is purposefully left undefined, the compiler raises the error shown in Figure 2.8 and terminates, preventing the incorrect use of the public API member `factorial`.

Furthermore, a C++ active library is capable of not only checking the correctness of its usage but also helping software engineers to effectively and efficiently figure out their mistakes. As shown in Figure 2.8, it is easy to see in line 3 that the error happens when the public API member `factorial` is being used for the exact reason shown in line 5 by the identifier of the private API member, namely that the predicate `valid_n` fails to hold for `n` being 13 as indicated in line 7.

Active Library Templates and Their Suitability in Programming Cyber-Physical Systems

The assembly program shown in Figure 2.7 and the error message shown in Figure 2.8 show the following important characteristics of the templates used by a C++ active library:

- The templates are discarded at compile-time, and therefore, instead of bloating the resulting programs by many slightly different subprograms produced by template instantiations with slightly different actual parameters, which is the usual objection against using templates in programming cyber-physical systems [75], the use of templates by C++ active libraries results in more efficient programs as demonstrated in Figure 2.7 line 9.

- The templates are capable of not only preventing the incorrect usage of the public API members of a C++ active library but also providing error messages that help to debug effectively and efficiently. Therefore, instead of making debugging harder, which is also the usual objection against using templates, the use of templates by C++ active libraries makes debugging easier, particularly by catching as many errors as possible at compile-time.

In other words, while the use of C++ templates in general might be unsuitable in programming cyber-physical systems, the use of C++ templates by C++ active libraries that in turn are used to program cyber-physical systems should be encouraged instead of being discouraged.

2.5 The Real-Time Aspect of a Program

To show the real-time aspect of a program, Figure 2.9 shows a C++ program that in line 16 waits for an integer b to arrive in the standard input using the public API member `scanf` of the *C++ standard library* `cstdio`, which is included in line 1. Once an integer arrives in the standard input (e.g., by typing some integer using the keyboard and pressing enter), the `scanf` *function call* in line 16 will *return* and assign the number of integers taken from the standard input to the variable `integer_count`. The *return value* of the `scanf` function call will then be checked in line 17 using an *if-statement* that evaluates a boolean expression that has two *subexpressions*: `integer_count == 1` and `b <= 1000`. The former subexpression checks whether the `scanf`

```
1  #include <cstdio>
2
3  namespace {
4    inline auto sum(int from, int to, int f(int)) {
5      auto result = 0;
6      while (from <= to) {
7        result += f(from++);
8      }
9      return result;
10   }
11 }
12
13 int main() {
14   using namespace std;
15   int b, integer_count;
16   integer_count = scanf("%d", &b);
17   if (integer_count == 1 && b <= 10) {
18     return sum(1, b, [](auto i) {return i * i;});
19   }
20   return 0;
21 }
```

Figure 2.9: A C++ program that waits for an integer b to arrive in the standard input before computing $\sum_{i=1}^{b} i^2$.

function matches one integer in the standard input. If it is not the case, since the *boolean and-operator* (i.e., `&&`) that operates on the two subexpressions performs a *lazy evaluation* (i.e., short-circuiting) by not evaluating the next subexpression once the result is already known, which in this case is the boolean false, the program will immediately terminate by jumping to line 20. Otherwise, the second subexpression is evaluated to check whether the integer taken from the standard input is less than or equal to ten. If it is not the case, the whole boolean expression evaluates to the boolean false as before, and the program will also immediately terminate by jumping to line 20. Otherwise, the program will proceed to line 18 to terminate by returning the value returned by calling the function `sum`.

Suppose the program shown in Figure 2.9 controls the emergency release of a control rod in a nuclear-fission power plant such that, if the emergency release button is pressed, a control rod of the right kind will be released within 100 ms, which is a *relative deadline*. Suppose the embedded system is designed such that, if the button is pressed, some integer will be made available in the standard input, and the program has to terminate returning some integer that will select the control rod to be released. Then, the *time point* at which the emergency release button is pressed is called the *release time*, which is also known as the *activation time*, of a *job*. A job has a start time, a finish time, a deadline, and a WCET. While ideally the start time of a job coincides with the job's release time, in practice, the start time is later than the release time due to several factors. Physically, the speed of light limits the fastest possible propagation of the electric signal generated by the pressed button to manifest as a processor interrupt in the computer hardware. Once the cyber part is entered by the processor interrupt, some further delay may await if another program is running the processor because the operating system needs to save the program's context before switching to the program shown in Figure 2.9, a process that is known as a *context switch*. Hence, once the emergency button press releases a job, the job's start time is the time point at which the program shown in Figure 2.9 resumes its execution within the function `scanf` called in line 16. On the other hand, the job's finish time is the time point at which the program terminates returning some integer obtained in line 18. The job's finish time is therefore later than the time point at which the function `sum` returns in line 18. Being real-time, however, a *real-time guarantee* must exist that the job's finish time will never be later than the job's deadline, which is none other than the time point that is obtained by adding the relative deadline (100 ms in this example) to the release time.

As shown in Figure 2.10, the real-time guarantee can be obtained by considering a job's release time as time zero and adding the job's WCET to the latest possible job's start time $t_{s,max}$, which ideally always coincides with the job's release time at time zero. Since a job's WCET guarantees the longest possible time that a job takes to finish its computation, adding a job's WCET to a job's latest possible start time gives the job's latest possible finish time $t_{f,max}$. Then, since the job's deadline d is none other than adding the relative deadline D to the job's release time, which is taken to be zero, a real-time guarantee exists if $t_{f,max} \leq d$ with $d = D$ if the

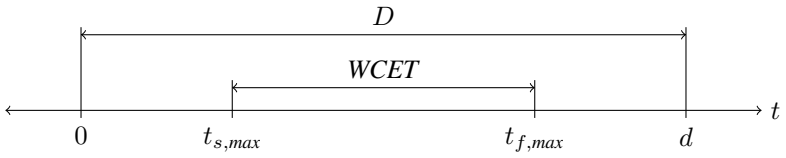


Figure 2.10: The real-time properties of a job whose release time is taken to be time zero.

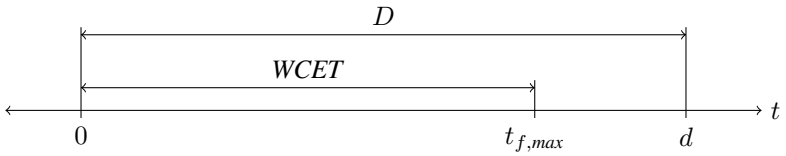


Figure 2.11: The simplification of Figure 2.10 to easily determine a job's real-time guarantee.

job's release time is taken to be time zero. To further simplify the real-time properties of a job, a job's $t_{s,max}$ is included in the job's WCET and the job's start time is taken to always coincide with the job's release time as shown in Figure 2.11. By the simplification, it is easier to determine whether a job has a real-time guarantee by the simple inequality $WCET \leq D$ (i.e., the real-time guarantee exists whenever the job's WCET is not greater than the relative deadline). Lastly, while a relative deadline is a requirement that comes from the physical world and is therefore usually given to software engineers as illustrated in Figure 1.1, a job's WCET is worked out by software engineers because a job's WCET depends on several factors, such as the job's program (e.g., based on Figure 2.9 lines 6–8, the `sum` function call in line 18 definitely takes longer to compute as the actual parameter of b takes on a greater value), the computer hardware that the program runs (e.g., how fast the processor executes the job's program), and the program's *execution environment* (e.g., the existence of other programs running the same computer hardware, such as an operating system).

While the program in Figure 2.9 is useful to show the real-time aspect of a program, a program controlling a cyber-physical system rarely has a single job. Instead of being *aperiodic* as the program in Figure 2.9, a program controlling a cyber-physical system is *periodic* by going through the same part of its program over and over again, each time instantiating a slightly different job. For example, the program in Figure 2.12 can be seen as controlling a cyber-physical system by taking a sensor's measurements from the standard input and sending commands to an actuator through the standard output over and over again until the sensor no longer provides further measurement data (i.e., when the equality operator in line 17 evaluates to the boolean false). In this case, referring to Figure 2.13, whenever the sensor places an integer in the standard input, a job is instantiated whose release time can be taken as time zero and whose finish time t_f is the time point at which the program calls the function `scanf` in line 17 again to take the next sensor's measurement. On the other hand, locally to the program, the *end-to-end delay* Δ of the sensor's measurement *data item* is the duration between the job's release time and the time point t_w at which the function `printf` in line 19 outputs some integer to the standard output. Lastly, the job's deadline can be taken as the period T by which the sensor places the next measurement in the standard input. For example, if $T = 200$ ms, then the sensor will place one new measurement data item in the standard input once every 200 ms, and therefore, one job is also released once every 200 ms and must finish before the sensor places the next data item in the standard input.

Instead of using the term program as in this section, real-time literature [8, 16, 73] uses the term *task* to mean the same thing. For example, the aperiodic program shown in Figure 2.9 is called an aperiodic task, while the periodic program shown in Figure 2.12 is called a periodic task. A task, however, can be either a *real-time task* (RT task) or a *non-real-time task* (NRT task). One of the objectives of real-time research is to find a *scheduling algorithm* that allows as many RT tasks as possible to run the same computer hardware while respecting their deadlines (a task's deadline is respected if all of the jobs generated by the task always finish not later than their deadlines). Furthermore, for every scheduling algorithm, real-time research also endeavors to obtain a sufficient and necessary *schedulability test*. Given a set of RT tasks to be scheduled by some scheduling algorithm, a schedulability test determines without executing the scheduling algorithm whether the *taskset* is *schedulable*, that is, all of the RT tasks in the set will respect their deadlines when scheduled using the algorithm. If a *sufficient* schedulability test decides that a taskset is schedulable using some scheduling algorithm, then the taskset indeed will be schedulable. However, if a sufficient schedulability test decides that a taskset is not schedulable using some scheduling algorithm, then it can be the case that the taskset is actually schedulable. On the other hand,

```
1  #include <cstdio>
2
3  namespace {
4    inline auto sum(int from, int to, int f(int)) {
5      auto result = 0;
6      while (from <= to) {
7        result += f(from++);
8      }
9      return result;
10   }
11 }
12
13 int main() {
14   using namespace std;
15   setbuf(stdout, nullptr); // prevents output buffering
16   int b, integer_count;
17   while ((integer_count = scanf("%d", &b)) == 1) {
18     if (b <= 10) {
19       printf("%d\n", sum(1, b, [](auto i) {return i * i;}));
20     }
21   }
22   return 0;
23 }
```

Figure 2.12: The program in Figure 2.9 with lines 16–19 being slightly modified for a control loop.

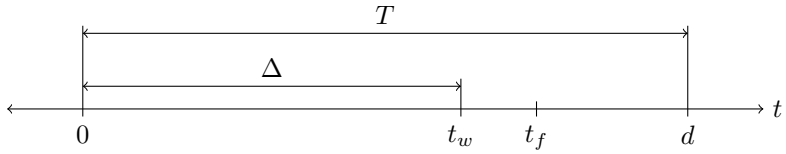


Figure 2.13: The local end-to-end delay of a data item processed by the program in Figure 2.12.

if a sufficient *and necessary* schedulability test decides that a taskset is not schedulable using some scheduling algorithm, then the taskset indeed is not schedulable. Lastly, real-time research also endeavors to accurately measure task WCET or investigate the kinds of computer hardware that makes task WCET more predictable.

Please skip this page.

3

The State of The Art

Many real-time languages have been proposed in the literature [10, 17, 34, 38, 39, 44, 47, 53, 65, 81, 84, 107, 108], starting in year 1983 with Esterel up to year 2018 with Timed C. All of them, however, have one thing in common: their programs cannot be compiled with off-the-shelf standard C++ compilers (e.g., GCC and Clang), which also means that their programs cannot be processed by off-the-shelf C++ tools (e.g., editors, debuggers, and program analyzers). Another commonality among the proposals is that they all highlight the importance of having real-time constraints as first-class programming constructs. However, since they all require the use of new software tools rather than reusing the existing tools already used in the software industry, they all experience the problem of adoption. Hence, although having real-time constraints as first-class programming constructs is indeed important, the intended users of the programming constructs have not adopted the proposals in their routine engineering practice. In contrast, this work brings some of the proposals as close as possible to the intended users by synthesizing them into a model-based real-time language called Tice that in turn is implemented as a C++ active library that is readily usable by the intended users owing to the fact that a C++ active library is just another ordinary C++ library [20, 23, 27, 41, 114] and the fact that C++ is one of the de facto languages to program real-time systems, in particular embedded systems [9, 18, 32, 68, 105]. The proposals that are synthesized into Tice are as follows:

- Giotto [44] proposes a real-time language that abstracts the real-time aspect of a program as a *logical execution time* (LET) that is time-triggered. The abstraction of the real-time aspect of a program as a logical time itself has seen success in the commercialization of *synchronous languages* (e.g., Esterel [10] and Lustre [17]) through SCADE Suite [3]. In contrast to synchronous languages that use *zero execution time* (ZET) that completely abstracts the actual passage of time as *logical ticks* and that assumes the *synchronous hypothesis* (i.e., every computation is started at the arrival of a logical tick and finishes before the arrival of the next logical tick), Giotto's time-triggered LET does not abstract the actual passage of time but abstracts the input/output activities performed by a real-time task to occur only at time points that are multiples of the task's period (i.e., if the task's period is P and the time point of the task's first release is t_0, then input/output activities occur only at $t_0, t_0 + P, t_0 + 2P, \ldots$) [64]. Considering the success of the synchronous languages as well as the need to work with the actual passage of time as demonstrated by the adoption of time-triggered LET in the automotive industry [12, 26], Tice takes Giotto's time-triggered LET as its *model of computation and communication* (MoCC). For certain kinds of computation, however, Tice lowers the time-triggered LET abstraction to the *bounded execution time* (BET) abstraction [64], which allows input/output activities to occur at any time point.

- TCEL [47] makes the key observation that real-time constraints should be applied only on externally observable events (e.g., an input signal received by a sensor and some desired action produced by an actuator) to maximize the number of design alternatives that are available in engineering the program that processes sensor input into actuator output. Tice takes the key observation to provide not only real-time constraints but also sensors and actuators as first-class programming constructs.

- A TCEL follow-up paper [40] proposes three kinds of real-time constraints: freshness, correlation, and separation. Since Tice already takes time-triggered LET as its MoCC, which dictates the rate at which a task produces its output and makes the third kind of real-time constraints (i.e., separation) not applicable, Tice takes only freshness, which in this work is called end-to-end delay, and correlation. According to the nomenclature that is introduced in [30], the semantics of the end-to-end delay constraints that Tice can express is called last-to-first.

A number of C++ active libraries have also been proposed in the literature:

- Deters, et al. [23] proposes an active library that can express a Liu-Layland taskset to be scheduled using the *rate-monotonic* scheduling algorithm [72]. A Liu-Layland taskset is a set of *multi-periodic* real-time tasks (i.e., different periodic tasks in the set can have different periods). At compile-time, the active library checks whether an expressed taskset is schedulable using the *response-time analysis* [71]. If the response-time analysis decides that the taskset is not schedulable using rate-monotonic, the compiler raises an error and terminates. Otherwise, the active library can generate the rate-monotonic schedule of the taskset offline so that at runtime, the tasks will run a computer according to the rate-monotonic schedule efficiently due to no need to run and no overhead of running the scheduling algorithm itself. The active

library also allows the tasks expressed in a taskset to be given a droppability priority so that at compile-time, if the active library finds the taskset to be not schedulable, the active library will use the droppability priorities of the tasks to remove one task from the taskset and then decide whether the resulting taskset is schedulable. The compiler will eventually raise an error and terminates if every droppable task has been removed and the resulting taskset is still not schedulable. In comparison, Tice provides a real-time abstraction level that is higher than the active library proposed by Deters, et al. because the MoCC of a Liu-Layland taskset gives no predictability about the input/output activities that will be performed by the set's tasks, which is not the case with Tice's MoCC. And, instead of using one scheduling algorithm, one schedulability analysis, and one taskset optimization mechanism, which is the droppability priority, Tice's MoCC in making input/output activities occur only at predictable time points and the fact that Tice requires the architectural description of the target hardware (i.e., the computer hardware that a set of tasks will run) allows for a greater flexibility in mapping an expressed taskset, including the use of more than one scheduling algorithm and its corresponding schedulability analysis and more than one taskset optimization strategy.

- Veldhuizen [114] proposes a C++ active library for making C++ programs much more efficient in matrix computations to rival the de facto language of scientific computing—Fortran [116]. The active library is relevant both to the hypothesis of this work and to the MDE adoption problem highlighted in [55]. With regard to the hypothesis of this work, the active library strengthens the hypothesis by showing that, toward C++ as a platform for language-oriented programming, a scientific computing language is also implementable as a C++ active library. Furthermore, the active library shows that programs as efficient as those obtained by the de facto language (Fortran) and its compilers are obtainable by a C++ compiler and a C++ active library [116]. Consequently, this shows that the adoption problem highlighted in [55] is solvable by C++ active libraries because C++ active libraries can produce efficient programs with the only cost of engineering the active libraries themselves due to existing standard C++ software tools being automatically reusable. Lastly, as statistical machine learning being increasingly used to make for more intelligent cyber-physical systems, and matrix computations are heavily used in statistical machine learning, the C++ active library proposed by Veldhuizen and Tice could be used synergistically in engineering cyber-physical systems where Veldhuizen's active library is used to engineer the needed matrix computations and Tice is used to describe the real-time aspect of the matrix computations.

- Gil and Lenz [41] propose a C++ active library that implements a database query language to interact with relational databases. At compile-time, the active library checks whether the expressed database queries are correct based on the schema of the relational database to access. In case of an incorrect query, the compiler will raise an error and terminates. Otherwise, the active library generates the needed program to access the relational database at runtime. The proposed active library strengthens the hypothesis of this work by showing that, toward C++ as a platform for language-oriented programming, a database query language is also implementable as a C++ active library. Furthermore, as intelligent cyber-physical systems will need to access some relational databases, the proposed active library and Tice could be used synergistically as well in engineering cyber-physical systems where the active library is used to engineer the needed database accesses and Tice is used to describe the real-time aspect of the database accesses.

Proposals also exist to make C++ serve as a modeling tool like MATLAB®/Simulink®:

- Tuscherer, et al. [112] propose to express MATLAB®/Simulink® models directly in C++ programs so that the models can be seamlessly composed and automatically integrated with the C++ object models for submicroscopic traffic-flow simulations of autonomous-driving vehicles. Similarly, Tice allows a model of the real-time aspect of a program to be expressed in a C++ program directly and to be used at compile-time to answer queries related to end-to-end delays and correlations by means of applying different real-time constraints interactively.

- C++ software engineers have also expressed their intention to bring MATLAB®/Simulink®-like feature as a C++ active library [43]. If their intention materializes, Tice can be used as a means to implement the semantics of a MATLAB®/Simulink®-like diagram that is expressed directly in a C++ program.

And, in addition to the basic techniques to engineer C++ active libraries shown in §2.4, the following engineering techniques have also been proposed:

- Expression templates proposed by Veldhuizen [114] have been used widely [27, 41]. Expression templates use C++ operator overloading to automatically construct the *abstract syntax tree* (AST) of a C++

expression. The constructed AST is then used to check whether the expression is indeed correct according to some domain-specific rules (e.g., the dimensions of two matrices involved in a multiplication) and to generate an efficient implementation of the expression by exploiting domain-specific knowledge (e.g., a matrix multiplication can indeed be done more efficiently when the multiplied matrices happen to satisfy some conditions). Therefore, expressions templates are very suitable to express the *functional aspect* of a program (e.g., matrix computations and database accesses) as the focus is on expression evaluations. Tice, on the other hand, deals with the *non-functional aspect* of a program, and therefore, finds expression templates not beneficial in implementing Tice as a C++ active library.

- Porkoláb [86] proposes to generate C++ template metaprograms from Haskell snippets inserted in C++ programs. Indeed, the basic techniques to engineer C++ active libraries shown in §2.4 are also the basic techniques used in the *functional* programming language Haskell. This engineering technique, however, requires a new software tool, which faces the adoption problem, and makes C++ source programs look foreign, which is intrusive. Therefore, Tice is implemented using techniques that need no new software tool and are not intrusive.

- Borok-Nagy, et al. [13] propose to instrument C++ template metaprograms so that they are debuggable using a new software tool. In addition to the adoption problem that a new software tool always faces, debugging requires the effort to figure out the specific reason of an error as well as its location. Tice, on the other hand, is implemented using techniques that not only need no new software tool but also can accurately point out the location of an error as well as the reason.

Lastly, the use of language-oriented programming has been shown to make for safer and sounder cyber-physical systems:

- Hickey, et al. [45] in engineering the autopilot program of a cyber-physical system started by engineering the most appropriate languages that are embedded in Haskell, reaping the benefits of more productivity in engineering the safety-critical software product as well as a more reliable software product. This work, on the other hand, takes the first step in embedding real-time languages in C++.

- Voelter, et al. [117–119] have been using a language workbench to develop embedded programs with positive results. This work, on the other hand, takes a bottom-up approach. Instead of starting with an independent platform for language-oriented programming, which faces the problem of adoption, this work takes a software tool already in widespread use to engineer embedded systems, namely off-the-shelf standard C++ compilers, to be the platform for language-oriented programming.

Please skip this page.

4 *Tice: a Model-Based Real-Time Language Embedded in C++*

Figure 4.1 shows how a C++ program that embeds a Tice program looks like. The program shown in Figure 4.1 is indeed the program already shown in Figure 2.12 with the following changes:

- An additional C++ library is included in Figure 4.1 line 1.
- The function `main` in Figure 2.12 is renamed to `fn`, placed in Figure 4.1 lines 21–30, and slightly modified by leaving the call to function `setbuf` in Figure 4.1's function `main`, replacing the *while-statement* with an if-statement, and the *return-statement* with an *else-clause*.
- Figure 4.1 lines 5–11 are new and embed a Tice program.
- Figure 4.1 lines 35–37 are new and execute the embedded Tice program.

More importantly, Figure 4.1 highlights the fact that the shown program is a standard C++ program [52]. Lastly, the term *Tice library* means some implementation of Tice, which in this work is the particular implementation at [87]. And, every Tice program is expressed in a C++ program using the public API members of Tice library, which are the *concrete syntax* of Tice.

```
1  #include <tice/v1.hpp>
2  #include <cstdio>
3
4  namespace {
5    using namespace tice::v1;
6    void fn(); // forward declaration
7    Program<
8      HW<Core_ids<0, 1, 2, 3>>,
9      Node<Comp(&fn, Ratio<10, 1000> /* 10 ms */),
10           Ratio<100, 1000> /* 100 ms */>
11   > p;
12
13   inline auto sum(int from, int to, int f(int)) {
14     auto result = 0;
15     while (from <= to) {
16       result += f(from++);
17     }
18     return result;
19   }
20
21   void fn() {
22     using namespace std;
23     int b, integer_count;
24     if ((integer_count = scanf("%d", &b)) == 1) {
25       if (b <= 10) {
26         printf("%d\n", sum(1, b, [](auto i) {return i * i;}));
27       }
28     }
29     else p.stop();
30   }
31 }
32
33 int main() {
34   std::setbuf(stdout, nullptr); // prevents output buffering
35   p.run();
36   if (p.get_error_code()) return p.get_error_code();
37   p.wait(); // until p.stop() is called
38 } // C++ indeed allows for a function main that ends without a return statement
```

Figure 4.1: Rewriting the program in Figure 2.12 using Tice library that embeds a Tice program in lines 7–11.

The embedded Tice program in lines 7–11 expresses the most minimal Tice program possible. The Tice program is constructed using the public API member `Program` with two actual parameters. The first actual parameter is the *architectural description* of the target hardware and is specified using the public API member `HW`, while the second actual parameter is a *computation node* and is specified using the public API member `Node`. This work uses the term "computation node" instead of the term "task" because depending on the architectural description of the target hardware, a computation node can be implemented by more than one task. The architectural description in Figure 4.1 line 8 describes the fact that the target hardware has four processor cores available to execute every computation node. The available processor core IDs are specified using the public API member `Core_ids`. On the other hand, the computation node in lines 9–10 specifies the computation as its first actual parameter and the computation period as its second actual parameter. The computation in turn is specified using the public API member `Comp` that takes a pointer to a C/C++ function as its first actual parameter and the function's WCET as its second actual parameter.

> The function's WCET is the longest possible duration that the function takes to return after being called. In particular, the function's WCET accounts for the longest possible *blocking time* experienced by the function `scanf` (i.e., the longest possible duration that the function `scanf` takes to wait for the availability of a data item in the standard input).

The function's WCET is specified in seconds using the public API member `Ratio` that expresses the rational number $10/1000$ (i.e., the function's WCET is 10 ms). Similarly, the computation period is specified using the public API member `Ratio` to be 100 ms to mean that the computation will be executed once every 100 ms provided that the computation respects its WCET. If the computation fails to respect its WCET, currently Tice leaves it as an *undefined behavior* (e.g., [87] takes the liberty to simply keep repeating the computation back-to-back for as many periods as there would be if the computation never failed to respect its WCET). Therefore, it should be clear that the semantics of the original program in Figure 2.12 is different from the semantics of the modified program in Figure 4.1. While the original program is driven by the availability of a data item in the standard input, the occurrence of which can be used in synchronous languages as a logical tick, the modified program is driven by the actual passage of time. Consequently, if a data item is not available in the standard input over a period of time that is longer than the stated period of 100 ms and then becomes available once again, the original program only executes the computation just once. In contrast, the behavior of the modified program is currently left undefined because the computation's WCET, which is 10 ms, is not respected. Lastly, as already stated in Chapter 3, Tice lowers the time-triggered LET abstraction to the BET abstraction for certain kinds of computation. The computation of `fn` is one of the kinds because it uses the function `printf` to perform output as soon as possible without waiting for the correct time points to do so under the time-triggered LET MoCC.

4.1 The Syntax of Tice

Tice language constructs (i.e., concrete syntax) are the following Tice library's public API members, which [87] places within the C++ namespace `tice::v1`:

- Class template `Core_ids`, which takes a variable number of integers to express zero or more processor core IDs.
- Class template `HW`, which takes an instance of `Core_ids` as its sole parameter.
- Class template `Ratio`, which takes two parameters, both of which are integers but the latter of which is an optional parameter: (1) the numerator of a rational number and (2) the denominator of the rational number, which defaults to one if not given.
- Macro `Comp`, which takes two parameters: (1) a pointer to a C/C++ function that will perform some real-time computation and (2) an instance of `Ratio` that specifies the function's WCET (i.e., the longest possible duration that the function takes to return after being called).
- Class template `Node`, which takes two parameters: (1) a call to `Comp` that specifies a computation and (2) an instance of `Ratio` that specifies the period by which the computation will be executed.
- Class template `Chan`, which takes two parameters: (1) the data type of an inter-node communication buffer and (2) a pointer to its initial value.
- Class template `Chan_inlit`, which takes two parameters: (1) the data type of an inter-node communication buffer and (2) its initial value.
- Class template `Feeder`, which takes three or more parameters to express inter-node producer-consumer relationships in a fan-in fashion: the consumer is always specified as the last parameter preceded by a number of producer-channel pairs feeding it. While the consumer and every producer are specified using instances of `Node`, every channel is specified using an instance of either `Chan` or `Chan_inlit`.

```
1  Tice_program = HW_dependent_part, ';', HW_independent_part, ';';
2
3  HW_dependent_part = 'typedef ', HW_desc, ';',
4                      'typedef ', WCET, {'; typedef ', WCET};
5  HW_desc = 'HW<', core_ids, '>', HW_desc_ident;
6  core_ids = 'Core_ids<', [ nonnegative_int, {',', nonnegative_int} ], '>';
7  WCET = positive_rational, wcet_ident;
8
9  HW_independent_part = 'typedef ', node, {'; typedef ', node}, ';',
10                       'typedef ', program;
11 node = 'Node<', computation, ',', period, '>', node_ident;
12 computation = 'Comp(', fn_ptr, ',', wcet_ident, ')';
13 period = positive_rational;
14 program = 'Program<', HW_desc_ident, ',',
15                       node_ident, {',', node_ident},
16                       [ ',', feeder, {',', feeder},
17                        {',', ETE_delay},
18                        {',', correlation} ], '>', Tice_program_ident;
19 feeder = 'Feeder<', producer, ',', channel,
20                     {',', producer, ',', channel}, ',', consumer, '>';
21 producer = node_ident;
22 consumer = node_ident;
23 channel = 'Chan_inlit<', type, ',', init_val, '>'
24         | 'Chan<', type, ',', init_val_ptr, '>';
25 ETE_delay = 'ETE_delay<', sensor, ',', actuator, ',',
26                           min_delay, ',', max_delay, '>';
27 min_delay = nonnegative_rational;
28 max_delay = positive_rational;
29 correlation = 'Correlation<', actuator, ',', threshold, ',',
30                               sensor, {',', sensor}, '>';
31 threshold = nonnegative_rational;
32 sensor = node_ident;
33 actuator = node_ident;
34
35 positive_rational = 'Ratio<', positive_int, [',', positive_int], '>';
36 nonnegative_rational = 'Ratio<', nonnegative_int, [',', positive_int], '>';
```

Figure 4.2: Tice syntax expressed in the ISO/IEC standard EBNF (Extended Backus-Naur Form) [50].

- Class template `ETE_delay`, which takes four parameters to specify a range of permitted end-to-end delays (i.e., an *end-to-end delay constraint*): (1) an instance of `Node` that specifies a source node, (2) an instance of `Node` that specifies a sink node, (3) an instance of `Ratio` that specifies the minimum delay, and (4) an instance of `Ratio` that specifies the maximum delay.
- Class template `Correlation`, which takes three or more parameters to specify some inter-data temporal correlation (i.e., a *correlation constraint*): (1) an instance of `Node` that specifies a sink node, (2) an instance of `Ratio` that specifies the *correlation threshold*, and (3) one or more source nodes, each of which is specified using an instance of `Node`.
- Class template `Program`, whose parameter list is divisible into the following segments to express a *complete* Tice program:
  - Segment-A that has exactly one instance of `HW`.
  - Segment-B that has at least one instance of `Node`.
  - Segment-C that has zero or more instances of `Feeder`.
  - Segment-D that has zero or more instances of `ETE_delay`.
  - Segment-E that has zero or more instances of `Correlation`.

More formally, Figure 4.2 shows Tice syntax (i.e., the *abstract syntax* of Tice) with the canonical C++ concrete syntax being shown in bold face enclosed within a pair of single quotes. Canonical C++ concrete syntax means that the parts in bold face can be written differently as long as the result is valid according to the C++ language and has the same semantics as the replaced parts. For example, instead of writing a separate typedef to instantiate `HW` in accordance with the grammar shown in Figure 4.2 lines 3 and 5, the instantiation of `HW` can be written directly in the parameter list of `Program` as shown in Figure 4.1 line 8.

The Tice syntax in Figure 4.2 uses the following nonterminal (symbol) definitions:

- Every nonterminal with suffix "_ident" represents a (valid) sequence of C++ terminals that expresses a C++ identifier.
- The nonterminals "nonnegative_int" and "positive_int" represent sequences of C++ terminals that express nonnegative and positive integers, respectively.
- In line 12, "fn_ptr" represents a sequence of C++ terminals that expresses a function pointer.
- In lines 23–24, "type" represents a sequence of C++ terminals expressing a C++ *object type*, which as specified in Section 3.9 Paragraph 8 of [51] is any type other than `void`, a reference type, and a function type.
- In line 23, "init_val" represents a sequence of C++ terminals that expresses a value whose type is compatible with the type expressed by the nonterminal "type" in the same line.
- In line 24, "init_val_ptr" represents a sequence of C++ terminals that expresses a pointer to an object whose type is compatible with the type expressed by the nonterminal "type" in the same line.

While every nonterminal with suffix "_ident" can formally be replaced with a single nonterminal, such as "identifier", it is not done to provide the following reading aid:

- In line 5, the C++ identifier represented by the nonterminal "HW_desc_ident" can be referred to in line 14 by making the line's "HW_desc_ident" represent the same identifier.
- In line 7, "wcet_ident" can be referred to in line 12 by making the line's "wcet_ident" represent the same identifier.
- In line 11, "node_ident" can be referred to in line 15/21/22/32/33 by making the respective line's "node_ident" represent the same identifier.
- In line 18, "Tice_program_ident" can be referred to in any place where it is valid to instantiate a class with a default constructor, which usually is in function `main` (a possible alternative is in the declaration of a global variable).

The reading aid makes it clear that a compile-time error will be raised when the identifier represented by the nonterminal "wcet_ident" in line 7 is used in line 14 by making the line's "HW_desc_ident" represent the identifier although doing so is valid according to the EBNF grammar.

To demonstrate the use of the Tice syntax shown in Figure 4.2 completely, the Tice syntax is fully exercised by the program shown in Figure 4.3. As indicated in Figure 4.2 line 1, the part of Tice syntax in lines 3–7 generates hardware-dependent constructs, while the part in lines 9–33 generates hardware-independent constructs. While the part of the syntax that generates hardware-dependent constructs is exercised in lines 6–10 of both Figure 4.3(b) and Figure 4.3(c), the part that generates hardware-independent constructs is exercised in lines 20–33 of Figure 4.3(d). More importantly, Figure 4.3 highlights the fact that a Tice program is seamlessly composable and automatically integrable with ordinary C++ libraries. The fact is evident in the use of the C++ standard library `array`, which is included in line 3 and used in lines 4–6 of Figure 4.3(a) and is used in line 14 and at runtime is checked in line 16 and initialized in lines 17–19 of Figure 4.3(d), by parts of the Tice program directly in line 30 and indirectly in lines 20–23 of Figure 4.3(d).

Lastly, the program shown in Figure 4.3 highlights the following important design decisions to improve Tice usability:

- Figure 4.3(d) lines 20–23 show that Tice requires pointers to the functions that will perform nodes' computations. Since two function pointers are equal if they point to the same function and not equal otherwise, Tice uses the required pointers as the identities of the nodes that are specified in a Tice program (i.e., in the parameter list of `Program`). This design decision improves Tice usability because otherwise node identities have to be specified either manually, which is burdensome and error-prone, or by some non-standard C++ feature, which is against the objective of this work.

- Figure 4.3(d) lines 28–30. show that the public API member `Feeder` is designed to fan-in by taking a number of producer nodes but only one consumer node. Two other possible designs are for `Feeder` either to fan-out by taking only one producer node and a number of consumer nodes or to pair by taking exactly one producer node and one consumer node. A fan-in design, however, supplies extra information on a graph arc $e \in \mathbb{E}$ implicitly, which otherwise must be specified explicitly when using the other two designs. This extra information specifies which producer supplies data item to which formal parameter of the consumer. For example, Figure 4.3(a) lines 5–6 shows that the producer node executing the function `f1` should supply the returned value (i.e., data item) as the actual parameter of the first formal parameter of `f4`, while the functions `f2` and `f3` should supply their returned values as the actual parameters of the second and third formal parameters of `f4`, respectively. In the fan-in design, the extra information is supplied implicitly by the positions of the producers in the `Feeder`'s parameter list as shown in Figure 4.3(d) lines 28–30. Since the extra information is important to ensure consistency, using fan-in design improves Tice usability accordingly.

```cpp
#ifndef SUBPROGRAMS_HPP
#define SUBPROGRAMS_HPP
#include <array>
typedef std::array<double, 5> X;
        double f1();     int f2();          X f3(double from_f1);
void f4(double from_f1, int from_f2, const X &from_f3);
#endif // SUBPROGRAMS_HPP
```

(a) File `subprograms.hpp`.

```cpp
#ifndef HW_1_HPP
#define HW_1_HPP
#include "subprograms.hpp"
#include <tice/v1.hpp>
using namespace tice::v1;
typedef HW<Core_ids<0>> target_hw;
typedef Ratio<1,10> f1_wcet;
typedef Ratio<3,10> f2_wcet;
typedef Ratio<1>    f3_wcet;
typedef Ratio<2,10> f4_wcet;
#endif // HW_1_HPP
```

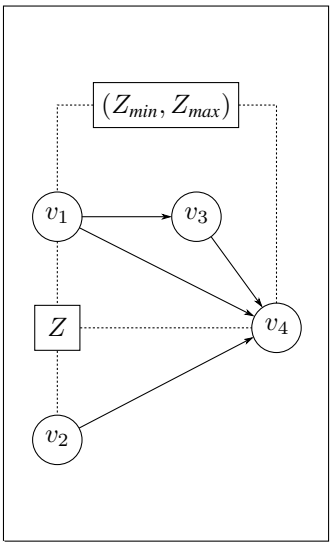
(b) File `hw-1.hpp`.

```cpp
#ifndef HW_2_HPP
#define HW_2_HPP
#include "subprograms.hpp"
#include <tice/v1.hpp>
using namespace tice::v1;
typedef HW<Core_ids<0, 1, 2, 3>> target_hw;
typedef Ratio<2,10> f1_wcet;
typedef Ratio<6,10> f2_wcet;
typedef Ratio<2>    f3_wcet;
typedef Ratio<4,10> f4_wcet;
#endif // HW_2_HPP
```

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

```cpp
#include <cstdlib>
#include <tice/v1.hpp>
#include "subprograms.hpp"
using namespace tice::v1;
#if   defined USE_HW_1
#include "hw-1.hpp"
#elif defined USE_HW_2
#include "hw-2.hpp"
#else
  typedef HW<Core_ids<>> target_hw;
  typedef Ratio<1> f1_wcet; typedef Ratio<1> f2_wcet;
  typedef Ratio<1> f3_wcet; typedef Ratio<1> f4_wcet;
#endif
namespace { X prms; double zero = 0; }
int main(int argc, char **argv) {
  if (argc != 1 + prms.size()) return -1;
  for (int i = 1; i <= prms.size(); ++i)
    prms[i - 1] = (argv[i]
                   ? std::strtod(argv[i], nullptr) : 0);
  typedef Node<Comp(&f1, f1_wcet), Ratio<2>> v1;
  typedef Node<Comp(&f2, f2_wcet), Ratio<3>> v2;
  typedef Node<Comp(&f3, f3_wcet), Ratio<5>> v3;
  typedef Node<Comp(&f4, f4_wcet), Ratio<2>> v4;
  Program<
  /*Segment-A*/ target_hw,
  /*Segment-B*/ v1, v2, v3, v4,
  /*Segment-C*/ Feeder<v1, Chan<double, &zero>, v3>,
                Feeder<v1, Chan<double, &zero>,
                       v2, Chan_inlit<int, 0> ,
                       v3, Chan<X, &prms>     , v4>,
  /*Segment-D*/ ETE_delay<v1, v4, Ratio<2>, Ratio<10>>,
  /*Segment-E*/ Correlation<v4, Ratio<10>, v1, v2>
  > p; p.run(); if (!p.get_error_code()) p.wait();
}
```

(d) File `main.cpp`.



(e) The expressed Tice model.

Figure 4.3: A C++ program that uses all Tice language constructs and is portable to different hardware.

```
1 $ g++ -Wfatal-errors -ftemplate-backtrace-limit=1 -Ilib -c -o /tmp/x \
2 tice-program.cpp 2>&1 | head -n2
3 In file included from tice-program.cpp:2:
4 lib/tice/v1.hpp: In instantiation of 'struct tice::v1::error::program::
5 correlation_is_within_threshold<false, 9, std::ratio<4000000, 1000000>,
6 std::ratio<0>, tice::v1::error::Path_forming_node_pos<2, 4, 5>, tice::v1
7 ::error::Path_forming_node_pos<3, 5>, 6>':
```

Figure 4.4: The compilation of the program shown in Figure 4.3(d) when line 32 instantiates `Ratio<10>`.

- Figure 4.3(d) lines 28–29 show that the main difference between using `Chan` and `Chan_inlit` is that the former takes a pointer to the initial value while the latter takes the initial value itself as their second parameter. The reason behind the difference is the restriction imposed by C++ on the data type of a template's non-type non-template parameter (a template's actual parameter can either be a type, a template, and a non-type non-template). For example, [51, 52] forbid `double` but neither `const double &` nor `const double *` from being the data type of a non-type non-template parameter. Therefore, Tice usability is improved by providing both `Chan` and `Chan_inlit` for the two different usage scenarios.

4.2 Turning C++ into a Model-Based Tool

Being model-based [95], Tice is founded on a formal ground that allows problems to be formally defined as models and their solutions to be formally analyzed on the models and implemented mechanically (i.e., automatically) based on the models. This means that questions on the behavior of a Tice program can be answered just by analyzing the model expressed by the Tice program without any need to first obtain the program's executable and then run the executable on the target hardware. Indeed, it should not be possible to obtain the executable in the first place if the behavior of the executable on the target hardware will violate the model's behavior. Therefore, Tice can turn a C++ compiler into a model-based tool, such as MATLAB®/Simulink®. Indeed, practitioners have expressed their intention to bring MATLAB®/Simulink®-features into C++ [43, 112]. Consequently, as demonstrated in this section, C++ as a platform for language-oriented programming could indeed do what language workbenches could do: interactive modeling by simulation.

When using an off-the-shelf standard C++ compiler as a model-based tool, the means to ask questions on the model expressed in a Tice program is by using some real-time constraint, which is either an end-to-end delay constraint or a correlation constraint. On the other hand, the means to answer the questions is the error messages that the compiler outputs when the compiler formally determines that the constraint cannot be respected. Using Figure 4.3(e) as an example, to ask a question on the correlation between v_1 and v_2 with respect to v_4, the correlation threshold in Figure 4.3(d) line 32 can be set to zero.

When compiled, the compiler will issue an error and report the first correlation that violates the constraint's threshold as shown in Figure 4.4. Specifically, line 5 shows that the predicate of "correlation is within threshold" is false for the correlation constraint that is found at position 9 in the `Program`'s parameter list because a computation of (4.7) shows that an absolute difference between the sensing times of data items read by v_4 can be as large as 4 s. Line 6 then continues on by showing that the constraint's threshold is zero and the first confluent path, which goes through nodes that can be found at positions 2, 4, and 5 in the `Program`'s parameter list. Lastly, line 7 finishes by showing the second confluent path, which goes through nodes that can be found at positions 3 and 5 in the `Program`'s parameter list, and the release index of v_4 at which the violation is observed, which is 6.

After receiving the answer, the correlation threshold in Figure 4.3(d) line 32 can then be modified to the value reported, which is 4 s, and then the compiler be re-asked whether the correlation threshold is indeed 4 s. If so, the compiler will terminate without reporting any error. Otherwise, the compiler will report another error that shows an absolute difference that is greater than 4 s, and the question-answering session can be repeated. This interactive manner is similar to what is done using model-based tools, such as MATLAB®/Simulink®.

4.3 Some Apparent Limitations of Tice

Tice has some features that at the first glance seem to be limitations:

- No two nodes can be assigned the same C/C++ function because the function pointer assigned to a node is used as the node's identity.

- The arcs connecting nodes must not form a *feedback loop*, which is commonly used in engineering the real-time control aspect of a cyber-physical system.

- Tice's time-triggered LET MoCC holds only for data items that are returned by functions implementing computation nodes. If a function has no return value (i.e., the function has `void` as its return type), Tice lowers the time-triggered LET abstraction to BET abstraction for the function's computation.

The features, however, are not real limitations because they can be addressed easily as follows:

- To assign the same C/C++ function to two or more nodes, each of the node can be assigned a wrapper function that simply calls the function to be shared. The cost of using a wrapper function is zero as an optimizing compiler will discard the wrapper function and calls the shared function directly.

- A feedback loop can be expressed at the lower abstraction level, which is that of a C++ program (i.e., Tice prevents no pair of producer-consumer nodes from sharing a C/C++ variable that is used to communicate some data item from the consumer back to the producer, which can be done correctly to keep implementing the semantics of Tice correctly).

- As time-triggered LET MoCC is usually used to control when data are visible, if the appearance of the data that are internally written by a function that has no return value needs to be controlled, the function can be wrapped within another function that simply calls the function and returns some dummy value. The dummy value can then be consumed by an extra sink node that is assigned an empty function that returns no value and acts as a global garbage collector. The extra sink node can then be assigned a WCET of zero to let Tice discard the extra sink node at compile time.

4.4 The Semantics of Tice

A complete Tice program expresses one *Tice model*, which is a DAG (directed acyclic graph) decorated with real-time constraints. The Tice model shown in Figure 4.3(e) is expressed by the Tice program shown in Figure 4.3(d) lines 24–33, precisely by parts of segment B that specify no WCET and everything in segment C up to segment E. Segment A and the WCET specifications in segment B, on the other hand, are used only to implement the model in some target hardware. In MDE terms, parts of segment B that specify no WCET and everything in segment C up to segment E express a *platform-independent model* (PIM), while segment A and the WCET specifications in segment B are altogether a *platform description model* (PDM), which is used to transform a PIM into a *platform-specific model* (PSM) [55]. The model in Figure 4.3(e) has its $Z_{\text{min}} = 2$ s and $Z_{\text{max}} = 10$ s as specified in the segment D (i.e., Figure 4.3(d) line 31) and has its $Z = 10$ s as specified in the segment E (i.e., Figure 4.3(d) line 32).

Informally, the Tice model shown in Figure 4.3(e) has the following semantics. The computations of v_1 (i.e., `f1`), v_2 (i.e., `f2`), v_3 (i.e., `f3`), and v_4 (i.e., `f4`) are to execute concurrently and periodically with periods 2 s, 3 s, 5 s, and 2 s, respectively, as specified in Figure 4.3(d) lines 20–23. In accordance with the time-triggered LET MoCC, each node in each of its periods reads one data item from every incoming arrow (if any) at the period's beginning, executes the computation during the period, and writes the computation result to every outgoing arrow (if any) at the period's end. On the other hand, every arrow represents a *data channel* (i.e., a means to communicate data items) with a *register buffer* (i.e., a buffer whose capacity is one, and hence, is capable of holding just one data item at any time point, just like a variable in a C++ program). While the arrows between the producer-consumer pairs v_1-v_3, v_1-v_4, and v_2-v_4 are all initialized with zero in Figure 4.3(d) lines 27, 28, and 29, respectively, the arrow between the pair v_3-v_4 is initialized in lines 17–19 with data items that are available only at runtime. Aside from that, Figure 4.3(e) shows that an *end-to-end delay constraint* is applied on v_1 and v_4 as specified in Figure 4.3(d) line 31 (i.e., segment D). The constraint guarantees that any data item generated by v_1 takes between $Z_{\text{min}} = 2$ s and $Z_{\text{max}} = 10$ s time units to undergo all of the followings: the generation by v_1, the flow to reach v_4 through every possible path, and the complete processing by v_4. On the other hand, Figure 4.3(e) shows that a correlation constraint is applied on v_1 and v_4 as specified in Figure 4.3(d) line 32 (i.e., segment E). The constraint guarantees that any data item generated by v_1 and v_2 that meet at v_4 differ in their generation times by at most $Z = 10$ s time units, which is the correlation threshold.

More formally, the Tice language $\mathcal{L}$ defines a set of valid Tice models where each model is a 10-tuple as in (4.1).

$$\mathcal{L} = \Big\{ \Big(\mathbb{V}, \mathbb{E}, \mathbb{T}_{\texttt{E2E}}, \mathbb{T}_{\texttt{Cor}}, f_{\texttt{P}} : \mathbb{V} \to \mathbb{Q}^{+}, f_{\texttt{C}} : \mathbb{V} \to \mathbb{Q}^{+}, f_{\texttt{t}} : \mathbb{E} \to \Theta_{\texttt{M}}, f_{\texttt{I}} : \mathbb{E} \to \bigcup_{\theta \in \Theta_{\texttt{M}}} A_{\theta}, \\ f_{\texttt{E2E}} : \mathbb{T}_{\texttt{E2E}} \to \left(\mathbb{Q}^{\geq 0} \times \mathbb{Q}^{+} \right), f_{\texttt{Cor}} : \mathbb{T}_{\texttt{Cor}} \to \mathbb{Q}^{\geq 0} \Big) \Big\} \tag{4.1}$$

The first element denoted $\mathbb{V}$ is a nonempty finite set of graph nodes; a graph node $v \in \mathbb{V}$ represents a computation. The second, third, and fourth elements denoted $\mathbb{E}$, $\mathbb{T}_{\texttt{E2E}}$, and $\mathbb{T}_{\texttt{Cor}}$ are possibly-empty finite sets of graph arcs, end-to-end delay constraints, and correlation constraints, respectively. A graph arc $e \in \mathbb{E}$ represents a data channel. The fifth and sixth elements denoted $f_{\texttt{P}}$ and $f_{\texttt{C}}$ are functions that map the computation represented by a graph node to its period of execution and to its WCET, respectively, both of which are positive rationals. The seventh and eight elements denoted $f_{\texttt{t}}$ and $f_{\texttt{I}}$ are functions that map the channel represented by a graph arc to its data type and to its initial value when no data item has flowed through the channel, respectively. Lastly, the ninth and tenth elements denoted $f_{\texttt{E2E}}$ and $f_{\texttt{Cor}}$ are functions that map an end-to-end delay constraint to its minimum and maximum delays and a correlation constraint to its threshold, respectively. The minimum and maximum delays and the threshold are all positive rationals.

In the rest of this section, $\mathbb{Q}^{+}$ denotes the set of all positive rationals, $\mathbb{Q}^{\geq 0}$ denotes the set of all nonnegative rationals, $\mathbb{N}^{+}$ denotes the set of all positive integers, $\mathbb{N}$ denotes the set of all nonnegative integers, and $\operatorname{lcm} S$ denotes the least common multiple of all members in some set $S \subset \mathbb{Q}^{+}$. Additionally, P_v and C_v abbreviate $f_{\texttt{P}}(v)$ and $f_{\texttt{C}}(v)$, respectively, and, when it is clear from the context which graph node $v \in \mathbb{V}$ is being referred to, P_v and C_v are denoted P and C, respectively.

Graph Nodes

A graph node $v \in \mathbb{V}$ represents a C/C++ function whose WCET $C \in \mathbb{Q}^{+}$ is assigned using Tice library's public API member `Comp`. Formally, the WCET assignment partially defines function $f_{\texttt{C}}$, which is the sixth element of a 10-tuple in $\mathcal{L}$. Since function $f_{\texttt{C}}$ maps to $\mathbb{Q}^{+}$, at compile-time Tice library checks that second actual parameter of `Comp` is a positive rational. Aside from that, Tice library implements the set $\mathbb{V}$ in terms of the function pointers given as the first actual parameters to calls to `Comp` that are evaluated in the instantiation of the public API member `Program`. Consequently, since a set abstracts duplicates, Tice library takes two or more function pointers pointing to the same function as a single node $v \in \mathbb{V}$ and any two function pointers pointing to different functions as two nodes $v_1, v_2 \in \mathbb{V}$ with $v_1 \neq v_2$. Aside from that, the C/C++ function represented by a graph node $v \in \mathbb{V}$ is also assigned its execution period $P \in \mathbb{Q}^{+}$ using Tice library's public API member `Node`. Formally, the period assignment partially defines function $f_{\texttt{P}}$, which is the fifth element of a 10-tuple in $\mathcal{L}$.

The semantics of assigning a C++ function some WCET C and some period P is that the C/C++ function is released (i.e., made available for execution) once every P time units as formalized in Definition 1 and Definition 2 and needs at most C time units to complete its computation. Tice requires and at compile-time Tice library checks at compile-time that the C/C++ function's WCET is not greater than its period.

Definition 1 (Node's Release Times)**.** *Let n be some value in the set $\mathbb{N}$ and k be $n+1$. Then, the time of the k-th release of a node $v \in \mathbb{V}$ is given by the function $f_{\texttt{r},v} : \mathbb{N} \to \mathbb{Q}^{\geq 0}$ defined as $f_{\texttt{r},v}(n) = nP_v$ and abbreviated as $r_{v,n}$.*

Definition 2 (Node's Release Timeset)**.** *The release timeset $\mathbb{A}_v$ of a node $v \in \mathbb{V}$ is the set $\{ r_{v,n} \mid n \in \mathbb{N} \}$.*

Graph Arcs

The second element of a 10-tuple in $\mathcal{L}$ is a possibly-empty finite set of graph arcs denoted $\mathbb{E}$ defined formally in (4.2). An arc $e \in \mathbb{E}$ represents a unidirectional data channel from a producer node v to a consumer node v''.

$$\mathbb{E} = \{ (v, v'') \mid v \in \mathbb{V}, v'' \in \mathbb{V}, v \neq v'' \}. \tag{4.2}$$

A data channel has a memory with data type θ to store *exactly one* data item $\alpha \in A_\theta$ at any time point where A_θ is the set of all values of type θ. For example, when θ is `uint_8`, α may be `0x07`, and A_θ is the set {`0x00`, `0x01`, ..., `0xFE`, `0xFF`}. Let $\Theta_{\texttt{M}}$ be the set of all C++ object types that are not qualified with either `const` or `volatile` (as mentioned in §4.1, C++ object types are C++ data types other than `void`, reference types, and function types). Then, $\Theta_{\texttt{M}}$ is a nonempty infinite set since C++ not only defines some types (e.g., `char`, `double`) but also allows further types to be defined (e.g., `class C`, `char **`). Tice requires that every data

channel is assigned some data type $\theta \in \Theta_{\texttt{M}}$ by specifying the data type as the first actual parameter of either the public API member `Chan` or `Chan_inlit`. Formally, each of the assignments partially defines the function $f_{\texttt{t}} : \mathbb{E} \to \Theta_{\texttt{M}}$, which is the seventh element of a 10-tuple in $\mathcal{L}$ and maps an arc $e \in \mathbb{E}$ to its represented data channel's data type $f_{\texttt{t}}(e)$. Furthermore, Tice requires that the memory of every data channel is initialized with some well-defined value $\alpha_0 \in A_\theta$ by specifying the initial value as the second actual parameter of either the public API member `Chan` or `Chan_inlit`. Formally, each of the specifications partially defines the function $f_{\texttt{I}} : \mathbb{E} \to \bigcup_{\theta \in \Theta_{\texttt{M}}} A_\theta$, which is the eighth element of a 10-tuple in $\mathcal{L}$ and maps an arc $e \in \mathbb{E}$ to its represented data channel's initial data item $f_{\texttt{I}}(e)$. Lastly, Tice requires and at compile-time Tice library checks that $f_{\texttt{I}}(e) \in A_{f_{\texttt{t}}(e)}$ for every $e \in \mathbb{E}$.

On the other hand, the set $\mathbb{E}$ is constructed by the public API member `Feeder`. Specifically, Tice requires that every instantiation of `Feeder` in the parameter list of `Program` formally constructs the set $\mathbb{E}'_{v''}$ defined in Definition 3. Therefore, at compile-time Tice library checks that every `Feeder` instance found in the segment C of a `Program`'s parameter list has a distinct consumer node. Additionally, since $\mathbb{E}$ is a set, at compile-time Tice library checks that every `Feeder` instance in the segment C has distinct producer nodes. Furthermore, since a Tice model is a DAG, any member of $\mathbb{E}$ forms no loop and no cycle. Consequently, at compile-time Tice library checks that every `Feeder` instance in the segment C has no producer node that is also a consumer node (i.e., no loop) and that the graph constructed by all `Feeder` instances in the segment C has no closed path (i.e., no cycle).

Definition 3 (Node's Incoming Arcs)**.** *Let v'' be some member of $\mathbb{V}$ such that $(\cdot, v'') \in \mathbb{E}$. Then, $\mathbb{E}'_{v''}$ is the largest possible set $\{(v_1, v''), \ldots, (v_k, v'')\}$ for some $k \in \mathbb{N}^+$ such that $\mathbb{E}'_{v''} \subseteq \mathbb{E}$.*

With $\mathbb{E}'_{v''}$ being the set defined in Definition 3, k being the cardinality of $\mathbb{E}'_{v''}$ (i.e., $k = |\mathbb{E}'_{v''}|$), and i being an integer satisfying $1 \leq i \leq k$, Tice requires and at compile-time Tice library checks that each arc $(v_i, v'') \in \mathbb{E}'_{v''}$ has:

- Its v_i represent a C++ function whose return type R is assignable to some C++ *object* (i.e., some region of memory that holds the value of some C++ data type) of type $f_{\texttt{t}}((v_i, v''))$.
- Its v'' represent a C++ function that is callable with k actual parameters of type $T_1, \ldots, T_k$, respectively, with the ordering of the k actual parameters being specified by the ordering of the producer nodes $v_1, \ldots, v_k$ in the `Feeder`'s parameter list and T_i being any type that accepts an actual parameter of type $f_{\texttt{t}}((v_i, v''))$.

Sensor, Actuator, and Intermediary Nodes

Tice partitions the members of $\mathbb{V}$ into three disjoint subsets to identify which kinds of computations have time-triggered LET or BET as their real-time abstractions and to support the syntax and the semantics of the third and fourth elements of a 10-tuple in $\mathcal{L}$, namely the sets of end-to-end delay constraints $\mathbb{T}_{\texttt{E2E}}$ and correlation constraints $\mathbb{T}_{\texttt{Cor}}$. The three disjoint subsets are the set of sensor nodes $\mathbb{V}_{\texttt{s}}$, the set of actuator nodes $\mathbb{V}_{\texttt{a}}$, and the set of intermediary nodes $\mathbb{V} \setminus (\mathbb{V}_{\texttt{s}} \cup \mathbb{V}_{\texttt{a}})$. Note that being disjoint, $\mathbb{V}_{\texttt{s}} \cap \mathbb{V}_{\texttt{a}} = \emptyset$.

The set of sensor nodes $\mathbb{V}_{\texttt{s}}$ is defined formally in (4.3) to include every node $v \in \mathbb{V}$ that has no incoming arc and *possibly* no outgoing arc.

$$\mathbb{V}_{\texttt{s}} = \{\, v \in \mathbb{V} \mid (\cdot, v) \notin \mathbb{E} \,\} \tag{4.3}$$

Consequently, the members of $\mathbb{V}_{\texttt{s}}$ represent C++ functions whose types are of the form $R\,\texttt{()}$, that is, whose function prototypes are of the form R `fn()` with `fn` being any valid C++ identifier. A sensor node that has no outgoing arc then represents a C++ function that has `void` as its return type R. A sensor node can represent a computation that extracts data from a sensor, for example, by reading I/O ports. Note that when a sensor node has no outgoing arc, it might be the case that the represented C++ function not only extracts data from some sensor but also commands some actuator (e.g., watchdog and logger) as illustrated in Figure 4.1 lines 21–30.

On the other hand, the set of actuator nodes $\mathbb{V}_{\texttt{a}}$ is defined formally in (4.4) to include every node $v \in \mathbb{V}$ that has at least one incoming arc but no outgoing arc.

$$\mathbb{V}_{\texttt{a}} = \{\, v \in \mathbb{V} \mid (v, \cdot) \notin \mathbb{E}, (\cdot, v) \in \mathbb{E} \,\} \tag{4.4}$$

Consequently, the members of $\mathbb{V}_{\texttt{a}}$ represent C++ functions whose types are of the form $\texttt{void}\,(T_1, \ldots, T_n)$, that is, whose function prototypes are of the form $\texttt{void fn}\,(T_1, \ldots, T_n)$ with n being in $\mathbb{N}^+$, $T_1, \ldots, T_n$ being any C++ type other than `void`, and `fn` being any valid C++ identifier. An actuator node can represent a computation that sends commands to an actuator, for example, by writing I/O ports.

It then follows that the set of intermediary nodes $\mathbb{V} \setminus (\mathbb{V}_{\mathrm{s}} \cup \mathbb{V}_{\mathrm{a}})$ satisfies (4.5) to include every node $v \in \mathbb{V}$ that has both incoming and outgoing arcs.

$$\mathbb{V} \setminus (\mathbb{V}_{\mathrm{s}} \cup \mathbb{V}_{\mathrm{a}}) = \{\, v \in \mathbb{V} \mid (v, \cdot) \in \mathbb{E}, (\cdot, v) \in \mathbb{E} \,\} \tag{4.5}$$

Consequently, the members of $\mathbb{V} \setminus (\mathbb{V}_{\mathrm{s}} \cup \mathbb{V}_{\mathrm{a}})$ represent C++ functions whose types are of the form $R'(T'_1, \ldots, T'_m)$, that is, whose function prototypes are of the form R' `fn` $(T'_1, \ldots, T'_m)$ with m being in $\mathbb{N}^+$, $R', T'_1, \ldots, T'_m$ being any C++ type other than `void`, and `fn` being any valid C++ identifier. An intermediary node can represent a computation that takes as input the data returned by one or more sensor and/or intermediary nodes and gives as output the result of its computation.

The partitioning of $\mathbb{V}$ into three disjoint subsets has two properties:

- The partitioning of $\mathbb{V}$ precludes no C++ function from being represented by a node $v \in \mathbb{V}$. This can be shown by letting Θ_{F} be the set of all possible C++ function types. Then, every member of Θ_{F} is a pair $(R, \mathbb{P})$ with R being the return type of a C++ function whose parameter types are expressed as the possibly-empty finite set $\mathbb{P}$. Since sensor, actuator, and intermediary nodes represent C++ functions whose function types are of the forms $(R, \emptyset)$, $(\mathtt{void}, \mathbb{P}')$, and $(R', \mathbb{P}')$, respectively, with R being any valid return type, $\mathbb{P}'$ being a nonempty set, and R' being any valid return type other than `void`, the partitioning of $\mathbb{V}$ precludes no member of Θ_{F}.

- The partitioning of $\mathbb{V}$ also partitions $\mathbb{E}$ into four disjoint subsets:
  - The set of sensor-to-actuator arcs $\{\, (v, v'') \in \mathbb{E} \mid v \in \mathbb{V}_{\mathrm{s}}, v'' \in \mathbb{V}_{\mathrm{a}} \,\}$.
  - The set of sensor-to-intermediary arcs $\{\, (v, v'') \in \mathbb{E} \mid v \in \mathbb{V}_{\mathrm{s}}, v'' \notin (\mathbb{V}_{\mathrm{s}} \cup \mathbb{V}_{\mathrm{a}}) \,\}$.
  - The set of intermediary-to-actuator arcs $\{\, (v, v'') \in \mathbb{E} \mid v \notin (\mathbb{V}_{\mathrm{s}} \cup \mathbb{V}_{\mathrm{a}}), v'' \in \mathbb{V}_{\mathrm{a}} \,\}$.
  - The set of intermediary-to-intermediary arcs $\{\, (v, v'') \in \mathbb{E} \mid v \notin (\mathbb{V}_{\mathrm{s}} \cup \mathbb{V}_{\mathrm{a}}), v'' \notin (\mathbb{V}_{\mathrm{s}} \cup \mathbb{V}_{\mathrm{a}}) \,\}$.

  The partitioning of $\mathbb{E}$ into four disjoint subsets implies that the remaining five possible types of arcs cannot exist in Tice because they contradict one of the definitions of sensor, actuator, and intermediary nodes stated in (4.3), (4.4), and (4.5), respectively. For example, the set of sensor-to-sensor arcs $\{\, (v, v'') \in \mathbb{E} \mid v \in \mathbb{V}_{\mathrm{s}}, v'' \in \mathbb{V}_{\mathrm{s}} \,\}$ cannot exist because the set requires not only $(v, v'') \in \mathbb{E}$ but also $v'' \in \mathbb{V}_{\mathrm{s}}$. By (4.3), however, the latter means that $(v, v'') \notin \mathbb{E}$, which is a contradiction.

Model of Computation and Communication

As stated in Definition 1, Tice's model of computation is to release the computations of all nodes at time $t = 0$, and thereafter, the computation of every node v is released at every time point that is a multiple of v's period. Hence, the time points at which v's computation are released are the members of the infinite set $\mathbb{A}_v$ as stated in Definition 2. Tice's model of communication, on the other hand, specifies the following time-triggered LET behavior of a data channel as driven by its producer and consumer nodes.

A data channel is used by exactly one producer and one consumer nodes and has a buffer with a capacity of one data item. The buffer can only be written by the producer and can only be read by the consumer. A write overwrites the previous data in the buffer, but a read does not consume the existing data in the buffer.

> Writes and reads take a zero logical time (i.e., a write/read can be taken to complete at the same time point it starts) and consequently can be seen as *non-blocking* (i.e., incurring no blocking time).

> A data channel is written by its producer v whenever the producer is released (i.e., at every time point in $\mathbb{A}_v$) and is read by its consumer v'' whenever the consumer is released (i.e., at every time point in $\mathbb{A}_{v''}$).

Hence, given a data channel, the input/output activities of the consumer/producer with respect to the data channel occur at predictable time points. When the producer and the consumer of a data channel happen to be released at the same time point, the data channel guarantees that the producer's write completes before the consumer's read starts. Consequently, because at the initial release time $t = 0$, each data channel is already read by its consumer while its producer has not completed any computation to write any data item, every data channel is assigned an initial data item so that every consumer has a well-defined behavior.

Since Tice's MoCC has already been described, two important points can now be pointed out below:

- Tice's MoCC and more generally the time-triggered LET MoCC while not imposing any *precedence constraint* on the communication between a producer-consumer pair over a data channel, the MoCC itself implicitly imposes a precedence constraint by making a producer's write occurs before a consumer's read when they happen to be released *synchronously* (i.e., at the same time point).

- Tice's MoCC provides the time-triggered LET abstraction only for the computation of every node that uses the data channel represented by some arc in a Tice model. Therefore, the time-triggered LET abstraction exists for the computation represented by any sensor node that has a consumer (i.e., not an *isolated node*), any actuator node with respect to its producers, and any intermediary node. In the other two cases, namely any isolated sensor node and any actuator node with respect to its actuating action, the time-triggered LET abstraction is lowered to the BET abstraction.

Data Loss Along an Arc

Given Tice's MoCC and an arc that represents a data channel, it is easy to see that a data item can be lost when being communicated through the arc by being overwritten with another data item before it is read from the arc. Since the the semantics of end-to-end delay and correlation constraints needs to account for a data loss along an arc to be sound, this section gives the semantics of a data loss along an arc after presenting the following conceptual building blocks:

- With respect to an arc's producer, an overwrite timeset is defined below:

  Definition 4 (Node's Overwrite Timeset)**.** *Let v be some node in $\mathbb{V}$ with period P such that $(v, \cdot) \in \mathbb{E}$, `fn` be the C++ function represented by v, and R be the return type of `fn`. Then, the overwrite timeset of v, which is denoted $\mathbb{A}_v^+$, is the set $\mathbb{A}_v \setminus \{r_{v,0}\}$ whose every member t is the release time at which v overwrites the buffer of every data channel represented by the arc(s) in the set $\{(v, \cdot) \in \mathbb{E}\}$ with the data item $\alpha \in A_R$ that `fn` returns at some time point t' such that $t - P < t' < t$.*

- With respect to an arc's consumer, a reading timeset can be defined after being capable of predicting the consumer's next release time by the following proposition:

  Proposition 1 (Node's Next Release Time)**.** *Let v be some node in $\mathbb{V}$. Then, the earliest release time of v found starting at time $t \in \mathbb{Q}^{\geq 0}$ is given by the function $f_v^{\triangleright} : \mathbb{Q}^{\geq 0} \to \mathbb{A}_v$ defined as $f_v^{\triangleright}(t) = P_v \left\lceil \frac{t}{P_v} \right\rceil$.*

  Proof. By the definition of ceil operator, $\left\lceil \frac{t}{P_v} \right\rceil \in \mathbb{N}$. Consequently, letting n being $\left\lceil \frac{t}{P_v} \right\rceil$, the definition of $f_v^{\triangleright}$ is equivalent to the definition of $f_{\mathtt{r},v}$, and therefore, by Definition 1, $f_v^{\triangleright}$ indeed gives the release times of some node v. Now it remains to show that $f_v^{\triangleright}$ indeed gives the release time of v that is either equivalent to t or the earliest one after t.

  Suppose v has a release time at time t'. Then, by Definition 1, t' is divisible by P_v. If $t = t'$, then t is also divisible by P_v, and ceil operator by its definition leaves the result of the division unchanged. Multiplying the result by P_v will produce t', which is v's release time that is equivalent to t. Otherwise, t satisfies the inequality $t' - P_v < t < t'$. Dividing the inequality by P_v gives $n' < \frac{t}{P_v} < n$ for some $n', n \in \mathbb{N}$ because t' is divisible by P_v. By the definition of ceil operator, $\frac{t}{P_v}$ will be mapped to n, which when multiplied by P_v gives t', the earliest v's release time after t. □

- By being capable of predicting the consumer's next release time, the consumer's reading timeset can be defined below:

  Definition 5 (Node's Reading Timeset)**.** *Let v be some node in $\mathbb{V}$ with period P_v such that $(v, \cdot) \in \mathbb{E}$, v^* be some node in $\mathbb{V}$ with period P_{v^*} such that $v \neq v^*$, and t be some member of $\mathbb{A}_v^+$. Then, with respect to v, the reading timeset of v^*, denoted $\mathbb{A}^*_{v,v^*,t}$, is the set $\{ f_{v^*}^{\triangleright}(t) + kP_{v^*} \mid f_{v^*}^{\triangleright}(t) + kP_{v^*} < t + P_v, k \in \mathbb{N} \}$ whose every member t^* is the release time of v^* at which the data item α on every data channel that v writes at time t stays the same.*

Therefore, it follows that a data loss along an arc $(v, v^*) \in \mathbb{E}$ occurs for any data item that v writes to the arc at any time $t \in \mathbb{A}_v^+$ such that $|\mathbb{A}^*_{v,v^*,t}| = 0$.

End-to-End Delay Constraints

If a path exists from some sensor node $v_{\mathtt{s}}$ to some actuator node $v_{\mathtt{a}}$ based on the set $\mathbb{E}$, then after some time duration denoted D and called end-to-end delay, each data item read by $v_{\mathtt{s}}$ will flow either to reach $v_{\mathtt{a}}$ to produce an actuating action, in which case $D \in \mathbb{Q}^+$, or to be lost along some arc before reaching $v_{\mathtt{a}}$, in which case $D = \infty$. A real-time constraint called end-to-end delay constraint therefore can be specified on the pair $(v_{\mathtt{s}}, v_{\mathtt{a}})$

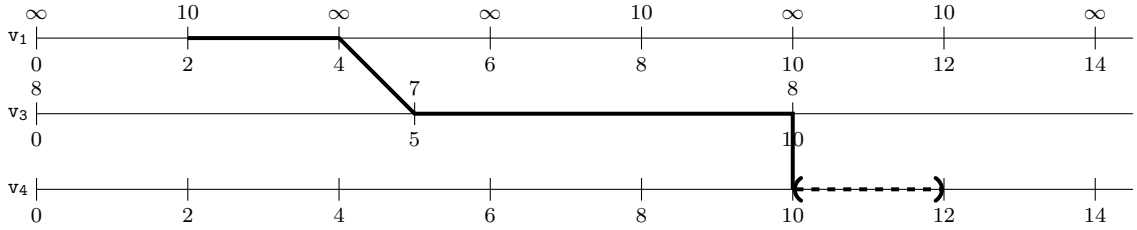


Figure 4.5: Release timelines of nodes $\mathtt{v}_1$ ($P_2 = 2$), $\mathtt{v}_3$ ($P_3 = 5$), and $\mathtt{v}_4$ ($P_4 = 2$) shown in Figure 4.3(e). Each tick has below it the global time t and above it either $g_{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}(t)$ on $\mathtt{v}_1$'s timeline or $g_{\overline{\pi}_{\mathtt{v}_3,\mathtt{v}_4}}(t)$ on $\mathtt{v}_3$'s timeline where $\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4} = \{(\mathtt{v}_1,\mathtt{v}_3)\} \cup \overline{\pi}_{\mathtt{v}_3,\mathtt{v}_4}$ and $\overline{\pi}_{\mathtt{v}_3,\mathtt{v}_4} = \{(\mathtt{v}_3,\mathtt{v}_4)\}$. The thick zigzag line shows a data item read by $\mathtt{v}_1$ at time 2 flowing to reach $\mathtt{v}_4$ at time 10, producing an actuating action at some time point along the dashed line.

to guarantee that every D stays within some given bound. The given bound's minimum and maximum are a nonnegative rational denoted Z_{min} and a positive rational denoted Z_{max}, respectively. In this setting, the end-to-end delay constraints specified using the public API member `ETE_delay` formally construct the set $\mathbb{T}_{\mathtt{E2E}}$, which is the third element of a 10-tuple in $\mathcal{L}$, by the $v_{\mathtt{s}}$-$v_{\mathtt{a}}$ node pairs specified in the constraints. Tice requires and at compile-time Tice library checks that the $v_{\mathtt{s}}$-$v_{\mathtt{a}}$ pairs specified by every end-to-end delay constraint is connected (i.e., a path exists from $v_{\mathtt{s}}$ to $v_{\mathtt{a}}$). Additionally, each end-to-end delay constraint also specifies its pair of end-to-end delay bounds (Z_{min}, Z_{max}). Formally, each of the specified end-to-end delay constraints partially constructs the function $f_{\mathtt{E2E}} : \mathbb{T}_{\mathtt{E2E}} \to \left(\mathbb{Q}^{\geq 0} \times \mathbb{Q}^{+}\right)$, which is the ninth element of a 10-tuple in $\mathcal{L}$. Tice requires and at compile-time Tice library checks that Z_{min} is not greater than Z_{max}. In order to show the precise semantics of an end-to-end delay constraint, however, the following conceptual building blocks will be presented first:

- The notation $\overline{\pi}_{v,v'}$ denotes a path from v to v', that is, $\overline{\pi}_{v,v'} = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$ with $n \geq 2$, $v_1 = v$, $v_n = v'$, and $\overline{\pi}_{v,v'} \subseteq \mathbb{E}$.
- The notation $v \rightsquigarrow v'$ denotes the proposition that some path $\overline{\pi}_{v,v'}$ exists.
- The notation $\overline{\mathbb{E}}_{v,v'}$ denotes the nonempty finite set of all possible paths from v to v' when $v \rightsquigarrow v'$.
- The end-to-end delay that a data item will experience when processed along some path $\overline{\pi}_{v,v'}$ can be defined for every data item that v reads at time $t \in \mathbb{A}_v$ as follows:

  Definition 6 (End-to-End Delay's Upper Bound)**.** *Let v and v' be any two nodes such that $v \rightsquigarrow v'$, v'' be any node such that $(v, v'') \in \overline{\pi}_{v,v'}$, and $\overline{\pi}_{v'',v'}$ be $\overline{\pi}_{v,v'} \backslash \{(v, v'')\}$. Then, function $g_{\overline{\pi}_{v,v'}} : \mathbb{A}_v \to \{\infty\} \cup \mathbb{Q}^{+}$ is defined inductively below:*

  Base case ($|\overline{\pi}_{v,v'}| = 1$): $g_{\overline{\pi}_{v,v'}}(t) = \min\left(\left\{\infty\right\} \cup \mathbb{A}^{*}_{v,v',t+P_v}\right) + P_{v'} - t$

  Inductive case ($|\overline{\pi}_{v,v'}| \geq 2$): $g_{\overline{\pi}_{v,v'}}(t) = \min\left(\left\{\infty\right\} \cup \left\{ t^* + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^{*}_{v,v'',t+P_v} \right\}\right) - t$

  Figure 4.5 illustrates Definition 6 where $\mathtt{v}_3$'s and $\mathtt{v}_1$'s timelines show the upper bounds of the end-to-end delays of the data items read by $\mathtt{v}_3$ and $\mathtt{v}_1$ at those times in the sets $\{0, 5, 10, \ldots, 20\} \subset \mathbb{A}_{\mathtt{v}_3}$ and $\{0, 2, 4, \ldots, 20\} \subset \mathbb{A}_{\mathtt{v}_1}$, respectively. Determining the values of $g_{\overline{\pi}_{\mathtt{v}_3,\mathtt{v}_4}}(t')$ for $t' \in \{0, 5, 10, \ldots, 20\}$ by Definition 6 is straightforward using the base case ($|\overline{\pi}_{\mathtt{v}_3,\mathtt{v}_4}| = 1$), for example, $g_{\overline{\pi}_{\mathtt{v}_3,\mathtt{v}_4}}(5) = 7$ due to $\mathbb{A}^{*}_{\mathtt{v}_3,\mathtt{v}_4,5+5} = \{10, 12, 14\}$ and $\min\{\infty, 10, 12, 14\} = 10$. Once determined, determining the values of $g_{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}(t)$ for $t \in \{0, 2, 4, \ldots, 20\}$ is also straightforward using the inductive case ($|\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}| = 2$), for example, $g_{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}(0) = \infty$ due to $\mathbb{A}^{*}_{\mathtt{v}_1,\mathtt{v}_3,0+2} = \emptyset$ (i.e., $\mathtt{v}_1$'s update at time 2 is lost along the arc $(\mathtt{v}_1, \mathtt{v}_3)$) and $g_{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}(2) = 10$ due to $\mathbb{A}^{*}_{\mathtt{v}_1,\mathtt{v}_3,2+2} = \{5\}$ and $g_{\overline{\pi}_{\mathtt{v}_3,\mathtt{v}_4}}(5) = 7$.
- Each of the producer nodes along some path $\overline{\pi}_{v,v'}$ can define the set of all end-to-end delays of all data items that the producer reads at all time points in $\mathbb{A}_v$ as follows:

  Definition 7 (End-to-End Delay's Upper-Bound Set)**.** *Let v and v' be any two nodes such that $v \rightsquigarrow v'$. Then, v's end-to-end delay's upper-bound set is denoted $\mathbb{G}_{\overline{\pi}_{v,v'}}$ and defined to be the nonempty set $\left\{ g_{\overline{\pi}_{v,v'}}(t) \;\middle|\; t \in \mathbb{A}_v \right\}$.*

Therefore, (4.6) describes the semantics of an end-to-end delay constraint formally.

$$Z_{min} \leq \min\left\{ \min \mathbb{G}_{\overline{\pi}_{v,v'}} \;\middle|\; \overline{\pi}_{v,v'} \in \overline{\mathbb{E}}_{v_{\mathtt{s}},v_{\mathtt{a}}} \right\} - P_{v_{\mathtt{a}}} < \max\left\{ \max \mathbb{G}_{\overline{\pi}_{v,v'}} \;\middle|\; \overline{\pi}_{v,v'} \in \overline{\mathbb{E}}_{v_{\mathtt{s}},v_{\mathtt{a}}} \right\} \leq Z_{max} \qquad (4.6)$$

Data Loss along a Path

Previously, the notion of a node's reading timeset was used to define a data loss along an arc that in turn was used to define the function $g_{\overline{\pi}_{v_s,v_a}}$ needed to measure the end-to-end delay's upper bound of a single data item along the end-to-end path $\overline{\pi}_{v_s,v_a}$. Analogously, the notion $g_{\overline{\pi}_{v,v'}}$ defined in Definition 6, which is the general definition of function $g_{\overline{\pi}_{v_s,v_a}}$, is used in this section to define the notion of a data loss along a path $\overline{\pi}_{v,v'}$ because the notion of a data loss along a path underlies the semantics of correlation constraints.

Let v and v' be any two nodes such that $v \rightsquigarrow v'$ and t be any release time in the set $\mathbb{A}_v$. Then, v is said to have a data loss at time t along a path $\overline{\pi}_{v,v'}$ if and only if $g_{\overline{\pi}_{v,v'}}(t) = \infty$. Intuitively, a data loss along the path $\overline{\pi}_{v,v'}$ happens when an event that v reads at time t is lost along some arc (v_p, v_c) on the path $\overline{\pi}_{v,v'}$. While it is obvious that a data loss along an arc cannot keep happening all the time, it is less obvious whether the same applies to a data loss along a path, especially when there are many nodes along the path and the configuration of the node periods happens to maximize the chance of having a data loss along some, if not all, arcs.

> In fact, given an arbitrary DAG and an arbitrary end-to-end path $\overline{\pi}_{v_s,v_a}$, the possibility that every data item flowing from v_s to v_a is lost along the path is addressed neither in [30] for the LET MoCC nor in [33] for the MoCC of BET with precedence constraints.

However, the semantics of correlation constraints is unsound if it is indeed possible that every data item flowing from v_s to v_a is lost along some path constrained by a correlation constraint. Consequently, Theorem 1 is developed in the next paragraph to show that it is indeed not the case, and therefore, the semantics of correlation constraints given in a later section is indeed sound.

Theorem 1 is developed in the following manner:

- The expression $S \oplus c$ denotes the possibly-empty finite set $\{\, s + c \mid s \in S \,\}$ for some $S \subset \mathbb{Q}^+$ and $c \in \mathbb{Q}^+$.

- The following proposition is shown to hold:

Proposition 2. *Let v be some node in $\mathbb{V}$ with period P and $\Delta \in \mathbb{Q}^+$ be a time duration such that Δ is divisible by P. Then, $f_v^{\triangleright}(t + \Delta) = f_v^{\triangleright}(t) + \Delta$.*

Proof.

$$
\begin{aligned}
f_v^{\triangleright}(t+\Delta) &= P_v \left\lceil \frac{t+\Delta}{P_v} \right\rceil \ \ldots \text{ Proposition 1} \\
&= P_v \left\lceil \frac{t}{P_v} + \frac{\Delta}{P_v} \right\rceil \\
&= P_v \left(\left\lceil \frac{t}{P_v} \right\rceil + \frac{\Delta}{P_v} \right) \ \ldots \ \tfrac{\Delta}{P_v} \in \mathbb{N} \\
&= P_v \left\lceil \frac{t}{P_v} \right\rceil + \Delta \\
&= f_v^{\triangleright}(t) + \Delta \ \ldots \text{ Proposition 1}
\end{aligned}
$$

□

- The following proposition can then be shown to hold:

Proposition 3. *Let v be any producer node, v^* be any node such that $v \neq v^*$, H be $\mathrm{lcm}\{P_v, P_{v^*}\}$, and Δ be any duration divisible by H. Then, $\forall t \in \mathbb{A}_v^+ : \mathbb{A}^*_{v,v^*,t+\Delta} = \mathbb{A}^*_{v,v^*,t} \oplus \Delta$.*

Proof.

$$
\begin{aligned}
&\forall t \in \mathbb{A}_v^+ : \mathbb{A}^*_{v,v^*,t+\Delta} = \mathbb{A}^*_{v,v^*,t} \oplus \Delta \\
\Leftrightarrow &\text{(by Definition 5)} \\
&\forall t \in \mathbb{A}_v^+ : \left\{ f^{\triangleright}_{v^*}(t+\Delta) + kP_{v^*} \;\middle|\; f^{\triangleright}_{v^*}(t+\Delta) + kP_{v^*} < t + P_v + \Delta, k \in \mathbb{N} \right\} \\
&= \left\{ f^{\triangleright}_{v^*}(t) + kP_{v^*} \;\middle|\; f^{\triangleright}_{v^*}(t) + kP_{v^*} < t + P_v, k \in \mathbb{N} \right\} \oplus \Delta \\
\Leftrightarrow &\text{(by Proposition 2)} \\
&\forall t \in \mathbb{A}_v^+ : \left\{ f^{\triangleright}_{v^*}(t) + kP_{v^*} + \Delta \;\middle|\; f^{\triangleright}_{v^*}(t) + kP_{v^*} + \Delta < t + P_v + \Delta, k \in \mathbb{N} \right\} \\
&= \left\{ f^{\triangleright}_{v^*}(t) + kP_{v^*} \;\middle|\; f^{\triangleright}_{v^*}(t) + kP_{v^*} < t + P_v, k \in \mathbb{N} \right\} \oplus \Delta \\
\Leftrightarrow &\text{(by algebra and the definition of } S \oplus c) \\
&\forall t \in \mathbb{A}_v^+ : \left\{ f^{\triangleright}_{v^*}(t) + kP_{v^*} + \Delta \;\middle|\; f^{\triangleright}_{v^*}(t) + kP_{v^*} < t + P_v, k \in \mathbb{N} \right\} \\
&= \left\{ f^{\triangleright}_{v^*}(t) + kP_{v^*} + \Delta \;\middle|\; f^{\triangleright}_{v^*}(t) + kP_{v^*} < t + P_v, k \in \mathbb{N} \right\}
\end{aligned}
$$

□

- The following lemma can then be shown to hold:

Lemma 1 ($g_{\overline{\pi}_{v,v'}}$ is periodic)**.** *Let H be* $\operatorname{lcm}(\{ P_v \mid (v,\cdot) \in \overline{\pi}_{v,v'} \} \cup \{P_{v'}\})$*, and Δ be any duration divisible by H. Then,* $\forall t \in \mathbb{A}_v : g_{\overline{\pi}_{v,v'}}(t) = g_{\overline{\pi}_{v,v'}}(t+\Delta)$.

Proof. By induction. Base case ($|\overline{\pi}_{v,v'}| = 1$):

$$
\begin{aligned}
&\forall t \in \mathbb{A}_v : g_{\overline{\pi}_{v,v'}}(t) = g_{\overline{\pi}_{v,v'}}(t+\Delta) \\
\Leftrightarrow &\text{(by Definition 6)} \\
&\forall t \in \mathbb{A}_v : \min \mathbb{A}^*_{v,v',t+P_v} + P_{v'} - t = \min \mathbb{A}^*_{v,v',t+\Delta+P_v} + P_{v'} - (t+\Delta) \\
\Leftrightarrow &\text{(by algebra)} \\
&\forall t \in \mathbb{A}_v : \min \mathbb{A}^*_{v,v',t+P_v} + \Delta = \min \mathbb{A}^*_{v,v',t+P_v+\Delta} \\
\Leftrightarrow &\text{(owing to } \mathbb{A}_v = \left\{ t - P_v \;\middle|\; t \in \mathbb{A}_v^+ \right\} \text{ by Definition 2 and Definition 4)} \\
&\forall t \in \mathbb{A}_v^+ : \min \mathbb{A}^*_{v,v',t} + \Delta = \min \mathbb{A}^*_{v,v',t+\Delta} \\
\Leftrightarrow &\text{(owing to } \min U = \min V \text{ iff } U = V \text{ and owing to } (\min S) + c = \min(S \oplus c)) \\
&\forall t \in \mathbb{A}_v^+ : \mathbb{A}^*_{v,v',t} \oplus \Delta = \mathbb{A}^*_{v,v',t+\Delta} \\
\Leftrightarrow &\text{(by Proposition 3)} \\
&\text{True}
\end{aligned}
$$

Inductive case ($|\overline{\pi}_{v,v'}| \geq 2$): Let (v, v'') be the arc in $\overline{\pi}_{v,v'}$ and $\overline{\pi}_{v'',v'}$ be $\overline{\pi}_{v,v'} \setminus \{(v,v'')\}$ and $\forall t'' \in \mathbb{A}_{v''}$:

$g_{\overline{\pi}_{v'',v'}}(t'') = g_{\overline{\pi}_{v'',v'}}(t''+\Delta)$ be the induction hypothesis.

$\forall t \in \mathbb{A}_v : g_{\overline{\pi}_{v,v'}}(t) = g_{\overline{\pi}_{v,v'}}(t+\Delta)$

$\Leftrightarrow$ (by Definition 6)

$\forall t \in \mathbb{A}_v : \min\left(\left\{\infty\right\} \cup \left\{\, t^* + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t+P_v} \,\right\}\right) - t$

$= \min\left(\left\{\infty\right\} \cup \left\{\, t^* + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t+\Delta+P_v} \,\right\}\right) - (t+\Delta)$

$\Leftrightarrow$ (by algebra and owing to $\mathbb{A}_v = \left\{\, t - P_v \;\middle|\; t \in \mathbb{A}^+_v \,\right\}$ by Definition 2 and Definition 4)

$\forall t \in \mathbb{A}^+_v : \min\left(\left\{\infty\right\} \cup \left\{\, t^* + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t} \,\right\}\right) + \Delta$

$= \min\left(\left\{\infty\right\} \cup \left\{\, t^* + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t+\Delta} \,\right\}\right)$

$\Leftrightarrow$ (owing to $\min U = \min V$ iff $U = V$ and owing to $(\min S) + c = \min(S \oplus c)$)

$\forall t \in \mathbb{A}^+_v : \left(\left\{\infty\right\} \cup \left\{\, t^* + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t} \,\right\}\right) \oplus \Delta$

$= \left(\left\{\infty\right\} \cup \left\{\, t^* + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t+\Delta} \,\right\}\right)$

$\Leftrightarrow$ (owing to $(\infty + \Delta) = \infty$ and owing to $(\{\infty\} \cup U) = (\{\infty\} \cup V)$ if $U = V$)

$\forall t \in \mathbb{A}^+_v : \left\{\, t^* + \Delta + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t} \,\right\} = \left\{\, t^* + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t+\Delta} \,\right\}$

$\Leftrightarrow$ (owing to $\mathbb{A}^*_{v,v'',t+\Delta} = \left\{\, t^* + \Delta \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t} \,\right\}$ by Definition 5)

$\forall t \in \mathbb{A}^+_v : \left\{\, t^* + \Delta + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t} \,\right\} = \left\{\, t^* + \Delta + g_{\overline{\pi}_{v'',v'}}(t^*+\Delta) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t} \,\right\}$

$\Leftrightarrow$ (by the induction hypothesis)

True □

- The following proposition is also shown to hold:

Proposition 4 (Data Loss along an Arc is Temporary)**.** *Let* $(v, v'') \in \mathbb{E}$*. Then,* $\exists t \in \mathbb{A}^+_v : \left|\mathbb{A}^*_{v,v'',t}\right| > 0$.

Proof. By contradiction. Suppose $\left|\mathbb{A}^*_{v,v'',t}\right| = 0$ for all $t \in \mathbb{A}^+_v$. Then, by Definition 5, $f^{\triangleright}_{v''}(t) + kP_{v''} \geq t + P_v$ for all $t \in \mathbb{A}^+_v$ and $k \in \mathbb{N}$. But, this fails for $t = H$ and $k = 0$ where $H = \operatorname{lcm}\left\{P_v, P_{v''}\right\}$ because $f^{\triangleright}_{v''}(H) \geq H + P_v$ iff $P_{v''}\left\lceil \frac{H}{P_{v''}} \right\rceil \geq H + P_v$ iff $\left\lceil \frac{H}{P_{v''}} \right\rceil \geq \frac{H}{P_{v''}} + \frac{P_v}{P_{v''}}$, which, owing to $\frac{H}{P_{v''}} \in \mathbb{N}^+$, is equivalent to $\frac{H}{P_{v''}} \geq \frac{H}{P_{v''}} + \frac{P_v}{P_{v''}}$, resulting in a contradiction due to $\frac{P_v}{P_{v''}} > 0$ as $P_v, P_{v''} \in \mathbb{Q}^+$. □

- The following proposition is also shown to hold:

Proposition 5 (Node's Previous Release Time)**.** *Let* v *be some node in* $\mathbb{V}$*. Then, the latest release time of* v *found not later than time* $t \in \mathbb{Q}^{\geq 0}$ *is given by the function* $f^{\triangleleft}_v : \mathbb{Q}^{\geq 0} \to \mathbb{A}_v$ *defined as* $f^{\triangleleft}_v(t) = P_v \left\lfloor \frac{t}{P_v} \right\rfloor$.

Proof. By the definition of floor operator, $\left\lfloor \frac{t}{P_v} \right\rfloor \in \mathbb{N}$. Consequently, letting n being $\left\lfloor \frac{t}{P_v} \right\rfloor$, the definition of $f^{\triangleleft}_v$ is equivalent to the definition of $f_{\mathrm{r},v}$, and therefore, by Definition 1, $f^{\triangleleft}_v$ indeed gives the release times of some node v. Now it remains to show that $f^{\triangleleft}_v$ indeed gives the release time of v that is either equivalent to t or the latest one before t.

Suppose v has a release time at time t'. Then, by Definition 1, t' is divisible by P_v. If $t = t'$, then t is also divisible by P_v, and floor operator by its definition leaves the result of the division unchanged. Multiplying the result by P_v will produce t', which is v's release time that is equivalent to t. Otherwise, t satisfies the inequality $t' < t < t' + P_v$. Dividing the inequality by P_v gives $n < \frac{t}{P_v} < n'$ for some $n, n' \in \mathbb{N}$ because t' is divisible by P_v. By the definition of floor operator, $\frac{t}{P_v}$ will be mapped to n, which when multiplied by P_v gives t', the latest v's release time before t. □

- The following proposition can then be shown to hold:

Proposition 6. *Let* v *be any producer node,* v^* *be any node such that* $v \neq v^*$*, and* t^* *be any time in* $\mathbb{A}_{v^*}$ *such that* $f^{\triangleleft}_v(t^*) \in \mathbb{A}^+_v$*. Then,* $t^* \in \mathbb{A}^*_{v,v^*,f^{\triangleleft}_v(t^*)}$.

Proof. Let t be $f_v^{\triangleleft}(t^*)$. Then, t is in $\mathbb{A}_v^+$ by the definition of t^*. By Proposition 5, t^* satisfies $t \leq t^* < t + P_v$. Since $t \leq t^*$ and $t^* \in \mathbb{A}_{v^*}$ and $f_{v^*}^{\triangleright}(t) \in \mathbb{A}_{v^*}$, it holds that $f_{v^*}^{\triangleright}(t) \leq t^*$. So, Definition 2, $f_{v^*}^{\triangleright}(t) + kP_{v^*} = t^*$ for some $k \in \mathbb{N}$. Since $t^* < t + P_v$, it holds that $f_{v^*}^{\triangleright}(t) + kP_{v^*} = t^* < t + P_v$ for some $k \in \mathbb{N}$, and by Definition 5, $t^* \in \mathbb{A}^*_{v,v^*,t}$. □

- The following proposition can finally be shown to hold:

Proposition 7 ($\mathbb{G}_{\overline{\pi}_{v,v'}}$ contains a positive rational)**.** *Let v and v' be any two nodes such that $v \rightsquigarrow v'$. Then, $\mathbb{G}_{\overline{\pi}_{v,v'}} \cap \mathbb{Q}^+ \neq \emptyset$.*

Proof. By induction noting that $\mathbb{G}_{\overline{\pi}_{v,v'}} \cap \mathbb{Q}^+ \neq \emptyset$ iff $q \in \mathbb{G}_{\overline{\pi}_{v,v'}}$ for some $q \in \mathbb{Q}^+$. Base case ($|\overline{\pi}_{v,v'}| = 1$): suppose $q \notin \mathbb{G}_{\overline{\pi}_{v,v'}}$ for all $q \in \mathbb{Q}^+$. Then, by Definition 7, it is the case that $\forall t \in \mathbb{A}_v : g_{\overline{\pi}_{v,v'}}(t) = \infty$, which by the base case of Definition 6 is equivalent to $\forall t \in \mathbb{A}_v : \left|\mathbb{A}^*_{v,v',t+P_v}\right| = 0$, which, owing to $\mathbb{A}_v = \{\, t - P_v \mid t \in \mathbb{A}_v^+ \,\}$, is equivalent to $\forall t \in \mathbb{A}_v^+ : \left|\mathbb{A}^*_{v,v',t}\right| = 0$. This contradicts Proposition 4 and shows for the base case that $\mathbb{G}_{\overline{\pi}_{v,v'}} \cap \mathbb{Q}^+ \neq \emptyset$. Inductive case ($|\overline{\pi}_{v,v'}| \geq 2$): let (v, v'') be the arc in $\overline{\pi}_{v,v'}$ and $\overline{\pi}_{v'',v'}$ be $\overline{\pi}_{v,v'} \setminus \{(v, v'')\}$ and $\mathbb{G}_{\overline{\pi}_{v'',v'}} \cap \mathbb{Q}^+ \neq \emptyset$ be the induction hypothesis. By Definition 7, the induction hypothesis implies that there exists $t'' \in \mathbb{A}_{v''}$ such that $g_{\overline{\pi}_{v'',v'}}(t'') \neq \infty$. Hence, to assert that $g_{\overline{\pi}_{v,v'}}(t) \neq \infty$ for some $t \in \mathbb{A}_v$ by the inductive case of Definition 6, it remains to show that $t'' \in \mathbb{A}^*_{v,v'',t+P_v}$, which is the case by Proposition 6 if $t + P_v = f_v^{\triangleleft}(t'')$ and $f_v^{\triangleleft}(t'') \in \mathbb{A}_v^+$. By Lemma 1, a selection of $t'' \in \mathbb{A}_{v''}$ can be made such that t'' satisfies not only $g_{\overline{\pi}_{v'',v'}}(t'') \neq \infty$ but also $f_v^{\triangleleft}(t'') \in \mathbb{A}_v^+$. Therefore, it follows that $g_{\overline{\pi}_{v,v'}}(t) \neq \infty$ for $t = f_v^{\triangleleft}(t'') - P_v$, and so by Definition 7, $\mathbb{G}_{\overline{\pi}_{v,v'}} \cap \mathbb{Q}^+ \neq \emptyset$. □

Lastly, Theorem 1 follows to show that it is never the case that every data item flowing from v to v' is always lost along some path connecting v to v'. More importantly, Theorem 1 shows that the semantics of a correlation constraint given in a later section is indeed sound.

Theorem 1 (Data Loss along a Path is Temporary)**.** *Let v and v' be any two nodes such that $v \rightsquigarrow v'$ and $\overline{\pi}_{v,v'}$ be any path in the set $\overline{\mathbb{E}}_{v,v'}$. Then, $\exists t \in \mathbb{A}_v : g_{\overline{\pi}_{v,v'}}(t) \neq \infty$.*

Proof. By contradiction. If $\forall t \in \mathbb{A}_v : g_{\overline{\pi}_{v,v'}}(t) = \infty$, then by Definition 7, $\mathbb{G}_{\overline{\pi}_{v,v'}} \cap \mathbb{Q}^+ = \emptyset$, contradicting Proposition 7. □

Earliest Data Item to Flow along a Path

The previous section starts the first step needed to define the semantics of correlation constraints by dealing with the data loss property along a path. This section, on the other hand, takes the second step to define the semantics of correlation constraints by first defining a special time point and lastly showing that the definition is sound.

Definition 8 (Earliest Success Time)**.** *Let v and v' be any two nodes such that $v \rightsquigarrow v'$ and $\overline{\pi}_{v,v'}$ be any path in the set $\overline{\mathbb{E}}_{v,v'}$. Then, the earliest success time denoted $t^{min}_{\overline{\pi}_{v,v'}}$ is the earliest v's release time such that the data item that v reads at that time successfully reaches v' by being communicated over $\overline{\pi}_{v,v'}$ and is formally defined below:*

$$t^{min}_{\overline{\pi}_{v,v'}} = \min\left\{\, t \in \mathbb{A}_v \;\middle|\; g_{\overline{\pi}_{v,v'}}(t) \neq \infty \,\right\}$$

Proposition 8 (Definition 8 is Sound)**.** *Proof.*

$$\begin{aligned}
&\text{Definition 8 is sound}\\
\Leftarrow\ &\text{(by Definition 8)}\\
&\min\left\{\, t \in \mathbb{A}_v \;\middle|\; g_{\overline{\pi}_{v,v'}}(t) \neq \infty \,\right\} \text{ is sound}\\
\Leftarrow\ &\text{(by the definition of } \min)\\
&\left\{\, t \in \mathbb{A}_v \;\middle|\; g_{\overline{\pi}_{v,v'}}(t) \neq \infty \,\right\} \neq \emptyset\\
\Leftarrow\ &\text{(by the set-builder notation's definition)}\\
&\exists t \in \mathbb{A}_v : g_{\overline{\pi}_{v,v'}}(t) \neq \infty\\
\Leftarrow\ &\text{(by Theorem 1)}\\
&\text{True}
\end{aligned}$$

Sensing Time Domain

In dealing with the notion of the earliest data item to flow along a path to define the semantics of correlation constraints, the soundness of the previous section is built upon the section preceding it. This section, on the other hand, takes the last step to define the semantics of correlation constraints by first defining a sensing time domain, and then showing that the definition is sound, and lastly showing that the domain is monotonically increasing along a path.

Definition 9 (Sensing Time Domain)**.** *Let v and v' be any two nodes such that $v \rightsquigarrow v'$ and $\overline{\pi}_{v,v'}$ be any path in the set $\overline{\mathbb{E}}_{v,v'}$ and $t^{min}_{\overline{\pi}_{v,v'}}$ be as defined in Definition 8. Then, the sensing time domain of v' is denoted $\mathbb{D}_{\overline{\pi}_{v,v'}}$ and is defined to be the nonempty infinite set $\left\{ t' \in \mathbb{A}_{v'} \;\middle|\; t' \geq t^{min}_{\overline{\pi}_{v,v'}} + g_{\overline{\pi}_{v,v'}}\left(t^{min}_{\overline{\pi}_{v,v'}}\right) - P_{v'} \right\}$.*

Proposition 9 (Definition 9 is Sound)**.** *Proof.*

$$
\begin{aligned}
&\text{Definition 9 is sound}\\
\Leftarrow\ &\text{(by Definition 9)}\\
&\left\{ t' \in \mathbb{A}_{v'} \;\middle|\; t' \geq t^{\min}_{\overline{\pi}_{v,v'}} + g_{\overline{\pi}_{v,v'}}\left(t^{\min}_{\overline{\pi}_{v,v'}}\right) - P_{v'} \right\} \text{ is sound}\\
\Leftarrow\ &\text{(by the definition of inequality and Definition 6)}\\
&t^{\min}_{\overline{\pi}_{v,v'}} \text{ exists}\\
\Leftarrow\ &\text{(by the definition of existence)}\\
&\text{Definition 8 is sound}\\
\Leftarrow\ &\text{(by Proposition 8)}\\
&\text{True}
\end{aligned}
$$

Lastly, to show that $\mathbb{D}_{\overline{\pi}_{v,v'}}$ is monotonically increasing along some path $\overline{\pi}_{v,v'}$, two steps are needed:

- The following proposition is first shown to hold:

  Proposition 10 (Reading Timesets are Disjoint and Ordered)**.** *Let t and t' be any two release times in $\mathbb{A}^+_v$ such that $t < t'$, v^* be any node such that $v \neq v^*$, and both $\mathbb{A}^*_{v,v^*,t}$ and $\mathbb{A}^*_{v,v^*,t'}$ are nonempty. Then, $\max \mathbb{A}^*_{v,v^*,t} < \min \mathbb{A}^*_{v,v^*,t'}$.*

  Proof. By Definition 5, $\max \mathbb{A}^*_{v,v^*,t} < t + P_v$ and $\min \mathbb{A}^*_{v,v^*,t'} = f^{\triangleright}_{v^*}(t')$. By Definition 4, $t' = t + nP_v$ for some $n \in \mathbb{N}^+$ due to $t < t'$, and by Proposition 1, $t + nP_v \leq f^{\triangleright}_{v^*}(t + nP_v)$. Hence, it holds that $\max \mathbb{A}^*_{v,v^*,t} < t + P_v \leq t + nP_v \leq f^{\triangleright}_{v^*}(t + nP_v) = \min \mathbb{A}^*_{v,v^*,t'}$. □

- The following proposition is then shown to hold:

  Proposition 11 (Success Times are First-In First-Out)**.** *Let t_1 and t_2 be any two release times in $\mathbb{A}_v$ such that $t_1 < t_2$ and $g_{\overline{\pi}_{v,v'}}(t_1), g_{\overline{\pi}_{v,v'}}(t_2) \in \mathbb{Q}^+$. Then, $t_1 + g_{\overline{\pi}_{v,v'}}(t_1) < t_2 + g_{\overline{\pi}_{v,v'}}(t_2)$.*

  Proof. By induction. Base case ($|\overline{\pi}_{v,v'}| = 1$):

$$
\begin{aligned}
&t_1 + g_{\overline{\pi}_{v,v'}}(t_1) < t_2 + g_{\overline{\pi}_{v,v'}}(t_2)\\
\Leftrightarrow\ &\text{(by Definition 6 and the assumption that } g_{\overline{\pi}_{v,v'}}(t_1), g_{\overline{\pi}_{v,v'}}(t_2) \in \mathbb{Q}^+)\\
&t_1 + \min \mathbb{A}^*_{v,v',t_1+P_v} + P_{v'} - t_1 < t_2 + \min \mathbb{A}^*_{v,v',t_2+P_v} + P_{v'} - t_2\\
\Leftrightarrow\ &\text{(by algebra)}\\
&\min \mathbb{A}^*_{v,v',t_1+P_v} < \min \mathbb{A}^*_{v,v',t_2+P_v}\\
\Leftrightarrow\ &\text{(by the definitions of } \min \text{ and } \max)\\
&\max \mathbb{A}^*_{v,v',t_1+P_v} < \min \mathbb{A}^*_{v,v',t_2+P_v}\\
\Leftrightarrow\ &\text{(by Proposition 10)}\\
&\text{True}
\end{aligned}
$$

Inductive case ($|\overline{\pi}_{v,v'}| \geq 2$): let (v, v'') be the arc in $\overline{\pi}_{v,v'}$ and $\overline{\pi}_{v'',v'}$ be $\overline{\pi}_{v,v'} \setminus \{(v, v'')\}$ and the induction hypothesis be:

$$\begin{aligned}\forall t_1'', t_2'' \in \mathbb{A}_{v''} : \big((t_1'' < t_2'') \wedge (g_{\overline{\pi}_{v'',v'}}(t_1'') \in \mathbb{Q}^+) \wedge (g_{\overline{\pi}_{v'',v'}}(t_2'') \in \mathbb{Q}^+)\big) \\ \rightarrow \big(t_1'' + g_{\overline{\pi}_{v'',v'}}(t_1'') < t_2'' + g_{\overline{\pi}_{v'',v'}}(t_2'')\big)\end{aligned}$$

Then, it can be shown that:

$t_1 + g_{\overline{\pi}_{v,v'}}(t_1) < t_2 + g_{\overline{\pi}_{v,v'}}(t_2)$

$\Leftrightarrow$ (by Definition 6 and the assumption that $g_{\overline{\pi}_{v,v'}}(t_1), g_{\overline{\pi}_{v,v'}}(t_2) \in \mathbb{Q}^+$)

$t_1 + \min\Big\{ t_1^* + g_{\overline{\pi}_{v'',v'}}(t_1^*) \;\Big|\; t_1^* \in \mathbb{A}^*_{v,v'',t_1+P_v} \Big\} - t_1$

$< t_2 + \min\Big\{ t_2^* + g_{\overline{\pi}_{v'',v'}}(t_2^*) \;\Big|\; t_2^* \in \mathbb{A}^*_{v,v'',t_2+P_v} \Big\} - t_2$

$\Leftrightarrow$ (by algebra)

$\min\Big\{ t_1^* + g_{\overline{\pi}_{v'',v'}}(t_1^*) \;\Big|\; t_1^* \in \mathbb{A}^*_{v,v'',t_1+P_v} \Big\} < \min\Big\{ t_2^* + g_{\overline{\pi}_{v'',v'}}(t_2^*) \;\Big|\; t_2^* \in \mathbb{A}^*_{v,v'',t_2+P_v} \Big\}$

$\Leftrightarrow$ (by the set-builder notation's definition, the assumption that $g_{\overline{\pi}_{v,v'}}(t_1), g_{\overline{\pi}_{v,v'}}(t_2) \in \mathbb{Q}^+$, and the inductive part of Definition 6)

$\exists t_1^{**} \in \mathbb{A}^*_{v,v'',t_1+P_v} : \exists t_2^{**} \in \mathbb{A}^*_{v,v'',t_2+P_v} : g_{\overline{\pi}_{v'',v'}}(t_1^{**}) \in \mathbb{Q}^+ \wedge g_{\overline{\pi}_{v'',v'}}(t_2^{**}) \in \mathbb{Q}^+ \wedge$

$t_1^{**} + g_{\overline{\pi}_{v'',v'}}(t_1^{**}) = \min\Big\{ t_1^* + g_{\overline{\pi}_{v'',v'}}(t_1^*) \;\Big|\; t_1^* \in \mathbb{A}^*_{v,v'',t_1+P_v} \Big\}$

$<$

$t_2^{**} + g_{\overline{\pi}_{v'',v'}}(t_2^{**}) = \min\Big\{ t_2^* + g_{\overline{\pi}_{v'',v'}}(t_2^*) \;\Big|\; t_2^* \in \mathbb{A}^*_{v,v'',t_2+P_v} \Big\}$

$\Leftrightarrow$ (by the induction hypothesis owing to the fact that $t_1^{**} < t_2^{**}$ by Proposition 10)

True $\square$

Proposition 12 can now show that $\mathbb{D}_{\overline{\pi}_{v,v'}}$ is monotonically increasing along some path $\overline{\pi}_{v,v'}$.

Proposition 12 (Sensing Time Domain is Monotonically Increasing)**.** *Let v and v' be any two nodes such that $v \rightsquigarrow v'$ and $\overline{\pi}_{v,v'}$ be any path in the set $\overline{\mathbb{E}}_{v,v'}$ such that $|\overline{\pi}_{v,v'}| \geq 2$ and (v, v'') be the arc in $\overline{\pi}_{v,v'}$ and $\overline{\pi}_{v'',v'}$ be $\overline{\pi}_{v,v'} \setminus \{(v, v'')\}$. Then, $\mathbb{D}_{\overline{\pi}_{v,v'}} \subseteq \mathbb{D}_{\overline{\pi}_{v'',v'}}$.*

Proof.

$\mathbb{D}_{\overline{\pi}_{v,v'}} \subseteq \mathbb{D}_{\overline{\pi}_{v'',v'}}$

$\Leftrightarrow$ (by Definition 9)

$\left\{ t' \in \mathbb{A}_{v'} \;\middle|\; t' \geq t^{\texttt{min}}_{\overline{\pi}_{v,v'}} + g_{\overline{\pi}_{v,v'}}\left(t^{\texttt{min}}_{\overline{\pi}_{v,v'}}\right) - P_{v'} \right\} \subseteq \left\{ t' \in \mathbb{A}_{v'} \;\middle|\; t' \geq t^{\texttt{min}}_{\overline{\pi}_{v'',v'}} + g_{\overline{\pi}_{v'',v'}}\left(t^{\texttt{min}}_{\overline{\pi}_{v'',v'}}\right) - P_{v'} \right\}$

$\Leftrightarrow$ (by the definitions of the set-builder notation and set-inclusion)

$t^{\texttt{min}}_{\overline{\pi}_{v,v'}} + g_{\overline{\pi}_{v,v'}}\left(t^{\texttt{min}}_{\overline{\pi}_{v,v'}}\right) \geq t^{\texttt{min}}_{\overline{\pi}_{v'',v'}} + g_{\overline{\pi}_{v'',v'}}\left(t^{\texttt{min}}_{\overline{\pi}_{v'',v'}}\right)$

$\Leftrightarrow$ (by Definition 6, the assumption that $|\overline{\pi}_{v,v'}| \geq 2$, and Definition 8)

$t^{\texttt{min}}_{\overline{\pi}_{v,v'}} + \min\left\{ t^* + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t^{\texttt{min}}_{\overline{\pi}_{v,v'}}+P_v} \right\} - t^{\texttt{min}}_{\overline{\pi}_{v,v'}} \geq t^{\texttt{min}}_{\overline{\pi}_{v'',v'}} + g_{\overline{\pi}_{v'',v'}}\left(t^{\texttt{min}}_{\overline{\pi}_{v'',v'}}\right)$

$\Leftrightarrow$ (by algebra)

$\min\left\{ t^* + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t^{\texttt{min}}_{\overline{\pi}_{v,v'}}+P_v} \right\} \geq t^{\texttt{min}}_{\overline{\pi}_{v'',v'}} + g_{\overline{\pi}_{v'',v'}}\left(t^{\texttt{min}}_{\overline{\pi}_{v'',v'}}\right)$

$\Leftrightarrow$ (by the set-builder notation's definition, Definition 8, and the inductive part of Definition 6)

$\exists t^{**} \in \mathbb{A}^*_{v,v'',t^{\texttt{min}}_{\overline{\pi}_{v,v'}}+P_v} : g_{\overline{\pi}_{v'',v'}}(t^{**}) \in \mathbb{Q}^+ \wedge$

$\min\left\{ t^* + g_{\overline{\pi}_{v'',v'}}(t^*) \;\middle|\; t^* \in \mathbb{A}^*_{v,v'',t^{\texttt{min}}_{\overline{\pi}_{v,v'}}+P_v} \right\} = t^{**} + g_{\overline{\pi}_{v'',v'}}(t^{**}) \geq t^{\texttt{min}}_{\overline{\pi}_{v'',v'}} + g_{\overline{\pi}_{v'',v'}}\left(t^{\texttt{min}}_{\overline{\pi}_{v'',v'}}\right)$

$\Leftrightarrow$ (by Proposition 11 owing to the fact that $t^{**} \in \mathbb{A}_{v''}$ by Definition 5 and the fact that $t^{**} \geq t^{\texttt{min}}_{\overline{\pi}_{v'',v'}}$ and $g_{\overline{\pi}_{v'',v'}}\left(t^{\texttt{min}}_{\overline{\pi}_{v'',v'}}\right) \in \mathbb{Q}^+$ by Definition 8)

True □

Correlation Constraints

Consider the case when $v_{\texttt{s}} \rightsquigarrow v_{\texttt{a}}$ for some $v_{\texttt{s}}$ and $v_{\texttt{a}}$, and there is exactly one path from $v_{\texttt{s}}$ to $v_{\texttt{a}}$ denoted $\overline{\pi}_{v_{\texttt{s}},v_{\texttt{a}}}$. Then, whenever $v_{\texttt{a}}$ is released at every time $t' \in \mathbb{A}_{v_{\texttt{a}}}$, $v_{\texttt{a}}$ will read the buffer of the arc $(v'', v_{\texttt{a}}) \in \overline{\pi}_{v_{\texttt{s}},v_{\texttt{a}}}$. Let α denote the data read from the buffer. Then, α is the result of processing either a data item read by $v_{\texttt{s}}$ when $v_{\texttt{s}}$ is released at some time $t \in \mathbb{A}_{v_{\texttt{s}}}$ or the initial value of the buffer of some arc in $\overline{\pi}_{v_{\texttt{s}},v_{\texttt{a}}}$. In the former case, α has t as its sensing time, while in the latter case, α has no sensing time. In the former case, the sensing time is denoted by $h_{\overline{\pi}_{v_{\texttt{s}},v_{\texttt{a}}}}(t')$ with $h_{\overline{\pi}_{v_{\texttt{s}},v_{\texttt{a}}}}$ being the function defined in Definition 10. In the latter case, $h_{\overline{\pi}_{v_{\texttt{s}},v_{\texttt{a}}}}$ is undefined at time t' since t' is not in the function's domain, which is defined in Definition 9. In general, sensing time is defined in Definition 10 for any consumer node in $\overline{\pi}_{v_{\texttt{s}},v_{\texttt{a}}}$, including $v_{\texttt{a}}$. More importantly, Proposition 13 shows that Definition 10 is sound.

Definition 10 (Sensing Time)**.** *Let v and v' be any two nodes such that $v \rightsquigarrow v'$, $\mathbb{D}_{\overline{\pi}_{v,v'}}$ be the sensing time domain defined in Definition 9, v'' be any node such that $(v, v'') \in \overline{\pi}_{v,v'}$, and $\overline{\pi}_{v'',v'}$ be $\overline{\pi}_{v,v'} \setminus \{(v, v'')\}$. Then, function $h_{\overline{\pi}_{v,v'}} : \mathbb{D}_{\overline{\pi}_{v,v'}} \rightarrow \mathbb{A}_v$ is defined inductively below.*

Base case ($|\overline{\pi}_{v,v'}| = 1$): $h_{\overline{\pi}_{v,v'}}(t') = f^{\triangleleft}_v(\quad t' \quad) - P_v$.

Inductive case ($|\overline{\pi}_{v,v'}| \geq 2$): $h_{\overline{\pi}_{v,v'}}(t') = f^{\triangleleft}_v\left(h_{\overline{\pi}_{v'',v'}}(t')\right) - P_v$.

Proposition 13 (Definition 10 is Sound)**.** *Proof.* The base case ($|\overline{\pi}_{v,v'}| = 1$) of of Definition 10 is sound iff Definition 9 is sound, which is true by Proposition 9. On the other hand, the inductive case ($|\overline{\pi}_{v,v'}| \geq 2$) of Definition 10 is sound iff the definition of $\mathbb{D}_{\overline{\pi}_{v,v'}}$ is sound and the definition of $\mathbb{D}_{\overline{\pi}_{v'',v'}}$ is sound and $\mathbb{D}_{\overline{\pi}_{v,v'}} \subseteq \mathbb{D}_{\overline{\pi}_{v'',v'}}$ (otherwise, there exists some $t' \in \mathbb{D}_{\overline{\pi}_{v,v'}}$ for which $h_{\overline{\pi}_{v'',v'}}(t')$ is undefined), all of which are true because the definitions of $\mathbb{D}_{\overline{\pi}_{v,v'}}$ and $\mathbb{D}_{\overline{\pi}_{v'',v'}}$ are sound by Proposition 9 and $\mathbb{D}_{\overline{\pi}_{v,v'}} \subseteq \mathbb{D}_{\overline{\pi}_{v'',v'}}$ by Proposition 12.

Definition 10 is illustrated in Figure 4.6 by referring to Figure 4.3(e) that shows that $\texttt{v}_4$ reads data items from the buffer of arc $(\texttt{v}_1, \texttt{v}_4)$. By Definition 9, the data items have no sensing time when $\texttt{v}_4$ is released at time zero. In contrast, when $\texttt{v}_4$ is released at other times, the data items have sensing times. The sensing times of the data items read by $\texttt{v}_4$ is determined straightforwardly using the base case ($|\overline{\pi}_{\texttt{v}_1,\texttt{v}_4}| = 1$).

On the other hand, if there are exactly two different paths from $v_{\texttt{s}}$ to $v_{\texttt{a}}$ denoted $\overline{\pi}_{v_{\texttt{s}},v_{\texttt{a}}}$ and $\overline{\pi}'_{v_{\texttt{s}},v_{\texttt{a}}}$, respectively, such that $\overline{\pi}_{v_{\texttt{s}},v_{\texttt{a}}}$ and $\overline{\pi}'_{v_{\texttt{s}},v_{\texttt{a}}}$ have only $v_{\texttt{s}}$ and $v_{\texttt{a}}$ as their common nodes, then whenever $v_{\texttt{a}}$ is released, $v_{\texttt{a}}$ will read the buffers of $(v_1, v_{\texttt{a}}) \in \overline{\pi}_{v_{\texttt{s}},v_{\texttt{a}}}$ and $(v_2, v_{\texttt{a}}) \in \overline{\pi}'_{v_{\texttt{s}},v_{\texttt{a}}}$ with $v_1 \neq v_2$. Let α_1 and α_2 denote the data items

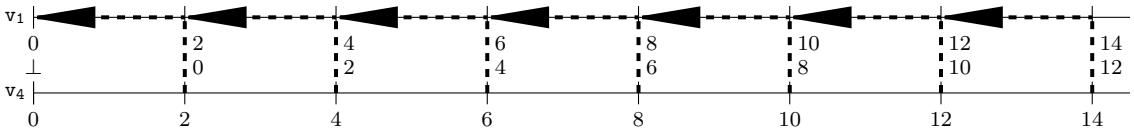


Figure 4.6: The nodes, timelines, ticks, and t values are identical to those of Figure 4.5. Each tick on $\mathtt{v}_4$'s timeline has above it the value of $h_{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}(t)$ or $\perp$ if $h_{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}$ is undefined at t where $\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4} = \{(\mathtt{v}_1, \mathtt{v}_4)\}$ (the dashed zigzag arrows point out the sensing times of the data items read by $\mathtt{v}_4$).

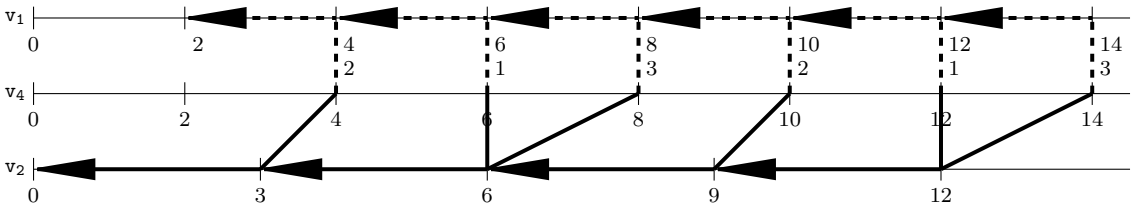


Figure 4.7: Release timelines of nodes $\mathtt{v}_1$ ($P_1 = 2$), $\mathtt{v}_4$ ($P_4 = 2$), and $\mathtt{v}_2$ ($P_2 = 3$) shown in Figure 4.3(e). Each tick has below it the global time t and above it only on $\mathtt{v}_4$'s timeline at every t that has both $h_{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}$ and $h_{\overline{\pi}_{\mathtt{v}_2,\mathtt{v}_4}}$ defined, the value of $\left|h_{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}(t) - h_{\overline{\pi}_{\mathtt{v}_2,\mathtt{v}_4}}(t)\right|$. A zigzag arrow starts at some t and points to the value of $h_{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}(t)$ if dashed or $h_{\overline{\pi}_{\mathtt{v}_2,\mathtt{v}_4}}(t)$ if solid.

that $v_{\mathtt{a}}$ reads from the buffers of $(v_1, v_{\mathtt{a}})$ and $(v_2, v_{\mathtt{a}})$, respectively. Then, α_1 and α_2 may have different sensing times. A correlation constraint therefore can be specified to ensure that the absolute difference in the sensing times is less than or equal to some given threshold denoted Z with $Z \in \mathbb{Q}^{\geq 0}$. More generally, a correlation constraint seeks to bound the absolute difference in the sensing times of the data items that a *confluent* node reads from two or more buffers belonging to two or more different paths. A confluent node is denoted $v^{\Diamond}$ and is a member of some confluent nodeset defined in Definition 11.

Definition 11 (Confluent Nodeset)**.** *Let v_1 and v_2 be any two nodes that are not necessarily distinct, v' be any node such that $v_1 \rightsquigarrow v'$ and $v_2 \rightsquigarrow v'$, and $\overline{\pi}_{v_1,v'}$ and $\overline{\pi}_{v_2,v'}$ be any two distinct paths. Then, the confluent nodeset of the two paths is the nonempty finite set denoted $\mathbb{V}^{\Diamond}_{\overline{\pi}_{v_1,v'},\overline{\pi}_{v_2,v'}}$ and defined below (a confluent nodeset is nonempty with either v' or some other node dominating v' as its member and is finite due to the number of nodes in $\overline{\pi}_{v_1,v'}$ and $\overline{\pi}_{v_2,v'}$ being finite):*

$$\left\{\, v'' \in \mathbb{V} \;\middle|\; (\cdot, v'') \in \left(\overline{\pi}_{v_1,v'} \setminus \overline{\pi}_{v_2,v'}\right) \right\} \cap \left\{\, v'' \in \mathbb{V} \;\middle|\; (\cdot, v'') \in \left(\overline{\pi}_{v_2,v'} \setminus \overline{\pi}_{v_1,v'}\right) \right\}$$

Definition 11 is illustrated by considering the case when there are exactly two different paths from $v_{\mathtt{s}}$ to $v_{\mathtt{a}}$ such that the two paths have only $v_{\mathtt{s}}$ and $v_{\mathtt{a}}$ as their common nodes, for example, $\overline{\pi}_{v_{\mathtt{s}},v_{\mathtt{a}}} = \{(\mathtt{v}_1, \mathtt{v}_2), (\mathtt{v}_2, \mathtt{v}_4)\}$ and $\overline{\pi}'_{v_{\mathtt{s}},v_{\mathtt{a}}} = \{(\mathtt{v}_1, \mathtt{v}_3), (\mathtt{v}_3, \mathtt{v}_4)\}$. Then, the confluent nodeset of the two paths is $\{v_{\mathtt{a}}\}$, which is $\{\mathtt{v}_4\}$ in the given example. If in addition to $v_{\mathtt{s}}$ and $v_{\mathtt{a}}$ the two paths have other common nodes, for example, $\overline{\pi}_{v_{\mathtt{s}},v_{\mathtt{a}}} = \{(\mathtt{v}_1, \mathtt{v}_2), (\mathtt{v}_2, \mathtt{v}_3), (\mathtt{v}_3, \mathtt{v}_5), (\mathtt{v}_5, \mathtt{v}_6)\}$ and $\overline{\pi}'_{v_{\mathtt{s}},v_{\mathtt{a}}} = \{(\mathtt{v}_1, \mathtt{v}_2), (\mathtt{v}_2, \mathtt{v}_4), (\mathtt{v}_4, \mathtt{v}_5), (\mathtt{v}_5, \mathtt{v}_6)\}$, then the confluent nodeset of the two paths is $\{v^{\Diamond}\}$ with $v^{\Diamond}$ being the node where the two paths conflate, which is $\mathtt{v}_5$ in the given example. In general, the two paths that define a confluent nodeset originate from two distinct sensor nodes.

Figure 4.3(e) shows a correlation constraint that is applied on two distinct sensor nodes $\mathtt{v}_1$ and $\mathtt{v}_2$ and one actuator node $\mathtt{v}_4$ with the threshold Z. The constraint covers exactly three paths $\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4} = \{(\mathtt{v}_1, \mathtt{v}_4)\}$, $\overline{\pi}_{\mathtt{v}_2,\mathtt{v}_4} = \{(\mathtt{v}_2, \mathtt{v}_4)\}$, and $\overline{\pi}'_{\mathtt{v}_1,\mathtt{v}_4} = \{(\mathtt{v}_1, \mathtt{v}_3), (\mathtt{v}_3, \mathtt{v}_4)\}$ that define three confluent nodesets $\mathbb{V}^{\Diamond}_{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4},\overline{\pi}_{\mathtt{v}_2,\mathtt{v}_4}}$, $\mathbb{V}^{\Diamond}_{\overline{\pi}_{\mathtt{v}_2,\mathtt{v}_4},\overline{\pi}'_{\mathtt{v}_1,\mathtt{v}_4}}$, and $\mathbb{V}^{\Diamond}_{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4},\overline{\pi}'_{\mathtt{v}_1,\mathtt{v}_4}}$, all of which are equal to $\{\mathtt{v}_4\}$. The constraint limits the absolute difference in the sensing times of the data items that $\mathtt{v}_4$ reads from the buffers of $(\mathtt{v}_1, \mathtt{v}_4)$ and $(\mathtt{v}_2, \mathtt{v}_4)$ as illustrated in Figure 4.7, from the buffers of $(\mathtt{v}_2, \mathtt{v}_4)$ and $(\mathtt{v}_3, \mathtt{v}_4)$, and from the buffers of $(\mathtt{v}_1, \mathtt{v}_4)$ and $(\mathtt{v}_3, \mathtt{v}_4)$. With $\mathbb{H}^{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}_{\overline{\pi}_{\mathtt{v}_2,\mathtt{v}_4}}$, $\mathbb{H}^{\overline{\pi}_{\mathtt{v}_2,\mathtt{v}_4}}_{\overline{\pi}'_{\mathtt{v}_1,\mathtt{v}_4}}$, and $\mathbb{H}^{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}_{\overline{\pi}'_{\mathtt{v}_1,\mathtt{v}_4}}$ being the sets defined by Definition 12, the constraint requires that $\max \mathbb{H}^{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}_{\overline{\pi}_{\mathtt{v}_2,\mathtt{v}_4}} \leq Z$, $\max \mathbb{H}^{\overline{\pi}_{\mathtt{v}_2,\mathtt{v}_4}}_{\overline{\pi}'_{\mathtt{v}_1,\mathtt{v}_4}} \leq Z$, and $\max \mathbb{H}^{\overline{\pi}_{\mathtt{v}_1,\mathtt{v}_4}}_{\overline{\pi}'_{\mathtt{v}_1,\mathtt{v}_4}} \leq Z$.

Definition 12 (Sensing-Time Absolute-Difference Set)**.** *Let v_1 and v_2 be any two nodes that are not necessarily distinct, v' be any node such that $v_1 \rightsquigarrow v'$ and $v_2 \rightsquigarrow v'$, and $\overline{\pi}_{v_1,v'}$ and $\overline{\pi}_{v_2,v'}$ be any two distinct paths.*

Then, the sensing-time absolute-difference set of v' is denoted $\mathbb{H}^{\overline{\pi}_{v_1,v'}}_{\overline{\pi}_{v_2,v'}}$ and defined to be the nonempty set $\left\{ \left| h_{\overline{\pi}_{v_1,v'}}(t') - h_{\overline{\pi}_{v_2,v'}}(t') \right| \;\middle|\; t' \in \left(\mathbb{D}_{\overline{\pi}_{v_1,v'}} \cap \mathbb{D}_{\overline{\pi}_{v_2,v'}} \right) \right\}$.

In general, a correlation constraint is specified on at least one sensor node and exactly one actuator node denoted as the pair $(\mathbb{V}'_{\mathtt{s}}, v_{\mathtt{a}})$. For example, the correlation constraint shown in Figure 4.3(e) is specified on the pair $(\{\mathtt{v}_1, \mathtt{v}_2\}, \mathtt{v}_4)$ using the public API member `Correlation`. Formally, each correlation constraint specified partially constructs both the set $\mathbb{T}_{\mathtt{Cor}}$ and the function $f_{\mathtt{Cor}} : \mathbb{T}_{\mathtt{Cor}} \to \mathbb{Q}^{\geq 0}$, which are the fourth and tenth elements of a 10-tuple in $\mathcal{L}$, respectively. Tice requires and at compile-time Tice library checks that every correlation constraint specifies a nonnegative threshold Z and a nonempty set of sensors that each is connected to the actuator. Therefore, (4.7) describes the semantics of a correlation constraint formally.

$$\max\left(\left\{ \max \mathbb{H}^{\overline{\pi}_{v_{\mathtt{s}},v^{\diamond}}}_{\overline{\pi}_{v'_{\mathtt{s}},v^{\diamond}}} \;\middle|\; \begin{array}{ll} \overline{\pi}_{v_{\mathtt{s}},v_{\mathtt{a}}}, \overline{\pi}_{v'_{\mathtt{s}},v_{\mathtt{a}}} \in \bigcup_{v''_{\mathtt{s}} \in \mathbb{V}'_{\mathtt{s}}} \overline{\mathbb{E}}_{v''_{\mathtt{s}},v_{\mathtt{a}}}, & \overline{\pi}_{v_{\mathtt{s}},v_{\mathtt{a}}} \neq \overline{\pi}_{v'_{\mathtt{s}},v_{\mathtt{a}}}, \\ v^{\diamond} \in \mathbb{V}^{\diamond}_{\overline{\pi}_{v_{\mathtt{s}},v_{\mathtt{a}}},\overline{\pi}_{v'_{\mathtt{s}},v_{\mathtt{a}}}}, \overline{\pi}_{v_{\mathtt{s}},v^{\diamond}} \subseteq \overline{\pi}_{v_{\mathtt{s}},v_{\mathtt{a}}}, & \overline{\pi}_{v'_{\mathtt{s}},v^{\diamond}} \subseteq \overline{\pi}_{v'_{\mathtt{s}},v_{\mathtt{a}}} \end{array} \right\} \cup \left\{ 0 \right\} \right) \leq Z \tag{4.7}$$

Please skip this page.

5 *Tice Decidability and Time-Complexity Analyses*

When it comes to compiling C++ source programs that use C++ active libraries, it is important to keep the programs *decidable* (i.e., their compilations are guaranteed to terminate, producing either error messages or target programs) because people have come to expect it. While the use of ordinary C++ libraries does not have the problem of being *undecidable* (i.e., their compilations will never keep the compiler keep running forever), the use of C++ active libraries have the potential to make their compilations undecidable because they are Turing-complete and therefore can create an infinite loop. Therefore, this work shows that the Tice language is decidable so that any of its implementation (i.e., any Tice library, such as [87]) should be decidable as well. After showing Tice decidability, this work will proceed to analyze the time complexity of one particular Tice library, which is publicly available at [87], and to validate the analysis results using GCC and Clang, two off-the-shelf standard C++ compilers in widespread use. In the rest of this chapter, the term Tice library is used to refer to the implementation at [87].

5.1 The Decidability of Tice

As already described in §4.4, Tice requires a Tice program to be checked for its validity, which includes checking for the presence of a cycle and the expressed real-time constraints. Since the decidability of common algorithms, such as the algorithm to check for the presence of a cycle, are already known, this section only shows the decidability of the real-time constraints, namely end-to-end delay and correlation constraints, because they are defined by this work from the ground up in §4.4.

The Decidability of an End-to-End Delay Constraint

Proposition 14. *The validity of an end-to-end delay constraint given in* (4.6) *is decidable in a finite amount of time.*

Proof. Based on (4.6), the decidability requires first the finiteness of $\overline{\mathbb{E}}_{v_s,v_a}$ and then the finiteness of $\mathbb{G}_{\overline{\pi}_{v,v'}}$ for any $\overline{\pi}_{v,v'} \in \overline{\mathbb{E}}_{v_s,v_a}$. The former is obvious by the fact that the number of distinct paths between any two distinct nodes in a DAG is finite, while the latter follows from Lemma 1. □

The Decidability of a Correlation Constraint

As in the previous section, the decidability of a correlation constraint is shown using a periodicity result. A periodicity result, however, cannot be shown for function $h_{\overline{\pi}_{v,v'}}$ defined in Definition 10 because it defines a sensing time as a time point instead of a time duration. Therefore, a relative sensing time is defined first below:

Definition 13 (Relative Sensing Time)**.** *Let v and v' be any two nodes such that $v \rightsquigarrow v'$ and $\overline{\pi}_{v,v'}$ be any path in the set $\overline{\mathbb{E}}_{v,v'}$. Then, function $h^{\star}_{\overline{\pi}_{v,v'}} : \mathbb{D}_{\overline{\pi}_{v,v'}} \to \mathbb{Q}^{\geq 0}$ defined as $h^{\star}_{\overline{\pi}_{v,v'}}(t') = t' - h_{\overline{\pi}_{v,v'}}(t')$ gives the relative sensing time of the data item found in the buffer of arc $(v'', v') \in \overline{\pi}_{v,v'}$ at time $t' \in \mathbb{D}_{\overline{\pi}_{v,v'}}$.*

It can now be shown below that a relative sensing time is periodic:

Proposition 15 ($h^{\star}_{\overline{\pi}_{v,v'}}$ is Periodic)**.** *Let H be $\operatorname{lcm}(\{ P_v \mid (v,\cdot) \in \overline{\pi}_{v,v'} \} \cup \{P_{v'}\})$, and Δ be any duration divisible by H. Then, $\forall t' \in \mathbb{D}_{\overline{\pi}_{v,v'}} : h^{\star}_{\overline{\pi}_{v,v'}}(t') = h^{\star}_{\overline{\pi}_{v,v'}}(t' + \Delta)$.*

Proof. By induction. By definition, $(t' + \Delta) \in \mathbb{D}_{\overline{\pi}_{v,v'}}$. Base case ($|\overline{\pi}_{v,v'}| = 1$): by Definition 13, the proposition is equivalent to $\forall t' \in \mathbb{D}_{\overline{\pi}_{v,v'}} : t' - h_{\overline{\pi}_{v,v'}}(t') = t' + \Delta - h_{\overline{\pi}_{v,v'}}(t' + \Delta)$ iff $\forall t' \in \mathbb{D}_{\overline{\pi}_{v,v'}} : h_{\overline{\pi}_{v,v'}}(t' + \Delta) = h_{\overline{\pi}_{v,v'}}(t') + \Delta$, which by the base case of Definition 10 is equivalent to $\forall t' \in \mathbb{D}_{\overline{\pi}_{v,v'}} : f_v^{\triangleleft}(t' + \Delta) - P_v = f_v^{\triangleleft}(t') - P_v + \Delta$ iff $\forall t' \in \mathbb{D}_{\overline{\pi}_{v,v'}} : f_v^{\triangleleft}(t' + \Delta) = f_v^{\triangleleft}(t') + \Delta$, which is true by $f_v^{\triangleleft}$ definition. Inductive case ($|\overline{\pi}_{v,v'}| \geq 2$): let (v, v'') be the edge in $\overline{\pi}_{v,v'}$, $\overline{\pi}_{v'',v'}$ be $\overline{\pi}_{v,v'} \setminus \{(v, v'')\}$, and $\forall t'' \in \mathbb{D}_{\overline{\pi}_{v'',v'}} : h^{\star}_{\overline{\pi}_{v'',v'}}(t'') = h^{\star}_{\overline{\pi}_{v'',v'}}(t'' + \Delta)$ be the induction hypothesis, which by Definition 13 is equivalent to $\forall t'' \in \mathbb{D}_{\overline{\pi}_{v'',v'}} : h_{\overline{\pi}_{v'',v'}}(t'' + \Delta) = h_{\overline{\pi}_{v'',v'}}(t'') + \Delta$. By Definition 13, the proposition is equivalent to $\forall t' \in \mathbb{D}_{\overline{\pi}_{v,v'}} : h_{\overline{\pi}_{v,v'}}(t' + \Delta) = h_{\overline{\pi}_{v,v'}}(t') + \Delta$, which by the inductive case of Definition 10 is equivalent to $\forall t' \in \mathbb{D}_{\overline{\pi}_{v,v'}} : f_v^{\triangleleft}\big(h_{\overline{\pi}_{v'',v'}}(t' + \Delta)\big) = f_v^{\triangleleft}\big(h_{\overline{\pi}_{v'',v'}}(t')\big) + \Delta$, which by Proposition 12 is equivalent to $\forall t' \in \mathbb{D}_{\overline{\pi}_{v'',v'}} : f_v^{\triangleleft}\big(h_{\overline{\pi}_{v'',v'}}(t' + \Delta)\big) =$

$f_v^{\triangleleft}\big(h_{\overline{\pi}_{v'',v'}}(t')\big)+\Delta$, which by the induction hypothesis is equivalent to $\forall t' \in \mathbb{D}_{\overline{\pi}_{v'',v'}} : f_v^{\triangleleft}\big(h_{\overline{\pi}_{v'',v'}}(t')+\Delta\big) = f_v^{\triangleleft}\big(h_{\overline{\pi}_{v'',v'}}(t')\big)+\Delta$, which is true by $f_v^{\triangleleft}$ definition. □

Lastly, Proposition 15 is used below to show the decidability of correlation constraints:

Proposition 16. *The validity of a correlation constraint given in* (4.7) *is decidable in a finite amount of time.*

Proof. Let S be the set $\bigcup_{v''_{\mathrm{s}} \in \mathbb{V}'_{\mathrm{s}}} \overline{\mathbb{E}}_{v''_{\mathrm{s}},v_{\mathrm{a}}}$. Then, based on (4.7), the decidability requires first the finiteness of S, second the finiteness of $\mathbb{V}^{\Diamond}_{\overline{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}},\overline{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}}$ for any two distinct paths $\overline{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}}, \overline{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}} \in S$, and then the finiteness of $\mathbb{H}^{\overline{\pi}_{v_{\mathrm{s}},v^{\Diamond}}}_{\overline{\pi}_{v'_{\mathrm{s}},v^{\Diamond}}}$ for the two distinct subpaths $\overline{\pi}_{v_{\mathrm{s}},v^{\Diamond}} \subseteq \overline{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}}$ and $\overline{\pi}_{v'_{\mathrm{s}},v^{\Diamond}} \subseteq \overline{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}$. The first is obvious by two facts: $\mathbb{V}'_{\mathrm{s}} \subseteq \mathbb{V}$ with $\mathbb{V}$ being a finite set, and the number of distinct paths between any two distinct nodes in a DAG is finite. The second is obvious by Definition 11. The last then follows from Proposition 15 because by Definition 12 $\Big|h_{\overline{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}}}(t') - h_{\overline{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}}(t')\Big| = \Big|\Big(t' - h_{\overline{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}}}(t')\Big) - \Big(t' - h_{\overline{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}}(t')\Big)\Big| = \Big|h^{\star}_{\overline{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}}}(t') - h^{\star}_{\overline{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}}(t')\Big| \in \mathbb{H}^{\overline{\pi}_{v_{\mathrm{s}},v^{\Diamond}}}_{\overline{\pi}_{v'_{\mathrm{s}},v^{\Diamond}}}$. As periodic functions, $h^{\star}_{\overline{\pi}_{v_{\mathrm{s}},v_{\mathrm{a}}}}(t')$ and $h^{\star}_{\overline{\pi}_{v'_{\mathrm{s}},v_{\mathrm{a}}}}(t')$ have $n \in \mathbb{N}^+$ and $m \in \mathbb{N}^+$ distinct members, respectively. Hence, $\Big|\mathbb{H}^{\overline{\pi}_{v_{\mathrm{s}},v^{\Diamond}}}_{\overline{\pi}_{v'_{\mathrm{s}},v^{\Diamond}}}\Big| \leq nm \in \mathbb{N}^+$. □

5.2 The Time Complexity of Tice Library

As already pointed out in §4.1, a Tice program has five segments:

- Segment-A that has exactly one instance of `HW`.
- Segment-B that has at least one instance of `Node`.
- Segment-C that has zero or more instances of `Feeder`.
- Segment-D that has zero or more instances of `ETE_delay`.
- Segment-E that has zero or more instances of `Correlation`.

A Tice program with a non-empty Segment-D or Segment-E must not compile if a specified real-time constraint is not respected. Else, if other model properties are valid (e.g., the `Feeder` configuration forms no cycle), the program must generate the real-time code that implements the model. Hence, Tice library compiles a Tice program in two stages: *Tice front-end*, which validates the expression of a Tice model, and *Tice back-end*, which generates the real-time code implementing the model. In analyzing their time complexities, the following notations and definitions are used:

- N is Segment-B's length (i.e., `Node` count).
- K is Segment-C's length (i.e., `Feeder` count).
- K_i is the `Node` count of i^{th} `Feeder`'s parameter list.
- W_E is Segment-D's length (i.e., `ETE_delay` count).
- W_C is Segment-E's length (i.e., `Correlation` count).
- W_i is the `Node` count of i^{th} `Correlation`'s parameter list.
- M is the arc count of a Tice model, that is, $M = \sum_{i=1}^{K} (K_i - 1)$.
- L is the least common multiple of the periods of all producer and consumer nodes divided by the least of the periods.
- $\exp_2$ is the exponentiation function with base two.
- A constrained node means a node that some end-to-end delay or correlation constraint is applied to.
- A constrained end-to-end path means some path that starts from and ends at some constrained nodes, respectively.

Readers uninterested with this section's details can skip to the next by noting that, while the time complexity of the real-time EDSL (segments A up to C) is polynomial due to the DAG of data flows, that of the real-time constraint check (segments D and E) is exponential.

Tice Front-End

Tice front-end traverses the parameter list of `Program` from left to right to check that every segment has the correct instances with the expected count (e.g., if the first three parameters are `HW`, `Feeder`, and `Node`, upon encountering `Feeder`, Tice front-end will conclude that Segment-B is empty and raise an error). This section will only describe checks whose time complexities are not linear.

Uniqueness Checks

Uniqueness checks are performed on the parameter lists of templates `HW`, `Feeder`, and `Correlation` as well as on Segment-B and Segment-C with time complexity $O\left(n^2\right)$ where for `HW` n is the number of core IDs specified in its first parameter using template `Core_ids`, for the i^{th} `Feeder` n is K_i, for `Correlation` n is W_i, for Segment-B n is N, and for Segment-C n is K. The quadratic time-complexity results from a uniqueness check that compares elements pairwise for their non-equality.

DAG Checks

To ensure that Segment-B and Segment-C altogether express a DAG, firstly every node in Segment-B is checked for their uniqueness with time complexity $O\left(N^2\right)$. Each checked node is also assigned an integer index, which grows by one starting at zero, so that the node can uniquely index an array later on.

Afterwards, the parameter list of every `Feeder` is checked to ensure that every node has been specified in Segment-B. The time complexity of this check across Segment-C is therefore $O(N(M+K))$ because every node that is not the last on every parameter list (i.e., every producer node), which expresses one distinct arc giving the term M, and every unaccounted last node (i.e., every consumer node), giving the term K, have to be sought linearly in Segment-B, giving the term N. Additionally, a node uniqueness check is also performed on each parameter list to ensure that (1) there is no multiple arc going from one producer to one consumer (to have multiple arcs, the same producer node must be duplicated on the parameter list) and (2) there is no loop (to have a loop, the consumer node must be duplicated on the parameter list). The time complexity of the uniqueness check across Segment-C is therefore $O\left(\sum_{i=1}^{K} K_i^2\right)$. Lastly, since the check to ensure no multiple arc and no loop can be worked around by specifying another `Feeder` that has the same consumer node, a uniqueness check is performed on Segment-C to ensure that every `Feeder` is unique in terms of their consumer nodes with time complexity $O\left(K^2\right)$. Therefore, the time complexity of analyzing Segment-C is $O\left(N(M+K)+K^2+\sum_{i=1}^{K} K_i^2\right)$.

Lastly, the presence of cycles in Segment-C is checked using an algorithm that visits every arc once. The set of source nodes is derived from some information that is obtained by traversing Segment-C from left to right in the following manner.

Before the traversal starts, a producer-node array is constructed as a boolean array of size N whose elements are initially false. Then, for every `Feeder`, the producer-node array is indexed with the index of every producer node on the `Feeder`'s parameter list to set the indexed element to true. At the end of the traversal, the producer-node array can decide with time complexity $O(1)$ whether a node is a producer node based on the node index. The construction of the array itself, however, has time complexity $O(NK)$ because the array is an immutable object after being updated at every `Feeder`, and hence, all elements in the array are copied to a new array to be updated with the information obtained from the current `Feeder`. In parallel, an adjacency list and a consumer-node list are also constructed and are turned into arrays at the end of the traversal.

The adjacency array is an array of N pointers that each points to an array of node indices whose last index is a sentinel value that signals the end of the array. The adjacency array can then be indexed by a node index to obtain the array that has the indices of the consumer nodes. If the indexing node is not a producer node, which means that the node is either a sink or an isolated node, the obtained array has the sentinel value as its sole element. While the adjacency array searches the consumer nodes of any given node with time complexity $O(1)$, the construction of the array itself has time complexity $O(NM)$ because at every `Feeder`, the adjacency list is updated for every producer node by locating the producer node's position in the list with time complexity $O(N)$ and then queuing the consumer node at that position, a process that is repeated M times for all producer-consumer pairs across every `Feeder` in Segment-C.

On the other hand, the consumer-node array is a boolean array of size N whose initial elements are all false. Every element indexed by the nodes in the consumer-node list is then set to true so that deciding whether a node is a consumer node based on the node index has time complexity $O(1)$. Despite queuing the consumer node of every `Feeder` to the consumer-node list itself has time complexity $O(1)$, the construction of the array itself has time complexity $O(N)$ due to initializing every element to false.

Lastly, using the producer-node and consumer-node arrays, a source-node array is first constructed as a boolean array of size N that decides a membership in the set of source nodes with time complexity $O(1)$. The construction of the source-node array itself has time complexity $O(N)$ as every element in the producer-node and consumer-node arrays is visited to set the corresponding element in the source-node array. The source-node array is then traversed with time complexity $O(N)$ to construct an array of source-node indices, which is the set of source nodes. Therefore, the time complexity of obtaining the set of source nodes is $O(N(M+K))$.

To visit every arc once, a depth-first traversal is performed using the source-node and the adjacency arrays. For every visited node, which starts with a source node, the node indexes the adjacency array to obtain its consumer nodes. If the node is not a sink node, it has at least one consumer node; otherwise, the node has no consumer node. If the node is not a sink node, it is indexed in an already-visited array, which is a boolean array of size N whose elements are initialized to false. If the indexing returns true, it means that Segment-C expresses a cycle, and hence, a compile-time error is raised. Otherwise, the node is indexed in a cycle-absent array, which is also a boolean array of size N whose elements are initialized to false. If the indexing returns true, it means that the arcs on every path that starts at the node and proceeds through any of its consumer nodes have already been visited, and hence, the arcs are not visited again. Otherwise, the node is marked as already visited in the already-visited array and then one of its consumer nodes is visited next. After all of its consumer nodes have been visited, the node is marked as cycle-free in the cycle-absent array. The time complexity of these arc visits is therefore $O(N+M)$ where N is for initializing the already-visited and cycle-absent arrays.

While visiting every arc once, a producer-sink adjacency matrix is also constructed with time complexity $O(MK)$. Letting X and Y be the numbers of producer and sink nodes, respectively, with X being exactly M and Y being at most K, the adjacency matrix is a two-dimensional boolean array of size $X \times Y$ whose elements need to be initialized to false, resulting in the $O(MK)$ time-complexity. The adjacency matrix is needed to decide with time complexity $O(1)$ whether there is a path from a given producer node to a given sink node as described in the next sections.

To conclude, the time complexity of the DAG checks is (5.1).

$$O\left(N^2 + N(M+K) + K^2 + MK + \sum_{i=1}^{K} K_i^2\right) \tag{5.1}$$

End-to-End Delay Checks

If Segment-D is not empty, every node on the parameter list of every `ETE_delay` is checked for their presence in Segment-B with time complexity $O(N)$ using a linear search. Then, the first of the constrained nodes is ensured to be a source node with time complexity $O(1)$ using the source-node array. Similarly, the second of the constrained nodes is ensured to be a sink node with time complexity $O(1)$ using a sink-node array, which is constructed with time complexity $O(N)$ in the same way the source-node array is constructed. Additionally, the constrained nodes are ensured to be connected with time complexity $O(1)$ using the producer-sink adjacency matrix. Lastly, the real-time constraint itself is ensured to be respected by a repeated identification-computation process. The identification-computation process first identifies a constrained end-to-end path and then computes the end-to-end delays experienced by the data generated by the source node that flow along the path. The process is then repeated for every possible constrained end-to-end path.

To identify a constrained end-to-end path, a depth-first traversal is performed starting from the constrained source node using the sink-node array, the adjacency array, and the producer-sink adjacency matrix. Note that an arc is visited more than once if the arc belongs to more than one constrained end-to-end path. Starting from the constrained source node, the visited node is indexed in the adjacency array to obtain the indices of its consumer nodes with time complexity $O(1)$. Every consumer node is then indexed in the sink-node array with time complexity $O(1)$. If true is indexed, a constrained end-to-end path has been identified, and the algorithm described in the paragraph after the next is executed to compute the end-to-end delay along the path. Otherwise, the consumer node and the constrained sink node are indexed in the adjacency matrix with time complexity $O(1)$. If false is indexed, the consumer node is not visited as no path to the sink node exists. Otherwise, the consumer node is visited as the next node in the traversal. The time complexity of identifying all constrained end-to-end paths is therefore $O\left(\sqrt{M}\,\exp_2\left((1/2)\sqrt{M}\right)\right)$ because, as detailed in the next paragraph, the number of arcs visited on every identified path and the number of the paths themselves have as upper bounds constant multiples of $\sqrt{M}$ and $\exp_2\left((1/2)\sqrt{M}\right)$, respectively.

The upper bounds can be derived by considering one particular way of constructing a DAG that maximizes its number of end-to-end paths when it has n nodes with $n \geq 2$ and exactly one source and one sink nodes. This paragraph calls the DAG a maximizing DAG. Starting with only two nodes that has just one end-to-end

path, every new node added such that neither new source nor sink node is introduced will introduce as many new arcs as there are existing nodes. The new arc that points to the sink node creates one new end-to-end path, while the remaining new arcs point to existing nodes whose numbers of possible end-to-end paths that start at those existing nodes, respectively, follow the sequence 2^0, 2^1, 2^2, …, 2^{k-2}, the sum of which together with the new end-to-end path is 2^{k-1} where $k \geq 2$ is the number of the existing nodes. This means that if a maximizing DAG has N nodes, the number of distinct end-to-end paths in the DAG is 2^{N-2} with 2^N being an upper bound. The 2^N upper-bound remains valid when the DAG has multiple source and/or sink nodes because the DAG can be derived from another maximizing DAG with $n = N + 2$ that is identical in every way except for one new node that points to all of the existing source nodes and another new node that is pointed by all of the sink nodes. Additionally, a maximizing DAG with n nodes also has $n(n-1)/2$ arcs because starting with one arc, whenever the k^{th} new node is added, the new node introduces $(k-1)$ new arcs. And, the longest end-to-end path in the DAG has $n-1$ arcs due to the way the DAG is constructed. This means that in general, the number of distinct end-to-end paths and the length of the longest end-to-end path in a DAG with M arcs have upper bounds that are none other than those of a maximizing DAG with n nodes such that $n(n-1)/2 = M$. That is, since $n = \lceil (1/2)\left(1+\sqrt{1+8M}\right)\rceil$, the upper bounds are $\exp_2\left(\lceil (1/2)\left(1+\sqrt{1+8M}\right)\rceil - 2\right)$, which is $O\Big(\exp_2\Big((1/2)\sqrt{M}\Big)\Big)$, and $\lceil (1/2)\left(1+\sqrt{1+8M}\right)\rceil - 1$, which is $O\Big(\sqrt{M}\Big)$, respectively.

Given a constrained end-to-end path, the end-to-end delay along the path is then computed by an algorithm whose time complexity is $O\Big(L\sqrt{M}\Big)$ because the algorithm visits every node on the constrained path but the sink node, the number of which has as an upper bound the number of arcs on the longest constrained end-to-end path, and for every visited node, the algorithm has to check at most L repetitions of the node's computation. This means that the time complexity of computing the end-to-end delay along all constrained end-to-end paths is $O\Big(L\sqrt{M}\exp_2\Big((1/2)\sqrt{M}\Big)\Big)$ because the $O\Big(L\sqrt{M}\Big)$ algorithm is executed once for every distinct constrained end-to-end path, the total number of which is $O\Big(\exp_2\Big((1/2)\sqrt{M}\Big)\Big)$.

To conclude, the time complexity of analyzing Segment-D is (5.2).

$$O\Big(W_E\left(N + L\sqrt{M}\exp_2\Big((1/2)\sqrt{M}\Big)\right)\Big) \tag{5.2}$$

Correlation Checks

If Segment-E is not empty, every constraint is checked as in the previous section with four differences. First, the check additionally ensures that every constrained source node is distinct and connected to the constrained sink node with time complexities $O\big(W_i^2\big)$ and $O(W_i)$, respectively, and that every constrained node exists in Segment-B with time complexity $O(NW_i)$. Second, the traversal maintains a stack of visited nodes that branch to two or more paths that lead to the constrained sink node. Third, when the traversal reaches the sink node and obtains one constrained end-to-end path, the path is duplicated, and the duplicate is backtracked to the branching node recorded on the stack's top to start an offshoot as the second traversal. When the second traversal reaches the constrained sink node, the first and the second traversals result in two distinct constrained end-to-end paths. Fourth, upon obtaining the two distinct paths, a temporal correlation is computed at every confluent node formed by the two paths as described in the next paragraph. If the correlation threshold is respected at every confluent node, the second traversal will continue obtaining other second paths until every branch of the branching node recorded on the stack's top has been traversed. At that point, using the next entry in the stack (the stack is read without popping), the backtracked path is further backtracked to the node recorded on the entry, and the whole process to obtain the second path starts all over again until every entry in the stack has been visited. At that point, the first traversal continues by popping the stack and then, referring to the popped entry, by backtracking its path to the recorded branching node and resuming its traversal by visiting the recorded consumer node. Before recommencing the traversal, however, the first traversal checks whether the consumer node to be visited is the last according to the adjacency array indexed by the popped entry's branching node. If it is, the traversal recommences; otherwise, before recommencing, a new entry is pushed to the stack such that the new and the popped entries differ only in their records of the next branch to take. Lastly, when the first traversal reaches the constrained sink node, the whole process of obtaining the second path by the stack starts all over again until every possible pair of constrained end-to-end paths have been checked. The time complexity of identifying two distinct constrained end-to-end paths is therefore $O\Big(\sqrt{M}\ \exp_2\Big(\sqrt{M}\Big)\Big)$ because the number of visited arcs on every constrained end-to-end path is $O\Big(\sqrt{M}\Big)$ as analyzed in §5.2 and the traversals obtain two constrained end-to-end paths in the same way two nodes are obtained by the pairwise-node check with time

complexity $O\left(n^2\right)$ described in §5.2 where n in this case is the number of distinct constrained end-to-end paths, which is $O\left(\exp_2\left((1/2)\sqrt{M}\right)\right)$ as analyzed in §5.2.

Given two constrained end-to-end paths, every confluent node formed by the two paths are identified by first constructing a position array of size N that can decide with time complexity $O(1)$ whether a node is on the first path based on the node's index. All elements of the position array are initially set to zero with time complexity $O(N)$ to indicate that every node is not on the first path. Every node on the first path is then visited starting from the source node with time complexity $O\left(\sqrt{M}\right)$, each of which indexes the position array to set the indexed element to the node's position on the first path. Once the position array is constructed, every confluent node can be identified by visiting every node on the second path starting from the source node and using the visited node to index the position array. If the visited node indexes non-zero, the identification process sets a boolean identification state to true, otherwise to false. Every visited node other than the source node that necessitates the identification state to change from false to true is then a confluent node. As a result, the identification of all confluent nodes has time complexity $O\left(\sqrt{M}\right)$. Altogether, the time complexity of identifying all confluent nodes for any two constrained end-to-end paths is $O\left(N + \sqrt{M}\right)$. This identification process, however, is repeated for every pair of distinct constrained end-to-end paths. Since the number of distinct constrained end-to-end paths is $O\left(\exp_2\left((1/2)\sqrt{M}\right)\right)$, the number of pairs of such paths is $O\left(\exp_2\left(\sqrt{M}\right)\right)$. Therefore, the time complexity of identifying the confluent nodes of all pairs of constrained end-to-end paths is $O\left(\left(N + \sqrt{M}\right)\exp_2\left(\sqrt{M}\right)\right)$.

Given a confluent node that is formed by two constrained end-to-end paths, the correlation at the confluent node is computed by an algorithm whose time complexity is $O\left(L\sqrt{M}\right)$ because the algorithm has to check at most L repetitions of the confluent node's computation and, in every repetition, the algorithm has to check every other node on the constrained paths that produce the data received by the confluent node, the number of which is $O\left(\sqrt{M}\right)$. This means that the time complexity of computing the correlations at all confluent nodes is $O\left(LM\exp_2\left(\sqrt{M}\right)\right)$ because the $O\left(L\sqrt{M}\right)$ algorithm is executed once for every confluent node, the total number of which is $O\left(\sqrt{M}\right)$, that exists in every pair of constrained end-to-end paths, the total number of which is $O\left(\exp_2\left(\sqrt{M}\right)\right)$.

To conclude, the time complexity of analyzing Segment-E is (5.3).

$$O\left(\sum_{i=1}^{W_C}\left(W_i^2 + NW_i\right) + W_C\left(LM + N + \sqrt{M}\right)\exp_2\left(\sqrt{M}\right)\right) \tag{5.3}$$

Tice Back-End

Once a Tice model is deemed valid by Tice front-end, Tice back-end will map the expressed nodes to a set of real-time tasks with time complexity $O(N)$ due to performing the real-time schedulability test of gEDF (global earliest-deadline first) [8]. A compile-time error will be raised if the mapping fails; otherwise, Tice back-end generates one C++ thread for one real-time task to be scheduled using the `SCHED_DEADLINE` policy available in the Linux kernel [63]. Every C++ thread simply invokes the C++ function that is assigned to the corresponding Tice node once in every period of the node. Beside generating the C++ thread, Tice back-end also generates the code that implements the communication of every producer-consumer pair expressed in the Tice model with time complexity $O\left(M^2\right)$ by visiting every arc expressed in the model and indexing the visited arc in a tuple using a linear search to obtain the channel details. Tice back-end, therefore, has time complexity $O\left(N + M^2\right)$.

5.3 Empirical Validations

The preceding section shows that as a real-time language (i.e., segments D and E are absent), Tice time complexity is determined by that of Tice front-end in (5.1) and Tice back-end in §5.2, which altogether result in:

$$O\left(N^2 + N(M+K) + K^2 + MK + \sum_{i=1}^{K} K_i^2 + M^2\right)$$

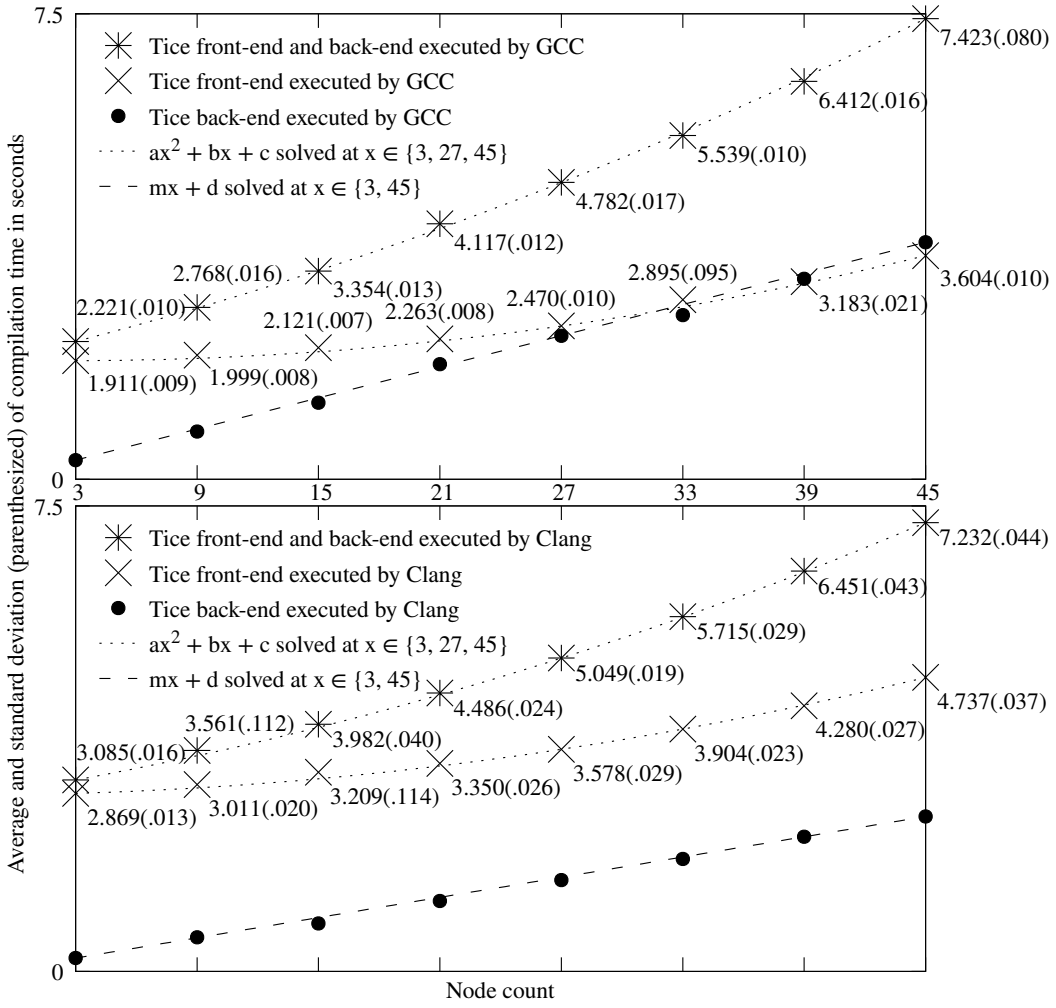


Figure 5.1: Compilation times of segments A and B.

This is also the case when a real-time constraint is present (i.e., segment D or E is present), except that the time complexity of Tice front-end is now exponential as shown in (5.2) and (5.3). The analysis results of §5.2 will now be validated while showing the practical limits of Tice library.

In validating the analyses, an ordinary laptop was used to approximate the computers of typical programmers. Specifically, the laptop was Lenovo E40-80 whose processor is Intel Core i3-5010U, which has four logical cores on two physical 64-bit 2.1-GHz cores, and whose DDR3 memory modules had capacity 2 GiB and 8 GiB, respectively, for a total of 10 GiB. The validations were designed so that the compilers did not cause any memory swapping to disk to make every measurement comparable. The laptop used Ubuntu 16.04.6 in its desktop version and downloaded the latest GCC (v9.1.0) and Clang (v9.0.0) using its package manager from `http://ppa.launchpad.net/ubuntu-toolchain-r/test/ubuntu` and `http://apt.llvm.org/xenial`, respectively. Their compilation times were measured by the built-in command `time` of Bash shell, taking the sums of the user and system times as the compilation times. The plotted average compilation times and their standard deviations were obtained by repeating every compilation five times. All compiled Tice programs were designed not to raise any compile-time error so that only complete compilation times were measured, and the period of every node, which affects L, is the same unless specified otherwise. Lastly, every Tice program was compiled with four core IDs and without core ID in their Segment-A so that the former case measured the compilation times of both Tice front-end and Tice back-end but the latter case measured only the compilation times of Tice front-end. That is, the compilation times of Tice back-end are inferred by taking the absolute differences between the compilation times in the former and latter cases, respectively.

Figure 5.1 shows that the compilation times of Tice programs with only segments A and B confirm the analysis results. In terms of N, the time complexity of Tice front-end is quadratic as $K = K_i = M = 0$ in (5.1), while that of Tice back-end is linear as $M = 0$ in §5.2.

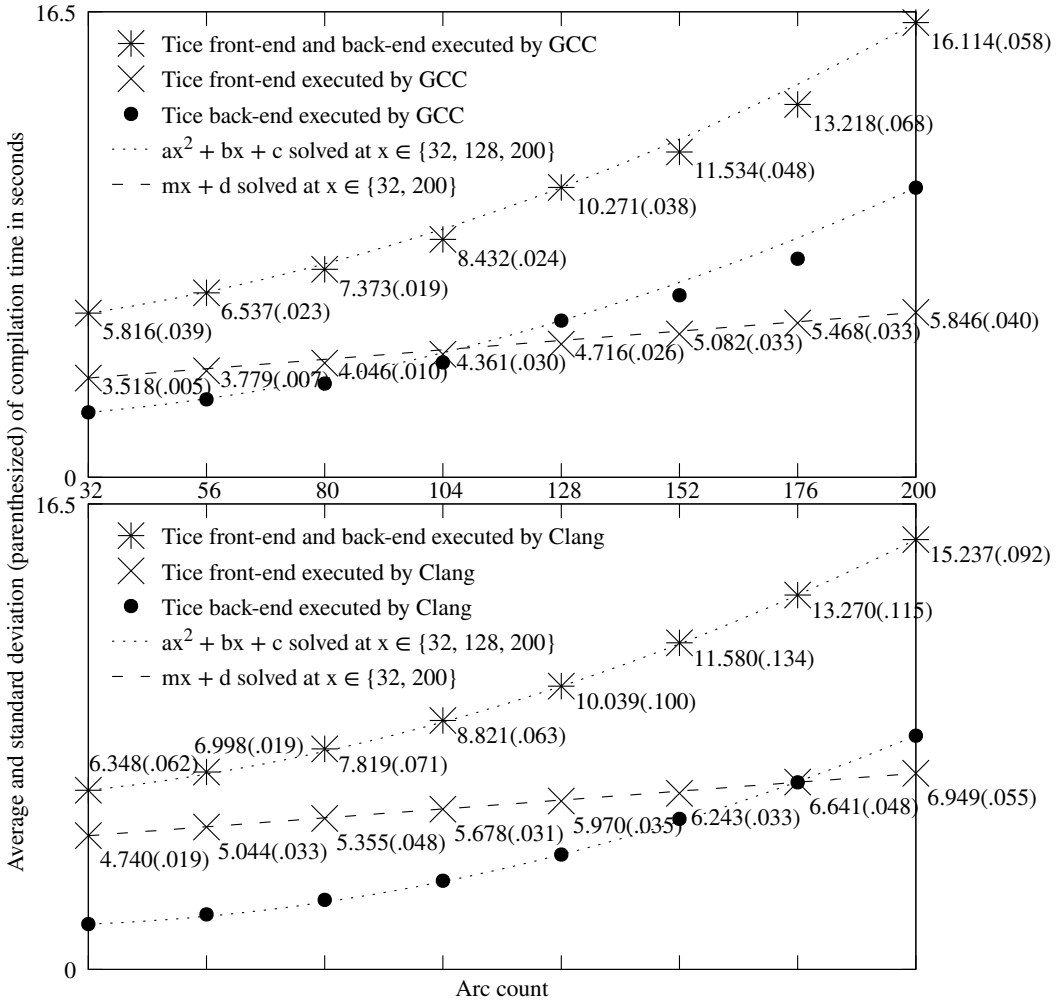


Figure 5.2: Compilation times of segments A up to C.

Figure 5.2 shows the compilation times of Tice programs with segments A up to C where $N = 21$ and $K = N - 1$ are fixed and distributed the arcs among the `Feeder` instances as equally as possible so that the last term of (5.1) has $N(N+1)/2$ as an upper bound. In this setup, in terms of M, the compilation time of Tice front-end grows linearly as predicted by (5.1), while that of Tice back-end grows quadratically as predicted by §5.2, both of which are confirmed in Figure 5.2.

Figure 5.3 shows the compilation times of Tice programs with segments A up to C as in the preceding paragraph but with $N = 15$ and additionally with Segment-D of length one that constrained the only existing source and sink nodes (no other existed since $K = N - 1$). In this setup, (5.2) predicts that the compilation time of Tice front-end will grow exponentially in terms of M. Looking at the $\log_2$-scaled graph of Figure 5.3, the prediction is indeed correct and is tight for arc counts less than or equal to 52 but is increasingly looser for the greater arc counts. This phenomenon is observed because both C++ compilers memoize/cache the results of instantiating template metaprograms so that, when they are executed with the same parameters again, they are not re-computed. Because N is fixed, as M grows, the cache-hit rate also increases, resulting in the observed phenomenon without invalidating (5.2). On the other hand, the prediction of §5.2 that the compilation time of Tice back-end will grow quadratically in terms of M is confirmed by Figure 5.3 as the solid dots is increasing, albeit slightly due to $M < 100$ as per Figure 5.2. Furthermore, measurements had also been made on the compilation times of Tice programs in the same setup but using two different periods: 1 and 4 so that $L = 4$, obtaining a plot that is virtually equal to Figure 5.3 but with vertically-scaled front-end's curve due to the greater L as predicted by (5.2). Hence, (5.2) is empirically valid.

Figure 5.4 shows the compilation times of Tice programs as in the preceding paragraph but with $N = 9$ and one constraint in segment E instead of D. In this setup, as before, §5.2 correctly predicts that the compilation time of Tice back-end will grow quadratically in terms of M, albeit slightly due to small M. Similarly, (5.3) also

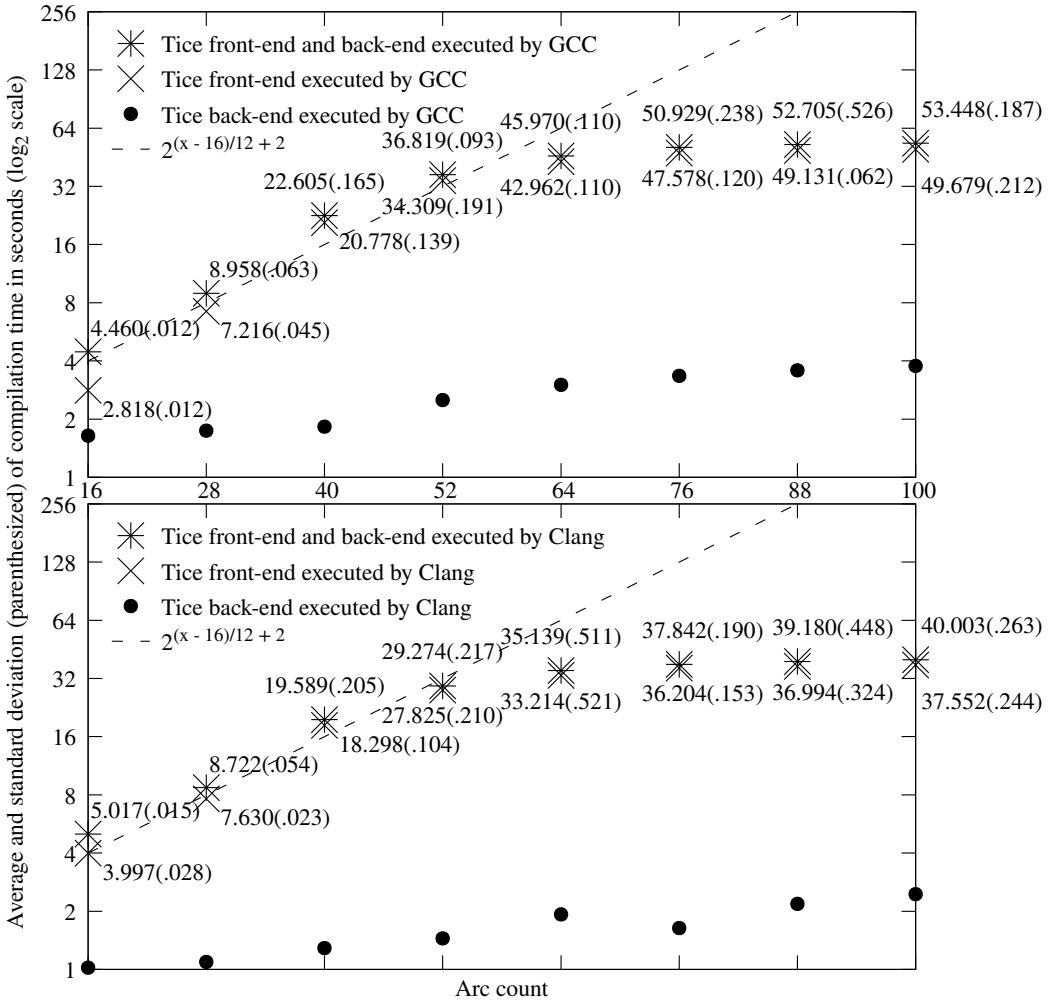


Figure 5.3: Compilation times of segments A up to C and D.

correctly predicts that the compilation time of Tice front-end will grow exponentially in terms of M, observing the same phenomenon of an increasingly higher cache-hit rate as M grows while N remains constant. Moreover, measurements had also been made on the compilation times of Tice programs in the same setup but using two different periods: 1 and 4 so that $L = 4$, obtaining a plot that is virtually equal to Figure 5.4 but scaled vertically due to the greater multiplication factor L as predicted by (5.3). Hence, (5.3) is empirically valid.

In summary, the analysis results obtained in §5.2 are empirically valid as (5.1), (5.2), and (5.3) predict the compilation times of GCC and Clang correctly. Additionally, the compilation times displayed in this section's figures show that GCC is faster than Clang at compiling a small number of template instantiations (cf. GCC in Figure 5.1 and Figure 5.2 and Clang in Figure 5.3 and Figure 5.4). Hence, while GCC would be faster than Clang at compiling C++ programs in general, Clang would be better suited than GCC to compile model-based C++ EDSLs as exemplified by Tice. More importantly, it can be seen that the compilation times of Tice library on GCC and Clang is practically feasible as the current practice of embedded software engineering usually uses a small number of nodes and arcs, fewer than what are used in the evaluations [15, 83].

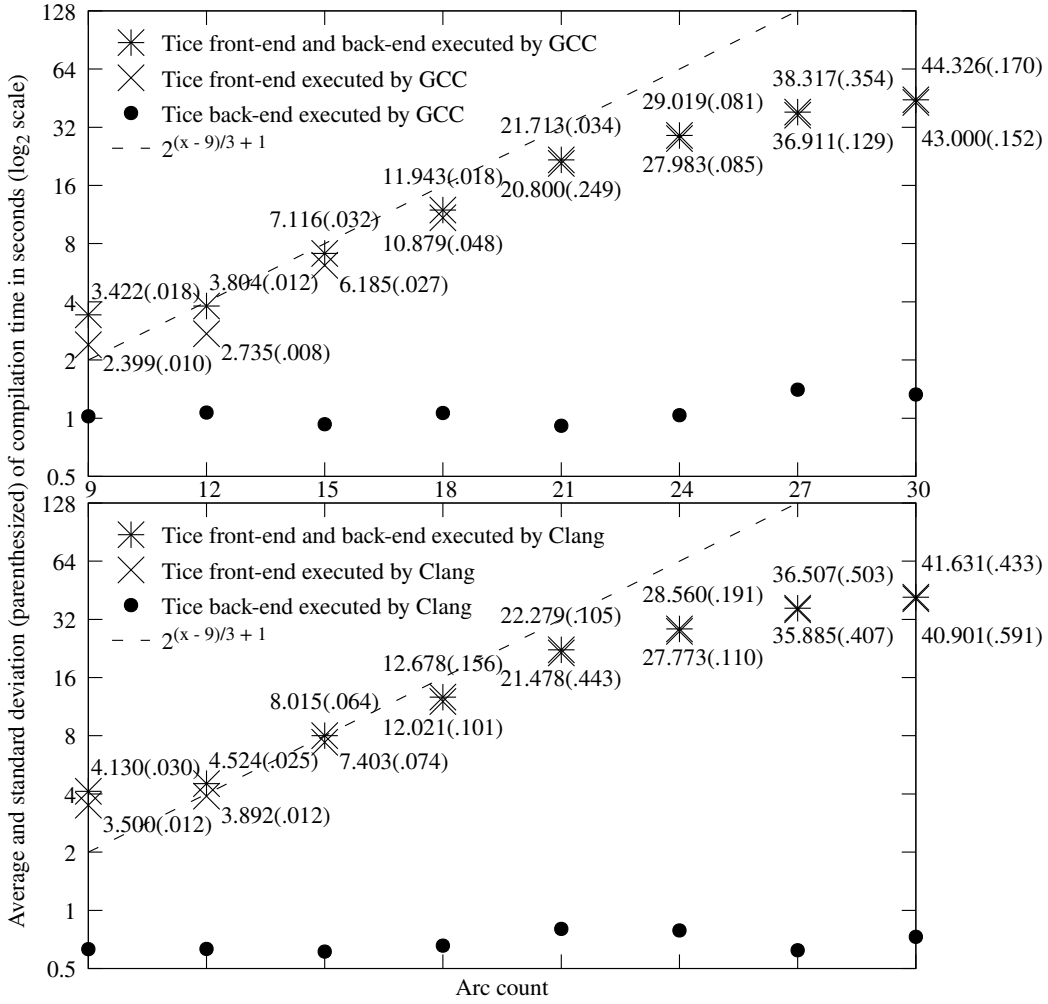


Figure 5.4: Compilation times of segments A up to C and E.

6 *Tice Engineering Techniques*

First of all, while the term Tice library indeed means some implementation of the Tice language, in the rest of this chapter, the term Tice library is used to refer to the particular implementation at [87]. Initially, Tice library was difficult to study, maintain, and evolve because its template metaprograms were difficult to read due to the essential information being obscured by the verbosity required by TMP (template metaprogramming) as exemplified by the template metaprogram shown on the left part of Figure 6.1, which implements (2.1) in a way that is different from the one shown in Figure 2.6 to illustrate the engineering techniques described in this section.

Consequently, an engineering technique has been developed to make Tice library's template metaprograms easier to understand as exemplified on the right part of Figure 6.1. For example, it becomes clear on the right part that lines 25–35 of the left part is conditioned only on the last template parameter. Additionally, it also becomes clear that the template metaprogram `fact__domain_check` only uses `n` for error reporting as it appears nowhere else in lines 25–35 on right part. Hence, analysis effort has been greatly simplified. Consequently, this simplifying technique will be elaborated further in the next section along with two techniques that has developed to afford Tice library smaller constants that are hidden beneath the big-O time-complexity analysis results (see §6.2) and better evolvability (see §6.3).

6.1 Template Patterns

The left part of Figure 6.1 shows in lines 10–48 that TMP involves a lot of repetitions. For example, the identifier `n` is repeated 14 times. Although two that follow `unsigned` in lines 10 and 18 can be omitted, the `unsigned` themselves cannot be omitted. The repetitions make comprehension difficult due to the following reasons:

- It is no longer clear which parameters are used by a particular class template and which parameters are only passing through the class template to another class template. For example, the `n` in lines 21–23 is just passing through `fact` to `fact__domain_check`. Additionally, for a particular class template, it is also not clear which parameters are important for which template specializations. For example, both `n` and `checker_msg` in lines 27 and 32 have no role in the template specializations because the specializations only care about the value of the third template parameter.

- Adding or removing or renaming a parameter is a herculean task because the parameter must be added to or deleted from or renamed in many places. While this problem can be solved using an IDE (integrated development environment) plugin [21], the solution is not general due to being tied to one specific IDE whose infrastructure has already performed the hard work of parsing C++ source programs in the first place.

- Owing to the previous two points, existing C++ template metaprograms, such as those in the Boost C++ libraries [22], tend to use short, if not cryptic, parameter names. This makes renaming harder and, therefore, discourages fixing misleading names, making comprehension harder.

Tice library solves the repetition problem by developing a set of C++ preprocessor macros along with its set of usage rules [87] to abstract the parameters of template metaprograms as systematic patterns, which are called *template patterns*. For example, the result of abstracting the template metaprograms on the left part of Figure 6.1 using template patterns is shown on the right part side-by-side and shows the following improvements:

- It is now clear that the `n` in lines 21–23 on the left part is just passing through to another class template as it is now hidden on the right part. Additionally, it is now clear in lines 27 and 32 on the left part that both `n` and `checker_msg` play no important roles as they are hidden on the right part, which now highlights the third parameter's value.

- While renaming the parameter `n` in lines 10–48 requires some error-prone work because a simple search-and-replace for "`n,`" will miss the "`n`" in line 37 on the left part, once abstracted with template patterns, renaming is done only in one place in the definitions of the template patterns as well as in places where `n` is indeed used, which are now easily identifiable on the right part of Figure 6.1 in lines 14, 28, and 42, especially if the name is more descriptive than just a single character. Therefore, renames are not as error-prone and hard as before, encouraging fixing misleading parameter names and updating them as template metaprograms evolve.

```
1  namespace error {
2   namespace fact {
3    template<bool is_in_domain, unsigned arg_1>
4    struct arg_1_is_in_domain {
5     static_assert(is_in_domain, "n is out of range");
6    };
7   }
8  }
9
10 template<unsigned n, template<bool, unsigned> class
11          checker_msg = error::fact::arg_1_is_in_domain>
12 struct fact;
13
14 template<unsigned n, template<bool, unsigned> class
15          checker_msg, bool is_in_domain = (n <= 12)>
16 struct fact__domain_check;
17
18 template<unsigned n>
19 struct fact__computation;
20
21 template<unsigned n, template<bool, unsigned> class
22          checker_msg>
23 struct fact : fact__domain_check<n, checker_msg> {};
24
25 template<unsigned n, template<bool, unsigned> class
26          checker_msg>
27 struct fact__domain_check<n, checker_msg, false>
28  : checker_msg<false, n> {};
29
30 template<unsigned n, template<bool, unsigned> class
31          checker_msg>
32 struct fact__domain_check<n, checker_msg, true> {
33  static constexpr fact__computation<n> result = {};
34  static constexpr unsigned value = result.value;
35 };
36
37 template<unsigned n>
38 struct fact__computation {
39  unsigned value;
40  constexpr unsigned compute() {
41   unsigned result = 1;
42   for (unsigned i = 1; i <= n; ++i) result *= i;
43   return result;
44  }
45  constexpr fact__computation() : value() {
46   value = compute();
47  }
48 };
```

```
namespace error {
 namespace fact {
  template<bool is_in_domain, unsigned arg_1>
  struct arg_1_is_in_domain {
   static_assert(is_in_domain, "n is out of range");
  };
 }
}

template<decl_2(_d1(error::fact::arg_1_is_in_domain),
                I, _, _)>
struct fact;

template<decl_3(_d1((n <= 12)),
                I, _, _, _)>
struct fact__domain_check;

template<prms_1(I, _)>
struct fact__computation;


template<prms_2(I, _, _)>
struct fact : fact__domain_check<args_3(I, _, _, X)> {};


template<prms_3(I, _, _, X)>
struct fact__domain_check<args_3(I, _, _, R(false))>
 : checker_msg<false, n> {};


template<prms_3(I, _, _, X)>
struct fact__domain_check<args_3(I, _, _, R(true))> {
 static constexpr fact__computation<args_1(I, _)> result = {};
 static constexpr unsigned value = result.value;
};

template<prms_1(I, _)>
struct fact__computation {
 unsigned value;
 constexpr unsigned compute() {
  unsigned result = 1;
  for (unsigned i = 1; i <= n; ++i) result *= i;
  return result;
 }
 constexpr fact__computation() : value() {
  value = compute();
 }
};
```

Figure 6.1: Verbose metaprograms (left) and distilled metaprograms (right) compared side-by-side.

- While formerly the repetitions discouraged descriptive parameter names, template patterns have encouraged them because the names are written only when they are needed.

While it seems that the benefits of template patterns can be afforded by using an IDE, to properly add, remove, and rename template parameters the IDE has to first understand C++ and capable of resolving its ambiguities (for a notable example, see `https://stackoverflow.com/a/14589567`). Such requirements, however, are not present when using template patterns because it is designed to be manipulable using the standard Unix text-manipulation tools that rely on regular expressions, such as *grep* and *sed*, allowing the use of various IDEs and even text editors (e.g., GNU® Emacs) to work on C++ template metaprograms. Furthermore, the IDE has to be sophisticated enough so as to know which template parameters should be hidden because, for example, the parameters are just passing through. This level of sophistication, however, is virtually unattainable because only the programmer knows which template parameters are actually important not to be hidden as underscores. In contrast, the programmer can express this important information explicitly using template patterns.

6.2 Faster Compilation and Array Usage

While template metaprograms are Turing-complete [20], their computations are slower than the computations of constant-expression functions (e.g., lines 40–44 and lines 45–47 of Figure 6.1). However, within any constant-expression function, templates cannot be instantiated using any parameter from the function's parameter list and any local variable declared within the function. As a result, it is not possible to raise a compile-time error by instantiating an externalized error message described in §6.3 if the error message needs to be supplied with information from any of the function's parameters or local variables. Additionally, it is also not possible to start

the computation of a template metaprogram from within the function if the template has to be instantiated using any of the function's parameters and local variables. However, it is possible to continue the computations of constant-expression functions in template metaprograms by creating constant-expression objects.

Referring to lines 37–48 in Figure 6.1, a constant-expression object has a constant-expression constructor (lines 45–47). When a constant-expression object is instantiated by a template metaprogram (line 33), its constructor can use the information supplied by the template metaprogram to perform computations whose results are then stored in the object's data members (line 39). Once the constant-expression object has been instantiated, the object's constant data members can be used to instantiate further template metaprograms or stored as static constant data members (line 34).

The use of constant-expression objects has been the key that enabled Tice library to use arrays in TMP to solve various decision and search problems with time complexity $O(1)$ as already detailed in §5.2. Constant-expression objects have been the key to use arrays because arrays cannot be passed around in TMP as objects but only as pointers. The pointers, however, point to static constant arrays that can be initialized either manually by specifying every single element or automatically by either a template metaprogram or the constant-expression constructor of a constant-expression object, which executes faster.

6.3 Precise Error Messages

Tice library externalizes error messages (e.g., lines 1–8 of Figure 6.1) and customizes the error messages of template metaprograms (e.g., line 10 on the right part of Figure 6.1) to help locate errors quickly with precise error messages without sacrificing the modularity of the library. For example, every member of Tice library API is a stand-alone unit with its own set of error messages and test cases (i.e., being modular). If the API member had no customizable error message, however, using the API member on the parameter list of another API member (e.g., `Program`) would render its error messages imprecise at best and misleading at worst. For example, `Node` as a stand-alone unit will raise the error message "Period is less than computation WCET" if the node's assigned period is less than the WCET assigned to the node's computation. While the error message quickly leads to the faulty `Node` when it is used independently (e.g., in a unit test), the error message becomes imprecise when it is on the parameter list of `Program` with, for example, another ten nodes because the error message does not help much to quickly locate the faulty node on the list. Furthermore, beside making it faster to locate a fault, customizable error messages also allow Tice as a language to be extensible, for example, by being composed with other C++ EDSLs and orchestrated as a whole new EDSL, which would need to customize Tice error messages to suit the new domain so as not to be misleading. On the other hand, externalized error messages have the benefits of being listed and easily browsable in an API documentation, making it easier to understand the semantics of the associated API members.

The error messages are precise because they are externalized as class templates with the following characteristics:

- The class template is named with a proposition that has to be satisfied for the error not to be raised. Furthermore, the proposition states the position of the parameter that is asserted. Additionally, the proposition is atomic by asserting only one criterion at a time instead of multiple criteria in a single proposition using "and"/"or" logical connectives.

- The class template has a parameter list with at least one parameter. The first parameter on the list is of type `bool` and is used to raise the error message itself. The remaining parameters are used to justify the raise of the error message, such as reporting the invalid parameter itself as well as providing additional information to ease debugging.

- The class template has exactly one static assertion that is conditioned on the template's first parameter with an error message that states the error in terms of domain-specific concepts. The domain-specific concepts could help to figure out the fault more quickly, especially if the error message gives a list of possible faults to be checked.

Lastly, every template metaprogram that uses as subprograms a number of other metaprograms with customizable error messages should allow the error messages of the subprograms to keep being customizable, except for those that can be proven to never raise errors due to the way the subprograms are used.

Please skip this page.

7 *Conclusions*

This work has shown that C++ indeed has the potential to be a platform where different software languages, including a model-based real-time language, are implementable as C++ active libraries and are composable and integrable as and with ordinary C++ libraries seamlessly and automatically as demonstrated in §4.1, particularly with Figure 4.3. This work also shows the potential of using a C++ compiler as an interactive modeling/simulation tool in §4.2. In implementing a software language on the platform, however, some engineering techniques may be needed to obtain the desired compile-time efficiency and long-term maintainability as demonstrated in Chapter 6.

With respect to overcoming the problem of adoption faced by many real-time languages already proposed in the literature, this work proposes a model-based real-time language that is readily adoptable by practitioners owing to the widespread use of C++ in the software industry, particularly in the domain of embedded systems. In proposing the model-based real-time language, this work has formally shown that the language definition is sound in §4.4 and that the language is decidable in §5.1. Furthermore, this work has also implemented the language and assessed the implementation's time complexity in §5.2. Lastly, this work has validated the analysis results empirically while at the same time showing the capability of off-the-shelf standard C++ compilers in processing a C++ active library with an exponential time-complexity in §5.3.

Lastly, some work remains to be done:

- With respect to real-time constraints, [30, 33] highlight the fact that there are several semantics of an end-to-end delay constraint, all of which could be adopted to further enrich Tice. With respect to real-time languages, on the other hand, it would be interesting to implement some of them as C++ active libraries to investigate how different different real-time language abstractions could help engineer different parts of a software product. Furthermore, owing to the seamless composability of C++ active libraries, interaction with different languages could start be investigated as well as user studies on the use of the languages because C++ active libraries are easily adoptable due to the widespread use of C++, especially in the domain of embedded systems.

- Work is needed to show that a C++ active library implementing a software language can be extended easily to transform models expressed in the implemented language into different programs based on the computers that the programs will run. The demonstration is important because model/language transformation is the cornerstone of using the most appropriate kinds of models to engineer different parts and aspects of a software product as explicitly pointed out in [106] for MDE, in [120] for language-oriented programming, and in [102] for intentional programming.

- Work is currently in progress to demonstrate a technique that enables a C++ active library to communicate with external tools, such as a WCET analyzer [5], despite being run by off-the-shelf standard C++ compilers.

Please skip this page.

Bibliography

[1] European Space Agency. 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 January 31, 2020.

[2] W. Aitken, B. Dickens, P. Kwiatkowski, O. De Moor, D. Richter, and C. Simonyi. Transformation in Intentional Programming. In *Proceedings of the 5th International Conference on Software Reuse*, pages 114–123. IEEE, June 1998. https://doi.org/10.1109/ICSR.1998.685736.

[3] ANSYS. Ansys scade suite. Online at https://www.ansys.com/products/embedded-software/ansys-scade-suite, accessed on January 31, 2020.

[4] Anya Helene Bagge and Vadim Zaytsev. Open and Original Problems in Software Language Engineering 2015 Workshop Report. *SIGSOFT Software Engineering Notes*, 40(3):32–37, June 2015. https://doi.org/10.1145/2757308.2757313.

[5] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010. https://doi.org/10.1007/978-3-642-16256-5_6.

[6] Michael Barr and Anthony Massa. *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly, 2nd edition, 2006. https://isbnsearch.org/isbn/978-0-596-00983-0.

[7] Niall Barr and Jeremy Singer. Solutions to Three Language Workbench Challenges using Wizards Workbench. Online at https://2016.splashcon.org/details/lwc2016/8/Solutions-to-Three-Language-Workbench-Challenges-using-Wizards-Workbench, accessed on January 31, 2020.

[8] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer, 1st edition, 2015. https://doi.org/10.1007/978-3-319-08696-5.

[9] Jacob Beningo. *Reusable Firmware Development: A Practical Approach to APIs, HALs, and Drivers*. Apress, 1st edition, 2017. https://doi.org/10.1007/978-1-4842-3297-2.

[10] Gerard Berry, Sabine Moisan, and Jean-Paul Rigault. Esterel: Towards a Synchronous and Semantically Sound High-Level Language for Real-Time Applications. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '83, pages 30–37. IEEE, 1983. https://site.ieee.org/tcrts/education/seminal-papers/.

[11] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The Concept Assignment Problem in Program Understanding. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 482–498. IEEE, May 1993. https://doi.org/10.1109/ICSE.1993.346017.

[12] A. Biondi and M. Di Natale. Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '18, pages 240–250, April 2018. https://doi.org/10.1109/RTAS.2018.00032.

[13] Z. Borok-Nagy, V. Majer, J. Mihalicza, N. Pataki, and Z. Porkolab. Visualization of C++ Template Metaprograms. In *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 167–176, September 2010. https://doi.org/10.1109/SCAM.2010.16.

[14] Antonio Bucchiarone, Jordi Cabot, Richard F. Paige, and Alfonso Pierantonio. Grand Challenges in Model-Driven Engineering: An Analysis of the State of the Research. *Software & Systems Modeling*, 19(1):5–13, 2020. https://doi.org/10.1007/s10270-019-00773-6.

[15] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Pearson, 4th edition, 2009. https://isbnsearch.org/isbn/978-0-321-41745-9.

[16] Giorgio Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 3rd edition, 2011. https://doi.org/10.1007/978-1-4614-0676-1.

[17] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188. ACM, 1987. https://doi.org/10.1145/41625.41641.

[18] Stephen Cass. Interactive: The Top Programming Languages 2019. Online at http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019, accessed on January 31, 2020.

[19] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative Programming and Active Libraries. In Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser, editors, *Generic Programming: International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27–May 1, 1998, Selected Papers*, pages 25–39. Springer, 2000. https://doi.org/10.1007/3-540-39953-4_3.

[20] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 1st edition, 2000. https://isbnsearch.org/isbn/0-201-30977-7.

[21] Samuel Davis and Gregor Kiczales. Registration-based Language Abstractions. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '10, pages 754–773. ACM, 2010. https://doi.org/10.1145/1869459.1869521.

[22] B. Dawes. Boost background information. Online at https://www.boost.org/users/index.html, accessed on January 31, 2020.

[23] Morgan Deters, Christopher Gill, and Ron Cytron. Rate-Monotonic Analysis in the C++ Type System. In *Proceedings of the RTAS 2003 Workshop on Model-Driven Embedded Systems (MDES)*. IEEE, May 2003. http://www.cse.wustl.edu/~cdgill/RTAS03/mdes_program.html.

[24] Lewin Edwards. *Embedded System Design on a Shoestring: Achieving High Performance with a Limited Budget*. Newnes, 1st edition, 2003. https://isbnsearch.org/isbn/978-0-750-67609-0.

[25] Bill Emshoff. Using C++ on Mission and Safety Critical Platforms. Online at https://www.youtube.com/watch?v=sRe77Mdna0Y, accessed on January 31, 2020.

[26] Rolf Ernst, Stefan Kuntz, Sophie Quinton, and Martin Simons. The Logical Execution Time Paradigm: New Perspectives for Multicore Systems (Dagstuhl Seminar 18092). *Dagstuhl Reports*, 8(2):122–149, 2018. https://doi.org/10.4230/DagRep.8.2.122.

[27] Joel Falcou and Vincent Reverdy. Edsl infinity wars: Mainstreaming symbolic computation. Online at https://www.youtube.com/watch?v=XH00wB_bbU4, accessed on January 31, 2020.

[28] Jean-Marie Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels — Episode II: Story of Thotus the Baboon. In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. http://drops.dagstuhl.de/opus/volltexte/2005/21.

[29] Jean-Marie Favre. Foundations of Model (Driven) (Reverse) Engineering : Models — Episode I: Stories of The Fidus Papyrus and of The Solarus. In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. http://drops.dagstuhl.de/opus/volltexte/2005/13.

[30] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A Compositional Framework for End-to-End Path Delay Calculation of Automative Systems under Different Path Semantics. In *Proceedings of the Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, CRTS '08. IEEE, 2008. https://research.chalmers.se/en/publication/139048.

[31] Daniel Feltey, Spencer P. Florence, Tim Knutson, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, Robby Findler, and Matthias Felleisen. Languages the Racket Way: Submission to the 2016 Language Workbench Challenge. Online at https://2016.splashcon.org/details/lwc2016/7/Languages-the-Racket-Way-Submission-to-the-2016-Language-Workbench-Challenge, accessed on January 31, 2020.

[32] Tully Foote. Celebrating 9 Years of ROS. Online at https://spectrum.ieee.org/automaton/robotics/robotics-software/celebrating-9-years-of-ros, accessed on January 31, 2020.

[33] J. Forget, F. Boniol, and C. Pagetti. Verifying End-to-End Real-Time Constraints on Multi-periodic Models. In *Proceedings of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA '17, pages 1–8. IEEE, September 2017. https://doi.org/10.1109/ETFA.2017.8247612.

[34] Julien Forget. *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. Online at https://hal.archives-ouvertes.fr/tel-01942421/file/jforget-thesis.pdf, accessed on January 31, 2020.

[35] Free Software Foundation. C++ Standards Support in GCC. Online at https://gcc.gnu.org/projects/cxx-status.html, accessed on January 31, 2020.

[36] M. Fowler. A Pedagogical Framework for Domain-Specific Languages. *IEEE Software*, 26(4):13–14, July 2009. https://doi.org/10.1109/MS.2009.85.

[37] Steven D. Fraser, Lars Bak, Rob DeLine, Nick Feamster, Lindsey Kuper, Cristina V. Lopes, and Peng Wu. The Future of Programming Languages and Programmers. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pages 63–66. ACM, 2015. https://doi.org/10.1145/2814189.2818719.

[38] Thierry Gautier, Paul Le Guernic, and Loïc Besnard. SIGNAL: A Declarative Language for Synchronous Programming of Real-Time Systems. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 257–277. Springer, 1987. https://doi.org/10.1007/3-540-18317-5_15.

[39] Narain Gehani and Krithi Ramamritham. Real-time Concurrent C: A Language for Programming Dynamic Real-Time Systems. *Real-Time Systems*, 3(4):377–405, December 1991. https://doi.org/10.1007/BF00365999.

[40] R. Gerber, S. Hong, and M. Saksena. Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '94, pages 192–203. IEEE, December 1994. https://doi.org/10.1109/REAL.1994.342716.

[41] Joseph (Yossi) Gil and Keren Lenz. Simple and safe SQL queries with C++ templates. *Science of Computer Programming*, 75(7):573–595, 2010. https://doi.org/10.1016/j.scico.2010.01.004.

[42] Paul Graham. *Hackers and Painters: Big Ideas from the Computer Age*. O'Reilly Media, 1st edition, 2004. https://isbnsearch.org/isbn/978-0-596-00662-4.

[43] Philippe Groarke. Re: [EXTERNAL] Re: Linear algebra library proposal. Online at https://lists.isocpp.org/sg14/2019/06/0149.php, accessed on January 31, 2020.

[44] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software: First International Workshop, EMSOFT 2001 Proceedings*, pages 166–184. Springer, 2001. https://doi.org/10.1007/3-540-45449-7_12.

[45] Patrick C. Hickey, Lee Pike, Trevor Elliott, James Bielman, and John Launchbury. Building Embedded Systems with Embedded DSLs. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 3–9. ACM, 2014. https://doi.org/10.1145/2628136.2628146.

[46] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the Naturalness of Software. *Communications of the ACM*, 59(5):122–131, April 2016. https://doi.org/10.1145/2902362.

[47] Seongsoo Hong and Richard Gerber. Scheduling with Compiler Transformations: The TCEL Approach. In *Proceedings of the 10th IEEE Workshop on Real-time Operating Systems and Software*, RTOSS '93, pages 80–84. IEEE, 1993. Online at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.1671, accessed on January 31, 2020.

[48] Pablo Inostroza and Tijs van der Storm. The Rascal Approach to Code in Prose, Computed Properties, and Language Extension. Online at https://2016.splashcon.org/details/lwc2016/9/The-Rascal-Approach-to-Code-in-Prose-Computed-Properties-and-Language-Extension, accessed on January 31, 2020.

[49] Intel Corporation. *Intel® 64 and IA-32 Architectures — Software Developer's Manual — Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A–Z*, May 2018. Order number: 325383-067US (latest revision is online at https://software.intel.com/en-us/articles/intel-sdm as of January 31, 2020).

[50] International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). *ISO/IEC 14977:1996 Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization, 1st edition, 1996. https://www.iso.org/standard/26153.html.

[51] International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. International Organization for Standardization, 4th edition, 2014. https://www.iso.org/standard/64029.html.

[52] International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. International Organization for Standardization, 5th edition, 2017. https://www.iso.org/standard/68564.html.

[53] Yutaka Ishikawa and Hideyuki Tokuda. Object-oriented Real-time Language Design: Constructs for Timing Constraints. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 289–298. ACM, 1990. https://doi.org/10.1145/97945.97980.

[54] JetBrainsTV. MPS Projectional Editor. Online at https://www.youtube.com/watch?v=iN2PflvXUqQ, accessed on January 31, 2020.

[55] Jean-Marc Jézéquel, Benoit Combemale, Steven Derrien, Clément Guy, and Sanjay Rajopadhye. Bridging the Chasm between MDE and the World of Compilation. *Software & Systems Modeling*, 11(4):581–597, 2012. https://doi.org/10.1007/s10270-012-0266-8.

[56] Alan Kay, Dan Amelang, Bert Freudenberg, Ted Kaehler, Stephen Murrell, Yoshiki Ohshima, Ian Piumarta, Kim Rose, Scott Wallace, Alessandro Warth, and Takashi Yamamiya. Steps Toward Expressive Programming Systems: 2011 Progress Report Submitted to the National Science Foundation (NSF). Technical Report TR-2011-004, Viewpoints Research Institute, October 2011. Online at http://www.vpri.org/pdf/tr2011004_steps11.pdf, accessed on January 31, 2020.

[57] Alan Kay, Yoshiki Ohshima, Dan Amelang, Ted Kaehler, Bert Freudenberg, Aran Lunzer, Ian Piumarta, Takashi Yamamiya, Alan Borning, Hesam Samimi, Bret Victor, and Kim Rose. Steps Toward the Reinvention of Programming: 2012 Final Report Submitted to the National Science Foundation (NSF). Technical Report TR-2012-001, Viewpoints Research Institute, October 2012. Online at http://www.vpri.org/pdf/tr2012001_steps.pdf, accessed on January 31, 2020.

[58] Alan Kay, Ian Piumarta, Dan Ingalls, Yoshiki Ohshima, Kim Rose, Dan Amelang, Ted Kaehler, Chuck Thacker, Scott Wallace, Alex Warth, and Takashi Yamamiya. Steps Toward The Reinvention of Programming: A "Moore's Law" Leap in Expressiveness. Technical Report TR-2007-008, Viewpoints Research Institute, December 2007. Online at http://www.vpri.org/pdf/tr2007008_steps.pdf, accessed on January 31, 2020.

[59] Alan Kay, Ian Piumarta, Dan Ingalls, Yoshiki Ohshima, Kim Rose, Dan Amelang, Ted Kaehler, Chuck Thacker, Scott Wallace, Alex Warth, Takashi Yamamiya, and Hesam Samimi. Steps Toward The Reinvention of Programming: 2008 Progress Report Submitted to the National Science Foundation (NSF). Technical Report TR-2008-004, Viewpoints Research Institute, October 2008. Online at http://www.vpri.org/pdf/tr2008004_steps08.pdf, accessed on January 31, 2020.

[60] Alan Kay, Ian Piumarta, Dan Ingalls, Yoshiki Ohshima, Kim Rose, Dan Amelang, Ted Kaehler, Chuck Thacker, Scott Wallace, Alex Warth, Takashi Yamamiya, and Hesam Samimi. Steps Toward The Reinvention of Programming: 2009 Progress Report Submitted to the National Science Foundation (NSF). Technical Report TR-2009-016, Viewpoints Research Institute, October 2009. Online at http://www.vpri.org/pdf/tr2009016_steps09.pdf, accessed on January 31, 2020.

[61] Alan Kay, Ian Piumarta, Dan Ingalls, Yoshiki Ohshima, Kim Rose, Dan Amelang, Ted Kaehler, Chuck Thacker, Scott Wallace, Alex Warth, Takashi Yamamiya, and Hesam Samimi. Steps Toward Expressive Programming Systems: 2010 Progress Report Submitted to the National Science Foundation (NSF). Technical Report TR-2010-004, Viewpoints Research Institute, October 2010. Online at http://www.vpri.org/pdf/tr2010004_steps10.pdf, accessed on January 31, 2020.

[62] Alan Kay, Ian Piumarta, Dan Ingalls, Yoshiki Oshima, and Andreas Raab. Steps Toward The Reinvention of Programming: A Compact And Practical Model of Personal Computing As A Self-Exploratorium. Technical Report RN-2006-002, Viewpoints Research Institute, August 2006. Online at http://www.vpri.org/pdf/rn2006002_nsfprop.pdf, accessed on January 31, 2020.

[63] Michael Kerrisk, Peter Zijlstra, and Juri Lelli. sched—overview of cpu scheduling. Online at http://man7.org/linux/man-pages/man7/sched.7.html, accessed on January 31, 2020.

[64] Christoph M. Kirsch and Ana Sokolova. The Logical Execution Time Paradigm. In Samarjit Chakraborty and Jörg Eberspächer, editors, *Advances in Real-Time Systems*, pages 103–120. Springer, 2012. https://doi.org/10.1007/978-3-642-24349-3_5.

[65] E. Kligerman and A. D. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, September 1986. https://doi.org/10.1109/TSE.1986.6313049.

[66] Donald E. Knuth. *The TEXbook*. Addison-Wesley, 1st edition, 1984. https://isbnsearch.org/isbn/978-0-201-13448-3.

[67] Gabriël Konat, Luis Eduardo de Souza Amorim, Sebastian Erdweg, and Eelco Visser. Bootstrapping, Default Formatting, and Skeleton Editing in the Spoofax Language Workbench. Online at https://2016.splashcon.org/details/lwc2016/4/Bootstrapping-Default-Formatting-and-Skeleton-Editing-in-the-Spoofax-Language-Workb, accessed on January 31, 2020.

[68] Christopher Kormanyos. *Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming*. Springer, 1st edition, 2013. https://doi.org/10.1007/978-3-642-34688-0.

[69] Ralf Lämmel. *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer, 1st edition, 2018. https://doi.org/10.1007/978-3-319-90800-7.

[70] Leslie Lamport. *LATEX: A Document Preparation System*. Addison-Wesley, 2nd edition, 1994. https://isbnsearch.org/isbn/978-0-201-52983-8.

[71] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *1989 Proceedings Real-Time Systems Symposium*, pages 166–171. IEEE, December 1989. https://doi.org/10.1109/REAL.1989.63567.

[72] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973. https://doi.org/10.1145/321738.321743.

[73] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 1st edition, 2000. https://isbnsearch.org/isbn/978-0-130-99651-0.

[74] David H. Lorenz and Boaz Rosenan. Cedalion's Response to the 2016 Language Workbench Challenge. Online at https://2016.splashcon.org/details/lwc2016/1/Cedalion-s-Response-to-the-2016-Language-Workbench-Challenge, accessed on January 31, 2020.

[75] Mark Maimone. C++ on Mars. Online at https://www.youtube.com/watch?v=3SdSKZFoUa8, accessed on January 31, 2020.

[76] Lockheed Martin. Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program, Document Number 2RDU00001 Rev C. Online at http://www.stroustrup.com/JSF-AV-rules.pdf, accessed on January 31, 2020.

[77] Peter Marwedel. *Embedded System Design: Embedded Systems, Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer, 3rd edition, 2018. https://doi.org/10.1007/978-3-319-56045-8.

[78] Toni Mattis, Patrick Rein, and Robert Hirschfeld. Ambiguous, Informal, and Unsound: Metaprogramming for Naturalness. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*, META 2019, pages 1–10. ACM, 2019. https://doi.org/10.1145/3358502.3361270.

[79] Motor Industry Software Reliability Association (MISRA). *MISRA-C++:2008 — Guidelines for the Use of the C++ Language in Critical Systems*. MIRA Limited, 1st edition, 2008. https://isbnsearch.org/isbn/978-1-906-40003-3.

[80] Motor Industry Software Reliability Association (MISRA). *MISRA-C:2012 — Guidelines for the Use of the C Language in Critical Systems*. MIRA Limited, 3rd edition, 2013. https://isbnsearch.org/isbn/978-1-906-40010-1.

[81] S. Natarajan and D. Broman. Timed C: An Extension to the C Programming Language for Real-Time Systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '18, pages 227–239, April 2018. https://doi.org/10.1109/RTAS.2018.00031.

[82] University of Illinois at Urbana-Champaign. C++ Support in Clang. Online at http://clang.llvm.org/cxx_status.html, accessed on January 31, 2020.

[83] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. The ROSACE Case Study: From Simulink Specification to Multi/Many-core Execution. In *Proceedings of the 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium*, RTAS '14, pages 309–318. IEEE, 2014. https://doi.org/10.1109/RTAS.2014.6926012.

[84] Luigi Palopoli, Giorgio Buttazzo, and Paolo Ancilotti. A C Language Extension for Programming Real-Time Applications. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, RTCSA '99, pages 103–110. IEEE, 1999. https://doi.org/10.1109/RTCSA.1999.811199.

[85] Petko Hristov Petkov, Tsonyo Nikolaev Slavov, and Jordan Konstantinov Kralev. *Design of Embedded Robust Control Systems Using MATLAB®/Simulink®*. The Institution of Engineering and Technology, 1st edition, 2018. https://isbnsearch.org/isbn/978-1-785-61330-2.

[86] Zoltán Porkoláb. Functional Programming with C++ Template Metaprograms. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *Central European Functional Programming School: Third Summer School, CEFP 2009, Budapest, Hungary, May 21–23, 2009 and Komárno, Slovakia, May 25–30, 2009, Revised Selected Lectures*, pages 306–353. Springer, 2010. https://doi.org/10.1007/978-3-642-17685-2_9.

[87] Tadeus Prastowo. The Repository of a Tice Implementation. Online at https://savannah.nongnu.org/projects/tice, accessed on January 31, 2020.

[88] Tadeus Prastowo, Luigi Palopoli, and Luca Abeni. C++ Hard-real-time Active Library: Syntax, Semantics, and Compilation of Tice Programs. *SIGBED Review*, 16(3):69–74, November 2019. https://doi.org/10.1145/3373400.3373411.

[89] Tadeus Prastowo, Luigi Palopoli, Luca Abeni, and Giuseppe Lipari. Analyses of a Model-Based Real-Time Language Embedded in C++. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, pages 1330–1339. ACM, 2020. https://doi.org/10.1145/3341105.3373994.

[90] N. Ramsey. Literate Programming Simplified. *IEEE Software*, 11(5):97–105, September 1994. https://doi.org/10.1109/52.311070.

[91] James Renwick, Tom Spink, and Björn Franke. Low-Cost Deterministic C++ Exceptions for Embedded Systems. In *Proceedings of the 28th International Conference on Compiler Construction*, CC 2019, pages 76–86. ACM, 2019. https://doi.org/10.1145/3302516.3307346.

[92] Microsoft Research. Intentional Programming Demo, Part 1: Editor. Online at https://www.youtube.com/watch?v=tSnnfUj1XCQ, accessed on January 31, 2020.

[93] Microsoft Research. Intentional Programming Demo, Part 2: Compiler. Online at https://www.youtube.com/watch?v=ZZDwB4-DPXE, accessed on January 31, 2020.

[94] Arch D. Robison. Impact of Economics on Compiler Optimization. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, JGI '01, pages 1–10. ACM, 2001. https://doi.org/10.1145/376656.376751.

[95] Alberto Sangiovanni-Vincentelli, Haibo Zeng, Marco Di Natale, and Peter Marwedel, editors. *Embedded Systems Development: From Functional Models to Implementations*. Springer, 1st edition, 2014. https://doi.org/10.1007/978-1-4614-3879-3.

[96] Eugen Schindler, Klemens Schindler, Federico Tomassetti, and Ana Maria Sutii. Language Workbench Challenge 2016: the JetBrains Meta Programming System. Online at https://2016.splashcon.org/details/lwc2016/6/Language-Workbench-Challenge-2016-the-JetBrains-Meta-Programming-System, accessed on January 31, 2020.

[97] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, February 2006. https://doi.org/10.1109/MC.2006.58.

[98] Sebastian Erdweg and Tijs van der Storm and Markus Völter and Laurence Tratt and Remi Bosman and William R. Cook and Albert Gerritsen and Angelo Hulshout and Steven Kelly and Alex Loh and Gabriël Konat and Pedro J. Molina and Martin Palatnik and Risto Pohjonen and Eugen Schindler and Klemens Schindler and Riccardo Solmi and Vlad Vergu and Eelco Visser and Kevin van der Vlist and Guido Wachsmuth and Jimi van der Woning. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Computer Languages, Systems & Structures*, 44:24–47, 2015. https://doi.org/10.1016/j.cl.2015.08.007.

[99] Xipeng Shen. Rethinking Compilers in the Rise of Machine Learning and AI (Keynote). In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, page 1. ACM, 2018. https://doi.org/10.1145/3178372.3183634.

[100] Charles Simonyi. The Death of Computer Languages, The Birth of Intentional Programming. Technical Report MSR-TR-95–52, Microsoft Research, September 1995. Online at https://www.microsoft.com/en-us/research/publication/the-death-of-computer-languages-the-birth-of-intentional-programming, accessed on January 31, 2020.

[101] Charles Simonyi. MODELS 2013 Conference Keynote: The Magic of Software. Online at https://www.youtube.com/watch?v=4pT9tDmxjlg, accessed on January 31, 2020.

[102] Charles Simonyi. The Intentional Domain Workbench. Online at https://www.youtube.com/watch?v=UBI33yXJZxg, accessed on January 31, 2020.

[103] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional Software. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 451–464. ACM, 2006. https://doi.org/10.1145/1167473.1167511.

[104] Riccardo Solmi and Enrico Persiani. Whole Platform Solution to a Selection of LWC16 Benchmark Problems. Online at https://2016.splashcon.org/details/lwc2016/5/Whole-Platform-Solution-to-a-Selection-of-LWC16-Benchmark-Problems, accessed on January 31, 2020.

[105] StackOverflow. Developer Survey Results 2019: How Technologies Are Connected. Online at https://insights.stackoverflow.com/survey/2019#technology-_-how-technologies-are-connected, accessed on January 31, 2020.

[106] Friedrich Steimann and Thomas Kühne. Coding for the Code. *Queue*, 3(10):44–51, December 2005. https://doi.org/10.1145/1113322.1113336.

[107] Alexander D. Stoyenko. The Evolution and State-of-the-art of Real-time Languages. *Journal of Systems and Software*, 18(1):61–84, April 1992. https://doi.org/10.1016/0164-1212(92)90046-M.

[108] Alexander D. Stoyenko, Thomas J. Marlowe, and Mohamed F. Younis. A Language for Complex Real-Time Systems. *The Computer Journal*, 38(4):319–338, 1995. https://doi.org/10.1093/comjnl/38.4.319.

[109] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 132–141. ACM, 2011. https://doi.org/10.1145/1993498.1993514.

[110] Juha-Pekka Tolvanen and Steven Kelly. Effort Used to Create Domain-Specific Modeling Languages. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, pages 235–244. ACM, 2018. https://doi.org/10.1145/3239372.3239410.

[111] Gregory Travis. 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 January 31, 2020.

[112] Daniel Tuchscherer, Alexander Weibert, and Frank Tränkle. Modern C++ As a Modeling Language for Automated Driving and Human-robot Collaboration. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 136–142. ACM, 2016. https://doi.org/10.1145/2976767.2976772.

[113] E. Van Wyk, O. de Moor, G. Sittampalam, I. Sanabria-Piretti, K. Backhouse, and P. Kwiatkowski. Intentional Programming: a Host of Language Features. Technical Report PRG-RR-01-21, University of Oxford, 2001. Online at https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.5286, accessed on January 31, 2020.

[114] Todd L. Veldhuizen. Blitz++: The Library that Thinks it is a Compiler. In Hans Petter Langtangen, Are Magnus Bruaset, and Ewald Quak, editors, *Advances in Software Tools for Scientific Computing*, pages 57–87. Springer, 2000. https://doi.org/10.1007/978-3-642-57172-5_2.

[115] Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University, Indianapolis, IN, USA, 2004. Online at https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.3916, accessed on January 31, 2020.

[116] Todd L. Veldhuizen and M. Ed Jernigan. Will C++ Be Faster Than Fortran? In *Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, ISCOPE '97, pages 49–56. Springer, 1997. https://doi.org/10.1007/3-540-63827-X_43.

[117] Markus Voelter, Bernd Kolb, Klaus Birken, Federico Tomassetti, Patrick Alff, Laurent Wiart, Andreas Wortmann, and Arne Nordmann. Using Language Workbenches and Domain-Specific Languages for Safety-Critical Software Development. *Software & Systems Modeling*, 18(4):2507–2530, 2019. https://doi.org/10.1007/s10270-018-0679-0.

[118] Markus Voelter, Zaur Molotnikov, and Bernd Kolb. Towards Improving Software Security Using Language Engineering and Mbeddr C. In *Proceedings of the Workshop on Domain-Specific Modeling*, DSM '15, pages 55–62. ACM, 2015. https://doi.org/10.1145/2846696.2846698.

[119] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. Mbeddr: An Extensible C-Based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 121–140. ACM, 2012. https://doi.org/10.1145/2384716.2384767.

[120] Martin P. Ward. Language-Oriented Programming. *Software—Concepts and Tools*, 15(4):147–161, 1994. Online at https://www.researchgate.net/publication/234125675_Language_Oriented_Programming, accessed on January 31, 2020.

[121] Tim Wescott. *Applied Control Theory for Embedded Systems*. Newnes, 1st edition, 2006. https://doi.org/10.1016/B978-0-7506-7839-1.X5000-4.

[122] English Wikipedia. AlphaGo. Online at https://en.wikipedia.org/wiki/AlphaGo, accessed on January 31, 2020.

[123] English Wikipedia. Deep Blue (chess computer). Online at https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer), accessed on January 31, 2020.

[124] English Wikipedia. Watson (computer). Online at https://en.wikipedia.org/wiki/Watson_(computer), accessed on January 31, 2020.

[125] Richard Zurawski, editor. *Embedded Systems Handbook*. CRC Press, 1st edition, 2005. https://isbnsearch.org/isbn/978-0-849-32824-4.

Please skip this page.

Index