big data and cognitive computing

MDPI

Article

Human-Error Prevention for Autonomous Edge Software Using Minimalistic Modern C++

Ryo Fukano [1,*] and Masato Ishikawa [2]

1 Komatsu MIRAI Construction Equipment Cooperative Research Center, Osaka University, Osaka 565-0871, Japan
2 Dept. of Mechanical Engineering, Osaka University, Osaka 565-0871, Japan; ishikawa@mech.eng.osaka-u.ac.jp
* Correspondence: fukano@jrl.eng.osaka-u.ac.jp; Tel.: +81-6-6875-7220

Received: 15 September 2019; Accepted: 18 October 2019; Published: 4 November 2019

Abstract: In science and engineering using edge-embedded software, it is necessary to demonstrate the validity of results; therefore, the software responsible for operating an edge system is required to guarantee its own validity. The aim of this study is to guarantee the validity of the sampled-time filter and time domain as fundamental elements of autonomous edge software. This requires the update law of a sampled-time filter to be invoked once per every control cycle, which we guaranteed by using the proposed domain specific language implemented by a metaprogramming design pattern in modern C++ (C++11 and later). The time-domain elements were extracted from the software, after which they were able to be injected into the extracted software independent from the execution environment of the software. The proposed approach was shown to be superior to conventional approaches that only rely on the attention of programmers to detect design defects. This shows that it is possible to guarantee the validity of edge software by using only a general embedded programming language such as modern C++ without auxiliary verification and validation toolchains.

Keywords: edge-computing architecture; real-time computing; software verification; ISO compliant

1. Introduction

1.1. Background

The increasing scale in which embedded software is being developed has become a matter of deep concern because of increasing design failures attributable to the carelessness of designers [1]. The complexity of large-scale embedded software is such that the rapid prevention of design and implementation failures is difficult, although science and engineering require rigorously reproducible software. Furthermore, today's embedded software is generally a part of a huge system linked to the cloud. This complexity is even greater.

Prevention of these failures is the duty of scientists and engineers using embedded software because embedded software in edge devices, advanced robots, and many types of uncrewed vehicles is designed to become part of our daily lives. Hence, methods to prevent these failures from being incorporated into embedded software are urgently required for both research and development purposes, and user safety.

In order for cloud applications to properly analyze data from an edge, the responsibility of edge software requires that its data uploading to the cloud shall be reproducible and deterministic (Figure 1). From an application-oriented viewpoint, machine learning and data analysis on edge devices [2,3] enable executing their applications with limited resources. On the other hand, from the building-block viewpoint, major cloud services such as AWS [4] and Azure [5] provide C/C++ language SDKs for edge software that cooperate with the clouds. These SDKs provide functions for sending and receiving data to and from the clouds. These two viewpoints do not provide a design method for the stability of overall edge software that works for a long period.

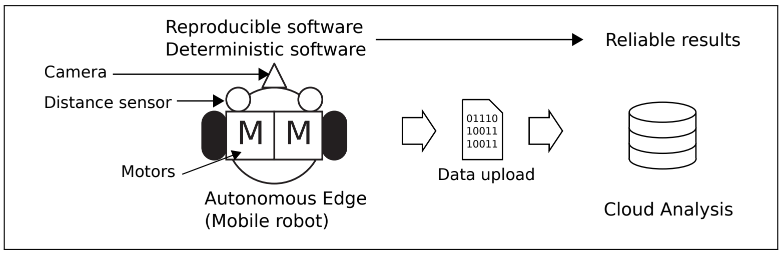

Figure 1. Reproducible and deterministic edge device yields reliable cloud-analysis results.

In terms of embedded software, typical factors for disturbing reproducibility and nondeterministic execution are the inclusion of the time domain as nonfunctional requirement, and the implementation of the human error of cyclic execution, such as sampled-time filters. Pure functionalities and the time domain are usually coupled in embedded software because the software not only has to comply with software-quality requirements, but also physical-boundary conditions. Additionally, this coupling often confuses programmers.

An autonomous robot is a kind of movable edge device for sensor-data acquisition in the real world. It can collect data using its sensors and also simultaneously move itself for data acquisition. Subsumption (SA) [6] architecture, installed on the early generations of the Roomba [7], is one of the most successful examples for this purpose. The feature of SA is that consists of tightly coupled functionalities, a time domain, software, and hardware. This tightly coupled architecture ensures robust behavior in the real world. However, software engineering warns that tightly coupled architecture is known to cause nondeterministic behaviors, which could lead to fatal disasters such as those reported in [8]. Motors, sensors, and a camera mounted on the robot (Figure 1) have different control rates (Figure 2). When the SA robot is working in the field, its software requires transaction processes to achieve a different control cycle and interference between SA layers.

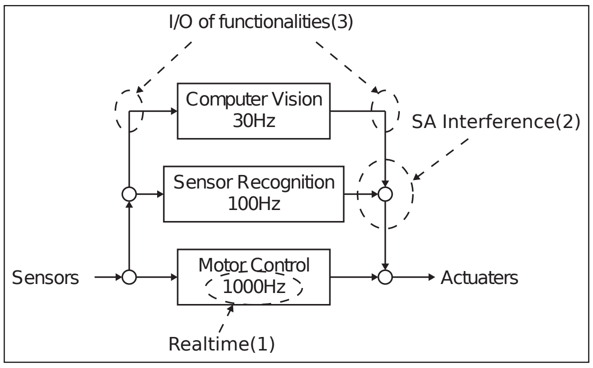

Figure 2. Software architecture of subsumption architecture in a mobile robot.

Object-Oriented Analysis and Design (OOAD) [9], generally used for large-scale software design, cannot solve the problems associated with tightly coupled SA. An essential purpose of OOAD is to decrease the connectedness between functionalities. However, first, robot software requires real-time capabilities classified as a nonfunctional requirement to execute its functionalities in the actual world. Second, the robustness of SA is provided by interference between SA layers by cross-referencing software functionalities. This ability to function in real time and to achieve interference are classified as cross-cutting concerns by Aspect-Oriented Analysis and Design (AOAD) [10] of the software and is difficult to implement using pure OOAD.

Furthermore, it was also necessary for us to consider sampled-time filters, which are fundamental elements of embedded-system design. These filters are defined as an all-online algorithm including internal states such as a Kalman filter, particle filter, and a finite-state automaton. The filters contain an update law that is inevitably required to be invoked once per one control cycle. Implementation of the algorithm as software does not prohibit the algorithm from invoking its update law twice or from ignoring the law by the carelessness of designers and programmers, because programming languages that are used to create robot software are designed for general purposes. These languages cannot prevent robot-specific failures.

This paper assumes all sampled-time filters to be implemented as classes. Classes *KalmanF* and *ParticleF* in Figure 3 are implementations of a Kalman filter and a particle filter. These two classes include member function *update* that has to be invoked once during each control cycle. The Kalman filter invokes its update law twice per one control cycle (♠), and the particle filter is not invoked (♡). Consequently, these invoking update laws are implementation failures. This loop also includes environmental dependency directly as *env_dependent_transaction* (♢).

```
KalmanF kf;
ParticleF pf;
// a lot of statements
while (true) {
  kf.update(); //♠
  // pf.update();//♡overlooking update
  // a lot of statements
  env_dependent_transaction(); //♢
  kf.update(); //♠excess update
}
```

Figure 3. Sampled-time filter update functions using traditional real-time loop implementation.

Programmers would not obtain any kind of noticeable feedback information, such as a compilation error, because implementation failures resulting from the invocation of update-law functions, such as those in Figure 3, are acceptable in terms of their programming-language grammars. Additionally, source-code inspection by review cannot theoretically guarantee the validity of robot software because reviewers' attention is limited. Therefore, implementation failures can only be detected by assessing the actual operation of the robot. Conventional methods of implementation implicitly assume that robots would experience runaway.

Domain Specific Language (DSL) [11] is a method used to enable a programming language to reinforce itself. The concept of DSL consists of "Internal DSL", which is constructed by the combination of its base language, and "External DSL", which is assisted by external tools. Internal DSL has the advantageous feature of being a lightweight method achieved by metaprogramming. Because C++, which is commonly used to pragmatically describe embedded software, provides a metaprogramming capability by using "template" grammar, internal DSL written by using C++ is expected to enforce robot-specific grammar and to improve the quality and productivity of embedded-software development. The lightweight method of internal DSL could overcome the defects of heavyweight methods such as middleware and formal methods.

1.2. Research Objectives

This paper proposes an internal DSL for autonomous edge software that only depends on the C++11 (ISO/IEC 14882:2011) [12] and later specification known as modern C++. Template specification has been greatly enhanced since modern C++. Therefore, it is not clear how much the quality of embedded software can be improved by the modern template.

The DSL is expected to isolate the time domain and functionality in edge software, and guarantee the invocation of update laws once per control cycle. This enables edge-software engineers to implement a tightly coupled embedded architecture, such as SA, as loosely coupled software. If the DSL, which only depends on C++ grammar, can guarantee the validity of embedded software, it would be possible to apply to all embedded software environments capable of running a C++ compiler.

This paper introduces the practically desirable goals listed below based on the above consideration.

1. Free from object-oriented constraints and simultaneously type-safe.
2. Using a minimal robot-software environment as a compiler of standard language specification.
3. No additional tools such as computer-aided software engineering (CASE) or software framework.
4. Providing well-designed building blocks easy to associate the result of a top–down approach.
5. Zero overhead in run-time.

2. Related Work

There are many challenges to improving the engineering quality of robot software. Existing works are roughly summarized into two approaches. One is the top–down approach, which includes formal methods and XX-Oriented design. The other is the bottom–up approach, which includes programming frameworks and programming techniques.

2.1. Top–Down Approach

Formal methods [13–15] and logic programming [16] extract consistency specification from embedded software. However, after extraction, executable software is generally implemented by programming language. The robot was designed by the model-based approach from the viewpoint of safety [17]; model conversion and execution code are required. Human errors occur during the implementation process. In addition, even if the executable source code is generated by automatic conversion, it is difficult to ensure readability for review.

Application-Oriented System Design [18] applied XX-oriented analyses to exhaustively operating system design in order to build a customizable OS. In this design process, object-oriented design decomposes OS functionalities from the viewpoint of a problem domain. At the same time, aspect-oriented design extracts nonfunctional requirements called "cross-cutting concerns" from the system. The aspects are independent from the objects. Results showed that it is necessary to apply multiple design methods to complex software products such as OSs. To apply aspect-oriented design [19–21] to robot software, its analysis can extract time elements as "cross-cutting concerns" to control its behavior. Implementation of the extracted elements remains an issue.

2.2. Bottom–Up Approach

Robot middleware such as OpenRTM [22] and ROS [23] cannot prevent implementation failure of a sampled-time filter and the dependency of an execution environment. For example, OpenRTM provides an $onExecute$ function for cyclic execution. However, a programmer manages a connection between filter and $onExection$ function only by attention, when the filter is implemented as OpenRTM component.

A testbed for robot software is one of the most promising applications [24,25] for robot middleware because its purpose does not require much portability. This kind of middleware usually provides bindings for multiple programming languages and simulators. Therefore, it is suitable as an environment to validate functionalities implemented in various languages. On the other hand, if the core functionality of the robot itself depends on specific middleware, coexistence with another testbed becomes an issue because typical middleware does not intend to coexist with another. A review of robot middleware types [26] shows that no middleware is superior to others in any respect. It is difficult to ignore portability by adopting the best one. In addition, these middlewares are heavyweight software, so their portability is an issue, although they can improve developmental productivity.

A simplified C language [27] eliminating some of its major features is proposed for robot education. This approach is classified as an external DSL [11] because its source code is parsed by Yacc [28] and does not support pass-by-reference and pointer operators as the features. Although these are notorious features of the C language, they are indispensable for the efficiency and portability of robot software.

To implement the extracted aspect using object-oriented language, two contrasting approaches are proposed: extending the language itself, and using programming techniques. AspectC++ [29] is an example of an extension of language specification; however, it deviates from ISO. Generics can be used to implement aspects because they can describe functionalities that are free from being object-constrained. C++, commonly known as an object-oriented language, is in fact a multiparadigm language. Thus, C++ can implement generics by macro and template means. The macro approach [30] is not type-safe. The template approach [31] can satisfy both ISO compliance and type safety. However, this approach does not have the capabilities of modern C++ because it based on C++03.

3. Dependency Injection DSL for Separation of Functionalities and Time Domain

3.1. Design Pattern: Type-Safe Portable Real-Time Control

1. Problem and Context: Robot software highly depends on the environment and cannot be ported to different platforms. It is especially difficult to port its simulator for testing.
2. Solution: Provide environment independency for robot software that includes different control cycles such as image processing, motor control, and sensing, in addition their interaction. In addition, because it is only implemented with the standard grammar of a programming language, it has portability and no additional tool is required.
3. Consequences and Trade-Offs: This is an extension of the bridge pattern of GoF [32] for robot software. Therefore, it inherits the issue of the bridge pattern.

3.2. Loosely Coupled Implementation of Tightly Coupled Architecture Using Modern C++

The robot architecture in Figure 1 includes a time factor that crosses its system (Figure 2 (1)), interaction among modules (Figure 2 (2)) and transaction processes for the input and output of the modules (Figure 2 (3)). For these items crossing over their system, AOAD [10] is proposed. OOAD encapsulates each object and weakens the association between objects, whereas implementations of AOAD often require a programming language with extensive capabilities.

This paper proposes an internal DSL using design patterns that only employ C++ specification to implement extracted objects and aspects without auxiliary toolchains. The patterns are shown in Figure 4, illustrated by UML 2. This approach injects a small aspect code, of which the validity is guaranteed, into objects. First, this design pattern extracts dependency from objects as an aspect in order to reduce module coupling. Then, this pattern assigns dependency by injection into an object after object implementation. In addition, thread functionality that is essential for parallel execution for the aspects was introduced as a standard from C++11.

3.3. Isolation of Objects and Real-Time Aspect

The bridge pattern [32] was introduced to isolate a real-time aspect (Figure 2 (1)) from other functionality objects, as shown as DI(1) in Figure 4. The isolation achieves writing independent robot functionalities from execution environments. The arrow direction from the RealtimeThreadIF class to the RealtimeFunctionalityIF one in Figure 4 represents independence.

3.4. Input/Output Aspect Implementation Independent from Execution Environment

A robot system based on SA has multiple running points because every functionality has specific control cycles. Hence, the Input/Output (I/O) aspect (Figure 2 (3)) should have transactional I/O that maintains consistency in all running points. If the transaction is improperly implemented, the robot system becomes a runaway, such that the system either aborts with an I/O deadlock or controller values of the system are not correctly copied. This behavior causes a fatal error in a critical robot system.

This paper proposes a "Dependency Injection" design pattern [33] for C++ to implement I/O aspects, as shown as DI(2) in Figure 4. Implementation is problematic in that each different type of I/O requires different transaction implementation because the language is statically typed. This feature causes reduplicated implementations. The problem of reduplication can be overcome by designing types of the I/O aspect as template arguments, as shown in Figure 5. Hence, functionality classes that require the aspect need to simultaneously inherit both the template class and the RealtimeFunctionalityIF class.

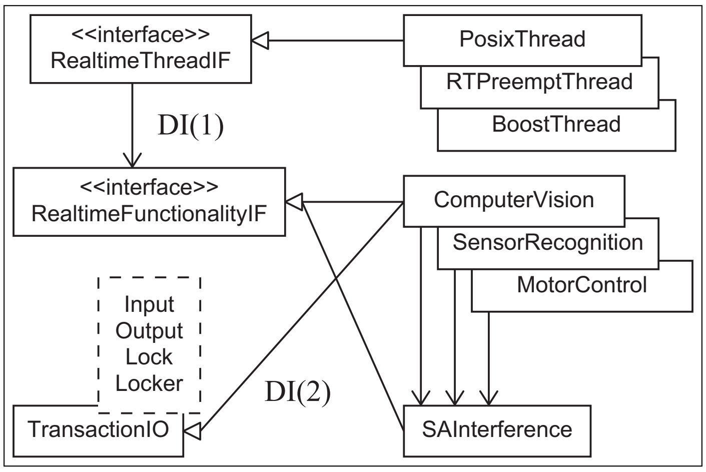

Figure 4. UML class diagram of software design by extracted functionalities and concerns from Subsumption (SA).

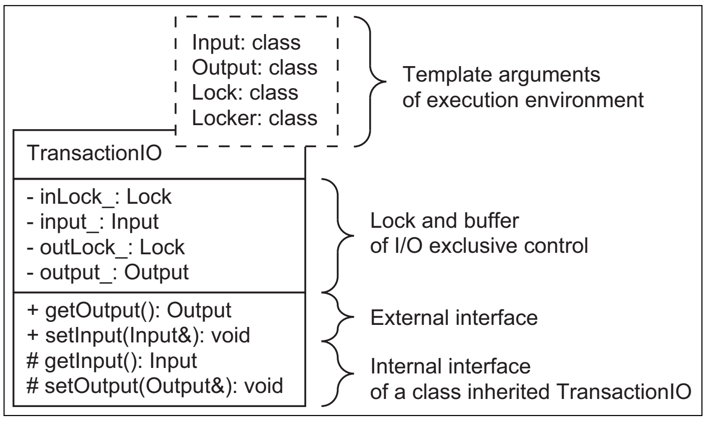

Figure 5. UML class diagram of Input/Output (I/O) of functionality aspect class.

The TransactionIO template class (Figure 5, implemented in Figure A1), is an implementation of the I/O aspect. The class connects public interfaces for external classes, and protected interfaces for subclasses via buffers as a transaction process. Implementation uses the "Parameterized Base Class" and "Thin Template" idioms [34]. The transaction process and I/O types are parameters bound at compilation time because of the effect of the "Parameterized Base Class". Their lock processes of exclusive access control are robust against exceptions due to the "Resource Acquisition Is Initialization" idiom [35].

Parameterization achieves that a transaction process provided by a real-time environment is independent from the environment, and a type-safe transaction process implemented by a statically typed language is simultaneously independent from the statically typed system. Additionally, the TransactionIO class inherited by all functionality classes requiring the I/O consumes few resources owing to the effect of the "Thin Template". This idiom reduces duplicate object code because the class only consists of an inline method.

3.5. Implementation of SA Interference Aspect

The features of the SA interference aspect (Figure 2 (2)) are encapsulated interference and the fastest control cycle to achieve infinite response speed when SA layers interfere with each other.

The combination of these two methods is a versatile design that implements real-time interaction as loosely coupled objects in embedded software.

A mediator pattern [32] is introduced to encapsulate the cross-reference of the objects. If the cross-reference of the objects were used to provide the interference of SA, the objects would involve tightly coupled implementation.

The control cycle of SA interference is configured at 1000 Hz, which corresponds to the fastest control cycle of the motor controller. The configuration of 1000 Hz provides a pseudoinfinite response speed from other SA layers with the same or slower control cycles from the viewpoint of interference.

4. Preventive Metaprogramming DSL for Real-Time Loop

4.1. Pattern: Compile as Right Control

1. Problem and Context: Robot software executes unexpected runaway by excessive or overlooked update of sampled-time filters.
2. Solution: Eliminate excessive or overlooked execution of sampled-time filters as compile error before robot-software execution. It can be applied to all types of robot software that use a sampled-time filter, e.g., signal filters, Kalman filters, and particle filters.
3. Consequences and Trade-Offs: It has portability and does not require an additional code generator because it is only implemented with the programming language's standard grammar. Compile-time check cannot detect errors that occur at run time. Therefore, an additional instance check is introduced in this paper.

4.2. Update-Law Metaprogramming Using Modern C++ Specification

This paper proposes compile-time metaprogramming to detect defective update-law functions before the robot software is put into operation.

The core of the proposed pattern is to invoke update functions by a destructor shown in Figure 6 when it is dismantled (Figure A2). The design pattern contrasts an idiom of must-release resource connected to a constructor and destructor, known as "Resource Acquisition Is Initialization" (RAII). RAII is achieved to define a class that manages resource acquisition and release. On the other hand, the pattern cannot be achieved in a single class; instead, the pattern can be achieved by using a design pattern that harmonizes more than one class. This reflects the complexity of robot software in comparison to resource management. This paper calls the pattern "Cyclic Execution Is Destruction" (CEID). CEID (Figure A3) theoretically prevents overlooked and excessive calls of the update law.

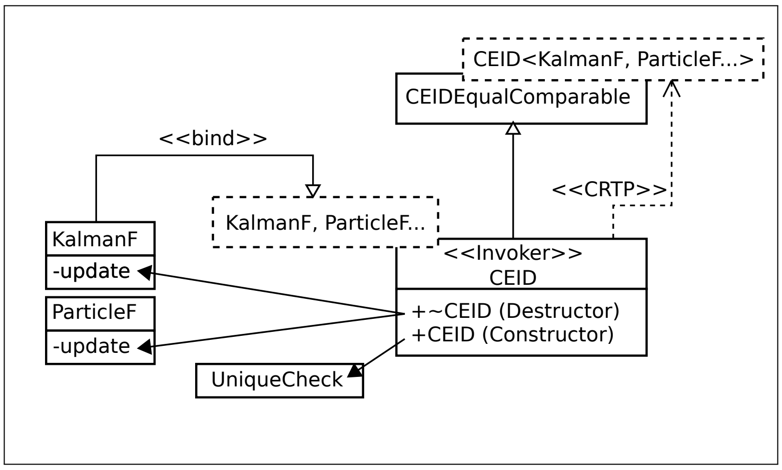

Figure 6. UML class diagram of "Cyclic Execution Is Destruction" template class.

In addition, to prevent overlooked registration of a sampled-time filter with CEID, it is necessary to inform the compiler of that. Therefore, the operator-existence check at compile time shown in

Figure 7, which is substantially enhanced from C++11, is combined with automatic operator generation. A compile error prevents embedded-software runaway beforehand, when there is no "=="operator. On the other hand, a unique check (Figure A4) for filter instances is executed at run time to prevent excessive registration.

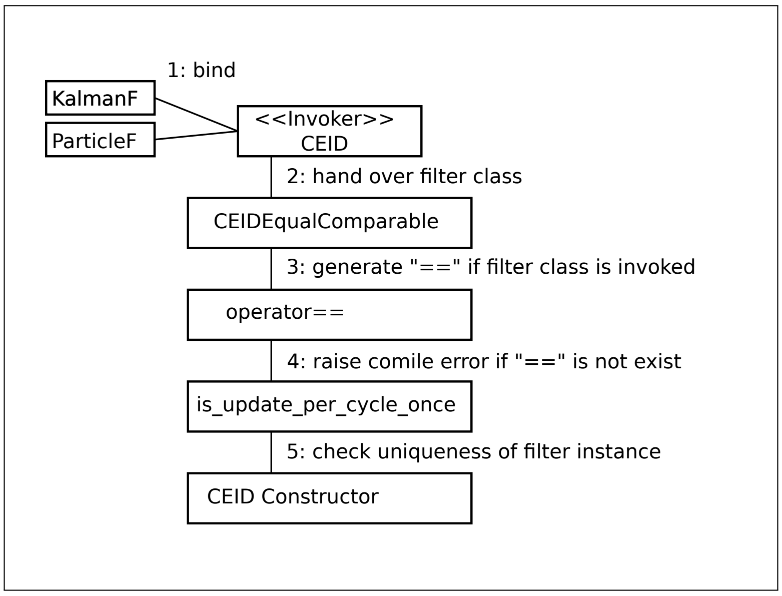
1: bind
KalmanF
ParticleF
<<Invoker>> CEID
2: hand over filter class
CEIDEqualComparable
3: generate "==" if filter class is invoked
operator==
4: raise comile error if "==" is not exist
is_update_per_cycle_once
5: check uniqueness of filter instance
CEID Constructor

Figure 7. UML collaboration diagram to eliminate excessive or overlooked execution of sampled-time filters.

These mechanisms are also implemented by variadic templates (Figure A5) introduced in C++11 to handle any number of filter classes at compile time. Variadic templates require recursive template declaration to handle any number of filter classes because the compiler does not know the number of sampled-time filters.

4.3. Overlooking to Invoke is a Compilation Error Using "Substitution Failure Is Not An Error" (SFINAE)

Automatic operator generation to invoke the update laws at once per cycle is closely related to C++11 grammar. Therefore, it is difficult to explain with a UML diagram, which is a general expression method. For this reason, this section represents the mechanism to guarantee update laws using source code.

C++ specification has a name-resolving mechanism to accept functions with the same name. This mechanism, SFINAE, does not judge substitution failure as a compilation error at compile time. The mechanism enables to detect overlooked sampled-time filters to invoke an update law of a sampled-time filter using automatic operator generation, taking the filter class and the special invoking class at compile time.

The proposed design pattern uses the "Curiously Recurring Template Pattern" [34] to automatically declare an operator function for detection. The "Curiously Recurring Template Pattern" (CRTP) was designed to substitute an operator member function that requires a complicated manual code description. Automatic declaration of the operator for the update-invoking class is shown in Figure 8.

The upper part of Figure 8 (■) shows a template class for automatic operator declaration. The declaration of the update-invoking class in the lower part of Figure 8 (▲) inherits an operator template class taken from the invoking class itself, and a target sampled-time filter class as argument. Therefore, member class function *equal_to* of the invoking class is executed as a "==" operator by the operator template class.

```
template<typename T, typename U>
class CEIDEqualComparable {//■
  friend bool operator==(T const &a, U const &b) {
    return a.equal_to(b);
  }
};

template
<class THead, class... TTails>
class CEID<THead, TTails...>//▲
  : private CEIDEqualComparable<CEID<THead, TTails...>, THead> {
public:
  bool equal_to(THead const& rhs) const {
    return false;
  };
};
```

Figure 8. Automatic operator generation by "Curiously Recurring Template Pattern".

The class that was used to inspect the absence or presence of the "==" operator at compile time is shown in Figure 9. "==" operators are generated for all filter classes corresponding to the special invoking class. The absence of a "==" operator even induces *overloading* for invoking inspection by SFINAE. Therefore, a filter class without this operator can be judged as an overlooked invocation.

To check the absence of a "==" operator, a trailing return type "−>" introduced by C++11 extracts a type of class corresponding to the truth value using *decltype*. The template-member function in the upper part of Figure 9 (♯) indicates "True" as return type by *overloading* name resolving when the "==" operator of the special invoking class and a filter class exists. The template-member function in the middle of Figure 9 (♭) indicates "False" as return type when the "==" operator does not exist.

An interface class to obtain the truth value at compile time is shown in the lower part of Figure 9 (♮). First, the class generates *overloading* of the above member function by using a transfer from its template arguments to the arguments of *is_update_per_cycle_impl*. Second, the class inherits an extracted type of the truth value using *decltype*. Finally, the class obtains the truth value corresponding to the absence or presence of the update invocation as a member value at compile time.

```
struct is_update_per_cycle_once_impl {
  template <class T, class U>
  static auto check(T*, U*) ->//♯
  decltype(std::declval<T>() == std::declval<U>(), std::true_type());
  template <class T, class U>
  static auto check(...) ->//♭
  std::false_type;
};

template <class T, class U> struct is_update_per_cycle_once //♮
: decltype(is_update_per_cycle_once_impl::check<T, U>(nullptr, nullptr)) {
};
```

Figure 9. Compile-time decision using *overloading* of functions.

The declaration of a sampled-time filter to raise a compile error if the filter is not invoked is shown in Figure 10. The truth value to raise is assigned to the "value" member parameter of *is_update_per_cycle*. Then, the "value" is assigned to an argument of *static_assert* that raises an error by the compile-time truth value. Consequently, this design pattern can control the raising of a compile error regardless of whether an update-law function is invoked. In practice, a template or a macro that defines an instance and *static_assert* at the same time is useful.

```
using CEIDAll = CEID<KalmanF, ParticleF>;
KalmanF kf;
static_assert(is_update_per_cycle_once<CEIDAll, decltype(kf)>::value,
              "Kalman␣filter␣is␣not␣updated.");
ParticleF pf;
static_assert(is_update_per_cycle_once<CEIDAll, decltype(pf)>::value,
              "Particle␣filter␣is␣not␣updated.");
```

Figure 10. Assertion of overlooked update law at compile time.

5. Results

5.1. Summary of Proposed Methods

The proposed methods achieve the following features that solve the prescribed requirements.

1. Free from object-oriented constraints, simultaneously type-safe (i.e., achieved by the proposed design patterns).
2. Using a minimal robot-software environment (i.e., implemented under C++11 compliance).
3. No additional tools (i.e., implemented by the C++11 compliance design patterns).
4. Providing well-designed building blocks (i.e., achieved by "Type-Safe Portable Real-time Control").
5. Zero overhead in run time (i.e., achieved by "Compile as Right Control").

5.2. Portability by ISO Compliant

This paper ported the two proposed design pattern for four types of computers: Linux PC and MacOSX as desktop, Raspberry Pi [36] and Arduino [37] as embedded (see Table 1). Both "'Type-Safe Portable Real-time Control" and "Compile as Right Control" work well on Linux PC, MacOSX, and Raspberry Pi. However, the patterns do not work on Arduino. The first pattern using a standard thread could not be compiled because Arduino is a single-process environment. In addition, the second pattern could not be compiled because the standard C++11 header is missing in the Arduino environment. This result shows that these design patterns achieve portability in any ISO C++11 environment.

Table 1. Portability of proposed design patterns.

	Linux PC	**MacOSX**	**Raspberry Pi 3B+**	**Arduino**
Architecture	Intel x86	Intel x86	ARM Cortex	AVR
OS	Linux 3.10.17	OSX 10.9.3 OSX 10.13.6	Linux 4.14	N/A (Arduino IDE 1.8.3)
Compiler	g++ 4.7.3	g++ 4.8.2 clang 9.0	g++ 6.3.0	g++ 5.4.0
Standard library	Yes	Yes	Yes	N/A
Porting	Yes	Yes	Yes	N/A

5.3. Development of SA Real-Time Simulation Using "Type-Safe Portable Real-Time Control"

The authors implemented controller software for an SA robot implemented by C++03 and C++11 (Table 2) using the proposed design patterns as a case study. C++03 also had the ability to implement the proposed design patterns, but it requires deviation from ISO. Robot software requires time-adjustment sleep to periodically execute a control cycle. The *std* :: *chrono* introduced in C++11 indicates that it can implement the aspect. On the other hand, a nonstandard sleep function is required to execute on a real-time OS. Even in this case, switching is easy due to the loosely coupled design.

Quantitative evaluation of this implementation by cccc [38] is shown in Table 3. The table shows the cyclical complexity of the implementation responsible for SA behavior, isolation to the environment, and parallel processing. From the perspective of metrics, C++11 and C++03 did not show any difference in complexity. The SAInterference object has maximum 8 of cyclomatic complexity. In addition, the sum of all implemented objects was 33. In general, software testing is difficult if complexity is 20 or more. Decomposition by the proposed methods kept maintainability in the applicable range.

Table 2. Comparison of robot software using C++11 and C++03.

	C++11	**C++03**
Real-time standard	No	No
Concurrency standard	Yes	No
Time-adjustment standard	Yes	No
Copy-control grammar	Yes	No
Strict override	Yes	No

Table 3. Comparison of cyclomatic complexity using C++11 and C++03.

	C++11	**C++03**
ComputerVision	6	6
MotorControl	6	6
SensorRecognition	5	5
SAInterference	8	8
RealtimeFunctionalityIF	0	0
ChronoCycleLogicalThread	1	N/A
RTPreemptCycleLogicalThread	3	3
TransactionIO	4	4
Total	33	32

According to qualitative analysis, C++11 has stronger grammar in terms of inheritance and parallel execution than C++03 (Table 2). For example, the *override* specifier keyword supports the compiler in finding the wrong member function to override. Although it is difficult to quantitatively evaluate the effects of these types of grammar, C++11 grammar might be able to automatically prevent human error.

Another "Dependency Injection" architecture validated as a commercial embedded C++ system [39] was invented independent from this paper. The effectiveness of the architecture was measured by the software metrics provided in Table 4.

Table 4. Productivity of another "Dependency Injection" architecture [39].

Total Man Hour	**New Code**	**Bugs**	**Bug Density**
85.5%	127.0%	31.7%	17.9%

From this result, the proposed "Type-Safe Portable real-time Control" can be expected to achieve equivalent productivity. Moreover, the proposed "Compile as Right Control" design pattern can also be expected to improve productivity because the pattern can save embedded software from common occurrence failures.

5.4. Robust Behavior Simulation to Verify Independence from Execution Environment

Simulation testing used the ChronoCycleLogicalThread class by *std* :: *chrono* (soft real-time) and the RTPreemptLogicalThread class (hard real time) by RT Preempt [40] as "Dependency Injection". The performance of the robot was tested to verify its integrated behaviors.

The mobile environment of the simulation included an obstacle and a reaching goal. The system test for the integrated behavior is represented in Figure 11. The path was plotted every 0.1 s, and total testing time was 7.0 s for each intensity. The figure demonstrates that the robot behaved rationally whenever the interference intensity of the concurrent avoidance and reaching was changed. Results showed that the simulator could exchange soft and hard real time by "Dependency Injection", although the simulator did not require modification of the core part of the controller, and the real-time functions were provided from different computational environments, respectively.

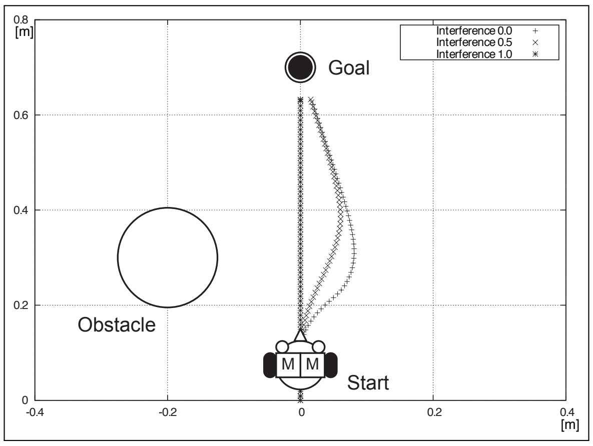

Figure 11. System test simulation of SA using proposed design pattern.

5.5. Defective Design Prevention Using "Compile as Right Control"

The proposed design pattern (code fragments of Figures 8, A4, and A5 were integrated as one class) demonstrated that it can prohibit the failure of update-law invocation of sampled-time filters, as shown in Figure 10. The truth value for inspection is stored in the value member parameter. Then, the truth value is passed to *static_assert* that generates an error by the passed value at compile time. Instance-uniqueness investigation of double-invoking can also stop executing the sampled-time filters by run time. Using the combination of metaprogramming and run-time investigation, the proposed method succeeded in preventing runaway before the robot started its control.

6. Discussion

In this paper, we proposed an internal DSL with two design patterns, "Type-Safe Portable Real-time Control" and "Compile as Right Control" as building blocks. The proposed DSL enables abstract design without additional modeling tools. This minimalistic approach can provide software-quality improvement opportunities for start-ups that investigate feasibility using a small controller, such as Raspberry Pi, and small-scale development organizations, such as university laboratories. In addition, the proposed method only uses a compiler that does not require external tools or language extensions, and it has high affinity with external software libraries. We can expect synergy with previous studies [2,3] and open-source artificial-intelligence products, for example, OpenCV [41], TensorFlow [42], and Autoware.AI [43], which can enlarge edge capability.

This minimalistic approach based on DSL makes full use of built-in programming-language grammar to detect design defects. The internal DSL eradicates implementation failures from sampled-time filters, which are fundamental elements of real-time signal-processing software. Furthermore, the method prevents failures in the time domain and increases the productivity of complex and robust autonomous-edge software, including the time domain, such as subsumption architecture. These results show that the minimalistic approach can provide a building block to wide domains of edge systems.

The defective design of software has traditionally faced the challenge of achieving effective prevention with minimal effort. Existing approaches, such as code review and formal methods, rely on programmers' attention and mathematical inspection, respectively, to eliminate design defects. To apply them to the time domain and existing assets, particularly in the case of embedded software, is a significant challenge that needs to be overcome [44] because the size of a practical system is often too large for reviewers' perception and the computational complexity of formal methods. The building block by the minimalistic approach seems to decompose complexity to support them.

The main part of the proposed method applies static class analysis by the compiler to software inspection. The double-registration prevention for a sampled-time filter causes very small run-time overhead because it checks pointers of sampled-time filters in the CEID constructor. C++ provides a limited run-time check mechanism because C++ was born as a statically typed language. One possibility for this purpose is compile-time reflection [45], but it is currently under discussion in the ISO committee. Another is to apply "Int-To-Type" idiom [34] to sampled-time filters. A constant integer of this idiom creates a new type that is identified by CEID from an existing sampled-time filter type.

Promotion of a pattern catalogue for edge software is future work because this paper proposes just two design patterns. Abstract modeling of the minimalistic approach will continue to expand its scope by strengthening language specification. In particular, the "concept" of C++20 [46] will make it easier to enhance the inspection of edge software. "Concept" is a grammar type that evaluates the existence of an operator. Therefore, the proposed automatic operator generation according to programmers' intention would be able to more directly investigate by C++20 and later.

7. Conclusions

To our knowledge, it has not previously been possible to use the proposed internal DSL approach for edge software as an established method that is widely applicable in practice, even though this approach only depends on the grammar of C++11 and later, which is one of the most popular edge-software languages. We expect this approach to replace the conventional defensive programming approach and lead to the widespread introduction of safe edge software.

Author Contributions: Conceptualization, R.F. and M.I.; methodology, R.F.; software, R.F.; validation, R.F.; investigation, R.F.; writing—original-draft preparation, R.F.; writing—review and editing, R.F. and M.I.; visualization, R.F.; and supervision, M.I.

Funding: This work was supported by the Komatsu MIRAI Construction Equipment Cooperative Research Center, endowed by Komatsu Ltd. https://home.komatsu/.

Acknowledgments: We wish to thank the timely help given by Editage and MDPI English Editing Service in English proofreading.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of the data.

Abbreviations

The following abbreviations are used in this manuscript:

DSL	Domain Specific Language
SA	Subsumption Architecture
UML	Unified Modeling Language
OOAD	Object Oriented Analysis and Design
AOAD	Aspect Oriented Analysis and Design
DI	Dependency Injection
RAII	Resource Acquisition Is Initialization
CEID	Cyclic Execution Is Destruction
CRTP	Curiously Recurring Template Pattern
SFINAE	Substitution Failure Is Not An Error

Appendix A. Implementation of Dependency Injection DSL for Separation of Functionalities and Time Domain

Figure A1 represents the implementation of an automatic transaction process. The object receives a value from an external object via the setInput method. The value is stored in the input_ buffer by exclusive access control using inLock_ atomically. Then, the required value by functionality objects is atomically loaded by the getInput method. The output process is also symmetrically implemented. The locks of the exclusive access control are robust against exceptions due to the "Resource Acquisition Is Initialization" idiom [35].

```
template <class Input, class Output, class Locker, class Lock>
class TransactionIO
{
  // Transaction for class external access
  public:
  inline void setInput(const Input& input)
  {
    Locker locker(&inLock_);
    input_ = input;
    return;
  }

  // Transaction for class internal access
  protected:
  inline const Input getInput()
  {
    Input tmp;
    {
      Locker locker(inLock_);
      tmp = input_;
    }
    return tmp;
  };
};

class DependencyInjected
  : public RealtimeFunctionalityIF,
    public TransactionIO
      <ConcreteIn, ConcreteOut, ConcreteLocker, ConcreteLock>
{
  // Automatic transaction process
  ConcreteIn val = getInput();
  setInput(val);
}
```

Figure A1. Dependency injection design pattern for transaction process.

Appendix B. Implementation of Preventive Metaprogramming DSL for Realtime Loop

Figure A2 provides a concentrated declaration of update-law functions, which are a vital part of robot software. This method can control the invoking sequence of filter classes from the declaration position near the start position of the control loop.

Figure A3 represents the prohibition of the mistaken invocation of an update law by programmers using a *private* access qualifier. CEID, which is a special class for invoking update-law functions, can access the *private* member function using a *friend* qualifier.

Figure A4 represents a run-time investigation of a double-invoking instance to supplement compile-time investigation. The recursive execution of the CEID constructor recursively searches the pointers of sampled-time filters because the constructor is always executed before the sampled-time filters are executed.

The variadic template class in Figure A5 is a recursive declaration that separates the filter list into a head item and remaining tail items, and inherits the remaining items (★). The template class uses invocation of an update-law function of the head class as an argument. This class automatically invokes all update-law functions because the template class is recursively declared. The recursive declaration terminates itself to inherit the specialized template class with no argument (△) when the declaration reaches the last argument class.

```
using CEIDAll = CEID<KalmanF, ParticleF>;
while (true) {
  auto&& cyclicExecution = CEIDAll(kf, pf);
  // a lot of procedures
  // implicit destructor calling
}
```

Figure A2. Automatic cyclic execution in realtime loop.

```
class KalmanF {
  template<class... TTails> friend class CEID;
private:
  void update();
};
```

Figure A3. Update member function of sampled-time filter in private access.

```
template <class ...>
class UniqueCheck;

template <>
class UniqueCheck<> {
public:
  UniqueCheck() {};
  bool const isUnique() const {
    return true;
  }
  const void* searchImpl(const void* p) const {
    return nullptr;
  };
};

template <class THead, class... TTails>
class UniqueCheck<THead, TTails...>
  : public UniqueCheck<TTails...> {
public:
  UniqueCheck(THead const& t,
              const TTails&... args)
    : UniqueCheck<TTails...>(args...),
    pointer_(&t) {
    assert(true == isUnique());
  }
  bool const isUnique() const {
    return (pointer_ == UniqueCheck<TTails...>::searchImpl(pointer_))
      ? false : UniqueCheck<TTails...>::isUnique();
  }
  const void* searchImpl(const void* p) const {
    return (p == pointer_)
      ? pointer_
      : UniqueCheck<TTails...>
      ::searchImpl(p);
  }
public:
  const void* pointer_;
};

// Unique check in constructor
template<class THead, class... TTails>
class CEID<THead, TTails...> {
public:
  explicit CEID(THead &cyclicObject,
                TTails&... args)
    : cyclicObject_(cyclicObject) {
    UniqueCheck<THead, TTails...> uniqueCheck(cyclicObject, args...);
  };
private:
  THead& cyclicObject_;
}
```

Figure A4. Run-time uniqueness inspection at constructor.

```
template <class... TTails>
class CEID;

template <>
class CEID<> {};//△

template <class THead, class... TTails>
class CEID<THead, TTails...>//★
  : public CEID<TTails...> {
public:
  explicit CEID(THead &filter, TTails... args)
    : CEID<TTails...>(args...), cyclicObject_(cyclicObject) {
  };
  ~CEID(){ cyclicObject_.update(); };

private:
  THead& cyclicObject_;
};
```

Figure A5. Variadic template invoking update-law functions.

References

1. Leveson, N.G. Chapter Computers and Risk. In *Safeware: System Safety and Computers*; Addison-Wesley Professional: Boston, MA, USA, 1995.
2. Yazici, M.; Basurra, S.; Gaber, M. Edge Machine Learning: Enabling Smart Internet of Things Applications. *Big Data Cogn. Comput.* **2018**, *2*, 26. [CrossRef]
3. Kang, K.D.; Chen, L.; Yi, H.; Wang, B.; Sha, M. Real-Time Information Derivation from Big Sensor Data via Edge Computing. *Big Data Cogn. Comput.* **2017**, *1*, 5. [CrossRef]
4. AWS IoT SDKs. Available online: https://docs.aws.amazon.com/iot/latest/developerguide/iot-sdks.html (accessed on 1 September 2019).
5. Understand and Use Azure IoT Hub SDKs. Available online: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-sdks (accessed on 1 September 2019).
6. Brooks, R.A. A robust layered control system for a mobile robot. *IEEE J. Robot. Autom.* **1986**, *2*, 14–23. [CrossRef]
7. *FORM S-1 iROBOT CORPORATION*; Chapter BUSINESS; 2005; p. 53. Available online: https://www.sec.gov/Archives/edgar/data/1159167/000095013505005611/b55709a2sv1za.htm#110 (accessed on 1 November 2019).
8. Leveson, N.G. Chapter Medical Devices: The Therac-25 Story. In *Safeware: System Safety and Computers*; Addison-Wesley Professional: Boston, MA, USA, 1995.
9. Meyer, B. *Object-Oriented Software Construction*, 2nd ed.; Prentice Hall: Englewood Cliffs, NJ, USA, 1997.
10. Jacobson, I.; Ng, P.W. *Aspect-Oriented Software Development with Use Cases*; Addison-Wesley Professional: Boston, MA, USA, 2005.
11. Ghosh, D. *DSLs in Action*; Manning Publications: Greenwich, CT, USA, 2010.
12. *The C++ Standards Committee*; ISO/IEC 14882:2011; ISO/IEC: Geneva, Switzerland, 2011.
13. Overture Project. Overture Tool Formal Modelling in VDM. Available online: http://overturetool.org (accessed on 4 January 2019).
14. Verifying Multi-Threaded Software with Spin. Available online: http://spinroot.com/spin/whatispin.html (accessed on 4 January 2019).
15. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics ISO/IEC 13568:2002*; ISO/IEC: Geneva, Switzerland, 2002.
16. Calegari, R.; Ciatto, G.; Mariani, S.; Denti, E.; Omicini, A. LPaaS as Micro-Intelligence: Enhancing IoT with Symbolic Reasoning. *Big Data Cogn. Comput.* **2018**, *2*, 23. [CrossRef]
17. Leite, A.; Pinto, A.; Matos, A. A Safety Monitoring Model for a Faulty Mobile Robot. *Robotics* **2018**, *7*, 32. [CrossRef]
18. Fröhlich, A.A. *Application Oriented Operating Systems*; GMD-Forschungszentrum Informationstechnik: Sankt Augustin, Germany, 2001.

19. Brugali, D.; Salvaneschi, P. Stable Aspects in Robot Software Development. *Int. J. Adv. Robot. Syst.* **2006**. [CrossRef]
20. Maoz, S.; Ringert, J.O.; Rumpe, B. Synthesis of component and connector models from crosscutting structural views. In Proceedings of the ESEC/FSE 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013.
21. Wehrmeister, M.A.; Freitasb, E.P.; Binottoc, A.P.D.; Pereirad, C.E. Combining aspects and object-orientation in model-driven engineering for distributed industrial mechatronics systems. *Mechatronics* **2014**, *24*, 844–865. [CrossRef]
22. OpenRTM-Aist Official Website. Available online: http://www.openrtm.org/ (accessed on 4 January 2019).
23. ROS Wiki. Available online: http://www.ros.org/wiki/ (accessed on 4 January 2019).
24. Correal, R.; Pajares, G.; Ruz, J. A Matlab-Based Testbed for Integration, Evaluation and Comparison of Heterogeneous Stereo Vision Matching Algorithms. *Robotics* **2016**, *5*, 24. [CrossRef]
25. Yan, Z.; Fabresse, L.; Laval, J.; Bouraqadi, N. Building a ROS-Based Testbed for Realistic Multi-Robot Simulation: Taking the Exploration as an Example. *Robotics* **2017**, *6*, 21. [CrossRef]
26. Jensen, K.; Larsen, M.; Nielsen, S.; Larsen, L.; Olsen, K.; Jørgensen, R. Towards an Open Software Platform for Field Robots in Precision Agriculture. *Robotics* **2014**, *3*, 207–234. [CrossRef]
27. Gonzalez, F.; Zalewski, J. Teaching Joint-Level Robot Programming with a New Robotics Software Tool. *Robotics* **2017**, *6*, 41. [CrossRef]
28. Johnson, S.C. Yacc: Yet Another Compiler-Compiler. Available online: http://dinosaur.compilertools.net/yacc/index.html (accessed on 18 June 2019).
29. Spinczyk, O.; Gal, A.; Schröder-Preikschat, W. AspectC++: An aspect-oriented extension to the C++ programming language. In *CRPIT '02 Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*; Australian Computer Society, Inc.: Darlinghurst, Australia, 2002; pp. 53–60.
30. Willink, E.D.; Muchnick, V.B. Preprocessing C++: Meta-class aspects. In Proceedings of the Eastern European Conference on Technology of Object Oriented Languages and Systems, Prague, Czech Republic, 7 July 1999.
31. Yao, Z.; Zheng, Q.L.; Chen, G.L. AOP++: A Generic Aspect-Oriented Programming Framework in C++. In Proceedings of the 4th International Conference, GPCE 2005, Tallinn, Estonia, 29 September–1 October 2005; pp. 94–108.
32. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Professional: Boston, MA, USA, 1995.
33. Fowler, M. *Inversion of Control Containers and the Dependency Injection Pattern*; Technical Report; ThoughtWorks: Chicago, IL, USA, 2004.
34. Tambe, S.; Many Other C++ aficionados. More C++ Idioms. Available online: https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms (accessed on 4 January 2019).
35. Sutter, H. *Exceptional C++*; Addison-Wesley Professional: Boston, MA, USA, 1999.
36. Raspberry Pi. Available online: https://www.raspberrypi.org/ (accessed on 22 June 2019).
37. Arduino. Available online: https://www.arduino.cc/ (accessed on 17 June 2019).
38. CCCC-C and C++ Code Counter. Available online: http://cccc.sourceforge.net/ (accessed on 22 June 2019).
39. Adachi, N. Model based development utilizing DI container for wide variety products and effective tests. In Proceedings of the JUSE Software Quality Symposium 2014, Tokyo, Japan, 14 September 2014. Available online: https://www.juse.jp/sqip/symposium/archive/2014/day1/files/happyou_D2.pdf (accessed on 31 October 2019). (In Japaneses)
40. Maria, A. RT PREEMPT HOWTO. Available online: https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/application_base (accessed on 4 January 2019).
41. OpenCV. Available online: https://opencv.org/ (accessed on 17 June 2019).
42. TensorFlow. Available online: https://www.tensorflow.org/ (accessed on 17 June 2019).
43. Autoware.AI. Available online: https://www.autoware.ai/ (accessed on 17 June 2019).
44. He, K.; Lahijanian, M.; Kavraki, L.E.; Vardi, M.Y. Towards manipulation planning with temporal logic specifications. In Proceedings of the 2015 IEEE international conference on robotics and Automation (ICRA), Seattle, WA, USA, 26–30 May 2015; pp. 346–352.

45. Sankel, D. N4766 Draft, C++ Extensions for Reflection. Available online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4766.pdf (accessed on 15 October 2019).
46. The C++ Standards Committee. Wording Paper, C++ Extensions for Concepts. Available online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0734r0.pdf (accessed on 22 June 2019).