Contracts for Real-Time, Safety Critical Systems

Development of a contract framework for the FASA platform

Software Systems Group,
ABB Corporate Research,
Baden-Daettwil

Laboratory for Automated Reasoning and Analysis,
School of Computer and Communication Sciences,
École Polytechnique Fédérale de Lausanne

Masters Thesis
August 8, 2014
Submitted in fulfillment of the
requirements for the degree of Master of Science
in Computer Science
by

Chandrakana Nandi

accepted on the proposal of the jury:
ABB supervisors: Dr. Manuel Oriol,Dr. Aurelien Monot
EPFL supervisor: Prof. Viktor Kuncak

Lausanne, EPFL, 2014

Karmanye Vadhikaraste, Ma Phaleshou Kada Chana,
Ma Karma Phala Hetur Bhurmatey Sangostva Akarmani.
—*Lord Krishna, Bhagvad Gita*

To my grandparents…

Acknowledgements

The author would like to express her sincere gratitude towards Dr. Manuel Oriol and Dr. Aurelien Monot for giving her the opportunity to work in their group at ABB Corporate Research. They have been very supportive and provided valuable suggestions. She would also like to thank Dr. Stefan Stattelmann of ABB Corporate Research, Germany, for his feedback, which was very helpful.

The author is extremely grateful to Prof. Viktor Kuncak for accepting her as a thesis student in his laboratory and for his support and guidance. She would like to thank Eva Darulova and Ravichandhran Kandhadai Madhavan for their kind advice.

She is deeply indebted to Prof. Joseph Sifakis for introducing her to the field of Formal Verification and Correctness by Construction and encouraging her to pursue research in this area.

Finally, she expresses her heartfelt gratitude towards God who gave her the strength to carry on with her work. She thanks her parents who have always had faith in her and given her the courage to move forward in life. She remains ever grateful to her siblings and Puma who are always there for her in all situations.

It is only due to the cumulative support of all that this thesis could be successfully compiled.

Lausanne, 08 August, 2014 C. N.

Abstract

Verifying real-time systems goes beyond the verification of functional properties: it also requires checking of real-time properties. This makes traditional contract-frameworks partially inept for checking real-time programs. This is a major problem because the failure of real-time and safety critical systems can have serious consequences. This thesis presents a solution to this problem by incorporating Design by Contracts to these systems. The main contribution of this thesis is the development of a contract framework for the Future Automation System Architecture (FASA) platform, used for developing *hard real-time* control applications. The contract framework allows the users to specify both functional and *temporal* properties for FASA. A novel approach of *empirical cdf* [3] based statistical inference is used for dynamically estimating temporal constraints and incorporating them in future contracts. The thesis illustrates the use of Real-time Logic (RTL) for formal specification of the statistical properties. The framework is validated by benchmarking and its performance is analyzed. The experiments show that this framework can be smoothly integrated to the FASA platform thereby increasing the reliability and dependability of the control applications while having a negligible impact on the performance of FASA.

Key words: Design by Contracts, Dynamic Verification, Real-time Applications, Statistical Inference, Component-based Software Engineering, Formal Logic

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In our day to day lives, our activities are influenced by many complex systems for which correct functioning is absolutely necessary. Any failure for such systems can have serious consequences. Let us take the example of the safety system of an automobile. Most modern cars have air bags for preventing the occupants from hitting the dashboard in case of an accident. The way it works is that there is an accelerometer which is capable of detecting collision forces. When this force becomes more than a particular threshold value, the collision is considered to be a crash and a crash sensor is notified. This sensor then causes the airbag to inflate [13]. If there is a slight delay in the transmission of the signals from the sensor to the controller that ejects the airbag or if there is a bug in the program that calculates the collision force, it may lead to failure of the safety system, causing loss of life. This is an example of a real-time, safety critical system. Other examples are pacemakers, nuclear reactors, the control system of an aircraft and online banking transaction systems [26].

In today's generation, most real-time, safety critical systems rely on computer systems and software in some form or the other. In order to ensure the reliability of safety critical systems, it is essential to ensure the reliability and correctness of the underlying computer hardware and software. In this sense, a real-time system can be considered as a layered system consisting of a software that implements various tasks and an execution platform (hardware) on which the software is run [40]. There are continuous interactions between these two layers which makes it a major challenge to ensure the dependability, safety and reliability of safety-critical systems. One common technique adopted in industries for developing complex real-time systems is to use a component-based approach. In this approach, a complex system is composed of several reusable components which interact with each other. For such systems, reliability is even more challenging because merely ensuring the reliability of the independent components will not be sufficient. One needs to think about the correctness of the system as a whole. Ensuring the correct construction of these systems requries verification of functional as well as real-time properties.

This thesis tackles the issue of correct construction of component-based real-time and safety critical systems by means of using the principle of *Design by Contracts* [33]. It revolves around the dynamic verification of a component-based framework, FASA(Future Automation System Architecture) [34], used for constructing real-time control applications, developed at ABB Corporate Research.

Our experiments show that for applications deployed on a single host, the average overhead added due to the contract framework is only 5.4 % for a cycle time of 10 ms, which renders it efficient and easily incorporable on top of FASA.

1.1 Statement of the problem

The challenge of this thesis is to bring *Design by Contracts* to real-time safety critical systems for allowing verification and testing. It entails the development of a contract framework for FASA which allows checking of preconditions, postconditions and invariants. The framework has been developed such that it can be turned on/off depending on whether the user wants to have contracts enabled at runtime. It allows users to specify functional as well as real-time properties for the applications while mainly focusing on the latter. What makes it challenging is that FASA targets a cycle time of 1 ms, which requires the contract framework to have as little overhead as possible.

While contracts have been used in the past to specify functional properties for programs, using it together with statistical inference for dynamically estimating and defining temporal properties of real-time systems is something that has not been explored in the past.

Moreover, this thesis uses Real Time Logic (RTL) [25] to formalize the stochastic temporal contracts. This is a unique contribution of this thesis because RTL has not been used so far for formalizing statistical properties.

1.2 Structure of the thesis

This thesis contains six chapters. Chapter 2 describes the principle of design by contract, the FASA framework, real time logic and presents an analysis of previous research. Chapter 3 describes the requirements and development principles of the contract framework. Chapter 4 presents the implementation details of the framework. Chapter 5 illustrates benchmarks and validates the techniques described in the previous chapters with results. We conclude in chapter 6 and describe possible future extensions.

2 State of the Art

2.1 Background

This section describes the concept of design by contract, introduces the FASA framework and also highlights some features of Real-time Logic (RTL) which is used for formalizing the contract framework developed in this thesis.

2.1.1 Design by Contracts

One of the first attempts towards verifiable programming was done in the 1970s using the language *Euclid* [28]. This language was not meant for writing large application but it allowed the use of external verification tools on moderately sized programs for verification [28]. Later, the concept of verifiable programs became more popular with the advent of contract programming. The term Design by Contract (DbC) [33] was coined by Bertrand Meyer, which is an approach for developing reliable Object Oriented Programming (OOP)-based software. Most programmers often adopt a defensive programming approach [33] in which the programmers makes numerous checks to make sure that all corner cases are handled well. This usually introduces several redundant checks and at times, increase the complexity of the program, thereby degrading its performance. DbC prevents this from happening. In most moderate to large sized applications, a program is broken down into several tasks and subtasks and a particular job is accomplished by making calls to these tasks and subtasks. In DbC terminology, the callers are called *clients* and the called routines are called *suppliers* and there is a *contract* between the client and the supplier in which the demands and requirements for each of them are listed. This is done by means of *assertions.* Assertions are boolean expressions. They can be of three types, *preconditions*, *postconditions* and *invariants.*
Eiffel Language: Contracts are inherently supported in the *Eiffel* programming language. The following pseudo code shows how pre and post conditions are written in Eiffel [33].

```
routine_name (argument declarations) is
require
      Preconditions
do
      Routine body
ensure
      Postconditions
end
```

Pre and post conditions: Theoretically, pre and post conditions can be expressed as a Hoare triplet [22] as:

$$P \{ S \} Q ,$$

where P is a precondition, S is the program and Q is the post condition.

A caller is expected to satisfy the preconditions if it wants to make a call to the routine and in turn, the routine is expected to satisfy the postconditions after its execution. In Eiffel, there is also the `old` construct in postconditions which is used to check the value of a variable after the execution of a routine. These assertions can be monitored at runtime. Failure of a precondition indicates a bug in the caller while failure of a postcondition indicates a bug in the routine.

Invariants: Class invariants are assertions that should be true for every instance of a class. Every routine must ensure that the class invariant is true upon exit if it was true at entry. Apart from class invariants, there are also loop invariants. A loop invariant is a condition which is true at the beginning of a loop, is preserved throughout all the iterations and also true when the loop condition fails and the loop is exited. A loop invariant is used to prove *partial* correctness of programs with loops, i.e it ensures that *if* the loop terminates, the postconditions will hold. A loop invariant is represented in Hoare Logic as:

$$\frac{\{ Condition \land Invariant \}\ loop\_body\ \{ Invariant \}}{\{ Invariant \}\ while\ (\ Condition\ )\ loop\_body\ \{ \neg Condition \land Invariant \}}$$

However, a loop invariant is not sufficient for proving the total correctness of a loop. In order to ensure total correctness, one must prove that the loop *terminates.* To ensure termination, we need a loop variant. *A loop variant is a non-negative integer which decreases atleast by one in each iteration.*

A key feature of OOP is inheritance. It is important to note how contracts behave with inheritance. For a parent class **A** and a child class **B** inheriting from **A**, it should be guaranteed by the programmer that the preconditions of **A** are not weaker than the preconditions of **B** and the postconditions of **B** are atleast as strong as the postconditions of **A**. In case of class invariants, **B** inherits the invariants of its parent **A**. Thus, if there are some new invariants for **B**, then the final invariant of **B** would be obtained by the logical *AND* operation on the invariants of **B** and

those that it inherited from **A**, thereby making the invariant of the child class stronger than the invariant of the parent class.
Other Languages supporting DbC: Apart from Eiffel, another programming language that supports contracts is SPARK [9], which is based on the ADA language. Several efforts have been made to introduce contracts explicitly into programming languages that did not have contract support before. For instance, Java Modeling Language (JML) is a behavioral specification language for defining contracts in Java [29]. Another technique for using DbC for Java is presented in [8]. This approach proposes two tools, Jtest and Jcontract for static analysis and dynamic verification of contracts respectively. Microsoft Research has developed a framework called Code Contracts [5] for .NET programming. The structure of this framework is quite similar to the classic framework from Eiffel. In [1], a list of other existing contract frameworks developed for Perl, Python, C++ etc. is provided.

2.1.2 FASA Framework

FASA stands for Future Automation System Architecture. It is a component-based framework developed at ABB Corporate Research, used for developing cyclic control applications [34]. FASA is based on the principle of keeping the applications, runtime framework and execution platform independent of each other [44] as shown in Figure 2.1. Due to this separation, it is very flexible in the sense that the applications can be executed on any platform, without requiring the application code to be changed. This allows a clear separation between the development of the applications and their deployment [34]. Application developers are only concerned with the code of the control application. In the background, the FASA framework is responsible for compiling the code, deciding upon a static schedule for the initial deployment of the application and also the best communication protocol to be used among the components which depends on their deployment with respect to each other [34].

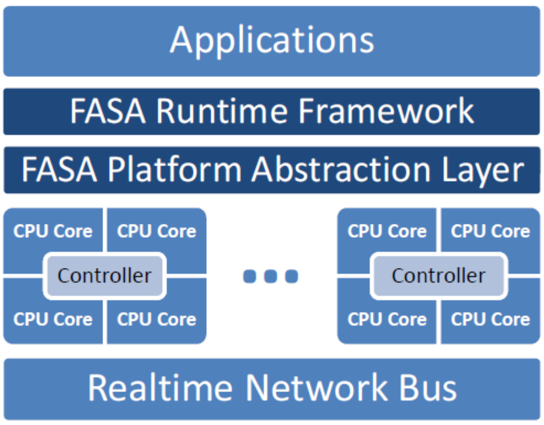


Figure 2.1: FASA Architecture [34]

Main constituents

In FASA, applications are composed of the following:

1. function blocks
2. components
3. ports
4. channels

Components are made up of one or more blocks which are the basic units of FASA. All the blocks enclosed by a component are deployed on the same host controller. Every block has a function, `construct()` meant for the initialization of the block, which is done only once when an application is launched. This includes initialization of variables and declaration of ports. The blocks also have a dedicated routine, `operator()` in which the behaviour is defined by the application developer. This routine is executed in every cycle.

Blocks have input and output ports for data transfer. Data can only be written on the output ports and read from the input ports. The channels are used to connect an input port to an output port. They are stateless and do not monitor the data they transmit. They are unidirectional.

A single FASA application may be executed on more than one host and the communication among the blocks depends on their deployment. If they are on the same core, they communicate by shared memory. If they are on different cores of the same host, they communicate by message passing. Finally, if they are deployed on different hosts, the blocks communicate through network proxies [34].

A typical FASA application

In Figure 2.2, a four block FASA application is shown which demonstrates the concepts of blocks, components, input and output ports and channels. The 4 function blocks are: a *Sensor*, a *Feed Forward block*, a *Current Controller* and a *Monitor*, each of which are enclosed by a component shown in the figure by the surrounding outer rectangles. The order of execution of the blocks is shown by the integers at the bottom of each block. In the actual FASA application, this information is provided in an xml file. The Sensor block collects data from the environment and sends them to the Feed Forward block and the Current Controller through ports **data_out_ff** and **data_out_cc** respectively. After receiving the input from the Sensor, the Feed Forward block performs some computations and sends the output to the Current Controller through port **data_out**. At this point, the Current Controller has the previous values from the Sensor which it received at port **data_in** and a new value from the Feed Forward block which it receives at **data_in_ff**, on which it then performs some computations and sends the obtained output to the Monitor through **data_out**. The Monitor is used for displaying the data on some form of console. This completes one cycle and it is repeated in the same way in each cycle.

File structure of FASA applications

Figure 2.3 shows the file structure of a typical FASA application. As we can see, there are three types of files that are required.

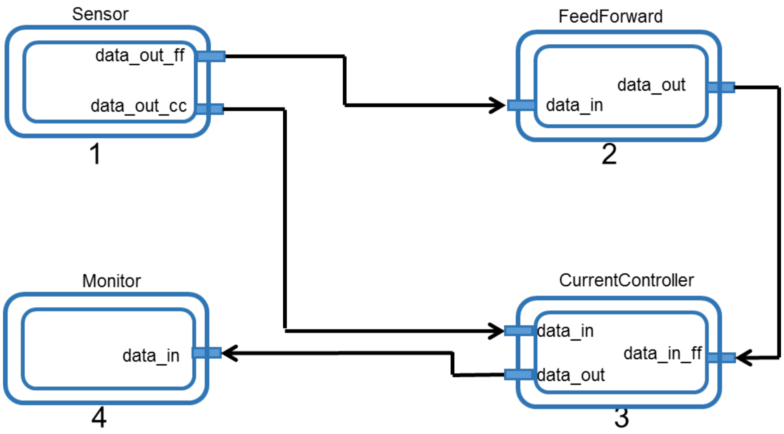


Figure 2.2: A four block FASA application

1. **source files**: The source files are written in C++. All blocks and components are classes that inherit from base classes *Block* and *Component* respectively. The behavior of the blocks are defined in these source files and they are instantiated in their corresponding component classes.

2. **xml files**: For every component, there is an associated xml file specifying the compiled dll file of the application, in which it will be used. This xml file is parsed by the FASA framework to identify the components to be executed. There is one main xml file for storing the description of the entire FASA application. The components that are actually used in an application are listed in this file. The channels are defined here in terms of the source port and the destination port. The other information that is provided in this main xml file is the schedule according to which the blocks are supposed to be executed.

3. **def files**: It contains information for the FASA framework regarding the header files, source files and namespaces used in an application.

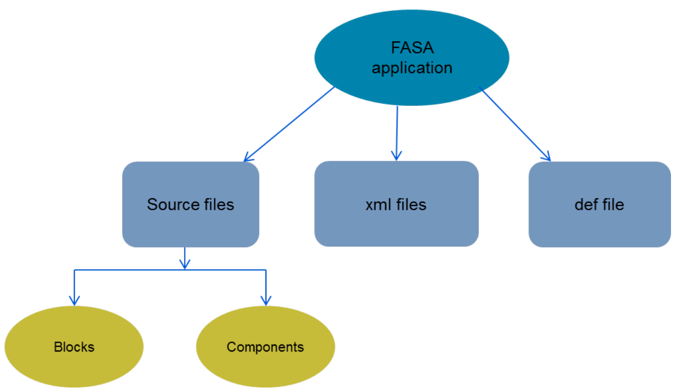


Figure 2.3: File structure of a typical FASA application

Details of the functioning of FASA

The scheduling of FASA applications is done offline. During this phase, the components are allocated to the available hosts and then time intervals are assigned to the blocks for execution. The output of this phase is a static non-preemptive schedule [34]. In order to run a FASA application, a host computer needs to run the FASA kernel which is implemented in C++. The framework has a parser which extracts information regarding the components, channels and the schedule of the FASA application (the order of execution of the function blocks) as described in the xml file. Finally, the application is launched and the blocks are executed by the FASA kernel. FASA applications are cyclic which means that after the execution of all the blocks of an application, the execution of the application is repeated [34]. After the execution of all the blocks in a cycle, there is usually some *slack time* during which blocks could be updated. If no such update activity is necessary, then the kernel sleeps until the start of the next cycle.

2.1.3 Temporal Logic (focusing on Real Time Logic)

Satisfying temporal constraints is vital for hard-real time systems. Missing a deadline for such systems is not acceptable because the consequences could be severe. FASA is used for developing control applications for hard-real time systems. Thus, in order to ensure the correctness of FASA applications, it should be ascertained that they meet the timing requirements. These timing requirements can be represented as a part of the *formal specification* of the system. Temporal logic has long been a popular mathematical tool for formal specification and verification of *safety* and *liveness* properties of reactive systems[31]. A safety property is used to ensure that nothing bad ever happens in a system and a liveness property ensures that eventually something good happens. Properties represented by temporal logic can be verified using techniques such as model checking to prove the correctness of a system. Initially, *classical* logic systems such as *propositional logic* and *first order logic* were used for system specification. However, these logic systems work well when the truth values of the assertions do not depend on time. This is the case for static systems. However, real-time systems are dynamic systems and *time* is an important aspect in this case. This led to developments in the field of *temporal* logic. Temporal logic is basically an extension of the classical logic systems using additional modal operators [27]. Table 2.1 shows the temporal logic symbols and their corresponding meanings.

Table 2.1: Symbols in temporal logic and their interpretations

symbol	meaning
$\bigcirc\phi$	ϕ is true in the next moment of time
$\square\phi$	ϕ is always true
$\diamond\phi$	ϕ is eventually going to be true
φ **until** ϕ	φ is true until ϕ is true
$\phi \trianglelefteq \varphi$	φ is true atleast as long as ϕ is true

Over time, there has been a lot of research on adapting temporal logic to different needs. Bellini et al. [11] have provided a review of the properties of different temporal logic systems. In this thesis, we are going to focus on temporal contracts for real-time systems. For our purpose, we have exploited Real Time Logic (RTL) [24]. RTL is an extension of first-order logic dedicated to specification of real time systems. Strictly speaking, despite what the name suggests, RTL is not a typical temporal logic system because it does not have the modal operators listed above [11]. One of the main advantages of RTL is that it allows us to reason about both absolute and relative time [25]. With the help of an *occurrence function*, @, RTL facilitates capturing of the time of occurrence of some event. In the contracts that we defined in this thesis, we needed to handle both these cases which is why RTL has been chosen for our specifications. Below, we provide an overview of the syntax of RTL.

RTL syntax

In Real-Time Logic defined by *Jahanian* & *Mok* [25], the notations defined for formal specification of a system are as follows:

Events: In RTL, an *event* is a temporal marker that describes the real-time behavior of a system [24]. An event is different from an *action*. An action requires system resources [25], while an event only gives us information regarding the time of occurence of an action. For every action, there are two associated events, a *start* event and a *stop* event. Following is a description of the different types of events in RTL.

1 Execution of an action is represented by two events, *start* and *stop*. For an action A, $\uparrow A$ and $\downarrow A$ respectively represent these two events.

2 Transition Events: These event occurs when the value of a state changes.

3 Ω EVENT_NAME: Events that impact the system behaviour but cannot be made to happen from within the system. Such events are called *external* events.

R: Occurrence relation R(*E,i,t*) is used to represent that the i^{th} occurrence of an event *E* takes place at time *t*, where, *i* is an integer such that $i > 0$ and t is an integer such that $t \geq 0$. Here time is considered to be a discrete quantity.

@: Occurrence function @(E,i) represents the time of the i^{th} occurrence of event *E*.

2.2 Analysis of previous research

Contracts have often been used in the past for specifying functional requirements of programs. Not only is it popular practice to use contracts in application software [46], but even in industrial software, the use of contracts has gained a lot of popularity because it allows both dynamic and static verification of the code. For example, DbC is incorporated in Ada 2012 [39] for functional specifications of real-time applications [15].

As described in [18], DbC is used in Component Based Software Engineering (CBSE) for describing the behavior of the components. From this point of view, contracts can be considered as a specification technique used in the *Design level* of complex systems.

In [41], [19], [38], [10], [12] and [43], the use of DbC in the system design phase is shown, focusing on the real-time aspects of the underlying systems.

Real-time contracts not only refer to temporal contracts but also to non-temporal contracts such as resource consumption related constraints [38], [43], [19], [12], [41].

Contracts using interface description languages (IDL):
Härtig et al. [19] and Barbacci et al. [10] show the use of interface description languages for specifying contracts.

In [19], contracts are specified for a real-time system in an OCL [7] based language called Extended Component Quality Modeling Language ($CQML^+$) and a dedicated runtime environment is designed to execute an application defined in $CQML^+$. This runtime environment is responsible for converting the component-specifications into executable task specifications for a real-time operating system (RTOS).

Barbacci et al. [10], represent the functional and temporal properties using two separate formalisms. The functional specifications are written in *Larch Interface Language (LIL)* [45] and the temporal requirements are written in an event expression language [10].

While in [10], the focus is on functional and temporal contracts, in [19], other non-functional aspects such as allocation of resources like CPU are also taken into account while defining the contracts.

The above approaches use a separate language for the specifications which would involve a lot of overhead due to parsing and interpreting the contracts. *Since this thesis aims at very small cycle times and dynamic verification of contracts, the approaches in [10] and [19] are not suitable for our need.*

Contracts for embedded systems:
Stierand et al. [43] show the use of interfaces and contracts for distributed embedded system design. In this paper, focus is mainly on the scheduling aspects. They have proposed a technique for designing an interface for a real-time system for which the scheduling policy and the contracts are *given*. They have used finite state machines for modeling the system and then they check whether this state machine satisfies the contracts.

Contrary to their objective, in this thesis, the contracts are not given and the objective is to *define* the contracts *themselves*.

Layered real-time contracts:
Sojka et al. [41], Benveniste et al. [12] and Sangiovanni et al. [38], have presented contracts in multiple layers of real-time systems.

Benveniste et al. [12] and Sangiovanni et al. [38] have used contracts for platform-based systems. The design of a complex system is classified into two orthogonal directions. The vertical direction relates to different levels in the design hierarchy, such as application level and platform level. The horizontal direction refers to various components which interact with each other in the *same* vertical level. In other words, platform-based design is an amalgamation of model-based (vertical) and component-based (horizontal) system design. Contracts in this setting are therefore classified as horizontal and vertical depending on the nature of the *assume-guarantee* pairs they represent. Contracts in the same horizontal level can undergo conjunction operation while contracts in the vertical layers undergo refinement [12], [38].

Our contract framework handles two levels of the contracts, block level and scheduler level. The details are given in chapter 3.

The layered contract framework of Sojka et al. [41] is a part of the FRESCOR project [6]. The two layers are *generic* and *resource specific*. The generic layer has an agent called a *broker* which acts as a mediator between the actual resources and the applications. The broker runs on every node in case of a distributed scenario.

If we try to model this approach according to our requirements, it would be equivalent to having an extra function block acting as the mediator. As we will see in chapter 4, having additional function blocks adds an overhead to the execution time. Moreover, the FASA framework is IEC 61131 compliant and it is a time-triggered system. The authors in [41] have mentioned that their framework has not yet been tested for time-triggered systems.

Use of RTL for real-time system specifications:
Barbacci et al. [10] and Jahanian et al. [25] illustrate the use of RTL for specifying properties of real-time systems. In this thesis as well, RTL has been used to formally define the real-time contracts.

What makes our approach distinct and unique is the use of RTL for specifying stochastic contracts, which has not be done before.

To conclude, contracts have been used for a wide range of applications, including several real-time systems. However, to the best of our knowledge, they have *not* been used for dynamically computing real-time contracts based on statistical estimates of parameters of the probability distributions of the execution times. Further, this thesis aims at performing these computations at runtime while keeping the overhead on the execution time as little as possible.

To summarize this section, table 2.2 shows the characteristic features of the existing contract frameworks which are relevant for our research and the characteristics of our own contract framework.

Table 2.2: Characteristics of different contract frameworks

	functional contracts	temporal contracts	other non-funtional contracts	stochastic analysis	OCL based contracts	RTL specs for formalization
AdaCore's framework [15]	✓	×	×	×	×	×
Barbacci et al. [10]	✓	✓	×	×	✓	✓
Härtig et al. [19]	✓	✓	✓	×	✓	×
Stierand et al. [43]	×	✓	✓	×	×	×
Sangiovanni et al. [38]	✓	×	✓	×	×	×
Benveniste et al. [12]	✓	✓	✓	×	×	×
Sojka et al. [41]	×	×	✓	×	×	×
FASA contract framework	✓	✓	×	✓	×	✓

As the table shows, our framework focuses on functional and temporal contracts only. It illustrates the use of statistical inference for computing the temporal contracts dynamically. The framework does not rely on any IDL in order to avoid additional overhead due to parsing and interpreting the contracts. Our framework has been formalized using RTL thereby highlighting its use for specification of statistical properties.

2.3 Preliminary experiments and problem exploration

Stochastic contracts: One of the key contributions of this thesis is the computation of stochastic temporal contracts for the function blocks.

In order to do this, we use a novel approach of *empirical cumulative distribution function (cdf)* of the execution times of the function blocks. The analysis is based on samples collected by executing the blocks for n cycles. Chapter 3 provides the details of this technique.

Here, we describe the preliminary experiments whose results motivated us to adopt this approach.

Although in many settings, the execution time of a program can be observed to have a well known probability distribution, such as the Pearson group of distributions [37], predicting this distribution is a research area on its own. Things would become much simpler if one could argue that for large number of executions, the probability distribution can be assumed to converge to a **Gaussian** distribution. This assumption is often far from the truth, thereby rendering statistical analysis of the execution times of programs a very challenging field. An example of such a scenario is presented in [32].

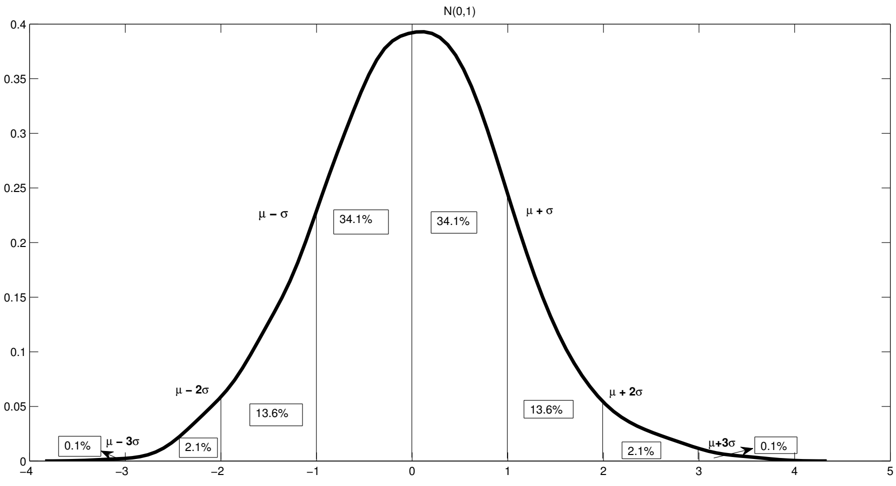


Figure 2.4: Standard Normal Distribution

Analysis of execution times can broadly be classified into *static*, *probabilistic* and *hybrid* approaches [35]. A lot of research has been done in the field of static analysis. However, the drawback of static analysis is that it requires knowledge of the code. This is not always feasible when systems are designed using CBSE. Components are meant to be used *off the shelf* and we cannot always expect to have access to the code of the components [35]. FASA is based on the idea of CBSE. Therefore, although static analysis is a solution for analyzing the WCET of the FASA function blocks, it is not always a plausible approach.

In this thesis, we consider two scenarios.

In the first scenario, we assume that WCET analysis for the function blocks has been done by some external static analysis tool and that we know the WCET values of the blocks. A part of the contract framework works on this assumption as we will see in Chapter 3.

In the second scenario, we assume that there is no information about the WCETs of the function blocks. In this setting , we use *statistical approaches* to compute the temporal contracts based on estimated upper bounds on the execution times of the function blocks.

For a Gaussian distribution, $N(\mu,\sigma^2)$, according to the **3-sigma rule**, 99.7% of the data drawn lies within $\mu \pm 3\sigma$, which can be seen in Figure 2.4. Now, to consider this as an upper bound for the execution times of the function blocks, the probability distribution of the execution times should conform to a Gaussian distribution. The distribution of the execution times depends on the nature of the computations being done within the blocks and it was observed that upon running the function blocks for 1000 cycles or more, although the distributions converge to a single peaked distribution, it is far from Gaussian.

We conducted the experiments for 11 FASA applications and in total 24 function blocks.

The results for one of the applications (the four block FASA application in Figure 2.2) is shown in Figure 2.5. This application is based on a Simulink case study. As it can be seen from the

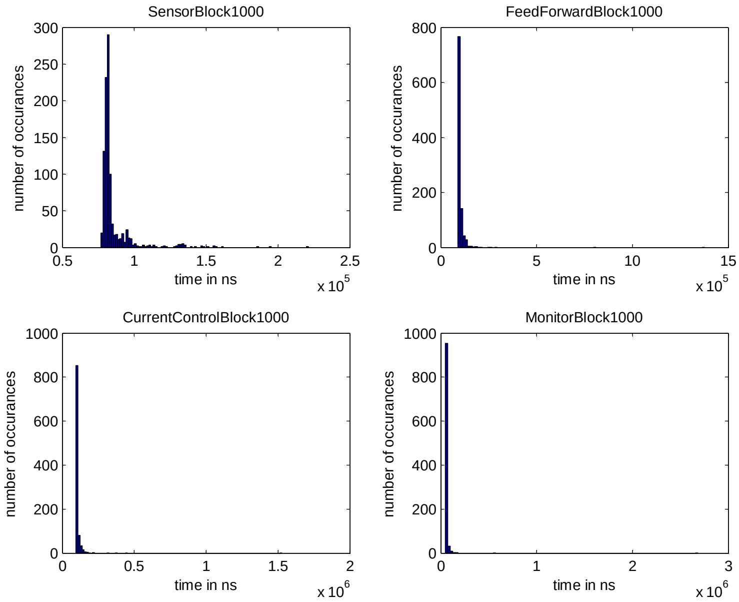


Figure 2.5: Block execution times of a FASA application with four function blocks. This application is based on a case study obtained from Simulink

histograms, the distributions are not Gaussian even for n as large as 1000. Thus, we could not use an estimated value of $\mu + 3\sigma$ as an upper bound on the execution times. This led us to adopt an approach based on the *empirical cdf* of the execution times of the blocks. We describe the approach in detail in chapter 3.

2.4 Conclusions

This chapter gave an overview of the underlying concepts used in the development of the contract framework. It presented a rigorous analysis of previous work on real-time contracts.

Finally, the chapter illustrated a very important experiment that we conducted showing that for a component-based platform such as FASA, it is **non-trivial** to estimate the probability distribution of the execution times of the function blocks. Known distributions can rarely be used to fit the execution times.

Therefore, an *empirical cdf*-based approach is used in the rest of the research. This approach subsumes the properties of the function blocks and as a result, it is a generic approach suitable for estimating the cdf of function blocks with varied behaviors.

3 Development of the Framework

FASA is a platform for developing real-time control applications. When we talk about contracts for such a platform, there are two important aspects to be considered - *functional* and *real-time*. By functional, we refer to the behavior of a function block. Functional contracts are thus related to properties such as variables, their values and relationships with other variables. For our research, real-time contracts refer to the temporal properties of a function block and also an application as a whole. This chapter first states the requirements of the contract framework, describes its features along with the underlying development principles and finally formalizes it using RTL.

3.1 Requirements of the FASA contract framework

The first step in the development of any software is the analysis of its requirements. For the contract framework developed in this thesis, there were *four* main requirements, which are given in tables 3.1, 3.2, 3.3 and 3.4.

Table 3.1: Requirement 1

Req1	Flexibility of the contract framework: it should not have any effect on the original FASA platform and the user should have the freedom to turn the contracts on/off as and when required.
Rationale	Contracts should preferably only be used in the debug mode and disabled in the production code in order to avoid any overhead. Therefore the framework should allow the user to enable or disable contracts according to the mode in which the applications are compiled.
Test	A *dummy* contract designed to fail, will be put in the base `Block` class. Applications should be executed with contracts enabled and disabled. In the latter case, the application should run as if no changes have been made to FASA. In the former case, a contract failure message should be logged in the slack time.
Classification	Required.

Table 3.2: Requirement 2

Req2	Allowing users to specify functional contracts at the function-block level.
Rationale	Contracts are required for ensuring the correct functionality of the blocks. The framework should provide the user a set of tools for defining functional contracts.
Test	*Dummy* pre and post conditions, class invariants and variable checks using the `Old` construct will be inserted in the applications. The dummies are designed to fail. The contract framework should be able to log the failure messages during the slack time when contracts are enabled.
Classification	Required.

Table 3.3: Requirement 3

Req3	Allowing the users to specify temporal contracts.
Rationale	Temporal correctness is vital for hard real-time, safety critical systems. The contract framework should allow defining of temporal contracts to ensure that the blocks meet the temporal requirements.
Test	*Dummy* temporal contracts will be inserted in the applications which are designed to fail. As an example, setting the WCET of a block to zero and checking the execution time of a block against it will always fail because execution time cannot be zero (assuming that the block gets started). The contract framework should be able to log failure messages corresponding to the contracts during the slack time when contracts are enabled.
Classification	Required.

Table 3.4: Requirement 4

Req4	Have minimum effect on the performance of the FASA platform, i.e limit the overhead due to contracts to the bare minimum.
Rationale	Since FASA is used for developing control applications, performance of the applications is a great concern. The contract framework should take this into account and ensure that the overhead is not more than 10% [23](section 3.1.1).
Test	Overhead should be measured when the contract framework is activated. The percentage increase in the execution time of the applications should not exceed 10.
Classification	Required.

3.1.1 Deciding the acceptable overhead level

In control systems, it is a common practice to fix a tolerance limit on the execution time overhead. This is necessary because control applications are hard real-time systems and missing of a deadline is considered as a failure. For the development of the contract framework, we had to decide how much overhead due to contracts would be considered as acceptable.

Huang et al. [23] have described a supervisory feedback control mechanism called SMCO to limit the overhead due to monitoring tools. The user specifies a *target* overhead which should not be exceeded by the monitoring software. A feedback mechanism is used to ensure that this bound is respected. In the experiments conducted by Huang et al. [23], the target overhead has been taken to be 10% and it is shown that a very high accuracy (in terms of the number of events monitored) can be attained by the monitoring services even with such a small target overhead. For our contract framework, we have fixed 10% as the maximum tolerance in the overhead.

3.2 Features and underlying principles

The contract framework allows the application developer to specify the contracts while defining the behavior of the function blocks in C++. It is important to note that the FASA platform does not require the application developer to interfere with the scheduling or the deployment of the application. In this sense, the contracts are *local* to the function blocks and do not have any effect on the platform abstraction layer. At the same time, contracts at the scheduler level are important for ensuring the temporal correctness of the applications. To tackle this situation, some contracts are made *implicit* in the FASA platform. They are checked by default whenever contracts are enabled. This is explained in more detail in section 3.2.2.

The objective behind the contract framework is to provide the user a set of tools for specifying the contracts according to the requirement of a particular application and then to dynamically verify whether the application satisfies the contracts in every cycle.

If a contract is not satisfied in a cycle, a message is logged and later the cause of the failure can be inspected. In the production code, the contract framework is disabled by default.

3.2.1 Functional Contracts

The FASA contract framework allows the users to define preconditions, postconditions, class invariants, loop invariants and loop variants for a function block. This feature satisfies **Req2**.

Preconditions are checked before the `operator()` method in the function block inside a routine dedicated to functional preconditions. An example of a functional precondition for a function block which has an output port for data transfer would be to ensure that the output port is connected to the input port of the destination function block.

Postconditions are similarly specified in a dedicated routine for functional postconditions and checked after the `operator()` method. An example of a functional postcondition for a function block that computes the cosine of an angle would be to ensure that the result lies in $[-1, +1]$.

Invariants are checked both before and after the `operator()` method. If the operator method has loops within it, one can also specify loop invariants and loop variants for checking the correctness of the loops. The contract framework will then dynamically check if they are satisfied in every cycle.

Contracts can also be defined for monitoring the value of a data member of a function block. This feature is inspired by the **old** construct of Eiffel.

3.2.2 Real-time (temporal) Contracts

Apart from functional properties, real time applications also have temporal properties which must be satisfied. The FASA contract framework ensures that such temporal contracts are satisfied by a FASA application, in order to fulfill **Req3**. Three real time aspects have been considered in this research:

1. WCET related properties
2. Cycle time related properties
3. Jitter related properties

Three *fundamental* temporal properties are checked by default whenever contract checking is enabled, without requiring the user to specify them explicitly. They are as follows:

1. WCET: Each function block has a WCET (worst case execution time) value. For the correct functioning of the application, it is necessary for each function block to meet the WCET requirement in each cycle. This can be verified at runtime using an implicit contract that is embedded in the FASA framework. For this, it is assumed that the WCET values of the function blocks are provided by an external static analysis tool. For the purpose of validating the framework, these values are assumed to be provided by the application developer. *This is a block level contract.*
2. CYCLE_TIME: The cycle time for FASA is a parameter that is predefined in a *configuration file*. A contract is defined that ensures that the total execution time of all the function blocks strictly respects this upper bound. *This is a scheduler level contract.*
3. JITTER_MARGIN: The contract framework allows the user to check the jitter margin. Jitter margin is a value that determines the maximum deviation from the required cycle time that is acceptable. At runtime, the user can specify an upper bound on the jitter that can be tolerated. The contract framework then dynamically verifies whether this bound is respected in every cycle. *This is a scheduler level contract.*

Apart from the above *fundamental* temporal contracts, the framework also allows the user to perform more complex real-time contract verifications. Following is a description of these properties.

Online estimation based

The contract framework allows the user to perform a dynamic estimation of an upper bound on the execution time of a block. Following are the rationales behind this approach:

- In situations when the WCET values of the function blocks are not known in advance (by means of static analysis or other techniques) the user has to compute the temporal requirements on the fly using statistical techniques. This is often the case for CBSE because components are meant to be used *off the shelf* and access to the component code is not always feasible. In such settings, statistical inferencing is a very useful technique for determining upper bounds on the execution times.

- Since contracts are dynamically verified, it is important to make sure that the current platform conditions are taken into account while computing the contracts. For instance, the execution time of the applications depends on the resources like memory and availability of the CPU. If the contracts are never updated, then they may become insignificant with time because the conditions under which they were originally defined may no longer be valid.
- The execution time of a function block also depends on the path followed by the execution. For instance, if in a certain cycle, there is an exception that has to be handled, it might take more time to execute the block as compared to the normal cycles. Online updating of the upper bound ensures that eventually the execution times for all such possible paths are subsumed.
- Additionally, when an application is started for the first time, the execution times are in a transient phase. With execution, they tend to become more stable. The present approach makes sure that this fact is not neglected while computing the contracts.

There are two possible scenarios to be considered here.

case 1 The probability distribution of the execution times is known and the population parameters such as mean and variance are also known.

case 2 The probability distribution of the execution times is not known and the population parameters are unknown as well.

Let us consider **case 2** first. In order to compute an estimate of the upper bound, first of all we need to estimate the population parameters. Here we only consider the first two moments of the probability distribution, *mean* and *standard deviation*, because computations are done dynamically while executing the applications and computing the higher order moments would degrade the performance of our framework. This approach respects **Req4**.

In order to compute the estimates, we need to use sample data. Samples are generated by running the function blocks for an $experimental\_cycle\_number$ number of times. This value is hard coded in the framework. For our experiments, this value is taken as 1000. After the block is executed $experimental\_cycle\_number$ times, the framework computes the sample mean, $\widehat{\mu}_n$ and sample variance, s_n^2 of the execution times and estimates the upper bound using statistical inference, where $n = experimental\_cycle\_number$. This upper bound is used for checking temporal contracts from the $(experimental\_cycle\_number + 1)^{th}$ cycle onward. It is shown below that the sample statistics, $\widehat{\mu}_n$ and s_n^2 are unbiased estimators of the population mean and variance respectively and thus, they can be used for estimating the upper bounds.

***Unbiased Estimator*:** Let $e_1, e_2, e_3 ... e_n$ be a random sample drawn from a population and let θ be an unknown parameter of the population which is to be estimated. Let $\widehat{\theta} = v(e_1, e_2, e_3 ... e_n)$ be a statistic (function of a random sample) based on the sample. By definition, $\widehat{\theta}$ is an *unbaised estimator* of θ if the expected value of $\widehat{\theta}$ is equal to θ.

$$E[\widehat{\theta}] = \theta \tag{3.1}$$

Let the execution time of a function block B be represented by e. Let the experimental cycle number be represented by n. Let $\widehat{\mu}_n$ and s_n^2 be statistics based on our sample, $e_1, e_2 ... e_n$ as defined in equations 3.2 and 3.3 respectively. Let the population mean of the probability distribution of the execution time be μ and the population variance be σ^2.

$$\widehat{\mu}_n = \frac{\sum_{i=1}^{n} e_i}{n} \tag{3.2}$$

$$s_n^2 = \frac{\sum_{i=1}^{n} (e_i - \widehat{\mu}_n)^2}{n-1} \tag{3.3}$$

Then,

$$E[\widehat{\mu}_n] = E\left[\frac{\sum_{i=1}^{n} e_i}{n}\right] \tag{3.4}$$

$$= \frac{\sum_{i=1}^{n} E[e_i]}{n} \tag{3.5}$$

$$= \frac{\sum_{i=1}^{n} \mu}{n} \tag{3.6}$$

$$= \mu \tag{3.7}$$

This result along with equation 3.1 implies that the sample mean is an unbiased estimator of the population mean. Further, let the sample variance be defined as:

$$\widehat{\sigma}_n^2 = \frac{\sum_{i=1}^{n} (e_i - \widehat{\mu}_n)^2}{n} \tag{3.8}$$

Then,

$$E[\widehat{\sigma}_n^2] = E\left[\frac{\sum_{i=1}^{n} (e_i - \widehat{\mu}_n)^2}{n}\right] \tag{3.9}$$

$$= \frac{E[\sum_{i=1}^{n} (e_i - \widehat{\mu}_n)^2]}{n} \tag{3.10}$$

$$= \frac{E[\sum_{i=1}^{n} {e_i}^2 + n\widehat{\mu}_n^2 - 2\widehat{\mu}_n \sum_{i=1}^{n} e_i]}{n} \tag{3.11}$$

$$= E\left[\frac{\sum_{i=1}^{n} e_i^2}{n} - \widehat{\mu}_n^2\right] \tag{3.12}$$

$$= \frac{\sum_{i=1}^{n} E[e_i^2]}{n} - E[\widehat{\mu}_n^2] \tag{3.13}$$

Now, we know that the population variance is defined as:

$$Var(e) = \sigma^2 = E[e_i^2] - [E[e_i]]^2 \tag{3.14}$$

$$\Rightarrow \sigma^2 = E[e_i^2] - \mu^2 \tag{3.15}$$

$$\Rightarrow E[e_i^2] = \sigma^2 + \mu^2 \tag{3.16}$$

and,

$$Var(\widehat{\mu}_n) = Var\left(\frac{\sum_{i=1}^{n} e_i}{n}\right) = \frac{\sigma^2}{n} \tag{3.17}$$

$$= E[\widehat{\mu}_n^2] - \mu^2 \because Equation\, 3.7 \Rightarrow [E[\widehat{\mu}_n]]^2 = \mu^2 \tag{3.18}$$

$$\Rightarrow E[\widehat{\mu}_n^2] = \mu^2 + \frac{\sigma^2}{n} \tag{3.19}$$

Using equations 3.16 and 3.19 in 3.13, we get:

$$E[\widehat{\sigma}_n^2] = \frac{n\sigma^2 + n\mu^2}{n} - \frac{\sigma^2}{n} - \mu^2 \tag{3.20}$$

$$\Rightarrow E[\widehat{\sigma}_n^2] = \frac{(n-1)\sigma^2}{n} \tag{3.21}$$

Thus, we see that the sample variance is not an unbiased estimator of the population variance. However,

$$E\left[\frac{n\widehat{\sigma}_n^2}{n-1}\right] = E[s_n^2] = \sigma^2 \tag{3.22}$$

Thus, s_n^2 is an unbiased estimator of the population variance.

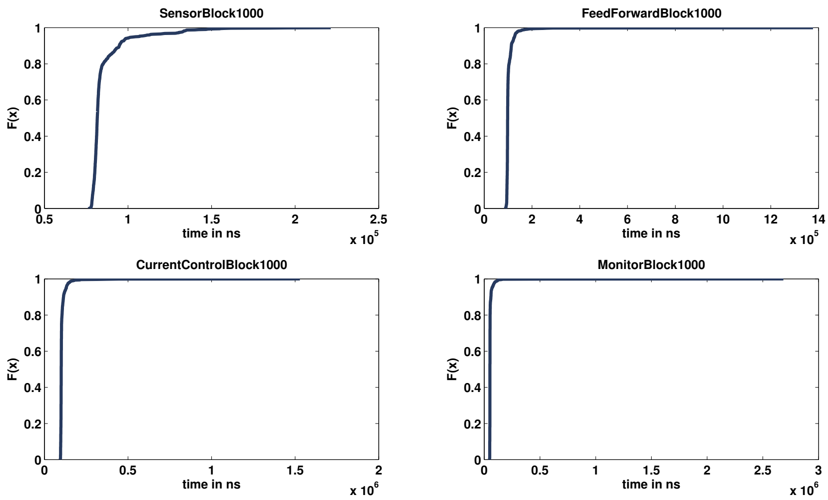


Figure 3.1: Empirical cumulative distribution functions of four function blocks of an application, based on sample size of 1000

Estimating the bound: For our sample $e_1, e_2, ...e_n$, let $f(x)$ denote the probability density function and $F(x)$ denote the cumulative distribution function(cdf). By definition, cdf of a random variable X is given by:

$$F(x) = P[X \leqslant x] \tag{3.23}$$

For our case, we can determine the *empirical* cdf of the execution times of the function blocks from the sample data obtained from $experimental\_cycle\_number$ cycles as shown in Algorithm 1. It classifies the data into bins and computes the cumulative sum of the number of items in each bin as shown in line 18. The number of bins is chosen using the *square root rule*, accoring to which, it is equal to the square root of the total number of data. Algorithm 1 describes the computation of the empirical cdf after the first $experimental\_cycle\_number$ cycles. Its also shows how the upper bound is estimated using equations 3.23.

Figure 3.1 shows the empirical cdfs of four function blocks of the application shown in figure 2.2. Now, using equation 3.23, we can find out the probability, $0 \leq \pi \leq 1$ of the execution time to be less than a certain value, say τ_π. If the value of π is given as the desired threshold probability value, then τ_π can be used as the estimation of the upper bound for computing contracts on the execution times. The value of π is defined by the user at runtime.

Further, the value of τ_π can also be used to compute a bound of the nature, $\mu + \gamma\sigma$, where,

$$\gamma = \frac{\tau_\pi - \mu}{\sigma} \tag{3.24}$$

Equation 3.24 simply tells us that the execution times must lie within γ standard deviations of the mean, where the value of γ is specific to every function block.

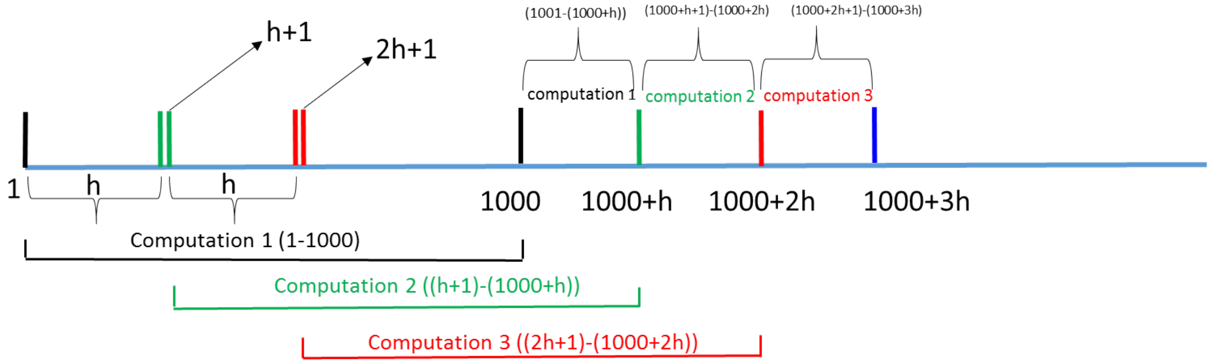


Figure 3.2: This figure shows the sliding window principle for dynamically updating the estimates of the parameters, μ and σ. The term h in the figure denotes $sliding\_window\_update\_interval$ and $experimental\_cycle\_number = 1000$.

Going back to **case 1** in item 3.2.2, if the *population* mean and variance of the execution times of the function blocks are known in advance, then the contract framework can test that the execution times of the function blocks do not exceed an upper bound that is computed the same way as described above. In this case, the estimation of the μ and σ is skipped.

1. SLIDING WINDOW BASED UPDATE: This technique is used to update the upper bound by using a sliding window principle. It requires the user to specify the value of a parameter, $h = sliding\_window\_update\_interval$, at runtime. This value tells the framework how often the parameter estimates have to be updated. First of all, the execution times of the function block are stored in a buffer until $experimental\_cycle\_number$ is reached. Then, the upper bound for the execution time is computed using equations 3.23 and 3.24. Depending on the value of $sliding\_window\_update\_interval$, this bound is used for specifying the contracts for $sliding\_window\_update\_interval$ cycles starting from the $(experimental\_cycle\_number + 1)^{th}$ cycle. When the cycle value is $(experimental\_cycle\_number + sliding\_window\_update\_interval)^{th}$, the framework updates the parameters taking the most recent values of the execution times. Let us take an example of $h = 5$. First of all, the values from cycle number 1 to 1000 are taken to estimate the parameters. These estimates are used for specifying the contracts from cycle number 1001 to 1005. Then, at the 1005^{th} cycle, the first 5 values of the execution times are removed from the buffer and the estimates are computed again, using values from cycle number 6 to 1005. These estimates are then used for the contracts from cycle 1006 to 1010 and the process continues. This is illustrated in Figure 3.2.

2. CONTINUOUS UPDATE: Another feature of the contract framework is that it can recompute and update this upper bound by taking into account the most recent value of the execution time in the computation of the mean and standard deviation, *without* removing the oldest value. Figure 3.3 illustrates this. In order to make the update efficient, the following mathematical results are used in the algorithm :

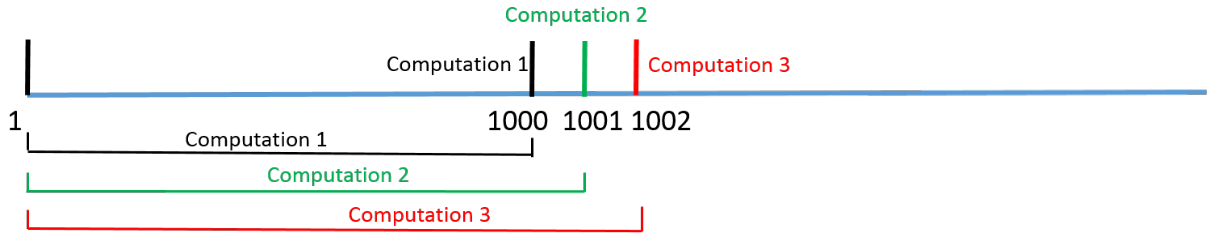


Figure 3.3: This figure shows the continuous update principle for dynamically updating the estimates of the parameters, μ and σ. The recomputation of the values is done in every cycle, starting from the $(experimental\_cycle\_number+1)^{th}$ cycle, where $experimental\_cycle\_number = 1000$.

From equation 3.2, we can say that,

$$\widehat{\mu}_{n+1} = \frac{\sum_{i=1}^{n+1} e_i}{n+1} \tag{3.25}$$

using (3.2) in (3.25),

$$\widehat{\mu}_{n+1} = \frac{n\widehat{\mu}_n + e_{n+1}}{n+1} \tag{3.26}$$

From equation 3.3, it follows that,

$$s_n^2 = \frac{\sum_{i=1}^{n} e_i^2 - n\widehat{\mu}_n}{n-1} \tag{3.27}$$

or,

$$s_n^2 = \frac{\sum_{i=1}^{n} e_i^2}{n-1} - \frac{n\widehat{\mu}_n}{n-1} \tag{3.28}$$

Similarly,

$$s_{n+1}^2 = \frac{\sum_{i=1}^{n+1} e_i^2}{n} - \frac{(n+1)\widehat{\mu}_{n+1}^2}{n} \tag{3.29}$$

Let $S_n^2 = \sum_{i=1}^{n} e_i^2$

Then, from (3.29), we get

$$s_{n+1}^2 = \frac{S_n^2 + e_{n+1}^2}{n} - \frac{(n+1)\widehat{\mu}_{n+1}^2}{n} \tag{3.30}$$

Equations (3.26) and (3.30) are used for updating the values of $\widehat{\mu}$ and $\widehat{\sigma}$. They represent the updated parameters in the $(n+1)^{th}$ cycle as functions of the parameters in the n^{th} cycle and thus recomputing the values from scratch can be avoided. This makes the algorithm more efficient.

Further, in the continuous update mechanism, after the first n cycles, it is no more necessary to store the execution times in the buffer because the only information required for continuously updating the parameter estimates are their estimates in the immediately preceding cycle and the current execution time value. This approach is more memory efficient compared to the sliding window technique where the most recent n execution times *have* to be stored.

3. CONSECUTIVE CYCLE BASED: The contract framework allows one to monitor the execution time of a block for l consecutive cycles, where l is specified by the user at runtime. This contract is important because if for $l > 1$ consecutive cycles, the function block keeps exceeding the WCET, then it means that there is a fault in the system and it needs to be investigated. Exceeding the WCET in one random cycle could happen by chance or due to platform related problems and it is not a strong enough evidence of a fault in the application code.

Parametric real-time contracts

An important aspect addressed in this thesis is to enable the FASA application developer to specify temporal contracts as functions of the properties of various data-structures used in the function blocks (for example the size of an array). We term this feature as *Parametric Real-Time Contracts*. Section 4.3 in chapter 4 highlights the details of this feature. Madhavan and Kuncak [30] illustrate the computation of bounds on execution times of functional programs. In their work, the user is allowed to define templated expressions to define the bounds. These templates contain unknowns which are then solved for. However, in this thesis, the idea of parametric contracts is different from the research in [30]. In our case, we do *not* solve for the unknown parameters and it is upto the user to determine these values externally. In fact, one way for the user to find out the values of the unknowns would be to use the algorithm presented in [30].

The implementation details for entire contract framework in described in chapter 4.

3.3 Temporal specification using RTL

This section uses RTL for formalizing the temporal part of our contract framework. As stated in chapter 2, RTL is a useful mathematical tool for real-time system specification. What makes our approach unique is that we illustrate the use of RTL for formalizing statistical temporal contracts.

Let $B_1, B_2, B_3 ... B_n$ be n blocks in a *cyclic* FASA application, $\mathscr{A}$, which are scheduled to be executed in the order: $B_1 \rightarrow B_2 \rightarrow ... \rightarrow B_n$ by a static non-pre-emptive schedule.
The j^{th} block is thus represented as B_j and the WCET of the j^{th} block is represented as C_j. Let the cycle time of an application be represented as P. Also, let the execution time of block B_j in cycle k be represented as $e_{j,k}$. Let the jitter margin be represented as J.

3.3.1 FASA Temporal Requirements

Based on the terminology of Real-Time Logic (RTL) given in 2.1.3, any FASA application should respect the following three specifications:

R1 The start of the kernel must be followed by the execution of the first block of the schedule: The following RTL formula states that for every i^{th} launch of the kernel at time $t \geq 0$, i.e. $\Omega KERNEL$, there exists a time $t^{'} \geq t$ such that the first block B_1 will start at $t^{'}$. Here, the launch of the kernel is treated as an *external* event in RTL terminology.

$\forall i > 0, \forall t \geq 0, R(\Omega KERNEL, i, t) \rightarrow [\exists t^{'} \mid R(\uparrow B_1, i, t^{'}) \wedge t^{'} > t]$

R2 Cyclic behaviour of the applications: This formula states that the function blocks must respect the cyclic behavior expected from all FASA applications. It says that the i^{th} termination of block B_n in cycle k at time$t \geq 0$ must be followed by the start of block B_1 in the $(k+1)^{th}$ cycle at time $t^{'} > t \geq 0$.

$\forall i > 0, \forall t \geq 0, \forall k > 0, R(\downarrow B_{n,k}, i, t) \rightarrow [\exists t^{'} \mid R(\uparrow B_{1,k+1}, i+1, t^{'}) \wedge t^{'} > t]$

R3 Respecting the schedule: The formula states that the function blocks must respect the order of execution, $B_1 \rightarrow B_2 \rightarrow ... \rightarrow B_n$ provided in the schedule of the application's xml file.

$\forall 1 \leq j \leq n, \forall i > 0, \forall t \geq 0, \forall k > 0, R(\downarrow B_{j,k}, i, t) \rightarrow [\exists t^{'} \mid R(\uparrow B_{j+1,k}, i, t^{'}) \wedge t^{'} > t] \wedge [\nexists m \in [1, n] \mid m \neq j+1 \wedge R(\uparrow B_{m,k}, i, t^{''}) \wedge t < t^{''} < t^{'}]$

3.3.2 FASA Real-Time Contracts

This section defines the temporal contracts for FASA using RTL.

C1 The start of the execution of block B_j should eventually be followed by the block's termination and respect the WCET constraint: The following RTL formula states that for all blocks $B_1, B_2, ..., B_n$, the i^{th} *start* event of a block's execution at time $t \geq 0$ in cycle k, i.e. $\uparrow B_{j,k}$ must be followed by the i^{th} *stop* event, i.e. $\downarrow B_{j,k}$ of its execution at time $t^{'} > t$ and the duration of execution of the block, $(t^{'} - t)$ must not exceed the WCET, C_j of the block.

$\forall 1 \leq j \leq n, \forall i > 0, \forall t \geq 0, \forall k \geq 1, R(\uparrow B_{j,k}, i, t) \rightarrow [\exists t^{'} \mid R(\downarrow B_{j,k}, i, t^{'}) \wedge (t^{'} - t) \leq C_j \wedge t^{'} > t$

C2 A block's execution should not reach or exceed the WCET in l consecutive cycles, where $l \geq 1$ is decided by the user at run-time:

$\forall k \geq 1, \forall 1 \leq j \leq n, \neg(e_{j,k} \geq C_j \wedge e_{j,k+1} \geq C_j \wedge \ldots \wedge e_{j,k+l-1} \geq C_j)$

C3 If τ_π is the estimated upper bound on the execution time of block B_j, computed from the empirical cdf obtained over 1000 cycles, then starting from the 1001^{th} cycle, the execution time of B_j should always lie within τ_π.

$\forall k > 1000, \forall 1 \leq j \leq n, e_{j,k} \leq \tau_\pi$

C4 This is a corollary of C3. If μ_j and σ_j^2 are the known mean and variance of the execution time of block B_j respectively, then the execution time of B_j should always be less than $\mu_j + \gamma\sigma_j$ in each $l \geq 1$ consecutive cycles where l is given at runtime:

$\forall k \geq 1, \forall 1 \leq j \leq n, (e_{j,k} \leq \mu_j + \gamma\sigma_j) \wedge (e_{j,k+1} \leq \mu_j + \gamma\sigma_j) \wedge \ldots \wedge (e_{j,k+l-1} \leq \mu_j + \gamma\sigma_j)$

where γ is computed using equation 3.24.

C5 This is a corollary of C3. If $\widehat{\mu}_j$ and $\widehat{\sigma}_j^2$ are the estimated mean and variance of the execution time of block B_j, executed over 1000 cycles, then starting from the 1001^{th} cycle, the execution time of B_j should always lie within $\widehat{\mu}_j + \gamma\widehat{\sigma}_j$ in each $l \geq 1$ consecutive cycles where l is given at runtime:

$\forall k > 1000, \forall 1 \leq j \leq n, (e_{j,k} \leq \widehat{\mu}_j + \gamma\widehat{\sigma}_j) \wedge (e_{j,k+1} \leq \widehat{\mu}_j + \gamma\widehat{\sigma}_j) \wedge \ldots \wedge (e_{j,k+l-1} \leq \widehat{\mu}_j + \gamma\widehat{\sigma}_j)$

where γ is computed using equation 3.24.

C6 This contract is based on a *sliding window* update technique of the parameters. The window size is taken to be 1000. The principle is to update the mean and variance of the execution time of a function block at an interval of h cycles where the value of $h \geq 1$ is decided by the user at runtime. The h oldest value of the execution times are replaced by the h most recent value and the parameters are recomputed for performing the contract checks. If $\widehat{\mu}_{j,((k-1)h+1)\ldots(1000+(k-1)h)}$ and $\widehat{\sigma}^2_{j,((k-1)h+1)\ldots(1000+(k-1)h)}$ are the estimated parameters of the execution time of block B_j, computed for every k^{th} set of 1000 cycles at an interval of h cycles, then the execution time of B_j from the $(1000+(k-1)h+1)^{th}$ cycle to the $(1000+(kh))^{th}$ should always be less than $\widehat{\mu}_{j,((k-1)h+1)\ldots(1000+(k-1)h)} + \gamma\widehat{\sigma}_{j,((k-1)h+1)\ldots(1000+(k-1)h)}$, where $k \geq 1$:

$\forall k \geq 1, \forall h \geq 1, \forall 1 \leq j \leq n$,
$\left[e_{j,(1000+(k-1)h+1)} \leq \widehat{\mu}_{j,((k-1)h+1)\ldots(1000+(k-1)h)} + \gamma\widehat{\sigma}_{j,((k-1)h+1)\ldots(1000+(k-1)h)}\right] \wedge$
$\left[e_{j,(1000+(k-1)h+2)} \leq \widehat{\mu}_{j,((k-1)h+1)\ldots(1000+(k-1)h)} + \gamma\widehat{\sigma}_{j,((k-1)h+1)\ldots(1000+(k-1)h)}\right] \wedge \ldots \wedge$
$\left[e_{j,(1000+(kh))} \leq \widehat{\mu}_{j,((k-1)h+1)\ldots(1000+(k-1)h)} + \gamma\widehat{\sigma}_{j,((k-1)h+1)\ldots(1000+(k-1)h)}\right]$

where γ is computed using equation 3.24.

To elaborate further, let us take the value of h as 5. For $k = 1$, we compute the estimates of the mean and variance based on the $first$ 1000 cycles as $\widehat{\mu}_{j,((k-1)h+1)\ldots(1000+(k-1)h)}$ and $\widehat{\sigma}^2_{j,((k-1)h+1)\ldots(1000+(k-1)h)}$ respectively.

If we substitute $h = 5$ and $k = 1$, we get these estimates as $\widehat{\mu}_{j,1...1000}$ and $\widehat{\sigma}^2_{j,1...1000}$.

These estimates are used for computing an upper bound for the contracts from the $(1000 + (k-1)h + 1)^{th}$ cycle to the $(1000 + (kh))^{th}$ cycle. Again substituting the value of $h = 5$ and $k = 1$ here, we get cycles from 1001 to 1005.

Then this contract states that from cycle 1001 to 1005, the execution time of block B_j should not exceed the estimated upper bound, computed as: $\widehat{\mu}_{j,1...1000} + \gamma\widehat{\sigma}_{j,1...1000}$, where where γ is computed using equation 3.24.

C7 This contract is based on the *continuous update* technique. It updates the estimated parameters in each cycle starting from the 1001^{th} cycle by taking into account the current value of the execution time of a function block. Let $\widehat{\mu}_{j,1000}$ and $\widehat{\sigma}^2_{j,1000}$ be the parameters for block B_j over the first 1000 cycles. Then the parameters can be updated in every $(1000+k)^{th}$ cycle using equations (3.26) and (3.28), $k \geq 1$. The contract is then given as:

$$\forall k \geq 1, \forall 1 \leq j \leq n, e_{j,1000+k} \leq \widehat{\mu}_{j,(1000+k-1)} + \gamma\widehat{\sigma}_{j,(1000+k-1)}$$

where γ is computed using equation 3.24

C8 The sum of the execution times of all the blocks in each cycle must be strictly less than the required cycle time:

$$\forall k \geq 1, \Sigma_{j=1}^{n} e_{j,k} < P$$

C9 This contract checks that the jitter margin J is respected by application $\mathscr{A}$. It states that for two consecutive starts of $\mathscr{A}$ at times t and $t^{'}$, the difference between the cycle time P and $(t^{'} - t)$ should not exceed J:

$$\forall k \geq 1, \forall 1 \leq j \leq n, \forall t, t^{'} \geq 0, [R(\uparrow \mathscr{A}, k, t) \wedge R(\uparrow \mathscr{A}, k+1, t^{'})] \Rightarrow |\,(t^{'} - t) - P\,| \leq J$$

C10 These contracts are used for specifying an acceptable time interval between the start (or end) of two function blocks with respect to each other.

For two blocks B_j and $B_m, 1 \leq m < j \leq n$ and $\forall t \geq 0, \forall i > 0$, it is possible to define contracts of the forms:

1. $@(\uparrow B_j, i) \leq @(\uparrow B_m, i) + t$
2. $@(\downarrow B_j, i) \leq @(\uparrow B_m, i) + t$
3. $@(\uparrow B_j, i) \leq @(\downarrow B_m, i) + t$
4. $@(\downarrow B_j, i) \leq @(\downarrow B_m, i) + t$

This set of contracts is particularly useful when function blocks are deployed in separate hosts and communicate over a network. We will see a related case study in section 5.5. In such applications, if a receiver block (which depends on data sent by a sender) is deployed in a host machine different from the one on which the sender is launched, there might be a large communication delay. In order to detect such delays, temporal contracts are introduced on the receiver end to check whether it received the data within a predefined time interval.

3.4 Conclusions

This chapter described the requirements of the contract framework, its features and the underlying principles. It showed how stochastic contracts are computed using online estimation techniques, based on the *empirical cdf* of the execution times of the function blocks. It also introduced the concept of *parametric real-time contracts*, which will be described in detail in the next chapter.

It illustrated the use of RTL for formally defining the temporal contracts. Using RTL for formalizing stochastic temporal properties is a new concept which this chapter introduced.

Algorithm 1 Empirical Cdf

```
procedure EMPIRICALCDF(execution_times)
    y ← 0
    buffer_size ← SIZE(execution_times)
    min ← 0
    max ← 0
    upper_bound ← 0
    cumulative_sum ← 0
    SORT(execution_times)
    min ← execution_times[0]
    max ← execution_times[buffer_size − 1]
    upper_bound ← max
    number_of_bins ← SQRT(buffer_size)
    bin_width ← (max−min) / number_of_bins
    upper_limit_of_bin ← (execution_times[0] + bin_width)
    while upper_limit_of_bin ≤ max do
        while y ≤ buffer_size do
            if execution_times[y] ≤ upper_limit_of_bin then
                cumulative_sum ← cumulative_sum + 1
            end if
            y ← y + 1
        end while
        probability ← cumulative_sum / buffer_size
        if probability ≥ π then
            upper_bound ← upper_limit_of_bin
            break
        end if
        upper_limit_of_bin ← upper_limit_of_bin + bin_width
        cumulative_sum ← 0
        y ← 0
    end while
    return upper_bound
end procedure
```

4 Implementation

This chapter describes the implementation of the contract framework. The entire coding was done in C++ to ensure compatibility with the FASA framework. ABB's Git Repository was used for version control. It required about 5000 loc to implement the contract framework and validate it. All the plots have been generated using MATLAB.

Figure 4.1 shows a high level view of the contract framework. It shows that the contract framework handles contracts in two levels, block-level and schedule level. The FASA scheduler is responsible for scheduling the logging of the failed contracts depending on the availability of slack time. This is shown in figure 4.3.

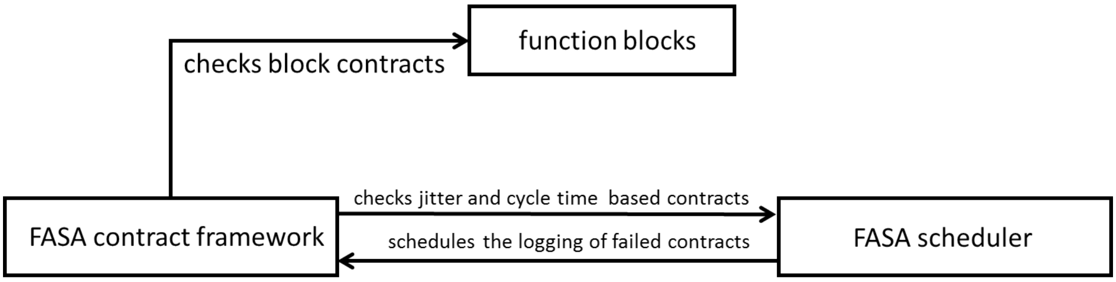


Figure 4.1: High level view of the contract framework

4.1 fasa_assert.h

The first step in developing the contract framework was to develop a dedicated library for contracts, which is the backbone of the entire framework. In this library, the following categories of macros are defined:

1. **preconditions :**
   PRECONDITION(expression)
2. **postconditions :**
   POSTCONDITION(expression)
3. **class invariants :**
   INVARIANT(expression)

4. **loop invariant :**
   LOOP_INVARIANT(expression)

5. **loop variant:**
   LOOP_VARIANT(expression)

4.1.1 Logging of the messages

Upon failure of a contract, which in the above syntax, is the value of the *expression*, a message is stored in a dynamic string array as shown in figure 4.2.

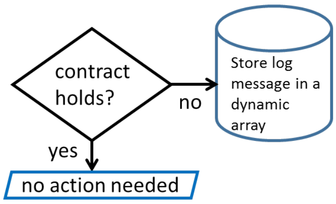


Figure 4.2: Fundamental mechanism of the contract framework

The structure of the message for all the categories, **except the loop variant** is:

<message> ::= <contract_type> "failed:" <file_name> ":" <line_number> ":" <contract_condition> ":" <time_of_failure>

<contract_type> ::= precondition | postcondition | invariant | loop invariant

For the loop variant, the message also contains information regarding the iteration number at which the variant expression failed. In this case, the message structure is:

<message> ::= "loop variant failed :" <file_name> ":" <line_number> ":" <contract_condition> ":" <time_of_failure> ": loop number : " <loop_number>

The messages in the array are logged during the **slack period**, which is the time that is left in each cycle after all the blocks are executed. In case the slack period expires before all the messages could be logged, the remaining messages are put into a buffer [16]. In the next cycle, before logging its own contract failure messages, the messages from this buffer are logged and the buffer is cleared. Figure 4.4 shows a screen shot to demonstrate the working of the contract framework. After every cycle, the dynamic array where the failure messages are stored, is also cleared. The logging mechanism is illustrated in figure 4.3.

FASA uses the *log4cpp* [20] library for logging. For the contract failure messages, we use the `DEBUG` level. It has the lowest priority (value=700) as defined by the library. This means that only messages with `DEBUG` priority and lower priorities are going to be captured by the logger. The only other log level for FASA which has lower priority than `DEBUG` is the `MONITORED` level [42], which is used by the monitoring facilities of FASA. However, the monitoring feature

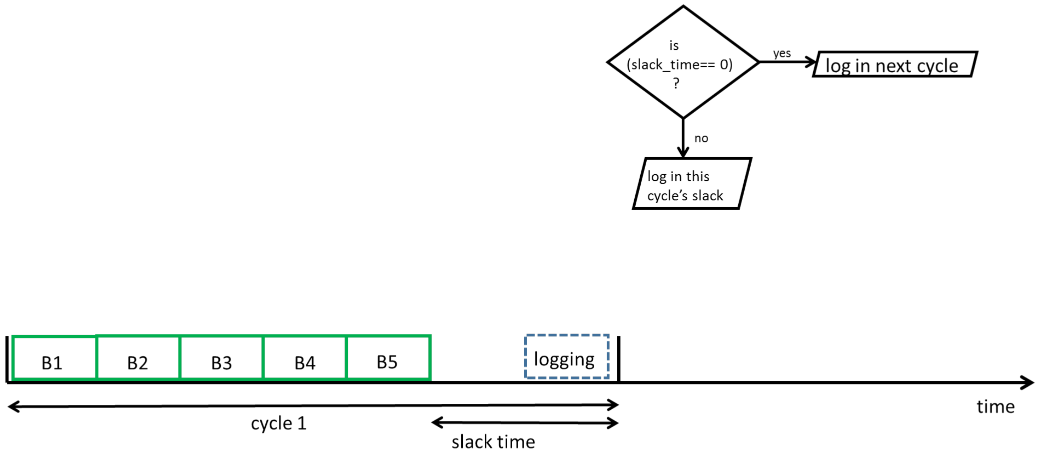


Figure 4.3: Logging for the contract framework

of FASA is by default turned off. Thus, using the `DEBUG` level for contracts ensures that the time taken for logging the contract failure messages in the slack time will be minimal as long as the monitoring feature is turned off. During the testing and validation phase of the contract framework, monitoring was always disabled, thereby keeping the logging time for the contracts as small as possible.

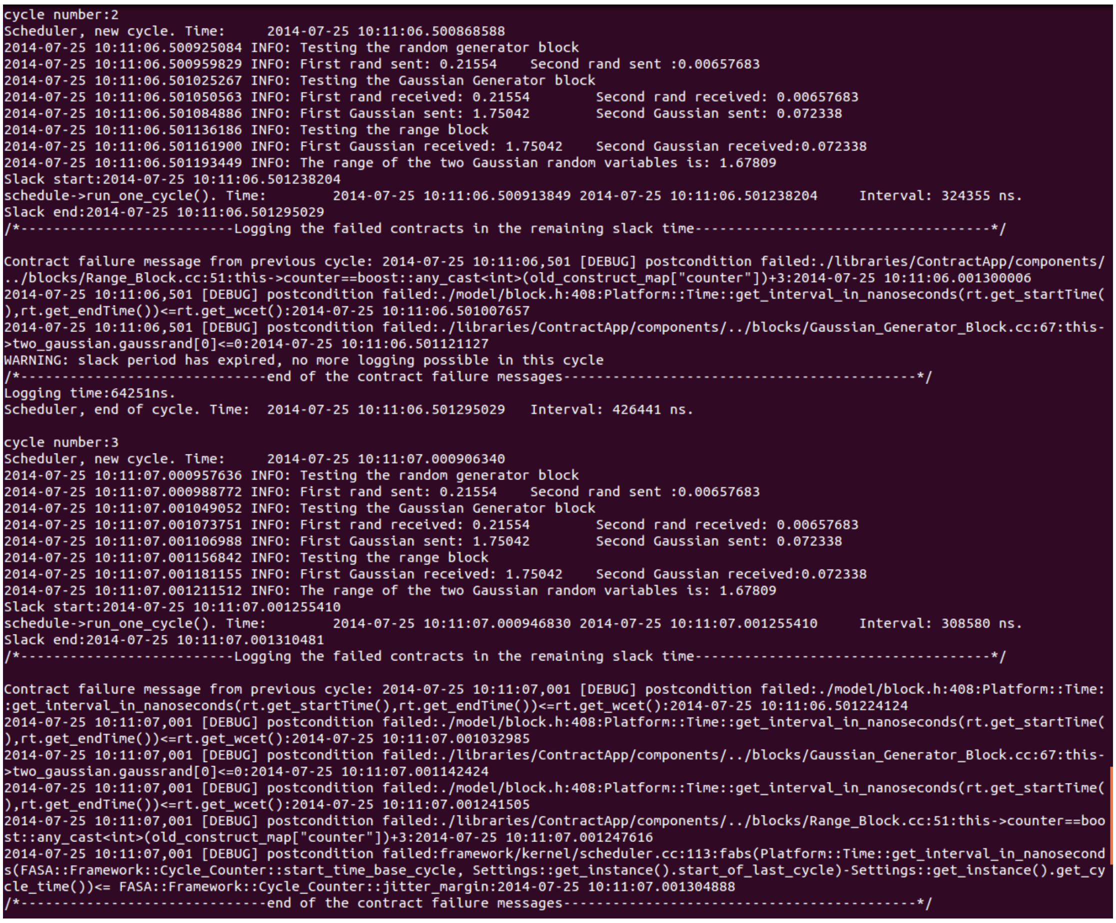
Old construct: The library also allows monitoring of the value of a variable similar to Eiffel. In section 4.4, this feature is described in detail.

For the first four contract categories listed at the beginning of this section, we need to check whether the value of *expression* evaluates to true. For the fifth category, i.e. the loop variants, the principle is different. As mentioned in 2.1.1, loop variants are conditions which ensure termination of a loop and are therefore important for proving the *total* correctness of a loop. Both loop invariants and loop variants are conditions that should be checked after every iteration. The loop invariants additionally must also be checked before the first iteration and after the termination of the loop.

The checking of loop variants for cyclic applications such as in FASA is non-trivial. This is because, we need to find a mechanism, that, starting from the second iteration will store the value of the variant expression in the previous iteration and compare it with the value in the current iteration, while being in the *same* cycle. Once a cycle is over, the value of the variant expression becomes invalid and we need to start the process again. Algorithm 2 describes the mechanism adopted for this purpose. In line 4 of the algorithm, the condition ensures that the variants are checked starting from the second iteration and that this is done only after the value of the `cycle_no` gets updated to the current cycle number.

4.2 Approaches towards the contract framework

This thesis demonstrates the use of two different approaches for developing the contract framework. The following sections will illustrate the merits and demerits of both and will



```
cycle number:2
Scheduler, new cycle. Time:     2014-07-25 10:11:06.500868588
2014-07-25 10:11:06.500925084 INFO: Testing the random generator block
2014-07-25 10:11:06.500959829 INFO: First rand sent: 0.21554    Second rand sent :0.00657683
2014-07-25 10:11:06.501025267 INFO: Testing the Gaussian Generator block
2014-07-25 10:11:06.501050563 INFO: First rand received: 0.21554        Second rand received: 0.00657683
2014-07-25 10:11:06.501084886 INFO: First Gaussian sent: 1.75042        Second Gaussian sent: 0.072338
2014-07-25 10:11:06.501136186 INFO: Testing the range block
2014-07-25 10:11:06.501161900 INFO: First Gaussian received: 1.75042    Second Gaussian received:0.072338
2014-07-25 10:11:06.501193449 INFO: The range of the two Gaussian random variables is: 1.67809
Slack start:2014-07-25 10:11:06.501238204
schedule->run_one_cycle(). Time:        2014-07-25 10:11:06.500913849 2014-07-25 10:11:06.501238204     Interval: 324355 ns.
Slack end:2014-07-25 10:11:06.501295029
/*-------------------------Logging the failed contracts in the remaining slack time----------------------------------*/

Contract failure message from previous cycle: 2014-07-25 10:11:06,501 [DEBUG] postcondition failed:./libraries/ContractApp/components/
../blocks/Range_Block.cc:51:this->counter==boost::any_cast<int>(old_construct_map["counter"])+3:2014-07-25 10:11:06.001300006
2014-07-25 10:11:06,501 [DEBUG] postcondition failed:./model/block.h:408:Platform::Time::get_interval_in_nanoseconds(rt.get_startTime(
),rt.get_endTime())<=rt.get_wcet():2014-07-25 10:11:06.501007657
2014-07-25 10:11:06,501 [DEBUG] postcondition failed:./libraries/ContractApp/components/../blocks/Gaussian_Generator_Block.cc:67:this-
>two_gaussian.gaussrand[0]<=0:2014-07-25 10:11:06.501121127
WARNING: slack period has expired, no more logging possible in this cycle
/*----------------------------end of the contract failure messages-----------------------------------------*/
Logging time:64251ns.
Scheduler, end of cycle. Time:  2014-07-25 10:11:06.501295029   Interval: 426441 ns.

cycle number:3
Scheduler, new cycle. Time:     2014-07-25 10:11:07.000906340
2014-07-25 10:11:07.000957636 INFO: Testing the random generator block
2014-07-25 10:11:07.000988772 INFO: First rand sent: 0.21554    Second rand sent :0.00657683
2014-07-25 10:11:07.001049052 INFO: Testing the Gaussian Generator block
2014-07-25 10:11:07.001073751 INFO: First rand received: 0.21554        Second rand received: 0.00657683
2014-07-25 10:11:07.001106988 INFO: First Gaussian sent: 1.75042        Second Gaussian sent: 0.072338
2014-07-25 10:11:07.001156842 INFO: Testing the range block
2014-07-25 10:11:07.001181155 INFO: First Gaussian received: 1.75042    Second Gaussian received:0.072338
2014-07-25 10:11:07.001211512 INFO: The range of the two Gaussian random variables is: 1.67809
Slack start:2014-07-25 10:11:07.001255410
schedule->run_one_cycle(). Time:        2014-07-25 10:11:07.000946830 2014-07-25 10:11:07.001255410     Interval: 308580 ns.
Slack end:2014-07-25 10:11:07.001310481
/*-------------------------Logging the failed contracts in the remaining slack time----------------------------------*/

Contract failure message from previous cycle: 2014-07-25 10:11:07,001 [DEBUG] postcondition failed:./model/block.h:408:Platform::Time:
:get_interval_in_nanoseconds(rt.get_startTime(),rt.get_endTime())<=rt.get_wcet():2014-07-25 10:11:06.501224124
2014-07-25 10:11:07,001 [DEBUG] postcondition failed:./model/block.h:408:Platform::Time::get_interval_in_nanoseconds(rt.get_startTime(
),rt.get_endTime())<=rt.get_wcet():2014-07-25 10:11:07.001032985
2014-07-25 10:11:07,001 [DEBUG] postcondition failed:./libraries/ContractApp/components/../blocks/Gaussian_Generator_Block.cc:67:this-
>two_gaussian.gaussrand[0]<=0:2014-07-25 10:11:07.001142424
2014-07-25 10:11:07,001 [DEBUG] postcondition failed:./model/block.h:408:Platform::Time::get_interval_in_nanoseconds(rt.get_startTime(
),rt.get_endTime())<=rt.get_wcet():2014-07-25 10:11:07.001241505
2014-07-25 10:11:07,001 [DEBUG] postcondition failed:./libraries/ContractApp/components/../blocks/Range_Block.cc:51:this->counter==boo
st::any_cast<int>(old_construct_map["counter"])+3:2014-07-25 10:11:07.001247616
2014-07-25 10:11:07,001 [DEBUG] postcondition failed:framework/kernel/scheduler.cc:113:fabs(Platform::Time::get_interval_in_nanosecond
s(FASA::Framework::Cycle_Counter::start_time_base_cycle, Settings::get_instance().start_of_last_cycle)-Settings::get_instance().get_cy
cle_time())<= FASA::Framework::Cycle_Counter::jitter_margin:2014-07-25 10:11:07.001304888
/*----------------------------end of the contract failure messages-----------------------------------------*/
```

Figure 4.4: This screen shot shows how the messages are logged in the remaining slack time after all blocks are executed. It also shows how the messages are carried to the next cycle in case the slack period is over in the current cycle. The screen shot is generated while testing the contract framework on case study-2, Gaussian Random Generator, described in Chapter 5.

justify the use of the second approach throughout the rest of the research. The two approaches are:

1. To have dedicated *function blocks* for contracts.
2. To have dedicated *routines* for contracts within the main function blocks.

4.2.1 Dedicated function blocks for contracts

In this approach, the contracts are specified in dedicated function blocks for pre-conditions, post-conditions and invariants. Figure 4.5 shows the structure of such an application. The schedule of the corresponding FASA application is modified such that the pre-condition block is executed before the actual function block and the post-condition block is executed after the actual block. The invariant block is executed both before and after the function block. The classes for the pre and post conditions and class invariants are declared as *friends* of the actual

Algorithm 2 Loop Variant Analysis

```
procedure LoopVariant(expression)
    static iteration_number ← 1
    static cycle_no ← 1
    if iteration_number > 1 and current_cycle_number = cycle_no then
        current_value ← expression
        if old_value > current_value then
            assertion holds
        else
            store failure message
        end if
    end if
    iteration_number ← iteration_number + 1
    old_value ← expression
    cycle_no ← current_cycle_number
end procedure
```

function blocks in order to allow them access to the private and protected members of the actual blocks. The advantages of this approach are:

- It imparts *flexibility* to the contract framework. Since the contracts are defined in separate function blocks, it makes the contract framework independent of the rest of the FASA application. As a result, in a setting with multiple cores, the contract blocks can be deployed in separate cores.
- They can be used *off the shelf*, respecting the concept of CBSE. For a new applications, the contracts need not be written down again and pre-existing contract blocks can be simply *plugged in* at appropriate locations as shown in figure 4.5.

The demerits of this approach are:

- It adds overhead to the FASA application in terms of the execution time thereby degrading its performance.
- This approach requires substantial additional work from the application developer. If an actual FASA application has n function blocks, this approach would require the developer to create $3n$ additional function blocks (for preconditions, postconditions and invariants).

4.2.2 Dedicated routines for contracts

In this approach, contracts are defined in dedicated routines within the actual function blocks. The merits of this approach are:

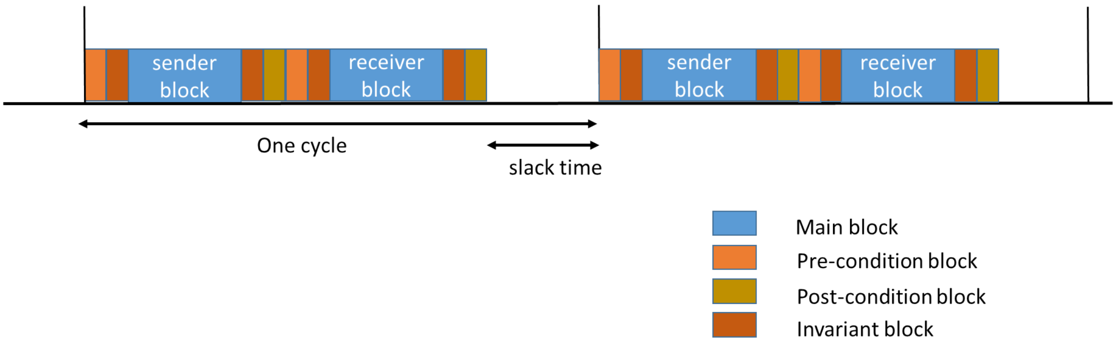


Figure 4.5: Sender-Receiver application with dedicated blocks for contracts

- It is more efficient than the dedicated function block approach in the sense that it adds less overhead to the application.
- It does not require much additional effort on the developer's side.

The demerit of this approach is that:

- It is less flexible compared to the dedicated function block approach. All contracts have to be executed on the same core as the application even in a distributed setting.

Both the approaches were tested on a simple two block sender-receiver application. Only one precondition and one postcondition was checked for both the sender and receiver blocks.

First, precondition and postcondition blocks were inserted before and after the function blocks respectively. Figure 4.6 shows the corresponding modifications in red in the application's xml file. For our experiment, the pre and post condition blocks were put inside the same component as the main function block implying that they were deployed on the same host machine.

Then, the same contracts were put inside the actual function blocks in dedicated routines. For each case, the execution time of the application was observed over 1000 cycles, repeated three times for precision.

Table 4.1 shows the mean execution time of the application in ns (nanoseconds) with dedicated contract blocks inserted in the application and with dedicated contracts routines within the actual function blocks. As it can be seen, the execution time of the application with dedicated contract blocks is more than twice the execution time with dedicated routines.

This shows that there is trade-off between the flexibility and the performance of the two approaches as shown in figure 4.7. Referring to **Req4** in chapter 3, one of the main requirements to be fulfilled by the contract framework is to keep the overhead to the bare minimum. From this point of view, the *dedicated routine* approach proved to be a better option and for the rest of the research, this approach has been adopted.

```xml
<schedule>
  <sequential>
    <execute component="Sender" block="preSender" />
    <execute component="Sender" block="sender" />
    <execute component="Sender" block="postSender" />

    <execute component="Receiver" block="preReceiver" />
    <execute component="Receiver" block="receiver" />
    <execute component="Receiver" block="postReceiver" />
  </sequential>
</schedule>
```

Figure 4.6: Modified schedule in the main xml file for the sender-receiver application that enables precondition and postcondition blocks.

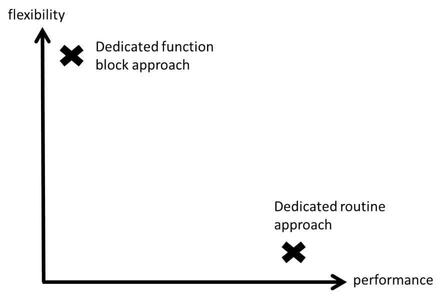


Figure 4.7: Performance-flexibility trade-off diagram

The entire process of developing the contract framework is comprised of several steps which are discussed in detail below:

1. **Flexibility of the contract framework**: Two levels of flexibility have been added to the FASA contract framework. This takes care of the **Req1** as described in chapter 3.

   (a) Compile time flexibility: The contract framework has been developed so that it can be turned on/off during the compilation of the FASA platform. This is necessary because during the release build, the developer might want to have the contract framework deactivated in order to avoid any overhead, while in the debug build, having the contract framework active is desirable. For this, the `makefile` of FASA has been extended. By default, the framework is always deactivated such that users who are not concerned with the dynamic verification aspect of FASA do not see any change in the actual FASA platform. The developer would have to add a flag in

Table 4.1: Execution time comparison for sender-receiver application

dedicated contract blocks	dedicated contract routines
167760 ns	74533 ns

order to activate it at compilation, which is defined as :

```
CONTRACTS::=enabled | disabled
```

Additionally, the directory structure for the contract framework was also changed in order to avoid mixing of this feature with the rest of the FASA platform. When contracts are disabled, the directory structure remains the same as before. When they are enabled, the depth of the directory tree is increased to an extra level, `enabled` and the compiled dll files of the FASA applications are stored here.

(b) Runtime flexibility: The FASA platform has a configuration file (xml file format) in which all the features of the framework are specified, such as the default cycle time, paths to the applications, enabled features like monitoring variables, logging messages etc. There is a default setting for this configuration file hardcoded in the FASA makefile, which can be updated by arguments passed at runtime. This configuration file has been augmented to facilitate contract features. A boolean valued element, `contracts` has been added with attributes, `jitter_margin` and `consecutive_cycles`, `threshold_probability` and `sliding_window_interval`. By default, the value of `contracts` is set to *false*. When FASA is compiled with contracts enabled, this value is set to *true*. For `jitter_margin`, the default value is 0. During runtime, the user can pass a value to this attribute. Additionally, the attribute `consecutive_cycles` is used for specifying contracts which check the execution time of a block for a certain number of consecutive cycles as explained in section 3.2.2. The default value of this attribute is set to 1. As described in 3.2.2, `threshold_probability` is used to estimate an upper bound for the block execution times and `sliding_window_interval` specifies the frequency of executing the sliding window algorithm. The default value of the former is set to 0.95 and for the latter, it is set to 1.

Figure 4.8 shows a snippet from the configuration file containing the contract framework addons. The figure corresponds to the case where the contract framework is turned off.

```xml
<contracts enable="false" consecutive_cycles="1" jitter_margin="0" sliding_window_interval="1" threshold_probability="0.95"/>
```

Figure 4.8: xml code snippet from the configuration file of FASA illustrating the contract-addons.

2. **Augmenting the base class `Block` with real-time properties and contract checking features**: The base class `Block` has a method `run_one_cycle()` which is executed in every cycle. In this method, the `operator()` method is invoked which is overriden in every derived function block and describes its behavior. For the contract framework, we add the following routines to this class which are overriden in each function block:

   (a) `functional_pre_conditions()`

   (b) `realTime_pre_conditions()`

   (c) `functional_post_conditions()`

(d) `realTime_post_conditions()`

(e) `invariants()`

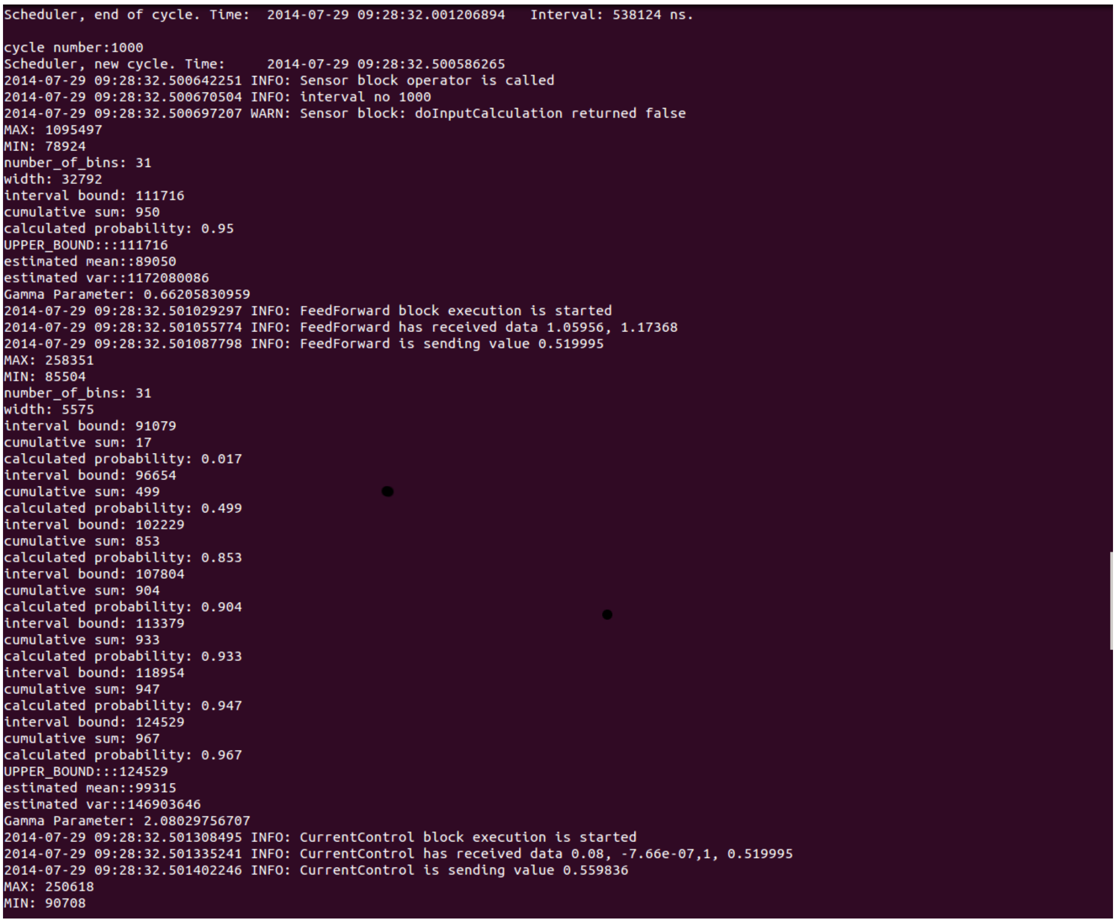
(f) `default_real_time_post_conditions()`

Out of the above, `default_real_time_post_conditions()` is used for checking the block-WCET related *fundamental* contract as described in section 3.2.2 and is invoked after the `operator()` method by default, whenever the contract framework is activated. For this, it is assumed that the WCET value used in this contract is known to us from some external source that does static analysis of the function block and computes this value. Figure 4.10 shows an example of a *fundamental* real-time post condition related to the WCET of a block and figure 4.11 shows a real-time contract computed using the sliding-window update technique. The routine `invariants()` is checked before and after the `operator()` method. It is used for class invariants. As the names suggest, the methods for the pre-conditions are invoked before the `operator()` method and the post-condition related routines are invoked after the `operator()` method. In addition to the above, the base `Block` also has real-time properties in the form of a `structure`, `Real_Time_Properties` which is inherited by all the function blocks. This structure contains information such as cycle time of an application, the WCET of the blocks, starting and ending time of the execution of a block and several other properties used for the statistical analysis of the execution times. In figures 4.10 and 4.11, **rt** is an instance of the structure containing real-time properties.

One point to noted here is that when we refer to the execution time of a function block, we mean the execution time of the main functionality of the block which is invoked in every cycle. It implies that the execution time of a block is the time it takes to execute `operator()` in each cycle because it is the only function that is invoked in every cycle which actually describes the behavior of a block.

For the contracts related to the statistical analysis of the temporal aspects of the FASA function blocks, there is a method, `statistical_analysis()`. This method takes as an argument, a circular buffer that contains the execution times of a function block, with capacity set to $experimental\_cycle\_number$ as introduced in chapter 3. Circular buffer is also used for checking contracts related to the the execution times of the function blocks for l consecutive cycle where the size of the buffer is set to l. It is in this method that the computations regarding the *sliding window* and *continuous* updating of the statistical parameters are done using the principles described in item 1 and item 2 in section 3.2.2 respectively. Additionally, the method `empirical_cdf()` computes the empirical cdf of the execution times of the function blocks using Algorithm 1. The screen shot in figure 4.9 illustrates the working of this feature.

3. **Contracts at one level higher than the blocks**: Above, we saw how the contract framework has been developed for assertions *local* to the function blocks. The block level is the lowest level in the hierarchical structure of a FASA application. While contracts related to execution times of the blocks can be specified at the block level, contracts related to the cycle-time and jitter margin of an application, have to be specified at the schedule level. Taking this into account, the contracts for the cycle time and jitter margin have been made implicit in the framework at the schedule level. The value of



```
Scheduler, end of cycle. Time:  2014-07-29 09:28:32.001206894   Interval: 538124 ns.

cycle number:1000
Scheduler, new cycle. Time:     2014-07-29 09:28:32.500586265
2014-07-29 09:28:32.500642251 INFO: Sensor block operator is called
2014-07-29 09:28:32.500670504 INFO: interval no 1000
2014-07-29 09:28:32.500697207 WARN: Sensor block: doInputCalculation returned false
MAX: 1095497
MIN: 78924
number_of_bins: 31
width: 32792
interval bound: 111716
cumulative sum: 950
calculated probability: 0.95
UPPER_BOUND:::111716
estimated mean::89050
estimated var::1172080086
Gamma Parameter: 0.66205830959
2014-07-29 09:28:32.501029297 INFO: FeedForward block execution is started
2014-07-29 09:28:32.501055774 INFO: FeedForward has received data 1.05956, 1.17368
2014-07-29 09:28:32.501087798 INFO: FeedForward is sending value 0.519995
MAX: 258351
MIN: 85504
number_of_bins: 31
width: 5575
interval bound: 91079
cumulative sum: 17
calculated probability: 0.017
interval bound: 96654
cumulative sum: 499
calculated probability: 0.499
interval bound: 102229
cumulative sum: 853
calculated probability: 0.853
interval bound: 107804
cumulative sum: 904
calculated probability: 0.904
interval bound: 113379
cumulative sum: 933
calculated probability: 0.933
interval bound: 118954
cumulative sum: 947
calculated probability: 0.947
interval bound: 124529
cumulative sum: 967
calculated probability: 0.967
UPPER_BOUND:::124529
estimated mean::99315
estimated var::146903646
Gamma Parameter: 2.08029756707
2014-07-29 09:28:32.501308495 INFO: CurrentControl block execution is started
2014-07-29 09:28:32.501335241 INFO: CurrentControl has received data 0.08, -7.66e-07,1, 0.519995
2014-07-29 09:28:32.501402246 INFO: CurrentControl is sending value 0.559836
MAX: 250618
MIN: 90708
```

Figure 4.9: The screen shot shows the computation of the γ parameter described in equation 3.24 for the function blocks, `Sensor` and `FeedForward` at the 1000^{th} cycle. This figure refers to the simulink based case study-4, described in chapter 5.

the cycle time can be retrieved from the configuration file of FASA and this is used to impose a contract that ensures that the cycle time is not exceeded by an application. For the jitter margin, the principle described in **C9** of subsection 3.3.2 is used. Figure 4.12 illustrates the structure of the contract framework with respect to FASA's architecture.

4.3 Parametric temporal contracts

As described in chapter 3, the contract framework allows the application developer to specify real-time contracts as functions of certain data-structures or variables used in the description of the function block behavior. Here we describe how this is done in practice. The *variadic*

```
POSTCONDITION(EXEC_TIME_THIS_CYCLE<=rt.get_wcet());
```

Figure 4.10: An example of a real-time post-condition related to block WCET.

```
POSTCONDITION(EXEC_TIME_THIS_CYCLE<=rt.estimated_mean_var[0]+rt.k_value*sqrt(rt.estimated_mean_var[1]));
```

Figure 4.11: Real-time contract based on sliding-window update technique for statistical parameters. In the figure, `k_value` represents the γ parameter as described in equation 3.24. `rt.estimated_mean_var` is a circular buffer of size two, which stores the mean and variance after every update. The mean is the first term and the variance is the second term in the buffer.

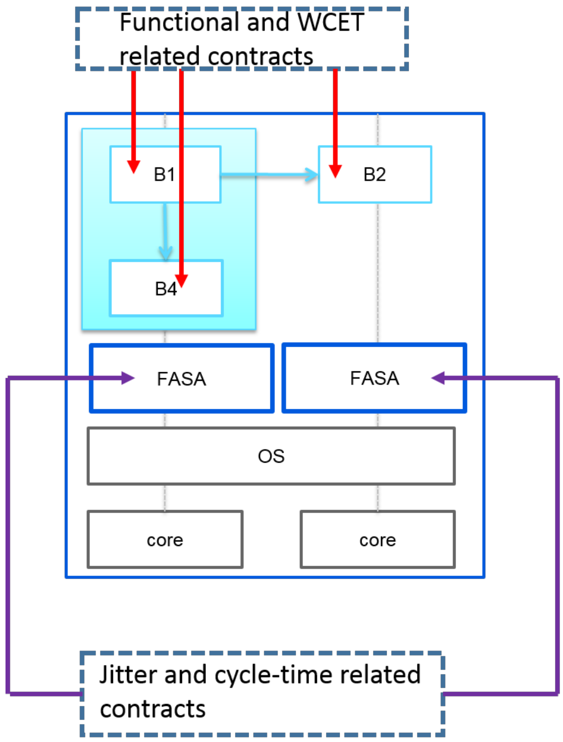


Figure 4.12: This figure illustrates the structure of the contract framework on top of the FASA architecture.

templates feature of C++ 11 has been used for this purpose. We have defined a `singleton` class, `User_Defined_Function`, which is instantiated once in the base `Block` class and we define a variadic macro in this file (`block.h`) as shown in figure 4.13. In this figure, the ellipsis argument to the macro `f(...)` indicates that this is a variadic macro. The term `__VAR_ARGS` in the macro definition is replaced by the actual arguments separated by comma [4].

The function `user_defined_function_for_parametric_contracts(Block b, Arguments... parameters)` as defined in figure 4.14 is a variadic template function. This function can take any number of any types of arguments as parameters, which is indicated by the ellipsis on the left hand side of the second argument to the function. It represents the *packing* of the parameters. The first argument to this function is a `Block` type, `b`. Inside this function as we can see from figure 4.14, we invoke yet another variadic function and pass on the arguments to it. For the second argument to this inner function call, we see that the ellipsis occurs on the right hand side of the argument, which indicates *unpacking* of the arguments passed to it [2].

block.h

```
#define f(...) unsigned(FASA::Model::User_Defined_Function::getInstance()->
                                        user_defined_function_for_parametric_contracts( __VA_ARGS__))
```

Figure 4.13: Use of variadic macro and variadic template functions from C++ 11 for parametric real-time contracts in FASA.

user_defined_function.h

```
/*using C++11 feature, variadic template*/
template<typename Block, typename... Arguments>
uint64_t user_defined_function_for_parametric_contracts(Block b, Arguments... parameters)
{
    uint64_t ret = function_definition(b, parameters...);
    return ret;
}
```

Figure 4.14: Definition of the variadic template function in the `User_Defined_Function` class used for parametric temporal contracts in FASA.

This inner function `function_definition()` has to be defined by the FASA application developer. Figure 4.15 shows an example of this definition. The figure explains why we need to pass the block as a necessary argument to this routine. To elaborate further, let us consider two function blocks. In FASA, every block has a unique id. Let the ids for these two function blocks be `block1` and `block2` respectively, as we can see from figure 4.15. Let us consider an application which uses both `block1` and `block2`. Let `block1` be performing merge sort of an array of length, say n, where n is an integer. In the worst case, the performance of merge sort is given by $O(n * log(n))$. Let `block2` be performing binary search in an array for which the length is again an integer. The worst case running time of binary search is $O(log(n))$. Let us now consider the situation in which the developer wants to represent a contract for the execution times of these two function blocks as a function of the length of the array on which they operate.

In other words, for both the blocks, the user would like to define the execution time as a function of an attribute of type **int**. However, it is not necessary that both the blocks will have the same function definition. In this particular example, we can see that for block `block1`, the function is $n * log(n)$ while for `block2`, the function is $log(n)$. If we do not pass the id of the block to `user_defined_function_for_parametric_contracts()`, then such a situation cannot be handled in which different function blocks within the *same* application have different dependencies on the variables or data structures of the *same type*, because we cannot have multiple definitions of the same function prototype. This is a way to make `user_defined_function_for_parametric_contracts()` return the correct function of n for the blocks. The return type of the function is an unsigned 64 bit integer which is sufficient for representing time in nanoseconds.

user_defined_function.h

```
inline uint64_t function_definition(Block *block, int n)
{
      cout<<"id"<<block->get_id()<<endl;
      stringstream strm;
      string st;
      strm<<block->get_id();
      st=strm.str();
      if(st==“block1")
      {
            return t*n*log(n)+constant1;
      }
      else if(st==“block2")
      {
            return t*log(n)+constant2;
      }
}
```

Figure 4.15: Examples of user defined functions in the `User_Defined_Function` class used for parametric temporal contracts.

Figure 4.16 shows an example of how to write parametric real-time contracts for function blocks. As we can see, the first argument is a pointer to the block itself, and the second argument in this case is the size of an array.

Binary_Search_Block.cc

```
void Binary_Search_Block::realTime_post_conditions()
{
      POSTCONDITION(EXEC_TIME_THIS_CYCLE<=f( this, sorted_array.size));
}
```

Figure 4.16: A parametric contract based on the concept of variadic template function.

4.4 The "Old" construct

The Eiffel programming language [33] has the *Old* construct that allows one to check assertions related to the correct updating of the value of a variable. For the FASA framework as well, this construct has been simulated. For this, two macros have been defined as shown in Figure 4.17.

```
#define OLD(X) old_construct_map[STRINGIZE(X)]=X
#define GET_OLD(DATA_TYPE , X) boost::any_cast<DATA_TYPE>(old_construct_map[STRINGIZE(X)])
```

Figure 4.17: Macros for the "Old" construct

The **OLD** macro is used before any change is made to a variable and **GET_OLD** is used to retrieve the *old* value. The principle behind this feature is to create a map data structure, `old_construct_map` that stores the name of the variable as the *key* and the current value as the *value*. This map is inherited by all the function blocks and thus, it is declared in the base class `Block`. The keys are of type `string` and the values have type `boost::any`. This is a feature of the boost::any class [21] which allows the values to be of any type. This feature is safe for use because it only allows the data to be of any arbitrary type but does not facilitate conversion between them. Thus, we can use this map to store any kind of value corresponding to its variable name. The only additional requirement from the user's side here is to pass the *type* of the variable as an argument to **GET_OLD** while defining the postcondition. In Figure 4.18, snippets from a sample function block depicting this feature are shown.

Sample_block.h

```
class Sample_Block: public Block
{
    int counter;
public:
    void operator ();
    void functional_pre_conditions();
    void functional_post_conditions();
};
```

Sample_block.cc

```
void Sample_Block::construct()
{
    counter=0;
}
void Sample_Block ::operator()
{
    OLD(counter);
    counter=counter+2;
}
void Sample_Block ::functional_post_conditions()
{
    POSTCONDITION(this→counter==GET_OLD(int , counter)+2);
}
```

Figure 4.18: Block code showing the "Old" Construct

4.5 Conclusions

This chapter illustrated the implementation details of the contract framework. Two approaches for developing the contract framework were explored, *dedicated function blocks* and *dedicated routines*. Experiments showed the trade-off between them. Finally, in order to fulfill **Req4** stated in chapter 3, the *dedicated routine* approach was chosen.

5 Results and Validations

In this chapter, we describe five case studies which have been used to validate the contract framework. The performance of the framework has been analyzed for each benchmark and discussed in detail. For every case study, we first describe the application, then summarize the contracts and finally provide an analysis of the performance of the contract framework.

Tables 5.1, 5.2, 5.3 and 5.4 illustrate the status of the requirements stated in chapter 3.

Table 5.1: Status of requirement 1

Req1	Flexibility of the contract framework: it should not have any effect on the original FASA platform and the user should have the freedom to turn the contracts on/off as and when required.
Status	Passed.
Proof	The contract framework when turned off, showed no change in the execution of the applications. It was exactly the same as in the original FASA platform. When the contract framework was activated, a failure message for the *dummy* contract was logged in the slack period.
Remarks	The contract framework provides two layers of flexibility, compile time and run-time. When it is disabled at compile time, no change is observed in the original FASA platform. The run-time flexibility allows the user to define complex temporal contracts by passing parameters.

Table 5.2: Status of requirement 2

Req2	Allowing users to specify functional contracts at the function-block level.
Status	Passed.
Proof	Failure messages for the *dummy* functional contracts were logged in the slack period when contracts were enabled. This showed that the contract framework supports functional contracts.
Remarks	Users can define pre and post conditions, class invariants, loop variants and loop invariants. The framework also simulates the "Old" construct of Eiffel.

Table 5.3: Status of requirement 3

Req3	Allowing the users to specify temporal contracts.
Status	Passed.
Proof	When the contract framework was activated, failure messages for the *dummy* temporal contracts were logged in the slack time. This showed that the contract framework supports temporal contracts.
Remarks	The contract framework can dynamically compute temporal contracts using statistical inference.

Table 5.4: Status of requirement 4

Req4	Have minimum effect on the performance of the FASA platform, i.e limit the overhead due to contracts to the bare minimum.
Status	Partially met.
Proof	The overhead due to the contract framework was measured. It was shown to be less than 10% for all cases except for those applications which involve network communication.
Remarks	This requirement is satisfied for applications that are deployed on the same host. When network communication is involved, the overhead is large, mainly due to network delay.

The applications were tested on MacMini computers, with 4 GB RAM running 64 bit Ubuntu 13.04 and quad core processors, each core running at 0.8 GHz. In actual embedded systems, performance of the contract framework is expected to be further enhanced because of dedicated hardware infrastructure and the use of RTOS. Nevertheless, the following results are definitely an indication of the feasibility of using the contract framework on top of FASA.

We use box plots to summarize the performance of the contracts for neat handling of outliers. The results are shown for 10000 executions of each application, repeated three times for precision. The desired cycle time is taken as **10ms** for the first four case studies while for the fifth case study, it is taken as **100ms** in order to take care of the network communication delay. Additionally, the unit for the WCETs is ns. In the contract summary tables below, the notations used are as follows:

pre	Preconditions
$post$	Postconditions
I	Invariant
C_I	Class Invariant
L_I	Loop Invariant
L_V	Loop Variant

5.1 Simple Counter Application

5.1.1 Application description

This is a basal one block FASA application. The block does not have any ports and is not connected to any other block through any channels. Figure 5.1 shows this application. An integer variable, `counter` is initialized to 1 in the block constructor. A macro MAX_ITER is assigned value 10. In every cycle, the block iterates MAX_ITER times to increment the value of `counter`. In section 5.1.2, z is the variable used in the iteration of the loop.
This application has been designed such that we could test the contracts related to the "Old" construct, loop variants and invariants.

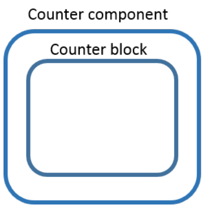


Figure 5.1: Simple one block FASA application.

5.1.2 Contract analysis

List of contracts

1. Functional:

   - POSTCONDITION(this->counter==GET_OLD(int, counter)+10)
   - LOOP_INVARIANT(z<=MAX_ITER)
   - LOOP_VARIANT(MAX_ITER-z)

   The explanation behind the loop variant (MAX_ITER-z) is that since z is incremented upto MAX_ITER, the value of (MAX_ITER-z) should decrease in every iteration. The same explanation follows for all the loop variants that are presented in the other case studies below.

2. Real-time:

   - **C1** with $WCET = 50000$
   - **C5** with $\pi = 0.95$ and $l = 1$
   - **C6** with $\pi = 0.95$ and $h = 1$
   - **C7** with $\pi = 0.95$

3. Scheduler level contracts:
   - **C8**
   - **C9** with $J = 100000$ ns

Summary table

Table 5.5 summarizes the contracts for case study 1.

Table 5.5: Case study 1-contract summary at block level

Type	**# pre**	**# post**	**# I**		# L_V	TOTAL
			# C_I	# L_I		
functional	0	1	0	1	1	3
real-time	0	4	0			4

Performance analysis

In figure 5.2, we show a summary of the execution times of case study 1 executed for 10000 cycles. The first plot shows the case *without* contracts and the second one illustrates the performance with contract enabled (logging feature of FASA deactivated). The third one shows the performance when the logging feature of FASA is also enabled. Table 5.6 shows the results of the analysis.

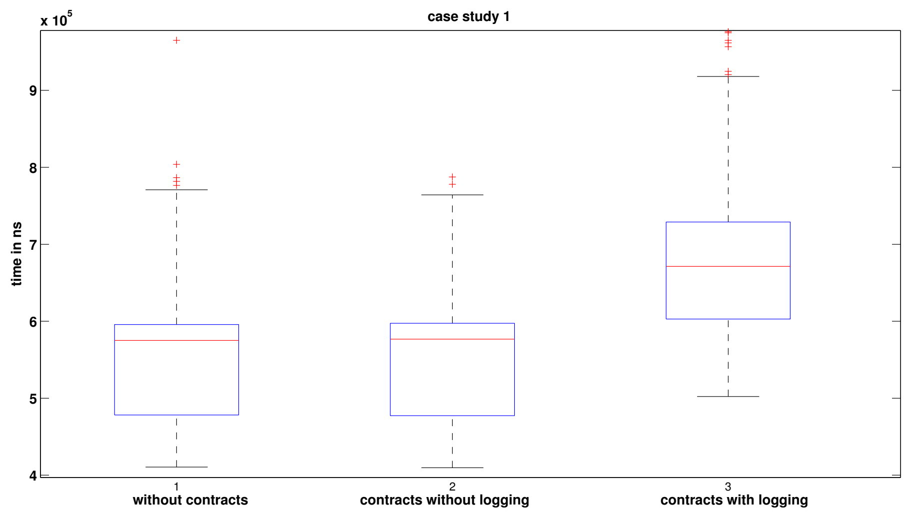


Figure 5.2: Box plot showing the execution times over 10000 cycles for case study 1 for the 1) original application, 2) with only contracts enabled and 3) with contracts enabled along with logging; the cycle time being 10 ms.

Table 5.6: Performance analysis of case study 1

enabled features	mean execution time	overhead
without contracts	0.5447 ms	-
contracts without logging	0.5574 ms	2.33%
contracts with logging	0.6640 ms	21.9 %

5.2 Gaussian Random Generator

5.2.1 Application description

This is a three block FASA application. It is used for generating random numbers having a Gaussian distribution. The mathematical principle on which this application is based is the *Box-Muller transformation*. Appendix A contains the details of this statistical technique. The first block, *Random_Generator_Block* generates two random numbers according to Uniform Distribution, $U[0,1]$. It then sends the two values to a second block, *Gaussian_Generator_Block* which generates two standard normal, $N(0,1)$ random values using the Box-Muller transformation. These two values are sent to a third block, the *Range_Block* which calculates the range of the two Gaussian random numbers. Figure 5.3 shows the structure of this application. This application illustrates functional contracts related to the connectivity of ports in channels and some contracts specific to the application.

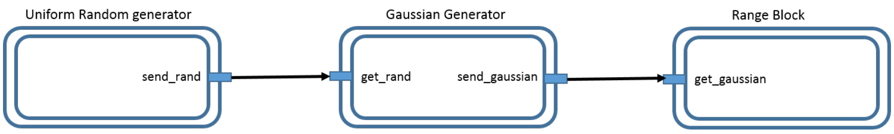


Figure 5.3: A application with three function blocks for generating Gaussian random values.

5.2.2 Contract analysis

List of contracts

1. Uniform Random Generator:

   (a) Functional:

   - PRECONDITION(send_rand.is_connected()) : This contract is related to the connectivity of the output port of the block.
   - POSTCONDITION(0<=this->two_rand.urand[0] && this->two_rand.urand[0]<=1)
   - POSTCONDITION(0<=this->two_rand.urand[1] && this->two_rand.urand[1]<=1)

   In the last two contracts above, we check whether the two random numbers that are generated indeed belong to $[0,1]$ because otherwise the Box-Muller transformation will fail.

   (b) Real-time:

   - **C1** with $WCET = 10000$
   - **C2** with $WCET = 10000$ and $l = 3$

- **C5** with $\pi = 0.97$ and $l = 3$
- **C6** with $\pi = 0.97$ and $h = 10$
- **C7** with $\pi = 0.97$

2. Gaussian Generator

   (a) Functional:

   - PRECONDITION(get_rand.is_connected())
   - PRECONDITION((*get_rand).urand[0]>=0 && (*get_rand).urand[0]<=1)
   - PRECONDITION((*get_rand).urand[1]>=0 && (*get_rand).urand[1]<=1)
   - PRECONDITION(send_gaussian.is_connected())
   - POSTCONDITION(this->two_gaussian.gaussrand[0]<=0)

   In the last contract above, we check whether both the Gaussian values generates belong to $[-\infty, 0]$. This is an example of an application specific contract. The rest of the contracts check the connectivity of the input port, asserts that the random numbers received at the port get_rand belong to $[0, 1]$ and checks the connectivity of the output port respectively.

   (b) Real-time:

   - **C1** with $WCET = 200000$
   - **C2** with $WCET = 200000$ and $l = 3$
   - **C5** with $\pi = 0.97$ and $l = 3$
   - **C6** with $\pi = 0.97$ and $h = 10$
   - **C7** with $\pi = 0.97$

3. Range Block

   (a) Functional:

   - PRECONDITION(get_gaussian.is_connected())
   - POSTCONDITION(range>=0)

   In the second contract above, we check whether the range is correctly computed and is always greater than or equal to zero.

   (b) Real-time:

   - **C1** with $WCET = 20000$
   - **C2** with $WCET = 20000$ and $l = 3$
   - **C5** with $\pi = 0.97$ and $l = 3$
   - **C6** with $\pi = 0.97$ and $h = 10$
   - **C7** with $\pi = 0.97$

4. Scheduler level contracts:

   - **C8**
   - **C9** with $J = 100000$ ns

Table 5.7: Case study 2-contract summary at block level

Block		# pre	# post	# I		# L_V	TOTAL
				# C_I	# L_I		
Random Generator	functional	1	2	0	0	0	3
	real-time	0	5				5
Gaussian Generator	functional	4	1	0	0	0	5
	real-time	0	5				5
Range Block	functional	1	1	0	0	0	2
	real-time	0	5				5

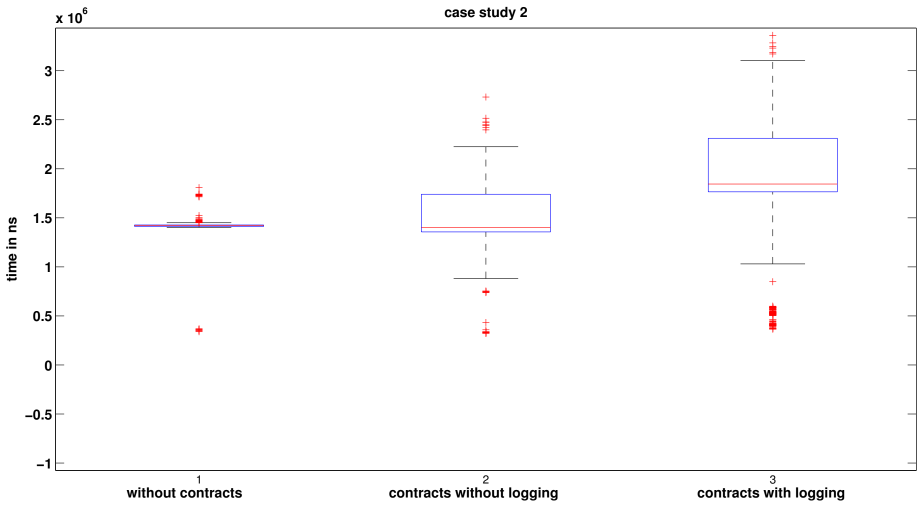


Figure 5.4: Box plot showing the execution times over 10000 cycles for case study 2 for 1) the original application, 2) with only contracts enabled and 3) with contracts enabled along with logging; the cycle time being 10 ms.

Summary table

Table 5.7 summarizes the contracts for case study 2.

Performance Analysis

In figure 5.4, we show the execution times of case study 2 executed for 10000 cycles. Table 5.8 shows the results of the analysis.

Table 5.8: Performance analysis of case study 2

enabled features	mean execution time	overhead
without contracts	1.4202 ms	-
contracts without logging	1.5161 ms	6.75%
contracts with logging	2.0114 ms	41.63%

5.3 Binary Search Application

5.3.1 Application description

In this application, there are five blocks. In each cycle, a *Random_Integer_Generator* block generates a random integer. This random number is then sent to a second block, *Create_Array_Block* which creates a dynamic array to store this number. It then sends the array to another block, *Sort_Array_Block* for sorting in ascending order. The sorted array is sent to *Binary_Search_Block*. This block has another input port where it receives a random number from a second instance of the *Random_Integer_Generator* block. It then searches for this random number in the sorted array that it receives from *Sort_Array_Block* and shows the index of the number if it is found, otherwise it prints -1 as the index. Algorithm 3 in appendix B shows the mechanism of binary search for reference. This completes one cycle. After every 10^{th} cycle, the memory allocated to the array is cleared. Thus, the maximum size of the array can be 10. Figure 5.5 illustrates this application. This application highlights the use of *parametric temporal contracts* among other features of the contract framework.

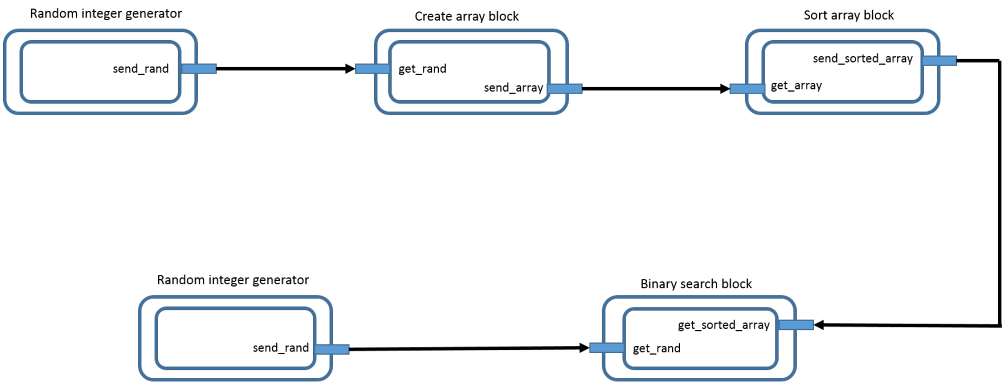


Figure 5.5: An application with 5 function blocks.

5.3.2 Contract analysis

List of contracts

1. Random Integer Generator:
   (a) Functional:
      - PRECONDITION(this->send_rand.is_connected())
   (b) Real-time:

- **C1** with $WCET = 70000$
- **C2** with $WCET = 70000$ and $l = 5$
- **C5** with $\pi = 0.98$ and $l = 5$
- **C6** with $\pi = 0.98$ and $h = 5$
- **C7** with $\pi = 0.98$

2. Create Array Block:

   (a) Functional:

   - PRECONDITION(this->get_rand.is_connected())
   - PRECONDITION(this->send_array.is_connected())
   - INVARIANT(count<=10)
   - LOOP_VARIANT(count-z)
   - LOOP_INVARIANT(z<=count)

   The variable `count` keeps track of the size of the array formed. Initially the value of `count` is 0 and it increments in every cycle until its value is equal to 10. At every 10^{th} cycle, its value is reinitialized to 0 because as described in the previous section, the maximum size of the array can be 10. Thus, for this block an invariant is to ensure that the value of `count` is never more than 10. In order to send the values of the array to the next function block, we have a loop that iterates `count` number of times. The loop variant and loop invariant above are meant to ensure the correctness of this loop. The rationale behind them is the same as described in section 5.1.2.

   (b) Real-time:

   - **C1** with $WCET = 100000$
   - **C2** with $WCET = 100000$ and $l = 5$
   - **C5** with $\pi = 0.98$ and $l = 5$
   - **C6** with $\pi = 0.98$ and $h = 5$
   - **C7** with $\pi = 0.98$
   - **C10, item 1** with $t = 10000$ ns and $j = m + 1$, where B_j represents the current block. This contract states that the current block should start no later than 10000 ns from the start of the previous block.

3. Sort Array Block:

   (a) Functional:

   - PRECONDITION(this->get_array.is_connected())
   - PRECONDITION(this->send_sorted_array.is_connected())
   - LOOP_VARIANT((*get_array).size-x)
   - LOOP_INVARIANT(x<=(*get_array).size)
   - LOOP_VARIANT(this->received_array.size-y)
   - LOOP_INVARIANT(y<=this->received_array.size)

In this block, we have two loops. The first loop is to receive the values from *Create Array Block* and store them in an array for sorting. The second loop is meant for sending the sorted array to *Binary Search Block*. The loop variants and invariants above correspond to these two loops respectively.

(b) Real-time:

- **C1** with $WCET = 200000$
- **C2** with $WCET = 200000$ and $l = 5$
- **C5** with $\pi = 0.98$ and $l = 5$
- **C6** with $\pi = 0.98$ and $h = 5$
- **C7** with $\pi = 0.98$
- **C10, item 1** with $t = 15000$ ns and $j = m + 1$, where B_j represents the current block. This contract states that the current block should start no later than 15000 ns from the start of the previous block.
- **parametric real-time contract:** POSTCONDITION(EXEC_TIME_THIS_CYCLE<=f(this,received_array.size));

4. Binary Search Block:

(a) Functional:

- PRECONDITION(this->get_rand.is_connected())
- PRECONDITION(this->get_sorted_array.is_connected())
- LOOP_VARIANT((*get_sorted_array).size-y)
- LOOP_INVARIANT(y<=(*get_sorted_array).size)
- LOOP_VARIANT(high-low)
- LOOP_INVARIANT(low<=high)

In the above list, the last loop variant and invariant is for the loop that performs the binary search. The rest are similar to the ones explained before.

(b) Real-time:

- **C1** with $WCET = 500000$
- **C2** with $WCET = 500000$ and $l = 5$
- **C5** with $\pi = 0.98$ and $l = 5$
- **C6** with $\pi = 0.98$ and $h = 5$
- **C7** with $\pi = 0.98$
- **parametric real-time contract:** POSTCONDITION(EXEC_TIME_THIS_CYCLE<=f(this,sorted_array.size))

5. Scheduler level contracts:

- **C8**
- **C9** with $J = 100000$ ns

Summary table

In table 5.9, we present a summary of the contracts for case study 3.

Table 5.9: Case study 3-contract summary at block level

Block		**# pre**	**# post**	**# I**		# L_V	TOTAL
				# C_I	# L_I		
Random Integer Generator	functional	1	0	0	0	0	1
	real-time	0	5				5
Create Array Block	functional	2	0	1	1	1	5
	real-time	1	5				6
Sort Array Block	functional	2	0	1	2	2	7
	real-time	1	6				7
Binary Search Block	functional	2	0	0	2	2	6
	real-time	0	6				6

Performance Analysis

In figure 5.6, we show the execution times of case study 3 executed for 10000 cycles. Due to the presence of outliers, the median is a better measure of central tendency in this case. Thus, we consider the median values in the analysis.

It should be noted here that the outliers in the execution time that can be seen in figure 5.6 are *not* due to the contracts because they are present in the case *without* contracts as well. A possible explanation for observing the outliers could be the complexity of the computations being done in the function blocks. Regardless of the outliers, we can see that the overhead due to the contract framework is still negligible. Table 5.10 shows the result of the analysis.

Table 5.10: Performance analysis of case study 3

enabled features	median execution time	overhead
without contracts	1.5865 ms	-
contracts without logging	1.6761 ms	5.65%
contracts with logging	1.8725 ms	18.03%

5.4 Energy-Pack-Core-Model example

5.4.1 Application description

The code for this application is generated from a Simulink case study. This application has been explained in detail in chapter 2 and the function block diagram is given in figure 2.2. The rationale behind choosing this as a case study is that it gives us an idea regarding how the contract framework can be integrated smoothly with pre-existing code. Additionally, we also see how the "Old" construct works successfully for **struct** variables as well.

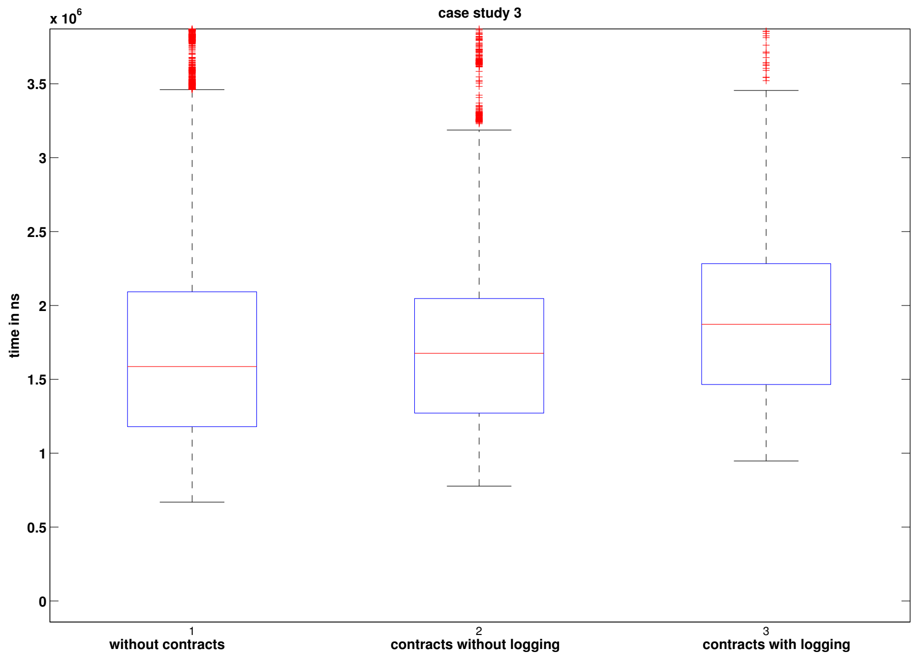


Figure 5.6: Box plot showing the execution times over 10000 cycles for case study 3 for 1) the original application, 2) with only contracts enabled and 3) with contracts enabled along with logging; the cycle time being 10 ms.

5.4.2 Contract analysis

List of contracts

1. Sensor Block:

   (a) Functional:

   - PRECONDITION(data_out_cc.is_connected())
   - PRECONDITION(data_out_ff.is_connected())
   - POSTCONDITION(this->state.interval_no==GET_OLD(unsigned int, state.interval_no)+1). In this contract, `interval_no` is an attribute of a structure, `state`, of type `unsigned int`. In the `operator()` method, this variable is incremented by one in every cycle, which explains this contract.

   (b) Real-time:

   - **C1** with $WCET = 250000$
   - **C2** with $WCET = 250000$ and $l = 7$
   - **C5** with $\pi = 0.95$ and $l = 7$
   - **C6** with $\pi = 0.95$ and $h = 5$
   - **C7** with $\pi = 0.95$

2. Feed Forward Block:

   (a) Functional:

   - PRECONDITION(data_in.is_connected())
   - PRECONDITION(data_out.is_connected())
   - INVARIANT(localB->Switch_g>=0). `localB` is a pointer to a structure variable with `Switch_g` as an attribute. In the `operator()` method, this variable is assigned the absolute value of another variable which means that its value cannot be negative. This is the rationale behind this invariant.

   (b) Real-time:

   - **C1** with $WCET = 210000$
   - **C2** with $WCET = 210000$ and $l = 7$
   - **C5** with $\pi = 0.95$ and $l = 7$
   - **C6** with $\pi = 0.95$ and $h = 5$
   - **C7** with $\pi = 0.95$
   - **C10, item 1** with $t = 20000$ ns and $j = m + 1$, where B_j represents the current block. This contract states that the current block should start no later than 20000 ns from the start of the previous block.

3. Current Control Block:

   (a) Functional:

   - PRECONDITION(data_in.is_connected())
   - PRECONDITION(data_out.is_connected())
   - PRECONDITION(data_in_ff.is_connected())
   - INVARIANT(this->state.TnIBatCtrol_TN!=0.0). `state` is a structure variable and `TnIBatCtrol_TN` is an attribute of the structure. This variable is used as a denominator in a division operation in the `operator()` method. This requries `state.TnIBatCtrol_TN` to be non-zero which is what this invariant checks.

   (b) Real-time:

   - **C1** with $WCET = 200000$
   - **C2** with $WCET = 200000$ and $l = 7$
   - **C5** with $\pi = 0.95$ and $l = 7$
   - **C6** with $\pi = 0.95$ and $h = 5$
   - **C7** with $\pi = 0.95$
   - **C10** with $t = 15000$ ns and $j = m + 1$, where B_j represents the current block. This contract states that the current block should start no later than 15000 ns from the start of the previous block.

4. Monitor Block:

   (a) Functional:

   - PRECONDITION(data_in.is_connected())

(b) Real-time:

- **C1** with $WCET = 100000$
- **C2** with $WCET = 100000$ and $l = 7$
- **C5** with $\pi = 0.95$ and $l = 7$
- **C6** with $\pi = 0.95$ and $h = 5$
- **C7** with $\pi = 0.95$
- **C10** with $t = 10000$ ns and $j = m + 1$, where B_j represents the current block. This contract states that the current block should start no later than 10000 ns from the start of the previous block.

5. Scheduler level contracts:
   - **C8**
   - **C9** with $J = 100000$ ns

Summary table

In table 5.11, we can see a summary of the contracts for case study 4.

Table 5.11: Case study 4-contract summary at block level

Block		**# pre**	**# post**	**# I**		# L_V	TOTAL
				# C_I	# L_I		
Sensor	functional	2	1	0	0	0	3
	real-time	0	5				5
Feed Forward	functional	2	0	0	1	0	3
	real-time	1	5				6
Current Control	functional	3	0	1	0	0	4
	real-time	1	5				6
Monitor	functional	1	0	0	0	0	1
	real-time	1	5				6

Performance Analysis

In figure 5.7, we show the execution times of case study 4 executed for 10000 cycles. As we can see, there is some jitter introduced by the contract framework. We use the median to analyze this case study. Table 5.12 shows the summary of the analysis.

Table 5.12: Performance analysis of case study 4

enabled features	median execution time	overhead
without contracts	1.7467 ms	-
contracts without logging	1.8657 ms	6.81%
contracts with logging	2.2961 ms	31.45%

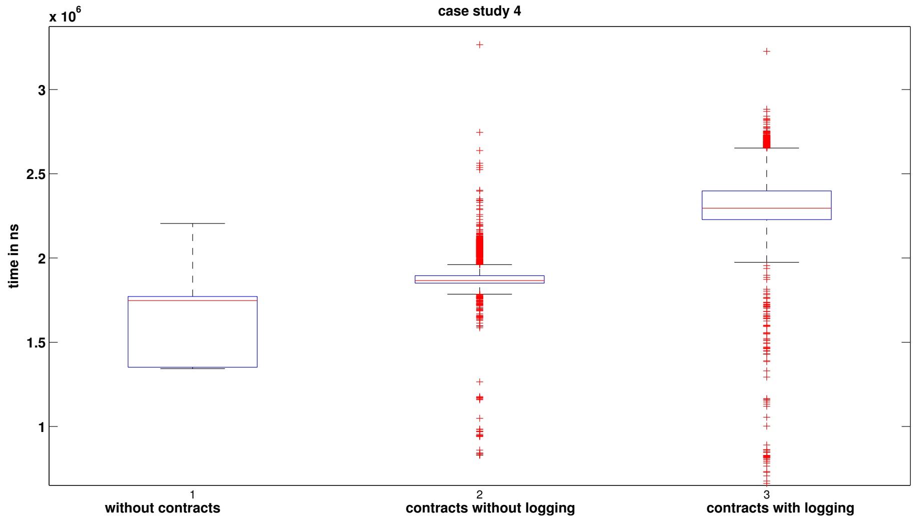


Figure 5.7: Box plot showing the execution times over 10000 cycles for case study 4 for 1) the original application, 2) with only contracts enabled and 3) with contracts enabled along with logging; the cycle time being 10 ms.

Table 5.12 shows that even though the contract framework added some jitter to case study 4 which can be seen in figure 5.7, its effect on the over all performance of the application is negligible.

5.5 NetProxy Application

5.5.1 Application description

In this case study, there are two separate applications launched on two separate hosts. The first application is a sender application with a `sender` block and a `net-proxy` block which sends data from the sender through the network. The second application is a receiver application with a `net-receive` block and a `receiver` block which are deployed on another host computer. The FASA kernel is launched on both the hosts. As can be seen from figure 5.8, **Application 1** is deployed on **Host 1** and **Application 2** is deployed on **Host 2**. The *sender* sends data to the *net_proxy_send* block. The latter communicates with *net_proxy_receive* over the local network and the *receiver* block receives the data from the *net_proxy_receive* block. The two net_proxy blocks belong to the FASA framework and can be used *off the shelf*. In order to synchronize the clock on the two hosts, the Precision Time Protocol(PTP) is implemented by the FASA platform. Before starting the FASA kernel on the hosts, we first need to start a PTP daemon in them. This application mainly highlights the real-time contracts in case of communication through a network proxy. This application illustrates how the communication delay is incorporated in the contract framework through the network.

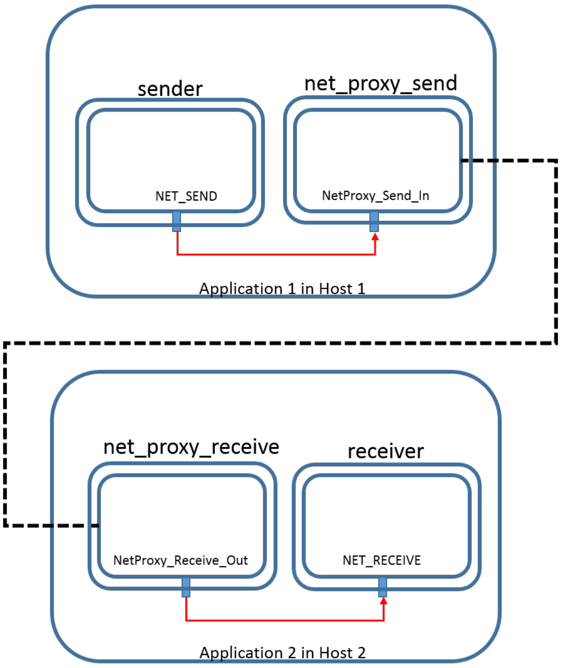


Figure 5.8: An application with communication through network proxy.

5.5.2 Contract analysis

List of contracts

1. Sender:

   (a) Functional:

   - PRECONDITION(NET_SEND.is_connected())
   - POSTCONDITION(this->sizeToSend==GET_OLD(int,sizeToSend)+1 || this->sizeToSend==STRING_MAX/2)
   - INVARIANT(this->sizeToSend<=STRING_MAX/2). The sender block is not supposed to send a value larger than STRING_MAX/2, where STRING_MAX is a macro whose value is set to 1024. This is the rationale behind having this invariant.

   (b) Real-time:

   - **C1** with $WCET = 500000$
   - **C2** with $WCET = 500000$ and $l = 10$
   - **C5** with $\pi = 0.99$ and $l = 10$
   - **C6** with $\pi = 0.99$ and $h = 10$
   - **C7** with $\pi = 0.99$

2. Receiver:

   (a) Functional:

   - PRECONDITION(NET_RECEIVE.is_connected())
   - POSTCONDITION(this->sizeToReceive==GET_OLD(int,sizeToReceive)+1 || this->sizeToReceive==STRING_MAX/2). The value received by the receiver in a cycle should either be one more than the value it had received in the previous cycle or it should be STRING_MAX/2, because STRING_MAX/2 is the maximum value that the sender can send.

   (b) Real-time:

   - **C1** with $WCET = 500000$
   - **C2** with $WCET = 500000$ and $l = 10$
   - **C5** with $\pi = 0.99$ and $l = 10$
   - **C6** with $\pi = 0.99$ and $h = 10$
   - **C7** with $\pi = 0.99$
   - **C10, item 2** with $t = 200000$ ns. This states that the receiver block should finish executing no later than 200000 ns from the start of the sender block.

3. Scheduler level contracts:
   - **C8**
   - **C9** with $J = 100000$ ns

Summary table

In table 5.13, a summary of the contracts for case study 5 is presented.

Table 5.13: Case study 5-contract summary

Block		**# pre**	**# post**	**# I**		**# L_V**	TOTAL
				# C_I	# L_I		
Sender	functional	1	1	1	0	0	3
	real-time	0	5				5
Receiver	functional	1	1	0	0	0	2
	real-time	0	6				6

Performance Analysis

For the analysis of the contract framework for this case study, we separately illustrate the performance of the two FASA application in the two hosts, for the case when contract are disabled and when they are enabled (with and without FASA logging). Figure 5.9 shows the performance of the sender application while figure 5.10 shows the performance of the receiver application. Tables 5.14 and 5.15 present the performance analysis of the sender and receiver blocks respectively.

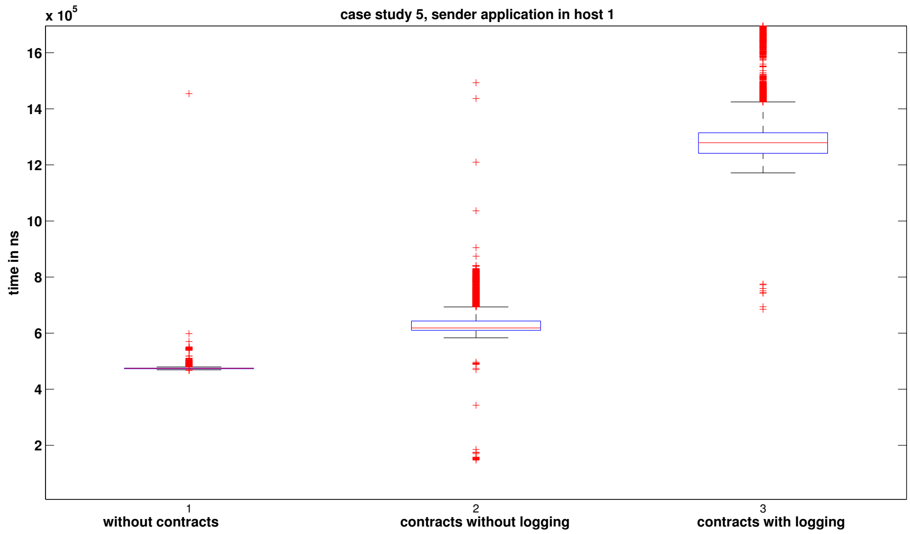


Figure 5.9: A sender application with communication through network proxy.

Table 5.14: Performance analysis of case study 5, sender application

enabled features	mean execution time	overhead
without contracts	0.4746 ms	-
contracts without logging	0.6359 ms	33.99%
contracts with logging	1.3006 ms	too large overhead

5.6 Conclusions

The contract framework has been validated using the 5 case studies described above. As described in chapter 3, our overhead tolerance limit is 10% [23].

1. The average overhead due to the contract framework (without any logging) for the first four case studies is 5.38% and the maximum observed value is 6.81%. Both these values are well within our overhead tolerance limit.

   Considering that the cycle time for these four case studies was 10 ms, a 5.38% overhead would amount to an overhead of 0.54 ms which is negligible.

2. In the fifth case study, the overhead is much more than 10% even without logging because of the fact that the function blocks communicate through the network. Since

Table 5.15: Performance analysis of case study 5, receiver application

enabled features	mean execution time	overhead
without contracts	0.8016 ms	-
contracts without logging	1.1531 ms	43.85%
contracts with logging	1.3677 ms	too large overhead

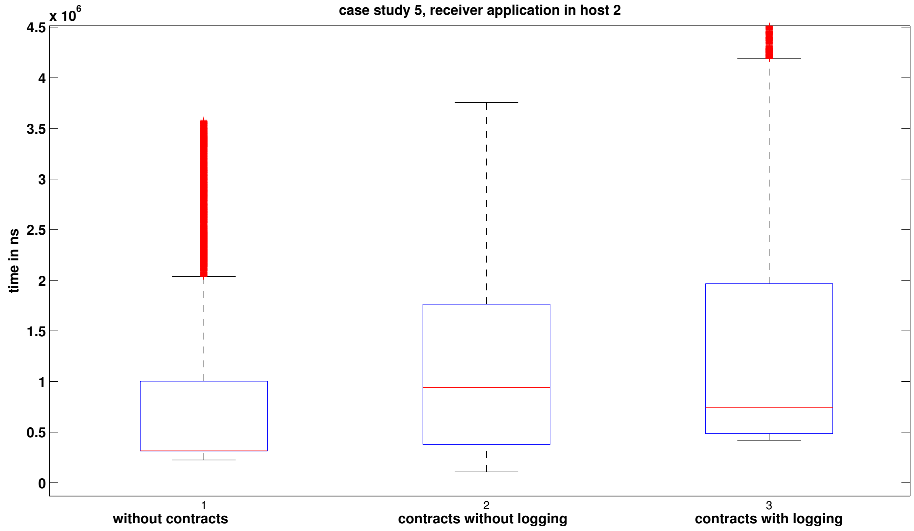


Figure 5.10: A receiver application with communication through network proxy.

some of the contracts are computed based on data received over the network, any communication delay results in an increase in the execution time of the application.

Overhead due to the communication delay cannot be avoided whether contracts are enabled or disabled.

3. It is observed that when the logging feature of FASA is enabled, the mean overhead for the first four case studies is 28.25%. This result however has nothing to do with the contract framework because these are application log messages [42] which were already there. The messages related to the failure of the contracts are logged in the *slack* time. As a result, they have no impact on the performance of the application.

Now, referring to the four requirements of the contract framework stated in chapter 3, table 5.16 gives a summary which illustrates the fulfillment of the requirements. Only for case study 5 (involving network communication), the framework failed to meet the requirement of having less than 10% overhead. Apart from that, the contract framework passed all the remaining test cases.

Table 5.16: Requirement fulfillment analysis

Test Case \ Req	**Req1**	**Req2**	**Req3**	**Req4**
case study 1	passed	passed	passed	passed
case study 2	passed	passed	passed	passed
case study 3	passed	passed	passed	passed
case study 4	passed	passed	passed	passed
case study 5	passed	passed	passed	failed

6 Conclusions and Future Work

Here, we present a summary of the entire work of the thesis and mention some possible extensions to this work.

6.1 Conclusions

In this thesis, we have developed a contract framework for a real-time platform, FASA. We formalized the framework using RTL, validated it on 5 different types of case studies and analyzed its performance for each case study.

6.1.1 Major Contributions

There are three major contributions of this thesis.

Development of a contract framework A contract framework for real-time control applications is developed in this thesis. We investigated two different approaches for developing the contract framework, *dedicated blocks* and *dedicated routines*. Based on experimental results, the *dedicated routines* approach proved to be *twice* as efficient as the *dedicated blocks* approach. The framework supports both functional and temporal contracts and is very flexible. It can be turned on during the debug mode while in the production code, it can be turned off. In order to keep the overhead due to the contracts to the bare minimum, failed contract messages are logged during the *slack* period to avoid having any effect on the execution time of the application. The framework is validated using five benchmarks and experiments have shown that the overhead due to the contract framework is less than 10% for applications deployed on the same host machine. In terms of functional contracts, the framework supports pre and post conditions, class invariants, loop variants and loop invariants. It also simulates the "Old" construct of Eiffel for monitoring variables. The temporal part of the framework supports contracts related to WCET, cycle time and jitter. The framework handles two levels of temporal contracts: block level (WCET) and schedule level (cycle time and jitter).It also allows the users to define *parametric temporal-contracts* which are functions of data-structures used in the blocks.

Stochastic temporal contracts A novel approach based on *empirical cdf* is used to dynamically estimate statistical parameters of the execution time-probability distributions. These estimates are incorporated in computing future temporal contracts. When static analysis tools are not avaliable and the WCET values of the function blocks are unknown, this approach is highly suitable for determining upper bounds on the execution times. Function blocks can have varied behaviors and often, a known probability distribution cannot be used to fit the probability distributions of their execution times. In such settings, the *empirical cdf* approach is ideal for subsuming the behavior of the blocks.

Using RTL specifications for formally defining statistical properties This thesis illustrates the use of RTL for formalizing stochastic temporal properties of a system. While RTL has been used in the past for system specifications, it has not been exploited for formalizing statistical properties.

6.1.2 Results

This thesis has presented a contract framework for the FASA platform for dynamic verification of real-time control applications. Our experiments have shown that the framework adds a very low overhead to the platform (5.38% on an average), when the function blocks do not communicate over the network.

Two novel contributions of the thesis are the use of an *empirical cdf* based approach for computing complex temporal contracts dynamically while having negligible effect on performance and the use of RTL for formal specification of statistical timing properties of cyclic hard real-time applications.

6.2 Future work

The work presented in this thesis can be extended further in certain ways. The first thing would be to test it on many more benchmarks and allow users to use it more. It will tell us how user-friendly the contract framework is. This is very important because the primary reason why contracts are usually not included in programs is that programmers find it tiresome and time consuming. They would rather debug the code later than add contracts in the development phase itself. The main objective of *correctness by construction* is to avoid this. Having an easy-to-use contract framework is a step towards this goal.

The framework can be futher extended by enabling the computation of other real-time contracts. Also, here we have used *statistical inference* to estimate the population parameters of the distributions of the execution times. One could use other machine learning techniques such as *neural networks*. That would however have a much higher overhead and it would then be difficult to perform the computations at runtime. It would be more suitable to perform the learning offline in that case.

Another direction that could be explored later is to automatically generate stochastic real-time contracts using pre-existing tools. For instance, Daikon [17] could be extended to allow the automatic generation of temporal contracts based on statistical techniques. It would then be interesting to compare these stochastic contracts with the ones defined by the application developers [36]. Further, formal verification could also be done as a part of the extension to this work using model checking tools such as CBMC [14]. However, this poses some issues regarding the compatibility of the tools with C++ code. The FASA framework and all its applications are written in C++ and when tools like CBMC are used with C++ code, it does not always work as expected. This is because many C++ standard header files and namespaces and are not recognized by the parsers used in these tools.

A Box-Muller Transform

The Box Muller transform is used for generating iid random numbers according to $N(0,1)$ distribution from $U[0,1]$ random numbers. The form of Box-Muller transformation used in the Gaussian Random Generator case study takes two samples from $U[0,1]$ and generates two iid $N(0,1)$ samples.

Let U_1 and U_2 be two iid random variables$\sim U[0,1]$. Their pdf is gives by:

$$f(x) = \begin{cases} 1 & 0 \leqslant x \leqslant 1 \\ 0 & \text{elsewhere} \end{cases}$$

The Box-Muller transformation is given as:

$$Z_0 = \sqrt{-2lnU_1}cos(2\pi U_2)$$
$$Z_1 = \sqrt{-2lnU_1}sin(2\pi U_2),$$

where Z_0 and Z_1 are iid standard normal variates,$N(0,1)$ with pdf:

$$f(x,\mu,\sigma) = \begin{cases} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} & -\infty < x < \infty \end{cases}$$

where $\mu = 0$ and $\sigma^2 = 1$.

B Binary Search Algorithm

The algorithm for Binary Search which is used in the Binary Search case study is as follows:

Algorithm 3 Binary Search

procedure BINARYSEARCH(*array*, *number*)
    $size \leftarrow$ size of $array$
    $low \leftarrow 0, high \leftarrow size, index = -1$
    while $low < high$ **do**
        $mid \leftarrow \frac{low+high}{2}$
        if $array[mid] == number$ **then**
            $index = mid$
        else if $array[mid] < number$ **then**
            $low = mid + 1$
        else
            $high = mid - 1$
        end if
    end while
    return $index$
end procedure

Bibliography

[1] Design by contract, May 2014. URL http://c2.com/cgi/wiki?DesignByContract.

[2] Parameter pack, 2014, URL http://en.cppreference.com/w/cpp/language/parameter_pack.

[3] Empirical distribution function, 2014, URL http://en.wikipedia.org/wiki/Empirical_distribution_function.

[4] Variadic macros, 2014, URL http://msdn.microsoft.com/en-us/library/ms177415.aspx.

[5] Code contracts, May 2014. URL http://research.microsoft.com/en-us/projects/contracts/.

[6] Framework for real-time embedded systems based on contracts, 2014, URL http://www.frescor.org/index.php?page=FRESCOR-homepage.

[7] Object constraint language (ocl), 2014, URL http://www.omg.org/spec/OCL/.

[8] Using design by contract to automate java software and component testing, 2014, URL http://www.parasoft.com/products/article.jsp?articleId=579&product=Jtest.

[9] Spark 2014, expanding the boundaries of safe and secure programming, May 2014. URL http://www.spark-2014.org/about/.

[10] Mario Barbacci and Jeannette M. Wing. Specifying functional and timing behavior for real-time applications. In *Proceedings of the Parallel Architectures and Languages Europe, Volume I*, pages 124–140, London, UK, UK, 1987. Springer-Verlag.

[11] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, March 2000.

[12] Albert Benveniste, Benoit Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim G. Larsen. Contracts for System Design. Rapport de recherche RR-8147, INRIA, November 2012.

[13] Marshall Brain. How airbags work, 2014, URL http://auto.howstuffworks.com/car-driving-safety/safety-regulatory-devices/airbag1.htm.

[14] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[15] Yannick Moy Cyrille Comar, Johannes Kanig. Integrating formal program verification with tesing, 2014, URL http://www.adacore.com/uploads_gems/Hi-Lite_ERTS-2012.pdf.

[16] Beman Dawes. Boost c++ libraries, 2014, URL http://www.boost.org/doc/libs/1_55_0/libs/libraries.htm.

[17] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[18] C.J.M. Geisterfer and S. Ghosh. Software component specification: a study in perspective of component selection and reuse. In *Commercial-off-the-Shelf (COTS)-Based Software Systems, 2006. Fifth International Conference on*, pages 9 pp.–, Feb 2006.

[19] Hermann Härtig, Steffen Zschaler, Martin Pohlack, Ronald Aigner, Steffen Göbel, Christoph Pohl, and Simone Röttger. Enforceable component-based realtime contracts. *Real-Time Syst.*, 35(1):1–31, January 2007.

[20] Dimitri Van Heesch. Log for c++ project, 2014, URL http://log4cpp.sourceforge.net/.

[21] Kevlin Henney. Boost c++ libraries, 2014, URL http://www.boost.org/doc/libs/1_55_0/doc/html/any.html.

[22] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[23] Xiaowan Huang, Justin Seyster, Sean Callanan, Ketan Dixit, Radu Grosu, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. Software monitoring with controllable overhead. *Int. J. Softw. Tools Technol. Transf.*, 14(3):327–347, June 2012.

[24] F. Jahanian and AK. Mok. Safety analysis of timing properties in real-time systems. *Software Engineering, IEEE Transactions on*, SE-12(9):890–904, Sept 1986.

[25] Farnam Jahanian, Aloysius K. Mok, and Douglas A. Stuart. Formal specification of real-time systems. Technical report, Austin, TX, USA, 1988.

[26] John C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, New York, NY, USA, 2002. ACM.

[27] Leslie Lamport. What good is temporal logic? In *IFIP Congress*, pages 657–668, 1983.

[28] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language euclid. *SIGPLAN Not.*, 12(2):1–79, February 1977.

[29] Gary T. Leavens and Yoonsik Cheon. Design by contract with jml, 2006.

[30] Ravichandhran Madhavan and Viktor Kuncak. Symbolic Resource Bound Inference. Technical report, 2014.

[31] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[32] A Mazouz, S. Touati, and D. Barthou. Study of variations of native program execution times on multi-core architectures. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 919–924, Feb 2010.

[33] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[34] Manuel Oriol, Michael Wahler, Robin Steiger, Sascha Stoeter, Egemen Vardar, Heiko Koziolek, and Atul Kumar. Fasa: A scalable software framework for distributed control systems. In *Proceedings of the 3rd International ACM SIGSOFT Symposium on Architecting Critical Systems*, ISARCS '12, pages 51–60, New York, NY, USA, 2012. ACM.

[35] Ricardo Perrone, Raimundo Macedo, George Lima, and Veronica Lima. An approach for estimating execution time probability distributions of component-based real-time systems. *J. UCS*, 15(11):2142–2165, 2009.

[36] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 93–104, New York, NY, USA, 2009. ACM.

[37] G. L. Reijns and A. J. C. van Gemund. Predicting the execution times of parallel-independent programs using pearson distributions. *Parallel Comput.*, 31(8-9):877–899, August 2005.

[38] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems*. *European Journal of Control*, 18(3):217 – 238, 2012.

[39] Edmond Schonberg. Towards ada 2012: An interim report. In *Proceedings of the ACM SIGAda Annual International Conference on SIGAda*, SIGAda '10, pages 63–70, New York, NY, USA, 2010. ACM.

[40] Joseph Sifakis. Modeling real-time systems - challenges and work directions. In *In Proceedings of the 1st International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science*, pages 373–389. Springer Verlag, 2001.

[41] M. Sojka and Z. Hanzalek. Modular architecture for real-time contract-based framework. In *Industrial Embedded Systems, 2009. SIES '09. IEEE International Symposium on*, pages 66–69, July 2009.

[42] Iosif Spulber. Lightweight monitoring for distributed control systems, 2013, Master Thesis, ABB Corporate Research.

[43] I Stierand, P. Reinkemeier, T. Gezgin, and P. Bhaduri. Real-time scheduling interfaces and contracts for the design of distributed embedded systems. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 130–139, June 2013.

[44] M. Wahler, M. Oriol, E. Ferranti, and A. Monot. Reconciling flexibility and robustness in industrial automation systems, and living happily ever after. In *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–8, Sept 2013.

[45] Jeannette M. Wing. Writing larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9:1–24, 1987.

[46] H. Conrad Cunningham Yi Liu. Software component specification using design by contract.

Chandrakana Nandi

Pflugstrasse 1
Zurich, 8006
(+41) 76 290 14 64

PERSONAL

Nationality: Indian
Email: chandrakana.nandi@epfl.ch
Website: https://epfl.academia.edu/ChandrakanaNandi

EDUCATION

Master of Science, Computer Science
École Polytechnique Fédérale de Lausanne(EPFL), 1024 Lausanne, Switzerland
Expected: August 2014
Thesis: Contracts for Real-Time, Safety Critial Systems, Supervisors: Prof. Viktor Kuncak, LARA, EPFL, Dr. Manuel Oriol, ABB Corporate Research

Bachelor of Science, Statistics, Mathematics and Computer Science
Banaras Hindu University (BHU), Varanasi 221005, India, June 2012
Concentration: Statistics
Thesis: Social Network-based Analysis of Behavior, Supervisor: Prof. R.D Singh
GPA: 9.68/10 , Valedictorian, Faculty of Science, BHU, 2012

COMPUTER SKILLS

Languages: C, C++, Java, Python (basic), C# (basic)
Database: SQL, XML
Operating Systems: Unix, Windows 7, 8
Web development: HTML, XML, Javascript, PHP
Software: Eclipse, MATLAB, Visual Studio, Unity3D, LaTeX
Libraries: OpenCV, AruCo, Bullet Physics

EXPERIENCE

Masters Thesis student Feb'14-Aug'14
ABB Corporate Research Center, Baden, Switzerland
Supervisor: Dr. Manuel Oriol

- Development of a contract framework for the FASA platform

Software Intern Aug'13-Jan'14
ABB Corporate Research Center, Baden, Switzerland
Supervisor: Dr. Manuel Oriol

- Development of a bi-directional model transformation tool between two state-of-the-art component based frameworks, BIP and FASA.

Summer Intern at BIOROB, EPFL Jun'10-Jul'10
Supervisor: Prof. Auke J. Ijspeert

- Analysis of the locomotion of a salamander from X-Ray movies
- Obtaining a graphical representation of the temporal variations of the angles at different joints on the salamander's body.

MAJOR PROJECTS

1. Semester Project: Using business rules for coordinating OSGI applications with the Behavior Interaction Priority (BIP) framework.
   Supervisor: Prof. Joseph Sifakis, Turing Award 2007, Head, Rigorous System Design Lab, EPFL.

2. Recognition of 3D images from the small NORB dataset.
   Instructor: Prof. Mathias Seeger, Head, Laboratory of Probabilistic Machine Learning, EPFL.
3. Developing a 3D bouncing ball game.
   Instructor: Prof. Ronan Boulic, Immersive Interaction Group, EPFL
4. Bachelors Thesis: Analysis of a dynamic social network data.
   Supervisor: Prof. R. D Singh, BHU

ACADEMIC ACHIEVEMENTS

1. Received 5 awards including 3 Gold medals in the 95^{th} convocation of BHU
   - Topper of the Faculty of Science, BHU
   - Topper of the Department of Statistics, BHU
   - Female Topper in Faculty of Science, BHU
   - Dr. Basudeo Sahni Gold Medal
   - Cash award and university scholarship holder for academic excellence
2. Awarded the Swiss Government Scholarship for September 2012-2014 for pursuing masters in computer science at EPFL
3. Selected for the M.Sc Research Scholar Program of the School of Computer and Commmunication Sciences at EPFL, by Prof. Joseph Sifakis at the Rigorous System Design Lab.
4. Secured All India Rank 14 in the IIT-Joint Admission Test for Mathematical Statistics in 2012 for graduate studies.
5. Attended Microsoft Theory Day, 2010 at IIT-Madras.
6. Accepted as a summer intern at Indian Institute of Information Technology, Allahabad in May, 2010.

EXTRA-CURRICULAR ACTIVITIES

1. Professionally trained Bharatnaytam dancer
2. Won first prize in Web designing in the tech fest TORQUE in 2010 conducted by Department of Computer Science, BHU
3. Reached the Semi-finals of Microsoft Imagine Cup-Worldwide Digital Media Contest-2010
4. Qualified for round 2 in ACM-ICPC coding contest in December 2010 and awarded ACM student membership
5. Member of the organizing team of a National Conference and Workshop on High Performance Computing and Applications and Graph and Geometric Algorithms organized by Banaras Hindu University from 08-02-1010 to 13-02-2010
6. Vice captain during my high school
7. Member of Student's editorial board in high school

LANGUAGE PROFICIENCY

English: fluent, TOEFL score 110/120, October 2013
Bengali: mother tongue
Hindi: fluent
French: basic