Formal Design and Verification of Digital PID Gain Scheduling Controllers

A Model Checking Approach

The University Of Sheffield.

Pablo Armando Ordóñez Aguileta

Department of Engineering
University of Sheffield

This dissertation is submitted for the degree of
Doctor of Philosophy

February 2018

I would like to dedicate this thesis to my loving parents Pedro and Socorro. …

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation, entitled **"Formal Design and Verification of Digital PID Gain Scheduling Controllers"**, is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Pablo Armando Ordóñez Aguileta
February 2018

Acknowledgements

My sincerest gratitude to my adviser, Tony Dodd, I could not have done any of this without his support and encouragement throughout this long journey. His continuous guidance, advice, and perspective made this possible even when I thought it was not. I will be forever in debt with you Tony.

I would like to thank Andy Mills and Jun Liu for continuously providing feedback, perspective, and more importantly, support during the development of this thesis. You have been wonderful mentors and both of you played a fundamental role for the development of this work.

I am truly honoured to had been part of the ACSE UTC group during my doctoral studies. Not only did I have colleagues but I made friends, a wonderful group of people willing to help or discuss ideas, especially if that meant a quick visit to The Red Deer. Special thanks to Andrew, Masz, Chris, Romain, Kacper, Ibrahim, and Dan. That being said, thanks to The Red Deer for always being open when I needed it.

To all my wonderful friends in this side of the world, thank you for being my family away from home: Ariel, Debbie, AndyLú, Toño, Alepa, Mario, Jesús Alejandro, Yessi, Maza, Robbie, Matamoros, Jaimito, Yess, Aldo, Harry, Oli, Jeff, Liliana, Celeste, Jorgito, Anna-Lena, Samantha, Matei, Larín, and Antito. To Raquelín for helping me dance my problems away at the very end of this journey. To Lucy Kemp for being so supportive during the homestretch. To Matou for helping me so much when I needed it the most. A Karlira por quererme cuando menos lo esperaba.

And last but not least, I would like to thank my family. They have always supported me in whatever new challenge I took. To my dad and mom for everything. To my brother Pedrito for always being there for me. To my uncle Pablo and my aunt Sonia, for being my parents all over again. To my cousins Sonia, Made, and Pabloco, because you are basically my siblings. My parents always thought that education and knowledge could open any door, this is a testament to that idea.

Abstract

The verification process of embedded systems is fundamental for their correct development. Embedded control is a popular choice among the engineering community, making the relationship between control systems and computer science very close. Gain scheduling is a typical approach for safety-critical systems (e.g. jet-engines). It is preferred due to a known route to certification. Nonetheless, stability and performance are hard to prove analytically. Consequently, safety and airworthiness are achieved by extensive testing, and therefore a new way for verification is desirable.

Model checking, an exhaustive verification technique, is a part of formal methods. Model checking can aid in detecting ambiguities and collisions in requirements, increasing and improving testing coverage and error-detection rate. However, there are still limitations and challenges to model checking. The *state-space explosion* problem limits its use to realistic dynamic control systems: Computational memory runs out or available data types are not appropriate for modelling.

This thesis addresses the formal design and verification of discrete PID gain-scheduled control systems. By the means of a novel abstraction methodology the control problem is resolved in a model checking environment; formally tuning the controller whilst systematically constructing a control schedule. The work in this overcomes typical constraints imposed by model checking. In this manner, the gain-scheduled controller can be efficiently generated and the resulting schedule is *correct-by-construction* with respect to high level performance requirements. This novel methodology incorporates computer science and control systems tools, proposing an *a priori* verification approach in contrast to current *a posteriori* testing activities. By combining computer science and control engineering, the gap between formal methods and control systems is reduced.

The next step in this line of research is to analyse the scalability of the approach using more realistic models and design cases; in this manner the *state-space explosion* problem can be addressed with a divide and conquer approach. Also, a trade-off analysis between benefits and the required effort learning the new approach in a real development cycle must be conducted to assess feasibility and capabilities of the approach.

Table of contents

List of figures

List of tables

Symbols and Acronyms

Symbols

b_0	Discrete PID coefficient zero
b_1	Discrete PID coefficient one
K_D	Controller Derivative gain
K_I	Controller Integral gain
K_P	Controller Proportional gain
T	System sampling period
δ	Gain delta
Δ_{K_I}	Controller integral gain total delta change
$\Delta_{K_I}SS$	Controller integral gain delta change step size
Δ_{K_P}	Controller proportional gain total delta change
$\Delta_{K_P}SS$	Controller proportional gain delta change step size
ε	Error
E(z)	Discrete-time System Error
F	Fractional Digit
I	Integral Digit
K_{ab}	Coefficients Scaling Gain
K	System gain

K_S	Input-Output Scaling Gain
K_U	Parametric Compensation Error Gain
ω_n	System natural frequency
A	Path quantifier - For all computation paths
E	Path quantifier - For some computation paths
<>	*Eventually* temporal operator
[]	*Globally* temporal operator
φ	State formula - Logical expression that can be evaluated in a state
$\mathbb{R}$	Real Numbers
R(z)	Discrete-time System Reference
e_{ss}	Steady State Error
θ	System transport delay
t_r	Rise Time
t_s	Settling Time
OS	UPPAAL query language Maximum Overshoot indicator
RT	UPPAAL query language Rise Time indicator
SSE	UPPAAL query language Steady State Error indicator
ST	UPPAAL query language Settling Time indicator
U(z)	Discrete-time System Input
Y(z)	Discrete-time System Output
$\mathbb{Z}^+$	Positive Numbers
ζ	System damping ratio

Acronyms

ARX	Auto-Regressive with Exogenous Input

CTL Computational Tree Logic

CTS Continuous Time Systems

DCS Distributed Control System

DES Discrete Event Systems

DO-178C Software Considerations in Airborne Systems and Equipment Certification

DO-333 Formal Methods Supplement to DO-178C and DO-278A

DPM Dynamic Power Management

EASA European Aviation Safety Agency

EEC Electronic Engine Control

FAA Federal Aviation Administration

FAR Federal Aviation Regulations

FBW Fly By Wire

FSM Finite State Machines

LTI Linear Time Invariant

LTL Linear Temporal Logic

LT Linear Time

MCDC Modified Condition Decision Coverage

MEA More Electric Aircraft

MEE More Electric Engine

MOC Model of Computation

NASA National Aeronautics and Space Administration

ODE Ordinary Differential Equation

PBW Power By Wire

PID Proportional Integral Derivative

PI Proportional Integral

POA Power Optimized Aircraft

PRISM Probabilistic Model Checker

RSM Runway Safety Monitor

SAT Propositional Satisfiability Problem

SCADE Safety-Critical Application Development Environment

SDV Simulink Design Verifier

SISO Single Input Single Output

SS Sequential Systems

TIMES Totally Integrated More Electric Systems

TS Transition Systems

UPPAAL-Cora UPPAAL for Cost Optimal Reachability Analysis

UPPAAL-TiGa UPPAAL for TImed GAmes based controller synthesis

UPPAAL Uppsala and Aalborg Universities Real-Time systems model checker

V&V Verification and Validation

Chapter 1

Introduction

1.1 Motivation

Cyber-physical systems comprise a wide range of computer-based systems which require the integration of various technologies such as computers, control, and communications in order to achieve stability, performance, reliability, efficiency and robustness when dealing with physical systems [133]. Nowadays and thanks to advances in computer science and technology, it is extremely common to use computer-based systems on a day to day basis [7, 86]. From mobile phones to power grids, they are part of our civilization. The development of the hardware and software that makes this possible is a major engineering activity in many sectors. Along with the development of such systems the concept of system verification emerges: to ensure the correctness of the system against requirements. When the system does not entirely meet requirements the damages could be minor (e.g. a smart-phone reset) or catastrophic (e.g. the loss of human lives).

1.1.1 System Verification

A system failure can be very costly and even if no human lives are lost, the consequences of a software or hardware error could have severe repercussions. Intel lost millions of dollars because of the Pentium processor error [7, 130]; in 1996 the Ariane-5 rocket crashed a few seconds after launch [45, 96]; the Mars Pathfinder rover suddenly stopped working [82]; and patients died of an overdose of radiation using the Therac machine [99]. All these failures happened because of a bug in the system and the consequences were monetary, loss of credibility and the loss of human lives. Over the years the case for better verification procedures has been made and even if current practices have improved compared to when all those errors occurred, every now and then we are reminded of the gaps we need to fill.

In computer science, the related activities to software testing are referred to as *Verification and Validation* (V&V). These activities are a cornerstone for software development and are a huge component of any software development life-cycle [113]. They are different activities and address two different aspects of product development [21]:

- *Verification*: Are we building the product right? The product meets requirements doing what it is supposed to do in the way it is supposed to do it. The product is bug-free.
- *Validation*: Are we building the right product? The product does what it is supposed to do, it fulfils high level functional requirements.

The V&V activities use different approaches, tools, and procedures. The more complex the project or product, the need for novel and better tools arises. Over the years the software development community has come up with new standards, tools, procedures, and practices in order to provide higher quality products and avoid fatal failures. The type of product under development drives the V&V efforts and the amount of time spent in these activities. For safety-critical applications, such as control systems in automotive or aerospace industries, not being too careful is not an option and the software should be stressed as much as possible to prove its correctness and safety. Besides being able to detect the errors of the design, it is desirable to detect such errors as early as possible during the development process. Detecting an error in a higher layer of the design process can increase the cost of fixing it several orders of magnitude compared to the lowest development stages.

For many years now dynamic control systems are implemented using embedded control. The control algorithm is thus implemented on a computer-based system in a software manner. This change in paradigm meant versatility because modifying a controller meant simply a software change. More advanced control algorithms became available and new control theory was born [86]. Software engineering and control systems engineering are a fundamental component in cyber-physical systems. The control community should care about the software community because it relies strongly on it, and the software community should care about the control community for the same reason. Even if most of the time these two fields of engineering are considered independently, in practice they are one discipline and this fact should be made clear to both software and control engineers, especially in safety-critical control applications.

1.1.2 Trends in Aerospace Systems

The trend pattern for the aerospace industry is one of growth. The Federal Aviation Administration (FAA) has predicted that within the next 2 decades air traffic will increase by 150% to

250% compared to the beginning of this decade [62]. The long-term vision for future aircraft is one of a more electric machine. Companies and organizations are currently developing technology to bring the concepts of More Electric Aircraft (MEA) [136] and More Electric Engine (MEE) [73, 116] to a reality. Projects like Totally Integrated More Electric Systems (TIMES) and Power Optimized Aircraft (POA) [52] are part of this effort [39, 118].

Power-By-Wire (PBW) [79, 90] technology will then become crucial. PBW means that the power to move an actuator comes from electric sources. Having electric-powered actuators will increase the complexity of current electric systems; if it is considered that in the future the main propulsive power for aircraft will come from electric sources, complexity increases even more. To integrate different electric technologies for long term operation presents new unseen challenges in several engineering and design aspects. This change in paradigm is partly enabled by current computational power in embedded applications for the aerospace industry, and to address these new challenges embedded systems will play a key role to provide solutions for control, health, and communications.

Computer-based control was a big leap in aviation and for decades centralized embedded control has been used to address control challenges. The inclusion of the computer as a control device made software a very important component for aviation. As technology progressed computational power increased, allowing to enhance the on-board software and a number of additional functionalities became standard, providing not only control but also health management and safety features. In a modern aircraft jet-engine the control architecture is centralized, and by including more features on the same computer software has become more complex. There exists a dependency relationship among the various features of the software.

Figure 1.1 shows a generic block diagram representation of a centralized jet engine Electronic Engine Control (EEC) unit and typical functional features that are implemented in the form of software [38]. The system contains 2 computers for redundancy and safety purposes. Both computers run the same software while sharing information via a data link. The software components become strongly connected with a dependency relationship. Therefore the development of aviation control software has become more complex over the years: adding or removing features generates development and certification efforts for the sake of safety. As a result, time is invested and a higher cost in development has to be paid. The next step in control system architectures is one towards Distributed Control System (DCS) [10]: control and health management functions are executed on different computers; instrumentation wired to communication buses and local control processes supervised by a top level controller [116]. A more modular approach with local control loops would help to reduce complexity in development and accelerate certification processes

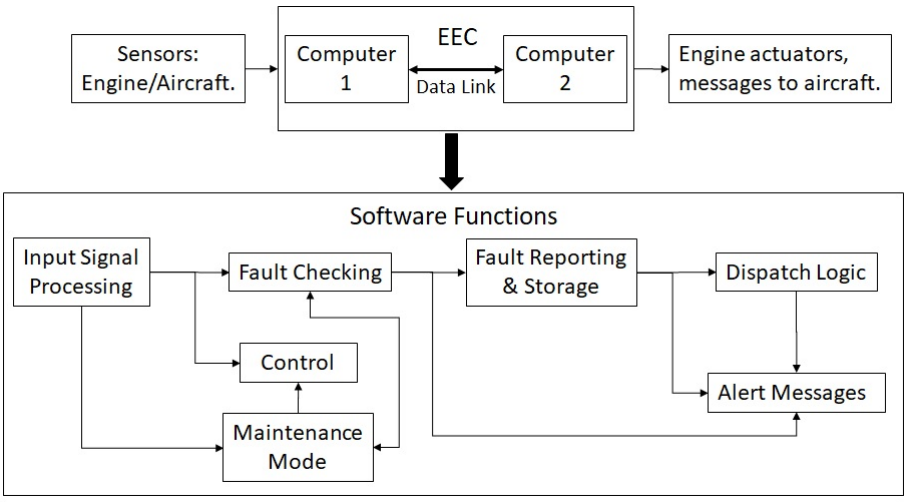

Fig. 1.1 Block diagram representation of an Electronic Engine Control (EEC) unit and the functional features it contains as software.

[62]. Nonetheless the DCS architecture is still under development with many open problems to solve. By partitioning functionalities the individual development effort may be reduced but the integration of all the features over a distributed architecture will require effort that is still unknown. Whether a distributed architecture becomes the standard practice or the centralized one remains the standard, software will become more complex in order to address the new challenges to come: more electric systems, more efficient systems, enhanced health management, more advanced control, etc.

1.1.3 Avionics Certification

For commercial aviation, the Federal Aviation Authority (FAA) and European Aviation Safety Agency (EASA) are the two major bodies in charge of aircraft certification [43]. Regulations to be fulfilled by a jet engine manufacturer can be found in Federal Aviation Regulations (FAR) parts 25 and 33 [50]. Part 25 is related to "Transport Category Airplanes" and Part 33 to "Aircraft Engines". All the certification activities for commercial airborne systems exist for safety reasons: they ensure that the right thing is being built in the right way, contemplating every aspect and detail of the aircraft.

Modern aviation systems use Fly By Wire (FBW) technology, evolving towards PBW technology. This means that control algorithms are and will be implemented in the form of software. It does not matter by which technique or method the controller has been developed; it will be implemented on an embedded computer system. Since the integration of computer systems into the aviation industry and as part of the certification process, software has to

be verified and validated versus requirements. DO-178C [51] is the standard created by regulation authorities to regulate and assess the safety of airborne software. This document does not specify how to make the software but rather describes what is to be expected of the software and the evidence to be provided for compliance. Standards like DO-178C identify a level of criticality of the related software according to its functionality by estimating the amount of damage that a malfunction in the software could generate. For the aviation industry the criticality levels go from *A* to *F*, level *A* being the most critical. Level *A* software malfunctions are classified as *catastrophic*, there exists a potential aircraft crash with potential human lives lost. A computer-based controller such as the EEC for a jet-engine is thus considered a safety-critical system.

1.1.4 Safety-Critical Software Development Life Cycle

It is especially for safety reasons that certification standards like DO-178C exist, and qualifying software for certification efforts is extremely critical and burdensome. In order to accomplish this, software development processes and practices have been established to facilitate and aid in the software development life cycle. Software development cycles were born in computer science and depending on the type of software under development one is chosen over another. For aviation systems, perhaps the most popular is the *V-Shaped* development cycle [113].

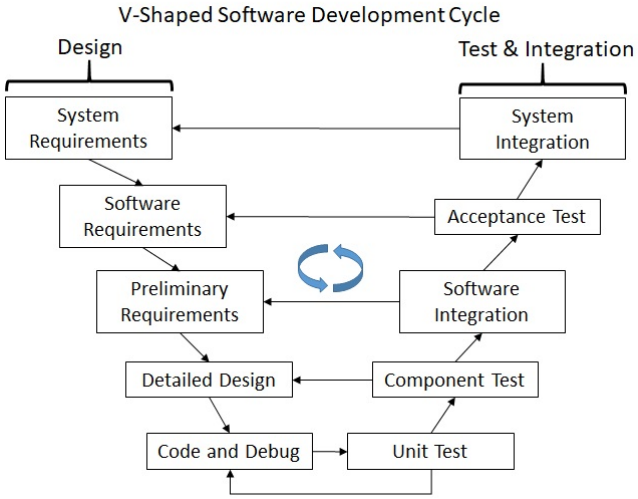

Fig. 1.2 V-Shaped software development cycle. Popular within the aerospace industry for software development. This process makes clear emphasis on testing activities throughout the whole cycle, which are fundamental when developing safety-critical software.

Figure 1.2 shows a block diagram description of the process. It is a popular choice because it is easy to follow and there are clear deliverables in each phase [113]. This model makes clear emphasis on the testing phase of the software. For safety-critical systems, verification and validation activities strongly rely on testing and a good amount of effort is spent on this activity: unit testing, integration testing, and acceptance testing [113]. Currently, 1 defect per 1000 lines of code is an accepted measure for today's software [7]. Considering Boeing's 787 avionics system, it is estimated to contain around 6 million of lines of code and therefore that would mean 6000 defects. In software development, testing all possible scenarios becomes harder when the software itself grows in size. Exhaustive testing practices like Modified Condition Decision Coverage (MCDC) are not sufficient to detect all defects. It is estimated that current testing activities amount to approximately 30% to 50% of the total cost of a software project [7].

We cannot forget that in the end there is an engineer coding the software, as Leveson and Turned have pointed out: *"it is still a common belief that any good engineer can build software, regardless of whether he or she is trained in state-of-the-art software-engineering procedures"* [99]. The most frequent reason for software crashes are due to programming bugs while doing the implementation of an abstract design [26, 139]. The more complex and large a design is, the more the chances for error. If the tasks can be separated in a more modular architecture with defined objectives for each task, the implementation and validation of the design can be benefited. It is therefore desirable to find a new approach to verification and validation which relies not only on testing activities.

Formal methods provide the means to analyse cyber-physical systems with a mathematical rigour. A report by NASA and FAA concluded that formal methods should become a standard part of engineering [7]. Within formal methods there is model checking, an exhaustive verification technique. Given a formal description of the system (a model) and a formal description of a property (a requirement for the system), model checking can verify if such property is fulfilled by the system. Model checking brings formality to both requirements and modelling, which is in turn useful when dealing with high level requirements expressed in natural languages for a complex system.

The usual verification process for a system versus requirements is an *a posteriori* one. A prototype of the product, or the product itself, is developed and the verification is performed using that prototype or product. Both the design and verification processes are driven by the system specifications but no formal approach is taken when bringing those specifications into low level requirements for the implementation or testing processes. Figure 1.3 shows a block diagram description of the process [7]. This is what most software development life-cycles

tend to follow: the verification is done in the actual product while it is under development. This falls well within the V-Shaped development cycle testing activities from Figure 1.2.

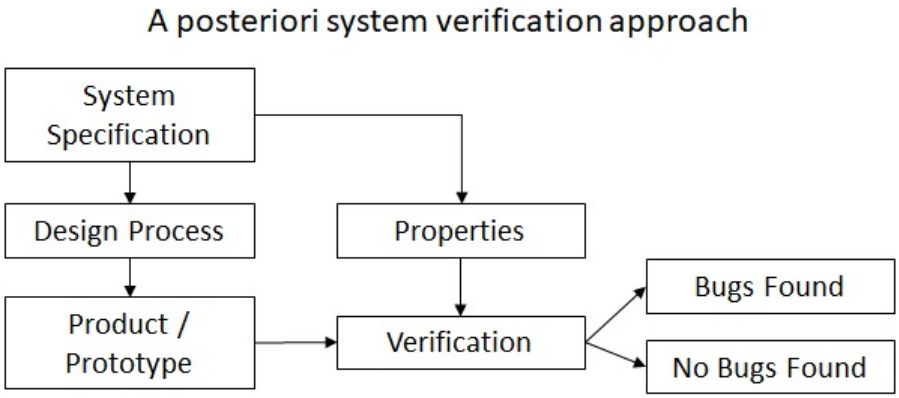

Fig. 1.3 System verification: *a posteriori* approach. The verification takes place in the product or a prototype of the product. The construction of the prototype or product is performed before the verification activities takes place. If bugs are found another iteration of the design and verification process is necessary.

In contrast model checking provides an alternative to verification that relies on a model of the system and a formal description of its requirements (properties). Figure 1.4 shows the model checking approach to verification. The model checking verification approach works over a model of the system, not the product itself (e.g. the final code in the software). The verification is performed in an *a priori* manner: errors, inconsistencies, and bugs can be detected before the product development process takes place. This is an important change in paradigm when it comes to development and verification. To fully exploit these benefits a greater effort in modelling has to be performed, so there is a trade-off when choosing which verification approach to take. However, the exhaustive verification capabilities and the formality of the model checking approach are appealing features for safety critical systems such as jet-engine control software.

1.1.5 Commercial Jet-Engine Control

A jet-engine is one of the most complex pieces of machinery ever created. No engine is the same and performance varies from engine to engine and on the same engine over time [78]. The engine components degrade at different rates and after an overhaul or repair performance is also affected. This presents a challenge for the control system because it should be able to cope with both slow degradations in performance and rapid changes due to overhaul [78].

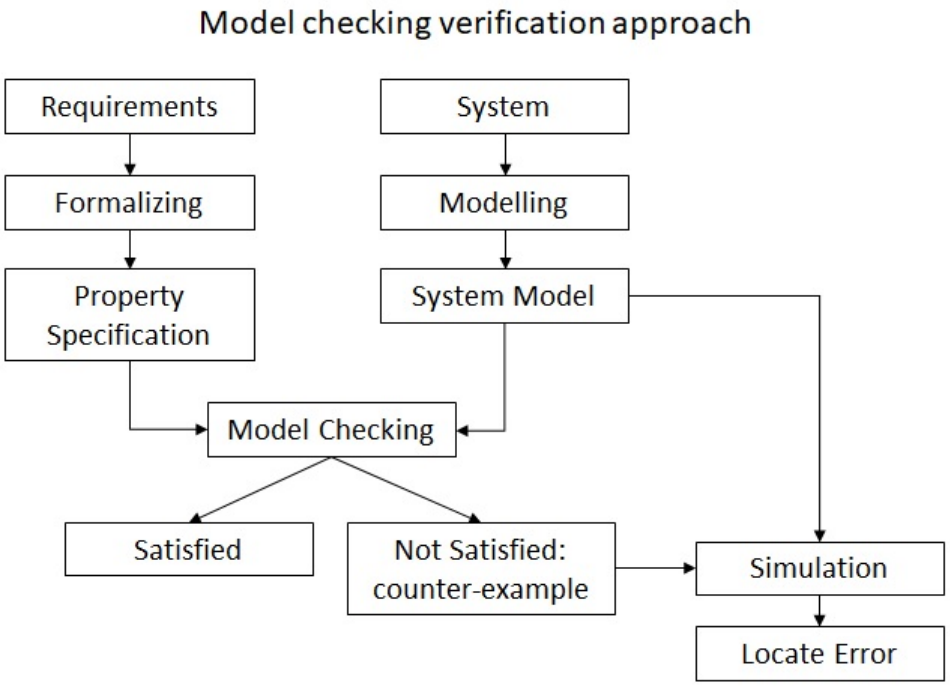

Fig. 1.4 Model checking verification approach: verification is performed in an *a priori* manner. The verification is performed in a model of the system under analysis before generating the product or a prototype.

In its early days, jet-engine control consisted mainly in regulating fuel flow into the combustor, controlling the air-fuel mix. Over the years new control inputs have been progressively added to the control scheme. Actuators such as guide vanes, variable exhaust nozzles, variable compressor stators, and variable bleed valves, among others, have been added to the jet-engine in order to obtain more thrust, better efficiency, and reduce weight [166]. This in turn created the necessity for a change in the control scheme, giving birth to more advanced algorithms.

A modern engine control system is in charge not only of control activities. As shown in Figure 1.1, there are many other functions running in the control computer (e.g. EEC). The three main functions are [38]:

- Communication with the aircraft to receive control commands and report status.
- Running the control algorithm to meet engine performance.
- Health management for diagnostics, prognostics, and control.

Advances in computer power allowed the possibility to add health management functions. Because most of the health monitoring functions are related to control actuators, health

management is tightly related to the control system itself [78]. Fault detection, isolation, and accommodation plays a fundamental role in engine control and performance [78, 166]. Jet-engine control must be fault tolerant and highly reliable for safety reasons, which means redundancy in both hardware and software components of the control system.

The main purpose of a jet-engine is that of providing a thrust output according to the pilot's demand [135, 149]. In whichever way thrust is regulated (e.g. shaft speed, pressure ratio), the control scheme for a commercial jet-engine is that of gain scheduling: a digital controller with multiple controller tunings (e.g. different sets of gains), the controller tuning is changed depending on the operating conditions. The controller is implemented in its digital form in the EEC and the gains for the controller are stored in a look-up table manner. To maintain engine variables within limits a min-select strategy is preferred, which means that the control strategy will comprise several control loops [149]. The controller will switch between control loops during different engine operating modes and conditions. This type of control is preferred because of its simplicity compared to more advanced control methodologies [127, 149]. Even if more advanced control architectures and approaches have been tested in real engines, this has either been in a prototype or military application. Certification of military applications is regulated by a different legal framework which is more flexible than that of commercial, which partially explains why intelligent and adaptive control schemes have already been implemented in military aircraft [18]. There are still gaps to be filled to bring more advanced control schemes into the commercial side of aviation, which may also include a change in paradigm about the way certification is conducted and what it entails.

This complexity makes the control system design for a jet engine a very challenging task. The complexity resides not only in the nature of the process but in the complexity of the control scheme itself. The control is implemented in the EEC and interacts with several functionalities creating data dependencies, and because different functionalities are constrained to different timing requirements this also adds to the complexity of the controller [78]. For these reasons, modern commercial jet-engine control software, under the levels of criticality designated by DO-178C, is classified as software level *A*: its failure is potentially catastrophic.

Before more advanced control schemes reach enough maturity to become part of commercial aviation, gain scheduling will be the default option in jet-engine control. Even though gain scheduling is a well-known control scheme and has been around for over 50 years, in practical applications it is still challenging to guarantee stability and prove its correctness [94, 97, 127, 138]. Demonstrating safety and requirements conformity for a gain scheduling controller is challenging from the design, verification, and implementation points of view.

To prove stability and performance in an analytical manner is complicated [97, 127, 138], also there are no performance guarantees in between design points [18]. Also, a jet-engine consists of several control-loops for different engine conditions, accounting for every possible variation is impractical at the least.

For a complex control system, such as a jet-engine control scheme that contains several limit restrictions, control loops, fault modes, and signal selection to handle faults, ensuring that a safe mode is reached in the event of multiple faults is non-trivial. Classical control design methods are hard to scale for such a complex system, therefore relying strongly in extensive testing activities in order to guarantee safety, performance, and certification compliance.

1.1.6 Formal Methods and Model Checking

The model checking verification framework (Figure 1.4) provides a potentially different approach to verification of gain scheduled controllers. The model checking exhaustive and systematic search for every possible reachable system state can help to increase the range of scenarios used during verification [139]. Formal methods and model checking have been around since the 1970's but their use have been limited mainly because of technological reasons and computational power [7, 35], just over 25 year ago its use in industrial applications was minimal [23]. In the last two decades, advances in computer systems and the increase of the available overall computing power, have made possible that the use of model checking transitioned from the research and academia environments into the industrial environment [7, 86]. This change in paradigm comes hand in hand with the increase of cyber-physical systems which depend on computer-based or embedded applications [86].

Formal methods and model checking have been successfully applied in industrial applications over a wide range of domains [8, 37, 61, 71, 86, 115, 148, 150, 162]. From computer science where companies like Microsoft and Intel have dedicated formal methods groups to manufacturing, aerospace, and motor industries. The National Aeronautics and Space Administration (NASA) has been a strong advocate of formal methods for some time now [29, 75], and over the years has improved and contributed to the incorporation of formal methods into the real applications domain [26]. The aerospace industry has become more interested in formal methods in recent years, analysing and experimenting with real-life scenarios case studies and in some cases even incorporating formal methods and model checking into the software life cycle for some particular features. Companies like Dassault-Aviation and Airbus have incorporated formal methods to replace some testing as early as 2001 [112]. For this reason regulation authorities like the FAA have updated their certification standards. DO-178B was updated into DO-178C, where the *C* release includes a supplement

on formal methods: DO-333 [137]. Benefits can be obtained from the incorporation of model checking into safety-critical control software development cycle (e.g. test coverage increase, automated test case generation, earlier error-detection, requirements clarification) [16, 62, 80, 108, 139, 143, 147, 153]. Recent case studies involving DO-333 are covered in a NASA report [36].

1.1.7 Challenges for a Formal Development Approach

Even though there are benefits to be obtained from including formal methods in the development cycle, the practicality to obtain them is not a straight forward process [16]. Most cases involve small scale studies and sometimes ad hoc tools have to be generated, which takes time and effort which may out-weight the benefits [16, 46, 47, 131, 158]. It is for reasons like this that it is hard to incorporate model checking into development frameworks. The full adoption of formal methods and model checking into industrial environments in the aerospace industry is still unlikely to happen soon, challenges and roadblocks have to be addressed before this transition becomes a reality. The exhaustive verification nature of model checking is also a hindrance.

This is referred to as the *state-space explosion* problem: the number of possible states of the system model becomes too high and performing the analysis of all of them is impractical and even impossible with current computational capabilities [7, 35]. All possible values for all the system variables have to be analysed in order to explore the full behaviour of the system model. This in turn derives in an exponential growth in the possible system states as the number of system variables increases. Floating-point data type and its arithmetic is part of this problem: a floating-point variable has potentially infinite possible values which results in a *state-space explosion* problem. So far, no model checker fully supports this data type which makes hard the modelling of dynamic control systems [108, 153].

Another challenge resides in the requirements formal description. Figure 1.4 shows how requirements are an input to a process in order to generate a formalization of them. The mathematical rigour of model checking needs that requirements be formalised as well. This is not a common practice and the usual definition of requirements is in the form of natural languages such as English. For this reason, a variety of model checking approaches exist: depending on the type of systems to be modelled and requirements to be verified, a formal language and modelling approach have to be chosen. There is extensive research on the formalization of requirements and the translation from natural languages into a formal description. However, dynamic control systems performance requirements are not directly expressible in a model checking formal language and complex control algorithms such as a feedback control loop with gain scheduling are problematic to implement in a model

checking framework [143]. The lack of expertise in this area presents an important roadblock towards the acceptance of model checking as a standard development tool. The use of model checking tools and formal approaches for software design most of the time requires high expertise and knowledge of formal methods from the software and control engineers [143]. In order to bring formal methods into common practices, the knowledge gap must be covered with easy to understand frameworks and practices.

Given the capabilities of formal methods and model checking, lately a new approach to control has emerged. The formal approach has not only been used for verification and validation activities but also to synthesize controllers, this is known as the *correct-by-construction* approach [67, 81, 152, 163]. One of the advantages of this approach is the use of requirements as a formal input into the controller design and synthesis process, which in turn can help to avoid ambiguities when the requirements are translated. The synthesized controller is one of symbolic nature, a state machine [40, 65, 67, 81, 152]. However, so far the *correct-by-construction* approach does not contemplate common controller structures (e.g. PID) and has not yet been applied to gain scheduling control. It is highly desirable to enable common modelling practices for control systems in a model checking environment thus allowing control engineers to exploit the benefits of model checking for a safety-critical application such as a jet-engine gain schedule control system.

1.2 Aims and Objectives

The aim of the research conducted in this thesis is to formally design and validate discrete Proportional Integral Derivative (PID) controllers in a gain scheduling control scheme. To demonstrate the applicability of the proposed methodology an aerospace application is selected: a commercial jet-engine thrust control system. Model checking is the selected tool within formal methods to conduct this research. To accomplish this, the following objectives are set:

1. Generate a formal design and verification framework for discrete PID-type controllers so that minimum intervention from the designer/engineer is required. In this manner the model checking capabilities can be exploited with the aim of incorporating the methodology in a design and verification software life-cycle.

2. Within a model checking environment, incorporate transient responses and system dynamics to the controller synthesis problem.

3. Generate a dynamic system abstraction methodology to incorporate dynamic behaviours into a model checking environment. The system abstraction must consider

modelling inaccuracies and compensate for them in order to provide safety guarantees when designing and verifying the original system using the abstraction.

4. Generate a model checking framework which allows to formally verify dynamic control systems high level performance requirements: maximum % overshoot, rise time, settling time, and steady state error. Using the abstraction methodology in order to generate the system dynamics, the model checking framework must allow to portray the high level performance requirements in a formal manner so that the model checker can verify them in the form of system properties. Once this has been accomplished, the requirements verification problem can be addressed as a model checking properties verification problem.

5. Generate a model checking design and verification framework for discrete PID-type controllers. The framework will take as formal input high level control performance requirements and using model checking will address the controller design problematic.

6. Generate a model checking design and verification framework for gain scheduling discrete controllers. The framework will take as a formal input high level performance requirements for a set of operating points of a dynamic system. Using a discrete PID-type controller the control schedule will be systematically generated until requirements are met in all operating points.

7. Perform a cross-verification of the model checking results with high-fidelity models to assess the accuracy and veracity of the proposed model checking approach.

1.3 Contributions

The fulfilment of these aims and objectives leads to the following contributions:

- The first main contribution is a novel abstraction methodology for dynamics systems that is described in Chapter 3. The input to the abstraction methodology is a continuous-time model: Single Input Single Output (SISO) Linear Time Invariant (LTI). The output of the methodology consists of a pair of discrete-time SISO LTI models. The abstraction methodology uses a fixed-point data type representation using integer data only, thus avoiding the necessity of floating-point data which makes the abstraction suitable for a model checking environment. The abstraction methodology considers modelling and data type errors. In this manner, upper and lower boundaries are generated: an *over-approximation* and an *under-approximation* of the original model.

Instead of abstracting the control system as a whole for its analysis in a model checking environment, as it is typically done, by the means of the methodology the components of a dynamic control system can be abstracted, implemented and analysed individually. The methodology allows to calculate the system response within a model checking environment, instead of pre-computing and preparing the data to import it into the model checker as is the common practice. This also allows to calculate the interaction between system components enabling a wider range of testing scenarios.

- The second main contribution is a novel formal verification framework for dynamic control systems presented in Chapter 4. The verification framework allows for the first time to formally verify the control system versus high level performance requirements: maximum overshoot percentage, settling time, rise time, and steady state error. The system abstraction (Chapter 3) is implemented in a model checking environment consisting of a set of timed-automata. In this manner, the verification is performed with a *push-button* approach in the model checker. The use of the *over-approximation* and the *under-approximation* allows to infer properties of the original system in the model checker. The methodology is demonstrated using an aerospace related control system example.

- The third main contribution is a novel PID controller formal tuning methodology presented in Chapter 5. For the first time a PID controller is tuned in a formal manner using high level performance requirements as the formal input. By the means of simulating the system response, the model checker non-deterministically searches for a set of controller gains which can drive the system to meet requirements. The controller tuning problem is solved as a *reachability* problem in model checking using a *push-button* approach. In this manner, the resulting PID controller is designed using the *correct-by-construction* formal approach but instead of generating a symbolic controller, a typical known-structure controller (e.g. PID) is tuned. The tuning methodology is also demonstrated using an aerospace related control system example.

- The fourth main contribution is a novel gain scheduling formal design and verification methodology presented in Chapter 6. The proposed methodology systematically builds a gain scheduled control scheme for a PID controller. For the first time a full model checking framework to address a gain scheduling problem is presented. The schedule design is solved as a *reachability* problem using a *push-button* approach. In this manner, the resulting schedule is designed with the *correct-by-construction* formal approach using a typical PID controller structure. The proposed framework enables an *a priori* approach for software design for safety-critical control applications. A partial version

of the methodology and its implementation was introduced in [123]. The novel formal framework could work as a baseline towards incorporating the *correct-by-construction* approach into the early design stages of the software life-cycle. As a direct result of this, the time spent in the design, verification, and validation activities for control software could potentially be reduced. The methodology uses a well-known model structure for control systems with no change of coordinates when using the abstraction; this feature makes debugging and analysis easier to the designer. The methodology is demonstrated using an aerospace related example: jet-engine thrust control. The gap between model checking and standard software development practices for aerospace applications can be reduced by the use of formal frameworks like the one proposed in this thesis.

1.4 Thesis Overview

This thesis is structured as follows:

- Chapter 2 presents the literature review and technical background regarding the main subjects for the development of the research presented in this thesis: formal methods and control systems. The literature review covers the trending towards the use of formal methods in industrial environments, the various uses of formal methods in software development, and current challenges to bring formal methods into a software development cycle regarding current safety-critical control applications.

- Chapter 3 presents the system abstraction methodology for dynamics recovery. Discrete SISO LTI models are proposed for the dynamical representation of the control system. An ad hoc fixed-point integer only data type is proposed. In this manner, the ad hoc data type only requires integer variables providing enough resolution to address the system dynamic simulation. The proposed abstraction methodology allows to portray a dynamic feedback control system without the need of floating-point or fixed-point data types, which makes it suitable for a model checker implementation. The abstraction methodology takes into account possible modelling errors and data type rounding errors, compensating for these inaccuracies so that the abstraction provides boundaries when reasoning about the original system. The output of this chapter is a novel dynamic system abstraction methodology suitable for model checking which provides safety guarantees regarding possible modelling errors.

- Chapter 4 presents the model checking approach to high level control system requirements verification. A *design for verification* approach is taken to create a set of

automata to enable the formal verification of high level control performance requirements: maximum % overshoot, rise time, settling time, and steady state error. Using the abstraction methodology presented in Chapter 3 to recover the system dynamics, the automata design is driven by the necessity of expressing control requirements as properties for the model checker. In this manner the performance requirements verification problem is addressed as a property verification in model checking. The output of this chapter is a novel formal verification methodology for discrete feedback control systems.

- Chapter 5 presents the formal discrete PID controller design and verification framework. By combining the system abstraction methodology from Chapter 3 and the high level performance requirements verification framework from Chapter 4, the automata framework is extended to address the controller tuning problem. The model checker is systematically used to generate a set of controller gains which drive the system into meeting high level performance requirements. The output of this chapter is a novel formal design and verification methodology for discrete PID controllers.

- Chapter 6 presents the formal discrete PID gain scheduling design and verification framework. The framework is underpinned by the abstraction methodology from Chapter 3, together with the high level control performance requirements verification framework from Chapter 4, and the formal discrete PID controller design and verification framework from Chapter 5. By bringing together all these elements, the timed-automata framework is extended to address the gain scheduling problem. The model checking framework is systematically used to generate the control schedule which drive the system into meeting high level performance requirements in every operating point. The model checking framework provides a novel push-button approach to design and verification of the schedule. The model checking framework also allows to perform coverage testing over the final schedule. In order to provide more information to the designer about possible behaviour of the system in case the controller switched gains in regions for which the tuning was not intended to operate.

- Finally, Chapter 7 presents the conclusions regarding the work presented in this thesis and the future work.

Chapter 2

Literature Review and Technical Background

2.1 Overview

Nowadays, embedded computer systems are widely used for control applications. The increase of computational power and the necessity of more complex control systems have made the verification for correctness and compliance with requirements a major part of the control systems development process [7]. In safety-critical applications where human lives are at stake, safety must always be a priority. The relationship of control systems and computer science has never been so close, the need to deliver safe and reliable computer-based controllers is a crucial growing activity in engineering. Formal methods and model checking can provide the means to improve the design, development, and verification of embedded control applications.

This chapter presents the literature review and technical background related to the work developed in this thesis. Section 2.2 presents a state of the art analysis regarding formal methods and model checking. The main applications of formal methods and model checking regarding software development and the recent advances in the area thanks to the increase of computational power were analysed. Also, how formal methods and model checking fit into an industrial development setting, the potential benefits from its application, and the current challenges to be addressed in order to fully exploit formal methods and model checking were analysed. This analysis was mainly focused on keeping in mind safety-critical control systems applications, which is the main area of interest in this work for the application of formal methods. Finally, Section 2.3 presents the necessary technical background to understand the type of systems under analysis in this thesis. Both the model checking

fundamentals and the selected type of control systems fundamentals are presented in order to cover the two main subjects of this work: formal methods and safety-critical dynamic control systems.

2.2 Literature Review

The following section is about the different applications of formal methods. First, their use for V&V activities is presented. Next, the controller synthesis application is presented along with automatic properties generation. Finally, the benefits of the use of formal methods are mentioned. Major challenges and roadblocks are presented as well as weaknesses of model checking. How these roadblocks have been addressed in the past is analysed in order to identify gaps in the current state of the art in the use of formal methods and model checking for the design and verification of safety-critical dynamic control systems.

2.2.1 Verification & Validation

Due to the nature of formal methods, their main application is within software V&V activities. Whether the software under analysis is a controller [26, 74, 102, 108, 119, 126, 139], diagnoser or fault management schemes [24, 63], network communication protocols [69], or some other type of application [62, 129], its use in this activity is wide.

The automotive industry has also shown interest in the application of formal methods and model checking for product development; the industry acknowledges the need for better modelling and software engineering practices [27, 55, 56, 84, 140, 155]. Hybrid systems for modelling an automotive engine and its power train control can be found in [9]. Here, formal methods were used for V&V and to synthesize the power train controller. In [84] a formal approach is used to verify that autonomous vehicles never violate safety requirements. The formal approach is also used to verify the actual code of the autonomous vehicles, not just the overall behaviour of the system. In [140] an incremental approach is presented in order to use a SAT solver in combination with an industrial embedded software verification tool to reduce runtime during the verification phase. In [155] system high level requirements were translated from natural language into a formal language in order to detect redundancy among requirements and test cases.

Given the computer science nature of formal methods the first power management challenges addressed with these techniques were computer related. The efficient use of energy for embedded systems has always been the subject of research. In the early 2000s probabilistic and statistical model checking were first used to address power management

issues, tools such as PRISM arose to perform such analysis [72]. Regarding V&V, Dynamic Power Management (DPM) policies under multiple constraints are verified and compared in [117]. Same activities are performed in [145], but also DPM as a two game player (power manager versus environment) is suggested. Hardware safety and efficiency requirements are verified in [91].

The safety-critical nature of aerospace systems demands the use of advanced formal verification methods which are capable of guaranteeing compliance with requirements. Both the FAA and NASA have expressed their interest in the use of formal methods [7]. Space-related applications have used formal methods for V&V activities, either in case studies or real life applications. Results are encouraging for formal methods as a way to locate software coding errors, outperforming traditional testing practices [26, 62, 63, 139, 146, 147]. Regarding the aerospace industrial environment, some companies have included in one way or another the use of formal methods at least partially to improve or replace testing activities [112]. The inclusion of formal methods in safety-critical control software development processes has been studied recently; it is not a straightforward process but efforts are being conducted in order to make this a reality and exploit the benefits of formal methods in industrial size aerospace projects [16, 143, 158].

Regarding control systems design, in particular when dealing with the controller itself, in [4, 5] a formal method approach for the verification of *control envelopes* is presented. This approach uses a logic prover to verify a family of input-output relationships regarding the controller and the plant. Instead of abstracting the system as a whole the verification is performed over the input-output families. This is a static check over an invariant set (the family of *control envelopes*) rather than a dynamic check of the closed loop behaviour of the system.

2.2.2 Controller Synthesis and Properties Generation

Over the last two decades the use of formal methods for the synthesis of logic based controllers has increased. This approach is known as *correct-by-design* because the control algorithm is synthesized using a formal approach with requirements as a formal input. This is an appealing application of formal methods and model checking because it provides a new way of generating control schemes using the advantages provided by the formal approach. The use of this approach is mainly in research activities and academia but the obtained results are promising and encouraging towards accepting this methodology in an industrial development application framework [9]. In particular, when the synthesis problem under analysis can be portrayed in a scheduling scenario manner, the formal approach of *correct-by-design* is popular [14, 17, 22, 31, 42, 81, 83, 95, 103, 105, 156].

Within model checking, linear-priced automata is the variant to address the control problem using an optimization criteria as part of the formulation to find a solution. An adversarial approach to solve a two player game (e.g. the controller versus the environment) has been used for DPM controller synthesis in [77] and a probabilistic model checking approach to DPM is analysed in [142]. The two player game-type approach to controller synthesis led to the realization of the UPPAAL-TiGa tool [11, 12] and linear-priced timed automata led to the realization of the UPPAAL-CORA tool [14], where the model checker is turned into an optimization schedule-solver. Adaptive schedule strategies for a pipeline system of a printer are generated using UPPAAL-TiGa in [2], this is an industrial-type case study. A power-grid relay controller was synthesized and verified using Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and the model checker tool UPPAAL [60]. This example addresses a case study of significant size regarding the problem it solves, showing promising results for UPPAAL as a working tool in industrial applications. Within the aerospace domain, power management for avionics has also been addressed by formal methods [163, 164]. Given a set of power sources, buses, and contactors find the best configuration possible to meet safety and performance requirements. Controllers are synthesized and verified to comply with the given requirements. Another important aspect of this work is the comparison between centralized and distributed control architectures and how to cope with them while using formal methods. This is important due to the trend towards DCS in the aerospace industry. Python was used to develop the in-house tool for this work and UPPAAL is mentioned as a good option for future work [163, 164].

Regarding dynamic control systems, a design and verification framework was developed in [58], using not model checking but theorem provers as the formal methods tool. In [40, 67] a control synthesis toolbox is presented combining the use of both Matlab and the model checker UPPAAL to construct a set of control laws by solving a game abstraction as a reachability property in the model checker. In this example the system dynamics are abstracted as time-invariant properties so the model checker can reason about them using clocks. In [65, 151, 152] the design of regulators for hybrid systems using requirements in the form of LTL formulae is explored. This work focuses on stability and regulation of hybrid systems, providing a working framework in Matlab for the synthesis of symbolic controllers. Also, this framework has been applied to motion planning as well. In [159–161] a working framework for motion and trajectory planning is presented. The toolbox TuLiP developed in Python takes a subset of LTL to describe control specifications as inputs and generates a symbolic controller which is provably correct.

Another interesting area of research for formal methods is the automatic generation of properties. As mentioned at the beginning of this section, not only the model of the system

has to be formal but also the language in which the properties are expressed. This means that high-level requirements have to be translated into a formal language and sometimes this is difficult, and because this activity is mostly conducted by hand and relying on the expertise of the designer and end-user, it is prone to human error as well. It is desirable to be able to translate requirements in a solid and systematic manner to avoid errors in the translation into properties expressed in a formal language such as LTL or CTL. This has been explored in [146] applied to an aerospace case study, and in [164] a property generation algorithm is proposed.

The use of formal methods for controller design, V&V, and the automatic translation of high level requirements into formal properties to address one particular problem, all at the same time, may seem ambitious at the moment. Nevertheless, in many cases the use of formal methods deals with at least two of these activities, particularly the combination of the controller design and V&V. Formal methods and model checking still need to address some weaknesses before becoming a standard development practice for software and hardware in every engineering field [16, 80, 143]. Particularly in the area of dynamic control systems for safety-critical applications, the *state-space explosion* is perhaps the main limitation due to the nature of the processes under analysis and the required data representation to deal with them. The following subsection elaborates this particular issue.

2.2.3 Challenges and Approaches

The benefits of using formal methods and model checking techniques to develop critical software applications have been mentioned. Safety and error-detection rates are benefited by the inclusion of such tools. The use of formal methods to verify and validate software as well as for synthesizing controllers has recently increased in both academic and industry environments [86]. The aerospace industry is willing to include these methods in the product development process but before this can become a reality, certain aspects and roadblocks have to be addressed.

One of the major drawbacks within the model checking tools is the *state-space explosion* [7]. When using floating point data-type, the *state-space explosion* problem becomes harder to address. A floating point variable has potentially infinite values which makes an infinite state-space to be analysed, which is at the least impractical if not impossible. For the correct representation of hybrid systems' dynamics and models the use of this data-type is needed. As mentioned in [108]: *"the floating point arithmetic has proved to be problematic. These kind of calculations are often used in the aerospace industry. There is no model checker that fully supports floating point arithmetic"*. This is still an open problem and one that

needs to be addressed to fully exploit the capabilities of formal methods and model checking. Different approaches are used to overcome the floating-point arithmetic limitation.

The approach in [83] was to first digitize a continuous time battery model using a fixed sampling period. Once the model was in digital form, it was used to generate data regarding the time-response of the system and the results were stored in the form of arrays for its use in the model checker UPPAAL. The data contained within the arrays was integer-type only. How the data was generated is not clear but the model checker was not involved in the calculations. The approach basically pre-computes a time response, prepares the data so the model checker can use it and imports it into the model checker. A limitation with this approach is that the testing scenarios have to be pre-computed as well. Calculating the dynamic interactions between components is not possible in this form. It is desirable to be able to calculate the system response within the model checker, this in turn also allows to verify for more scenarios and perform a more extensive verification.

Another way to deal with the floating-point limitation is reducing the complexity of the model so that the variables of interest can be calculated using integer only data type. In [156] a thermal model is simplified from a complex one so that the calculations can be performed within the model checker. By doing so the dynamic behaviour is in a way abstracted from a high-fidelity model. This approach may restrict the type of models to be used and it is possible that transient behaviour is completely lost by doing so. Especially when dealing with control feedback loops this approach may not be sufficient because of the interaction and dependencies among components such as the plant and the controller.

Removing transient and dynamic behaviours is another way of avoiding the use of models which require floating-point computations. In this approach the problem is then restricted to its steady state formulation. In [108] even if the selected tool (e.g. SCADE) contains a development environment and formal methods tools , the case study selected to show the advantages of the approach entirely disregards linear arithmetic calculations. Such calculations are supported by the development environment but not by the formal methods tool. The properties which can be verified thus omit dynamic behaviour from the verification process.

Another way to avoid the use of floating-point data operations to recover the system dynamic behaviour when using a particular model structure, is to use an abstraction of the dynamic behaviour using a different coordinate plane. In this way, a surrogate of the original model is constructed. Some model checkers, especially the ones that deal with real-timed systems, can deal with timing constraints (time invariants) using clock variables. Clock variables are essentially floating-point variables but their use is very limited because they are aimed at simulating time progression only. The model checker UPPAAL meets this

requirement and in [126] this option is explored: abstract dynamic behaviour using time invariants constraints as surrogates. Another interesting remark is that MATLAB-Simulink is used to calculate the timing constraints by using a complex model. Again, the model checker is not in charge of dealing with the heavy burden of calculating the system dynamics but to verify compliance with requirements.

The model checker UPPAAL and MATLAB-Simulink have also been used to synthesize a temperature controller in [81]. The control problem was modelled as a two player game using UPPAAL-TIGA and an S-Function for MATLAB-Simulink is automatically generated once the model checker satisfies a property. This is an interesting application towards a fully integrated implementation process. The heat exchange process is simulated within the model checker using a model abstraction of a more complex non-linear process. As long as the required properties to synthesize the controller are portrayed by the simpler model a correct result can be obtained.

A divide and conquer approach is also suggested when dealing with the *state-space explosion* problem. The approach relies on partitioning the system into smaller subsets and generating a model abstraction consisting of a combination of under-approximations and over-approximations. The correctness of individual components can assure the correctness of system-level properties [62]. The verification is then performed on a family of models and system-level properties can be analysed as a group of properties in the model abstraction. By doing so the verification is done in smaller models, avoiding running out of memory and generating an overall result by aggregating individual verifications.

Following the partitioning idea of the state-space and using the model checker to solve reachability optimal problems, an extension of Linear-Time Priced Automata is proposed in [95]. The concepts of Priced Zones and Faucets are introduced in order to deal with the state-space explosion by reducing the amount of states that have to be visited to reach an optimal solution. The methodology does not deal with system dynamics in any particular way but it does alleviate the memory burden when solving problems.

The Runway Safety Monitor (RSM) was developed by NASA and its main purpose is to generate alerts for possible crash scenarios. To deal with the state-space explosion (an infinite set of positions of multiple airplanes in multiple runways) a discrete model was used instead. By definition RSM only considers snapshots of data, discretizing the input data. In this way an abstraction of the much more complex model is generated. Part of the software was verified using model checking helping to detect errors that had been missed by common testing practices [147].

NASA explored the option of using fixed-point instead of floating-point to limit the variables' range. The model checker used was developed by NASA as well (Java Pathfinder)

and the fixed-point module was developed from scratch and added to the model checker [68, 128, 154]. The idea is the same: take an infinite range variable and reduce its range just so it is still useful enough for model checking techniques to use it. MATLAB-Simulink design verifier performs a similar task when dealing with floating-point data but it is not clear how is this process performed [109].

2.2.4 Remarks

The development of cyber-physical systems is a non-trivial task. The interaction between components and the environment keeps growing which adds to the complexity of the task. Safety and reliability are important aspects to the development of cyber-physical systems, and this becomes even more important for safety-critical systems. Efforts to incorporate formal methods into this arena are currently being conducted. Several approaches, techniques, and tools comprise a good body of knowledge in this subject. The problem under analysis and design strongly drives the formal method approach (e.g. technique and tools) to be used to tackle the problem; a good summary on formal frameworks and tools can be found in [120]. Most tools still have to deal with scalability issues and state-space explosion limitations [120].

The *state-space explosion* is still a major limitation in the use of formal methods and model checking to address dynamic control systems. The necessity of generating system abstractions is required. The system abstraction must preserve the original system qualities needed for the verification process. The generation of the abstraction is thus driven by the type of requirements to be verified. In one way or another addressing the state-space problem by using abstractions means the partition of the operating space into smaller versions of it. This generates either an over-approximation of the system or an under-approximation of it. Combining both the abstractions can provide boundaries regarding the original and more complex system model.

It must not be overlooked the fact that model checking assumes a correct representation of the system and it cannot do anything to prevent the user from generating incorrect results: if the model is incorrect, then incorrect results could be obtained [7]. The generation of good system abstractions to generate smaller models so that they can be used in model checking still requires high level of expertise. Particularly when dealing with dynamic control systems, abstractions are required to reduce the state-space to be analysed. This is strongly correlated to the *state-space explosion* problem. Even if memory is still a limitation for the use of high-fidelity models in model checking, with the increase of memory capabilities and better abstractions methodologies, the use of model checking tools to address dynamic control system problems is increasing.

It has been shown that model checking can help to identify gaps and errors in requirements, models, testing scenarios, and design practices. Testing is and will be needed for certification because it is performed on the product itself and not on a model. But results show that the early inclusion of model checking in the design process is more productive than usual verification techniques [147]. Also, errors can be found earlier in time when they are cheaper to fix [7, 62]. This shows that the integration of model checking in current software development practices is relatively easy.

Another benefit of model checking is the potential use of *push-button* technology. This approach requires minimum interaction and involvement from the end user. In this respect, running verification or design activities with the *push-button* approach does not require a high degree of expertise in model checking [7].

The use of formal methods and model checking from academic applications and case studies, into industrial case studies or even industrial applications, is a trending pattern. Particularly this can be verified by looking at the use of the UPPAAL tool over the years, going from basic examples as case studies to industrial size case studies [11, 28, 48].

Formal methods and model checking have been successfully applied in industrial environments and over the last two decades the use of the *correct-by-design* approach has become an active research area pushing the development of tools. The application of model checking for both controller design and V&V activities seem to be mature enough for industrial-size applications, particularly when combining model checking tools with other simulating and code-generating tools to provide an end-to-end solution from design to implementation [32]. There are still technical limitations to the model checking approach, particularly the state-space explosion.

When dealing with dynamic control systems from a model checking perspective this becomes an obvious problem to tackle and to circumvent this limitation the use of model abstractions becomes very helpful. The usual approach is to construct an abstraction of the closed-loop behaviour using a particular controller [4]. This limits the capability of reasoning and designing the controller itself because the system is abstracted as a whole, when the controller design process has already taken place, which is a shortcoming of the approach. Also, this approach is usually applied to trivial examples where one can abstract the controller and plant system as a single entity [4]. These reasons partially explain why model checking is yet to be applied into the design and verification of safety-critical control systems such as gain scheduling PID control schemes, specially with the aims of certification practices in mind. Not only the technical limitations have to be addressed but the existing gap between current software engineering practices and model checking must also be addressed. Model checking is not a common approach partially because it relies on the designer's expertise on

the subject, making it not user-friendly. Easy to follow model checking practices for a control engineer must be developed so that the benefits of model checking can be fully exploited.

2.3 Technical Background

The use of formal methods and model checking to address the design and verification of dynamic control systems is the main subject of this thesis. Formal methods and control systems are broad areas of engineering, which creates many possible applications and areas of interest when combining them both. Particularly one aim of the work presented in this thesis is to bring model checking and control systems together to address safety-critical industrial applications type of problems. The following section explains the particular type of dynamic control systems to be considered in this thesis and the type of model checking technique which will be used to generate a formal design and verification framework.

2.3.1 Dynamic Control Systems

The evolution of control systems as we know them today has seen drastic changes over the years generating alongside a wide body of knowledge around it. Early control systems relied on analogue control and its implementation was performed using operational amplifiers [19]. Their understanding and design generated early control theory based on frequency domain analysis by Bode [20], Evans [49], and many others. With the arrival of the transistor and digital computing, digital control was born around the 1950s [15]. The arrival of computer systems and the ability to perform fast computations allowed to perform algorithmic calculations before generating the desired control signal, improving control algorithms and enabling a variety of control theory such as state-space, predictive, fuzzy, linear, non-linear, adaptive, robust, hybrid, among others [86]. Among all the different types of control (and controllers), PID control is probably the most widely used type of control [59, 85, 101, 121, 122, 124]. It is simple to implement and the fact that it only consists of 3 parameters for tuning (proportional, integral, and derivative gains) makes it easy to configure.

The technological progress in computer science, both in hardware and software, has enabled the incorporation of computer systems everywhere. This is referred to as embedded systems, and when they are used for control purposes, embedded control systems. When an embedded control system is used to regulate the behaviour of a physical system, a hybrid system is born: continuous and discrete dynamics interacting. Figure 2.1 shows a generic hybrid control system, the controller is implemented in a computer-based system for the regulation of a physical variable. The actuator and the sensing device are the bridges

between the discrete and continuous worlds. This configuration is very common and flexible. Changing a controller means in most cases a software update, making it very versatile.

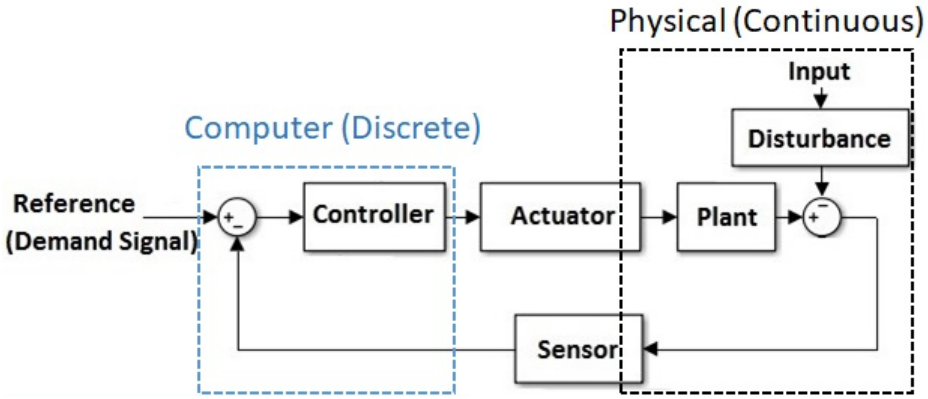

Fig. 2.1 Generic dynamic hybrid control system. A physical system regulated by a computer-based controller.

During the late 1950s computer-based control was being incorporated into aerospace applications. Jet-engine control was a direct beneficiary of such an approach. Highly non-linear processes such as thrust regulation in a jet-engine require more complex control approaches. The early attempts to apply adaptive control in an embedded control system for aerospace applications resulted in the gain scheduling approach [94, 149]: the gains of a known structure controller are varied depending on external factors and different operating points. The typical choice for the controller structure is a variation of the PID controller. In this way, a non-linear control problem is solved as a series of linear control problems, which in turn are easier to handle both from a design point of view and an implementation point of view. The controller gains are stored as arrays or look-up tables in the computer.

Even if more advanced control techniques and higher computer power are available with today's technology, PID gain scheduling control is still a popular approach [97, 127]. In particular, for safety-critical commercial aerospace control systems, gain scheduling is preferred because it is a well known approach, there is a lot of experience using it, it is simple to implement, and the road to certification for an airborne system is known. Airworthiness certification for commercial applications requires evidence to show the correct behaviour of the system prior to operation, which can be done with gain scheduling control but not with an adaptive system scheme [18].

However, certification practices are not exempt from mistakes and human errors. Proving safety and compliance with requirements is an arduous process where testing activities play a key role. The development of safety-critical control software can be improved by the

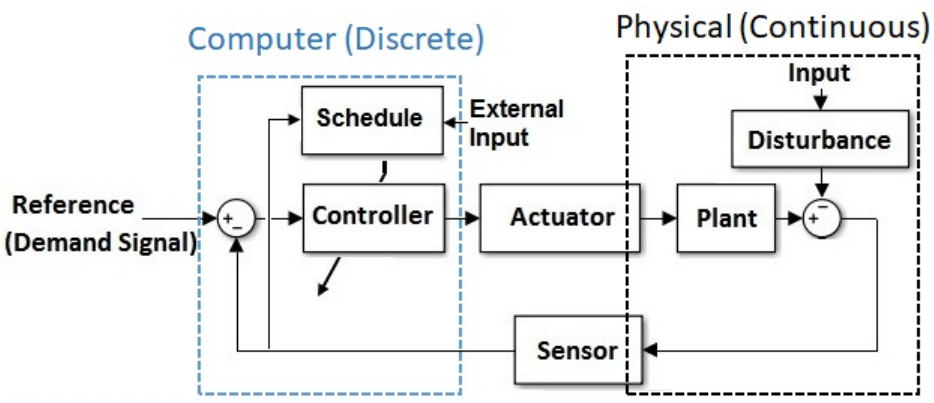

Fig. 2.2 Generic gain scheduled dynamic hybrid control system. A physical system regulated by a computer based controller. The controller gains are configurable while in operation depending on external factors and operating points.

incorporation of formal methods in the design and verification cycles, bringing together tools from computer science and control engineering.

Final Value Theorem

To know if a system output settles eventually to a given input gives an stability verification; also this calculation provides an indication of the system gain. To calculate the final value of the system response to a given input, the *final value theorem* can be applied. The following equation can be used to calculate the final value of a discrete SISO transfer function:

$$\lim_{z \to 1} (z-1)F(z) \tag{2.1}$$

The step response for a discrete SISO system $G(z)$ with input $U(z)$ and output $Y(z)$, can be calculated as follows:

$$G(z) = \frac{Y(z)}{U(z)}$$
$$Y(z) = G(z)U(z)$$
$$Y(z) = G(z)U(z) = G(z)\frac{z}{z-1} = F(z)$$

Using Equation 2.1 to calculate the final value of the transfer function $F(z)$:

$$\lim_{z\to 1}(z-1)F(z) = \lim_{z\to 1}(z-1)G(z)\frac{z}{z-1} = \lim_{z\to 1} zG(z) \tag{2.2}$$

The calculation of the final value of a system will be a useful tool for the proposed abstraction methodology in this thesis. Chapter 3 will present the abstraction methodology in more detail.

2.3.2 Formal Methods and Model Checking

The following section summarizes the basic concepts of formal methods and model checking. A description of what they are and the key building blocks of these methods are provided. In software and hardware engineering formal methods refer to the mathematically-based techniques for the specification, development and verification of software and hardware systems [87]. It is expected that the mathematical rigour which underpins formal methods can improve reliability, quality, and robustness of the design [75].

Within formal methods a model-based verification technique is model checking. Model checking is an exhaustive verification technique that given a formal description of the system and of a required property, it automatically determines whether the property is satisfied or not by the system [7]. To do so it has to explore all possible scenarios (this is why it is exhaustive). When the property is violated it returns an execution that exhibits the problem, which is called a counter example. This technique verifies the correctness of a model to meet certain properties. Model checking is an exhaustive and systematic search of the reachable states of a model of a given system [139]. Perhaps the most recognized limitation of model checking is the *state-space* explosion [7, 139]: the necessary states to represent the desired behaviour of the system is too high so that it easily exceeds the amount of computer memory.

To perform the formal verification of a system using model checking, two components are required:

1. Model of the system, i.e. formal description of it.

2. Formal language to describe the properties of the system.

The two main components of model checking are explained in more detail in the following sections.

Models: Transition Systems and Hybrid Systems

The models required for model checking are part of the Transition Systems (TS) family. TS consist of nodes (states) and edges (transitions) which model node changes. Hybrid systems

use this modelling principle. Figure 2.3 shows a generic transition system consisting of five nodes and eight edges. Events are related to edges so they trigger transitions to change states.

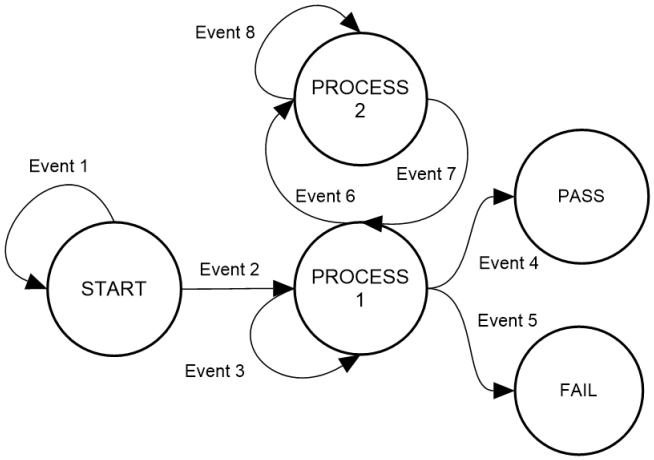

Fig. 2.3 Generic transition system consisting of five nodes and eight edges. Every node represents a state of the system and the edges represent how actions make the system move between states.

A hybrid system is composed of elements of both continuous and discrete dynamics. These 2 components interact with each other to determine the overall behaviour of the system. Due to the complexity of such systems a variety of modelling techniques have emerged. These are known as Model of Computation (MOC) e.g. Markov Chains [30], Petri Nets [30], Finite State Machines (FSM) [33, 57], Discrete Event Systems (DES) [3], Continuous Time Systems (CTS) [144], Sequential Systems (SS) [92]. By using two or more MOCs a more realistic model of a system can be built, which is becoming a common practice in order to meet industry requirements [3, 25, 30, 88, 98, 104, 114].

Properties: Formal Languages

Once having a model of the system a formal language to express its properties is needed. Such a language should be expressive enough to capture any possible property to be verified, and formal enough to avoid vagueness or ambiguity. Requirements come in the form of plain text written in human-spoken languages, which is not useful due to the lack of formalism required for model checking. LTL and CTL are formal ways to express linear-time (LT) properties for TS [7, 89]. The main difference between LTL and CTL is the concept of 'time'. In LTL time is linear and only one successor can follow each state, in CTL time is considered to be able to branch thus allowing more than one successor. CTL made possible the inclusion of explicit time-values in the verification process (in the form of clocks), which is not possible in LTL where time is rather a concept with no specific value to be verified.

Some properties can be expressed in one and not in the other and equivalences exist between certain properties as well. For a better understanding of both refer to [7] and [89].

2.3.3 Model Checking and Hybrid Systems

The verification of the system and its properties is performed in the model of the system, not in the product itself or the prototype. This is an important point because it means the model checking approach and the results obtained by it are only as good as the model. Therefore, the verification is limited by how good the model is and by what kind of properties can be expressed with the formal language [7]. Depending on the type of properties to be verified and the system component's interaction a modelling approach is selected. The selected approach must preserve the properties of interest for the problem to be addressed. Regarding dynamic control systems, transient behaviours and interactions among the various components are of important significance when designing and verifying the system compliance with design requirements. It is therefore desirable that the formal model is able to capture the system dynamics as closely as possible to the real system.

Regarding dynamic control systems and due to the nature of model checking, an exhaustive verification technique, the type of formal models to address a dynamic control problem become restricted because of memory limitations. High fidelity commercial off-the-shelf models used for control design purposes are not suitable for model checking. For this reason, a system abstraction is the usual approach to construct a formal model of a system which in turn is suitable for model checking. The system abstraction to be implemented in model checking must preserve as closely as possible the dynamic system properties and interactions of interest in the dynamic control problem.

2.3.4 Timed-Automata and Computation Tree Logic

The following section gives a brief description of the type of automata to be used in this thesis and the formal language used to express the properties of the systems modelled using the automata.

Timed-Automata

A timed-automaton is a finite-state machine extended with clock variables. A system can be modelled as a network of timed-automata. Clock variables evaluate to a real number and all the clocks in the system progress synchronously. A state of the system is defined by the locations of all automata, clock constraints, and the values of the system variables.

Every automaton may fire an edge independently or via a synchronize operation with another automaton [13].

A timed-automaton (TA) is a tuple *(L,l_0,C,A,E,I)*, where:

- L is a set of locations or states.
- l_0 is the initial location.
- C is a set of clocks.
- A is a set of actions.
- E is a set of edges between the locations. Edges are constituted by actions, a guard, and a set of clocks to be reset.
- $I: L \rightarrow B(C)$ assigns invariants to locations.

B(C) is the set of conjunctions over simple conditions of the form $x \bowtie c$ or where x is a clock and c is an integer value. The allowed operations for $\bowtie$ are $\{<, \leq, =, \geq, >\}$.

Consider the following example of a TA modelling the behaviour of a gate that opens and closes. The timing requirements for the open and closing behaviour of the gate are modelled using the clock x.

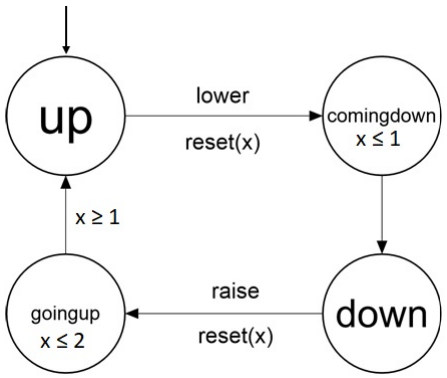

Fig. 2.4 Timed-Automaton example of a gate that open and closes on request. The timing requirements regarding the opening and closing actions are modelled using the clock x.

The automaton definition is given by:

- L = {up, comingdown, down, goingup}.
- l_0 = {up}.
- C = {x}.

- A = {lower, raise}.
- E = {up $\xrightarrow{true:lower,x}$ comingdown, comingdown $\xrightarrow{true:\tau,\varnothing}$ down, down $\xrightarrow{true:raise,x}$ goingup, goingup $\xrightarrow{x\geq 1:\tau,\varnothing}$ up}.
- I = {comingdown $\longrightarrow x \leq 1$, goingup $\longrightarrow x \leq 2$}.

The model checker UPPAAL is used to conduct the formal verification and design activities in the work presented in this thesis. UPPAAL uses TA to model systems. The particular implementation of TA in the model checker UPPAAL contains semantics and definitions particular to the tool. For further reference about TA in UPPAAL and how to construct networks of TA to model systems refer to [13, 41].

Computation Tree Logic

In this section the basic syntax and semantics for CTL are presented. CTL has a two stage syntax, state formulae and path formulae [7, 35]:

- State formulae: assertions about the atomic propositions (AP) in the states and their branching structure.
- Path formulae: express temporal properties of paths.

CTL *state formulae* over the set of AP are formed according to the following grammar:

$$\Phi ::= true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

where $a \in AP$ and φ is a path formula. $\exists$ is the *exist* path quantifier (for *some* path), and $\forall$ is the *all* path quantifier (for *all* paths).

CTL *path formulae* are formed according to the following grammar:

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \bigcup \Phi_2$$

where Φ, Φ_1, and Φ_2 are state formulae, $\bigcirc$ is the *next* operator, and $\bigcup$ is the *until* operator .

Consider the following example. Let $AP = \{x = 1, x < 2, x \geq 3\}$ be a set of atomic propositions. The following are examples of syntactically valid CTL formulae:

- $\exists \bigcirc (x = 1)$.
- $\forall \bigcirc (x = 1)$.
- $(x < 2) \vee (x = 1)$.

- $\exists((x < 2) \bigcup (x \geq 3))$.

State formulae express a property of a state, while *path formulae* express a property of a path (e.g. a sequence of state). Formula $\bigcirc\Phi$ holds (evaluates to true) if Φ holds at the *next* state in the path. $\Phi_1 \bigcup \Phi_2$ holds for a path if there is some state along the path for which Φ_2 holds, and Φ_1 holds in all the states prior to that state. By combining both *state* and *path* the properties verification can take place. For example, $\exists\varphi$ holds in a state if there exists *some* path satisfying φ that starts in that state, and $\forall\varphi$ holds in a state if *all* paths that start in that state satisfy φ.

Figure 2.5 shows a graphic representation of the *state formulae* and *path formulae* combination for the $\forall\varphi$ and $\exists\varphi$ cases. Figure 2.5-a shows how φ is satisfied in a state for all the available paths beginning in state *Init*. Figure 2.5-b shows how there exists at least one path beginning at state *Init* where φ is satisfied in a state.

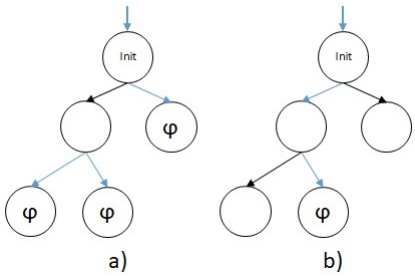

Fig. 2.5 Graphic representation of *state formulae* in combination with *path formulae*. a) $\forall\varphi$ case and b) $\exists\varphi$ case.

The model checker UPPAAL works with a subset of CTL. The query language of UPPAAL uses basically a combination of *path formulae* and *state formulae* with the here presented path quantifiers. The syntax of the query language is particular to the tool, for further reference see [13, 41].

The type of properties that can be verified using TA and CTL with the model checker UPPAAL are (this is revisited in Section 4.4 again):

1. *Reachability:* It is possible to reach a system state.
2. *Safety:* Something can never happen.
3. *Liveness:* Something will eventually happen.

They all check different behaviours in the system, by using a combination of these properties the design and verification problem for a dynamic control system can be portrayed. The type of automata (TA), properties formal language (CTL), and available properties will drive the modelling approach.

Remarks

A brief description of the type of models and formal language that will be used in the novel formal design and verification framework developed in this thesis was presented. For a more detail description of both modelling approaches and formal languages refer to [7, 35]. For a more detailed description of the modelling tool UPPAAL and its use refer to [13, 41].

2.4 Final Remarks

The design and verification of safety-critical applications are challenging activities. Formal methods and model checking can provide tools to help improving the current design and verification processes for safety-critical software applications. Dynamic control systems present challenges from both modelling aspects and requirements formalization in order to be fully addressed in a formal manner. The state-space explosion problem in model checking is perhaps the main limitation when modelling, particularly this makes the use of floating-point data and its arithmetic an open challenge for current model checking tools and techniques: computational memory is still a major limitation.

The necessity of generating a system abstraction arises, where such abstraction must maintain the particular features of interests to be formally verified. In dynamic control systems, transitory and dynamic behaviours drive the level of abstraction to be used. Regarding feedback control loops it is desirable to be able to reason about the plant and the controller individually, particularly when designing the controller. Current safety-critical control applications rely on known control schemes such as PID gain scheduled controllers; one of the main reasons for this is the certification of this type of software.

The incorporation of formal methods and model checking into the software life-cycle for safety critical control applications requires the generation of easy to follow practices and frameworks. In this way control and software engineers can embrace the use of formal methods more easily and the benefits of their use to be fully exploited. The work developed in this thesis targets the formal design and verification of discrete gain scheduled PID controllers. The selected type of models and formal methods tools to address this problem were presented in this chapter. The following chapters explain the methodology in detail, from the selected approach to generate the system abstraction to the formal modelling of the system in the model checker environment. The methodology is presented within the context of an aerospace safety-critical control application.

Chapter 3

Dynamic System Abstraction Methodology

3.1 Overview

Model checking is only as good as the model being used. As mentioned in Chapter 2 the main limitation of model checking is the *state-space explosion* problem: when the system behaviour is expanded to its full possible states, the number is too high to perform an exhaustive analysis over them [7, 35]. The use of the floating-point data type is part of this limitation and currently no model checker fully supports this data type [108, 153]. This limitation makes it hard to simulate dynamic control systems and limits the use of existing engineering models. The common approach to solve this limitation when addressing dynamics systems in a model checking environment is to generate an abstraction of the system dynamic [47, 86, 134, 152]. The abstraction must contain the desired dynamic behaviour related to the problem to be verified. The abstraction therefore is a reduced version of the original model. This usually means a change of coordinates by mapping the original search space into another one where model checking can solve the problem. However, it is desirable to use the same coordinates in the system abstraction to avoid one more level of transformation. In this way, when requirements are translated into properties no extra conversion is required and the reasoning can be performed using the original coordinates.

To overcome the floating point limitation, a model abstraction constructed with a fixed point data type representation is proposed. To construct the fixed point representation only integer data is used. By doing so the same coordinates are preserved but the search space is limited so that model checking can address the design and verification problem. The drawback of using this approach is the level of resolution that can be obtained when using

integer data to represent real numbers. Nonetheless the approach enables the use of well-known dynamics models within a model checking framework. By using this abstraction methodology the PID controller gain scheduling problem formulation can be addressed using a formal verification approach with model checking. The key contribution of this chapter is therefore presenting the underlying abstraction methodology that underpins the formal design and verification approach for a gain scheduling control scheme in Chapter 6.

This chapter is structured as follows: Section 3.2 presents the type of dynamic models to be used in the abstraction. Section 3.3 presents the fixed point representation and the considerations when using it. Section 3.4 presents the error compensation approach to be used by the methodology in order to provide safety guarantees when the final abstraction is to be used. Section 3.5 presents the full abstraction methodology and presents an example of its use. Finally, Section 3.6 presents final remarks and discusses the abstraction methodology.

3.2 Discrete-time SISO LTI Models

Considering a control scheme such as the one in Figure 2.1 and inspired by a classical control approach, a discrete representation of the entire system is to be considered. The controller is typically implemented in a computer-based system hence a discrete implementation is fairly common. It is thus convenient to consider the entire system in the discrete domain and address the problem in this domain. Also, from a model checking implementation point of view, it is convenient to implement a discrete solver rather than a continuous one (e.g. ODE solver).

In order to model dynamic systems (e.g. a closed control loop) and to address the gain scheduling control design problem, discrete SISO LTI models have been selected. The results can be generalized to the multi-variable case but it is not in the scope of this work. Discrete-time SISO LTI deterministic models can be described by an auto-regressive with exogenous input (ARX) model (Equation 3.1). Using the z transform notation:

$$\frac{Y(z^{-1})}{U(z^{-1})} = \frac{b_1 z^{-1-n} + b_2 z^{-2-n} + ... + b_{n_b} z^{-n_b-n}}{1 + a_1 z^{-1} + a_2 z^{-2} + ... + a_{n_a} z^{-n_a}} \tag{3.1}$$

where:

$$b_i, a_i \in \mathbb{R}$$
$$Y, U \in \mathbb{R}$$
$$n, n_a, n_b \in \mathbb{Z}^+$$

where the output of the system is Y, the input is U and the system response delay to the input is n. The order of the system is determined by the number of the coefficients a (n_a) and b (n_b). The output calculation is therefore the weighted sum of previous input and output values:

$$Y(k) = \sum_{i=1}^{n_a} a_i Y(k-i) + \sum_{i=1}^{n_b} b_i U(k-i-n) \tag{3.2}$$

Every element in the control scheme (Figure 2.1) can be described by a SISO LTI. In this manner the interaction between all the elements can be constructed by solving the recurrence equations of the system components.

Inputs, outputs, and coefficients are real numbers which are best represented by floating-point variables. The use of the floating-point data-type is currently very limited for model checking due to the state space explosion problem. The range of a floating-point variable is potentially infinite which makes the search space grow exponentially leading to a *state-space explosion*. To overcome this limitation a fixed-point representation which uses integer data-type is proposed. The following section explains the considerations in order to select a fixed-point representation for the SISO LTI models.

3.3 Fixed Point Representation Using Integer Data

When considering modelling a dynamic control system the obvious numeric representation of the related variables is real numbers. For computational purposes real numbers are expressed as either floating-point data or fixed-point data. The use of integer data to represent real-valued variables usually implies a transformation or equivalence between the two domains - e.g. a change of coordinates. Within a model checking framework dealing with floating point data is an open problem and no model checker fully supports this data type [108, 153]. On the other hand integer data type is supported by most model checkers even if with some restrictions (e.g. data type size).

A fixed-point representation using integer data-type is proposed to overcome the floating-point limitation in model checking. Limiting the operating range of the system variables and using a scaling approach to recover a fixed number of digits with integer data type, a fixed-point representation is constructed avoiding the necessity of floating-point data [66]. The selection of the fixed-point representation is driven by:

- Operating space range: input U and output Y range.
- Resolution: number of integer and fractional digits.
- Data operations: addition, subtraction, etc.

- Available size in integer data type (e.g. 16-bit signed, 16-bit unsigned).

The previous considerations will determine the type of ad hoc fixed-point data representation needed. The following sections explain how these considerations have an effect on the final fixed-point selection and its implementation.

3.3.1 Data Types for Data Representation

Input U, output Y, a and b coefficients related to the discrete SISO LTI models directly affect the first two items in the considerations listed in Section 3.3. Depending on the numbers and range to be represented a suitable format must be chosen. Table 3.1 shows the data ranges for different integer data sizes. The available number of digits and range will influence the decision of where to separate the integer part from the fractional part in the fixed point representation.

Table 3.1 Integer data type ranges.

Bits	Available Digits	Signed		Unsigned	
		Lower Limit	Upper Limit	Lower Limit	Upper Limit
8	3	-128	127	0	255
16	5	-32768	32767	0	65535
32	10	-2147483648	2147483647	0	4294967295

Table 3.2 shows possible configurations for a fixed-point representation using a 16-bit integer data type. Five digits are available to use in both the integer and fractional parts. Choosing one configuration within the available options depends on the range and resolution needed.

Table 3.2 Possible fixed point representations with 16-bit integer data type

Integer Digits	Fractional Digits	Signed		Unsigned	
		Lower Limit	Upper Limit	Lower Limit	Upper Limit
4	1	-3276.8	3276.7	0	6553.5
3	2	-327.68	327.67	0	655.35
2	3	-32.768	32.767	0	65.535
1	4	-3.2768	3.2767	0	6.5535

The way data is chosen to be represented is highly relevant to the information listed in Table 3.1 and Table 3.2. If the number of digits required to represent the data (in whichever

configuration for integer and fractional parts is chosen) is higher than 5, then 16-bit integers are not suitable for representation purposes. Following the same reasoning, if the number of total elements (e.g. data range) is higher than 65,535 (either positive or negative values) 16-bit integers are not suitable to fit such data range. A compromise involving range and presentation must be made when selecting how to represent the data.

3.3.2 Fixed Point Data Size Considerations

To calculate the system dynamics using a discrete SISO LTI model a weighted sum of previous input and output values has to be performed (Equation 3.2). Performing this operation affects the required size of the data type to be used. Regardless of the choice for the fixed-point representation (e.g. number of digits for integer and fractional parts) the operations are performed using integer data. Therefore it is important to consider the limits of the chosen data representation and the type of operations to be performed with such data to ensure there is no overflow, risking the end result of the operation. These considerations will determine how to perform arithmetic operations with the selected data type.

When considering arithmetic operations using integer data the risk of overflow has to be assessed. Multiplication and addition are the main risk of potential overflow. Table 3.3 shows the required size (binary case) to store the result (C) when performing addition (subtraction is considered to be a signed addition), multiplication, and divide operations with two integer numbers (A, B) of size *n* for both upper and lower limits worst cases.

Table 3.3 Size considerations for data operations - binary case.

Inputs		A+B	A*B	A/B
		Output		
A	B	C	C	C
2^n	2^n	2^{n+1}	2^{2n}	2^0
2^n	2^{-n}	2^{2n}	2^0	2^{2n}

Performing addition and subtraction operations for a chosen fixed-point representation using integer data is a straightforward procedure (e.g. using primitive operations with integer data will provide the fixed-point result). Multiplication can also be performed using primitive operations but the result will need a correction to compensate for the magnitude of the integer part in the data representation. Most importantly a consideration of size to store the result of the operation has to be considered. From Table 3.3 it can be seen that when multiplying two integers of size *n* a *2n* result will be generated.

Divide operations with integer data also present an overflow risk (Table 3.3), as in the multiplication case a possible size of *2n* to store the result may be necessary. However, the fixed-point representation has other implications regarding overflow and data size. Consider the fixed-point representation in Table 3.4. The total number of bits (binary case) is equal to the sum of the number of bits in the integer and fractional parts.

Table 3.4 Fixed point representation - binary case

Total Bits	♯ Integer Bits	♯ Fractional Bits	Representation
n = i+f	i	f	$2^i.2^{-f}$

Considering Table 3.4 the necessary bits to accommodate the largest result from a divide operation A/B is 2n-i (Table 3.5).

Table 3.5 A / B = C: Largest possible result - binary case

A	B	A/B=C	Required Representation	♯ Bits Required
2^i	2^{-f}	$\frac{2^i}{2^{-f}} = 2^{i+f}$	$2^n.2^{-f}$	n+f=2n-i

These considerations will determine the data type size required to correctly use a fixed-point representation and perform arithmetic operations. Arithmetic operations are also driven by the selected representation. Some operations may be performed with standard primitive integer data operations. The data type availability in the model checker, the type of arithmetic operations to be performed with such data are relevant considerations to the methodology because they will determine if the chosen fixed-point representation and its arithmetic operations can be correctly implemented. However, if the required size is bigger than the available data size in the model checker, an ad hoc implementation to compensate for such limitations will be required.

3.3.3 Fixed Point Arithmetic Using Integer Data

In order to properly perform data operations with a fixed-point representation which uses integer data only the size restrictions from Section 3.2, specifics of the fixed-point representation (e.g. digits/bits in fractional and integer parts), and the available data types must be considered. Table 3.6 shows the required size to perform arithmetic operations for a given number of digits. The limitation is mostly driven by the multiply operation requirement. The number of bits needed to perform a multiplication is double the size of the bits needed to represent a number.

Table 3.6 Data size requirements.

Representation			Operations		
Data Type	Bits	Digits	Data Type	Bits	Digits
Byte	8	3	Word	16	5
Word	16	5	Long	32	10
Long	32	10	Longword	64	19

The required data type is thus the one needed to perform arithmetic operations. To have a 5 digit representation (in whichever way the digits are distributed for integer and fractional parts) with a 16-bit data range (e.g. -32,768 to 32,767) a total of 32 bits are needed to perform data operations; hence *Long* data type is required. A case can be made when restricting upper and lower bounds in the data but the reasoning is the same. In general, the number of bits needed for operations are double the number of bits used to represent the highest magnitude value in the data. The naming convention may vary from computer language or platform but the consideration is driven by the number of bits needed to perform data operations.

Considering the information in Table 3.4 and Table 3.6, to use a 5 digit fixed point representation using the full range of a signed 16-bit integer, 32 bits are required. If only 16-bit signed integers are available in the platform an ad hoc arithmetic is required to correctly represent and perform arithmetic operations with the data.

3.3.4 Ad Hoc Data Type

The purpose for generating an ad hoc data type is to be able to run the discrete time solver for the SISO LTI models presented in Section 3.2 using 16-bit integer only data. In this manner the dynamic models can be implemented in a model checker environment even if only 16-bit integer only data is available. This is a common limitation in model checking because increasing the data type size availability could potentially turn into a state space explosion due to an even bigger data range. Solving a recurrence equation with the form of Equation 3.2 requires addition, subtraction, and multiplication operations. To correctly perform all these operations considering a 5-digit fixed-point representation using 16-bit signed integer data type, the data representation from Table 3.7 is proposed. This ad hoc data type allows to use 5 digits for data representation with the range of a signed 16-bit integer. This allows a magnitude of over 30,000 values to be distributed as required depending on the fixed-point selection. Also, the ad hoc data type allows to perform addition, subtraction, and multiplication arithmetic operations which are necessary to solve the recurrence equation of a discrete SISO LTI model.

Table 3.7 Ad hoc data type for abstraction

Limits		Required Bits			Signed Integer Elements
Lower	Upper	Sign	Magnitude	Operations	
-32767	32767	1	15	30	2

By modifying the lower limit of the data to -32,767 instead of the native -32,768 limit of a 16-bit signed integer, two 16-bit signed integers are used to perform operations within the specified range. The sign bit is handled independently so that data representation is strictly limited to use 15-bit magnitude range.

A discrete SISO LTI model structure (Section 3.2) has been proposed to model the system's dynamics in combination with an ad hoc fixed-point data type (Section 3.3.4) in order to perform data operations without the need of floating-point data. The combination of the model structure with the fixed-point data type representation will enable the use of such models in an environment where neither floating point nor fixed-point (in its native form) data are available, enabling the use of this type of models in a model checking environment. By enabling the use of discrete SISO LTI models in a model checking environment will allow to formally verify typical dynamic control systems.

In order to provide guarantees when using the proposed abstraction, the amount of error added by the approach has to be considered. The following section explains the type of inaccuracies to be considered during the modelling process.

3.4 Modelling Error Compensation

In order to provide safety guarantees when using the model abstraction, modelling errors, and data inaccuracies have to be integrated in the modelling process. The model abstraction will be used in a model checking environment to reason about the performance of the original control system, therefore it is important to provide safety guarantees when performing formal verification and validation of the system. Therefore modelling errors and data type rounding effects have to be considered before using the model abstraction to reason about the behaviour of the system. The following sections explain the three different type of modelling errors to be considered and how to address them.

3.4.1 Parametric Compensation - ε_1 Error

Considering the continuous domain closed loop control system in Figure 3.1, it is assumed that:

- This is a single variable feedback control problem. The process *G(s)* is regulated by the controller *C(s)*.
- The controller structure *C(s)* is known.
- The process structure *G(s)* is unknown thus subject to modelling.
- The process is modelled by a transfer function in the continuous-time domain.

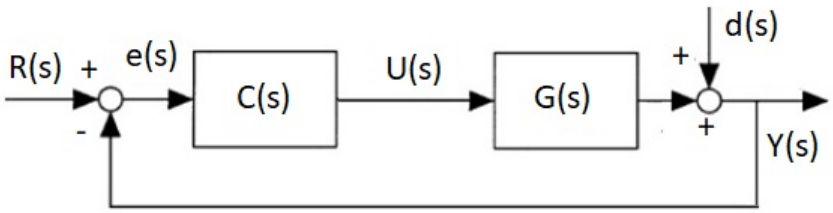

Fig. 3.1 Generic continuous time closed loop control system.

Taking into account the previous assumptions, the modelling error can be considered in two forms: additive and relative [1].

Considering:

a G(s): True Plant.

b G_0(s): Nominal Model.

The *Additive Model Error* is given by:

$$G_{E_A}(s) = G(s) - G_0(s) \tag{3.3}$$

Considering *Additive Model Error* (Equation 3.3) the true plant is given by:

$$G(s) = G_0(s) + G_{E_A}(s) \tag{3.4}$$

where $G_{E_A}(s)$ can be considered as extra dynamics in the nominal model $G_0(s)$ and factored in with it. The true plant can therefore be represented as:

$$G(s) = G_0(s)G_\Delta(s) \tag{3.5}$$

$G_\Delta(s)$ can represent different types of dynamic errors:

1. Gain error - $G_\Delta(s) = (1 + K_U)$
   Thus $G(s) = G_0(s)(1 + K_U)$

2. Delay and gain error - $G_\Delta(s) = (1 + K_U)e^{-sT_U}$
   Thus $G(s) = G_0(s)(1 + K_U)e^{-sT_U}$

3. Pole and gain error - $G_\Delta(s) = \frac{(1+K_U)}{(\tau_P s+1)}$
   Thus $G(s) = G_0(s)\frac{(1+K_U)}{(\tau_P s+1)}$

4. Zero and gain error - $G_\Delta(s) = (1 + K_U)(\tau_Z s + 1)$
   Thus $G(s) = G_0(s)(1 + K_U)(\tau_Z s + 1)$

where:

- K_U = Uncertainty gain magnitude.
- T_U = Uncertainty time delay magnitude.
- τ_Z = Uncertainty zero time constant.
- τ_P = Uncertainty pole time constant.

The following assumptions regarding the modelling uncertainties $G_\Delta(s)$ are made:

- The plant is modelled as a stable system.
- Tolerance errors do not change the structure of the plant.
- The error has the same order as the plant: no zeros or poles added. This may lead to closed-loop instability due to pole shifting.

These assumptions reduce the type of modelling errors to be considered, so only *Gain error* remains. Only this type of error will be considered when compensating the plant. The proposed models for the system abstraction are of discrete nature (Section 3.2). The same reasoning about modelling error applies in a discrete formulation, Figure 3.2.

Considering *Gain Error* only, the true plant *G(z)* can be calculated as:

$$G(z) = G_0(z)(1 + K_U) = G_0(z) + K_U G_0(z) \tag{3.6}$$

In open loop form, the plant model to be considered in this problem formulation is the one in Figure 3.3.

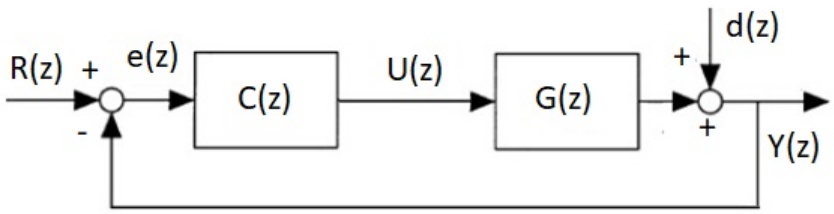

Fig. 3.2 Generic discrete time closed loop control system.

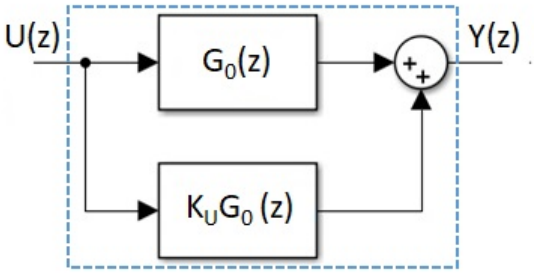

Fig. 3.3 Open loop representation of the plant model with gain error compensation.

The parametric compensation will be given by the selected value of K_U.

$$K_U \in \mathbb{R} : -\delta \leq K_U \leq \delta \tag{3.7}$$

The magnitude of K_U will determine the amount of system's gain uncertainty to take into account. The bigger the magnitude of K_U the greater the distance between the nominal model $G_0(z)$ and the abstraction. This could lead to having a model abstraction which is too conservative so the real system dynamics are far from the abstraction. In the same way, a small K_U value can cause that the real system dynamics are not included in the abstraction.

The purpose of generating the abstraction is to enable addressing a dynamic control design problem in a model checking environment. Both extremes (too small or too big a value fo K_U) could lead to false results in the model checker due to being too conservative or too tight of an abstraction. After compensating the model dynamics, the data type restriction has to be addressed. The following section explains how rounding effects to generate a fixed-point representation contribute to the amount of error the abstraction will contain and how to address it.

3.4.2 Fixed Point Representation Compensation - ε_2 Error

The proposed data type for representation and data operations is fixed-point but using integer data type only. In this way the necessity for fixed-point or floating-point data is removed from the approach, making the abstraction suitable for a model checking implementation. The amount of digits selected for the integer part and for the fractional part will determine how much error is added by using a fixed-point representation instead of floating-point. When using fixed-point data a rounding or truncating operation has to be performed to limit the data to the selected representation, this amount of error has to be accounted for.

In general, when using a fixed-point representation for a real number X described by Equation 3.8:

$$X = 2^i.2^{-f} \tag{3.8}$$

with i bits for the integer part and f bits for the fractional part. The amount of error ε_2 when truncating or rounding off a datum's value is given:

$$|\varepsilon_2| = \frac{1}{2}2^{-(f_b+1)} \tag{3.9}$$

Alternatively, in decimal form:

$$|\varepsilon_2| = \frac{1}{2}10^{-(f_d+1)} \tag{3.10}$$

The amount of error per datum is described by Equation 3.9 in *binary* base and Equation 3.10 in *decimal* base. Thus, error is then bounded by the limits in the *decimal* base form:

$$-\frac{1}{2}10^{-(f_d+1)} \leq \varepsilon_2 \leq \frac{1}{2}10^{-(f_d+1)} \tag{3.11}$$

When using the fixed-point representation the amount of error added by the selected format has to be considered and accounted for in order to properly bound the arithmetic operations in the proposed models to describe the system's dynamics.

3.4.3 Scaling Compensation - ε_3 Error

When using integer data to perform arithmetic operations with a fixed-point representation, certain considerations have to be made when performing multiply operations:

- The original fixed-point data has to be scaled to recover the fractional part. This is equivalent to performing a left-shift operation.
- Multiply operations have to be performed differently: multiply then divide.

- The extra divide operation concerns the effect of scaling. Such effect has to be removed once the operation has been performed.

Consider any number in a fixed-point representation, where every I element represents an integer digit and every F element a fractional digit. In order to bring n fractional digits into the integer part, the original number must be multiplied by 10^n. In this way the fractional part can be represented using an integer number:

- A = I.FFFF. B = I.FFF.
- $n_A = 4$. $n_B = 3$.
- 10^{n_A} = 10,000. $10^{n_B} = 1,000$.
- $A' = A \times 10^{n_A}$.
- $B' = B \times 10^{n_B}$.

A data representation format has to be chosen depending on the range of the variables involved in the operations. This selection has to be done beforehand to make sure the results fit in the chosen format. Disregarding how this is done, assume the selected integer-only format consist of 5 digits: 1 integer digit (I = 1) and 4 fractional digits (F = 4). This means that the result (C) from the multiply operation will also be represented in this format. In the same way as number *(A)*, a value of $10^4 = 10,000$ is required to bring this number into the integer domain:

- C = I.FFFF.
- $n_C = 4$.
- $C' = C \times 10^{n_C}$.

The operation $A \times B = C$ is thus performed in the following way:

$$A' \times B' = (A \times 10^{n_A})(B \times 10^{n_B}) = C \times 10^{n_A + n_B} \tag{3.12}$$

Because $n_A = n_C$:

$$A' \times B' = C' \times 10^{n_B} = C'' \tag{3.13}$$

To recover *C'* from *C"* an extra division is required:

$$A' \times B' = \frac{C''}{10^{n_B}} = C' \tag{3.14}$$

When multiplying to scaled-up values and portraying the result in the format of one of the elements involved (A element in the previous example), the result must be divided by the scaling factor of the other element (B element in the previous example). Using the ad hoc data representation from Section 3.3.4 (two 16-bit signed integers to construct a 30 bit unsigned integer), doing division by performing bitwise right-shift operations makes computations easier. Choosing a scaling factor as a power of 2 will make sure that removing the scaling effect can be done by performing bitwise right-shift operations.

Representing coefficients a and b from Equation 3.2 in the ad hoc format requires a scaling factor of 10^4 to recover 4 fractional digits. This value (10,000) is not a power of 2, hence a value of 16,384 (2^{14}) is selected because it is the closest power of 2 that allows to recover 4 fractional digits. Removing this scaling effect from a multiplication becomes a 14 bit right-shift operation.

Because the scaling process is performed using a 2^{14} factor, rounding effects are added and hence the scaled value is subject to error. Consider the following example where the value A will be converted into the ad hoc fixed-point format which uses integer data only:

$$A = 1.8529$$
$$A_1 = A \times 10^4 = A \times 10,000 = 18,529.0$$
$$A_2 = A \times 2^{14} = A \times 16,384 = 30,357.9$$

Value A_1 fits exactly into the 5 digit format but value A_2 does not. Depending on the rounding operation (e.g. up or down), the result could either be 30,357 or 30,358. This is the nature of the scaling factor error: the rounding operation in the scaling process when a scaling gain which is not a power of 10 but a power of 2 is used. The power of 2 gain is used for the sake of performing divisions with shift operations. The rounding effect contributes to error. In a similar manner as for the calculation of ε_2 (Section 3.4.2) the amount of error is given by the number of digits used in the selected data representation. Thus, the amount of error added by the ad hoc data type ε_3 is described by:

$$|\varepsilon_3| = \frac{1}{2}10^{-(f+1)} \tag{3.15}$$

Rounding up or down will generate different results and for this reason representing error in the form of boundaries is a better approach:

$$-\frac{1}{2}10^{-(f+1)} \leq \varepsilon_3 \leq \frac{1}{2}10^{-(f+1)} \tag{3.16}$$

Three different sources of error when translating the original model into the discrete model using the ad hoc data type have been described. The following section explains how to consider all these sources of error and generating bounds for the system abstraction.

3.4.4 Global Error

Discrete SISO LTI models have been selected to describe the system dynamics. A fixed-point data representation which uses integer data type only will be used to overcome the floating-point data restriction in model checking. Three different sources of error regarding the modelling process have been described and will be considered when implementing the system abstraction:

1. ε_1 - Parametric: modelling inaccuracies, *Gain Error* only.
2. ε_2 - Fixed Point: data representation error.
3. ε_3 - Ad hoc data type: scaling factor error.

Error ε_1 is determined by gain K_U (Equation 3.7) which is a design parameter to be selected. Errors ε_2 and ε_3 have the same magnitude. The ad hoc data representation (Section 3.3.4) consist of 5 digits total, using 4 digits for the fractional part. Therefore the value for f in Equations 3.10 and 3.15 is 4, which leads to:

$$|\varepsilon_2| = |\varepsilon_3| = \frac{1}{2}10^{-5} \tag{3.17}$$

Considering Table 3.2 the selected data representation can be ignored since all data will be handled as integers. The value of f is thus 0 when considering every data as an integer number, therefore the error compensation can be done in the integers' domain:

$$|\varepsilon_2| = |\varepsilon_3| = \frac{1}{2}10^{-1} \tag{3.18}$$

Considering the discrete SISO LTI model given by Equations 3.1 and 3.2, the global error can be included in the coefficients a_i and b_i. Coefficients a and b can be referred to generically as c coefficients where each c_i coefficient will be calculated as follows:

$$\bar{c_i} = c_i(1 \pm |\varepsilon_1|) \pm |\varepsilon_2| \pm |\varepsilon_3| = c_i(1 \pm |K_U|) \pm |\varepsilon_2| \pm |\varepsilon_3| \tag{3.19}$$

According to Equation 3.19, intervals have to be calculated when adding the error compensation into the coefficients. The following section explains how these intervals are calculated.

3.5 Safety Guarantees

In whichever way the system behaviour is represented in the model checker environment, guarantees about the accuracy of what the model checker is saying about the system must be provided. Using a fixed-point representation implemented with integer data allows the use of the same model structure (discrete SISO LTI) to calculate the control system dynamics without making any coordinates change in the process. However, modelling errors are present along with rounding errors due to the use of fixed-point to represent floating-point data.

In order to compensate the modelling process, three sources of error have been identified (Section 3.4). Including the global error into the modelling process will provide a level of confidence when using the model abstraction for design and verification purposes. Incorporating Equation 3.19 into Equation 3.1:

$$\frac{Y(z^{-1})}{U(z^{-1})} = \frac{\bar{b_1}z^{-1-n} + \bar{b_2}z^{-2-n} + ... + \bar{b_{n_b}}z^{-n_b-n}}{1 + \bar{a_1}z^{-1} + \bar{a_2}z^{-2} + ... + \bar{a_{n_a}}z^{-n_a}} \tag{3.20}$$

Each coefficient has to be recalculated to include the global error effect. The compensation process consists of the selection of a K_U value and using it in combination with Equations 3.18 and 3.19 to come up with a new coefficient value. Given there are three sources of error and the effect is to add or subtract the amount of each type of error, there are 8 possible combinations for how to apply the add/subtract operations, Table 3.8:

Table 3.8 Possible compensation operations considering the aforementioned sources of error.

	Type of Error		
	ε_1	ε_2	ε_3
Operation	+	+	+
	+	+	-
	+	-	+
	+	-	-
	-	+	+
	-	+	-
	-	-	+
	-	-	-

The purpose of the error compensation is to take into account possible modelling errors and encapsulate the true behaviour of the process under modelling. An interval arithmetic approach is proposed [110, 111]. In this way upper and lower bounds for the model will be provided. A criteria to determine how to calculate upper and lower bounds for the coefficients compensation is required. All the vertices of uncertainty in Table 3.8 have to be checked to

find the extreme values (maximum and minimum) which will correspond to the upper and lower bounds.

3.5.1 Over and Under Approximation

Using an interval arithmetic [110, 111] approach to calculate the upper and lower bounds for the coefficients compensation will result in two different models: over approximation (upper bound) and under approximation (lower bound). The plant behaviour is then bounded by both the *under approximation* and the *over approximation*.

The final value of the system output to a step input (using the *final value theorem* presented in Section 2.3.1) is selected as the criterion to determine the upper and lower bounds for the system abstraction. This criterion is selected because the type of parametric compensation (Section 3.4.1) in this abstraction methodology only considers a system gain compensation (ε_1 error), which in turn has a direct effect on the settling value of the system.

By applying Equation 2.2 with all the combinations in Table 3.8, the models with the extreme gain values (e.g. minimum and maximum) will correspond to the *under approximation* and the *over approximation* respectively. Finally, from a continuous SISO LTI model a discrete SISO LTI abstraction can be generated. The following section explains this procedure.

3.5.2 Abstraction Generation

The system abstraction consists of an *under approximation* and an *over approximation* of the original system. To recover the original system response both over and under approximations are needed because they provide bounds for the original system. The system abstraction is generated so a feedback control problem can be addressed in a model checking environment. Modelling errors and computational errors are taken into account to compensate for inaccuracies and include them in the final abstraction. Algorithm 1 indicates the inputs, outputs, and steps in order to generate the model abstraction.

From a continuous time SISO LTI model an abstraction which is suitable for a model checking environment implementation is generated. The model abstraction consists of two discrete SISO LTI models, and two scaling gains (K_{ab} and K_S). The model abstractions use integer data only to represent the models' coefficients.

Data Processing Cycle

Once the abstraction has been generated it can be implemented to process the input/output relationship given by the transfer functions of the over and under approximations. The data

Algorithm 1: Over and Under Approximation Generation Procedure.

Input : Continuous SISO LTI Model, Sampling Period (T), K_U gain.
Output : Over and Under Approximations: Discrete SISO LTI, coefficients scaling gain K_{ab}, Input/Output scaling gain K_S.

1 Digitize the SISO LTI model using sampling period T.
2 Define operating space of the model: Input-Output range.
3 Based on Step 2 range, select fixed-point representation (I, F).
4 Based on Step 3 fixed-point representation select Input-Output scaling gain K_S.
5 According to the selected fixed-point representation, convert the discrete SISO LTI model to fixed-point: round-off to the selected amount of fractional digits F.
6 Verify non-zero fractional digits in coefficients a and b can be recovered using the selected fixed-point representation with scaling K_{ab} gain.
7 Based on Step 3 fixed-point representation, calculate the value of the coefficients scaling gain K_{ab}: closest value to 10^F which is a power of 2 .
8 Recalculate coefficients using Equation 3.17, K_U, and Table 3.8.
9 Apply final gain value theorem (Equation 2.2) to all generated transfer functions.
10 Select the transfer functions with the highest and lowest final value gains from Step 9.
11 Scale coefficients using K_{ab} gain and round off to integer values.

cycle for processing the system output requires an update to the normal data cycle when using discrete SISO LTI models (given by Equation 3.2).

To calculate the output for a discrete SISO LTI model described by Equation 3.2, a weighted sum of previous inputs and outputs values must be performed. The general idea of the process is described by Algorithm 2. Whether the system consists of one or more discrete SISO LTI models the general data flow described by Algorithm 2 has to be maintained.

Algorithm 2: Discrete SISO LTI Data Processing Cycle.

Input : Discrete SISO LTI model(s): a and b coefficients, Total run time (iterations), Input signal(s) U.
Output : Output signal(s) Y.

1 Iteration=0.
2 **while** *Iteration<Total Run Time* **do**
3 Update input(s) for current iteration: U(Iteration).
4 Calculate Output(s) for current iteration: Y(Iteration).
5 Update previous input/output values.
6 Iteration++.
7 **end**

The reason for this change is the restriction in the use of floating-point or fixed-point data. As explained in Section 3.3 an ad hoc fixed-point data representation is proposed. The

proposed representation uses integer data only. It is the considerations when implementing the arithmetic (Section 3.3.3) and the scaling effect added when processing multiplication operations (explained in Section 3.4.2) that generate a change in the data processing cycle. The updated data processing cycle is described by Algorithm 3.

Algorithm 3: Modified Discrete SISO LTI Data Processing Cycle.

Input : Discrete SISO LTI model(s): a and b coefficients, Total run time (iterations), Input signal(s) U, Coefficients Scaling Gain K_{ab}.
Output : Output signal(s) Y.

```
1 Iteration=0.
2 while Iteration<Total Run Time do
3 |   Update input(s) for current iteration: U(Iteration).
4 |   Calculate Output(s) for current iteration: Y(Iteration).
5 |   Remove K_ab effect from weighted sum: Y(Iteration)=Y(Iteration)/K_ab.
6 |   Update previous input/output values.
7 |   Iteration++.
8 end
```

After step 4 (Algorithm 3) an extra operation is added to the processing cycle: the removal of the K_{ab} gain effect. The reason for doing so is the need to remove the scaling effect added when representing fixed-point data with integers. By using Algorithm 3 the simulation of the system can be computed. The following section explains with a practical example how the abstraction is generated (Algorithm 1) and the simulation of the process is achieved using Algorithm 3.

Example

To show the applicability of Algorithm 1 with the proposed data-type, a practical example is presented. From a continuous SISO LTI system a discrete SISO LTI abstraction consisting of integer data only will be generated. A simulation of the system response will be generated using the abstraction and will be compared to the original discrete-time system. Taking into account the restrictions in data range and type that Algorithm 1 presents, it guarantees that a system abstraction can be generated from any dynamic system described by a continuous SISO LTI model.

Consider a generic second order system described by:

$$G(s) = \frac{Y(s)}{U(s)} = \frac{K\omega_n{}^2 e^{-\theta s}}{s^2 + 2\zeta\omega_n s + \omega_n{}^2} \tag{3.21}$$

where, the following arbitrary values to demonstrate the abstraction methodology are selected:

- System gain $K = 1$.
- System transport delay $\theta = 1$ sec.
- System natural frequency $\omega_n = 0.45$ rad/sec.
- System damping ratio $\zeta = 0.5$.

The system transfer function becomes:

$$G(s) = \frac{Y(s)}{U(s)} = \frac{0.45^2 e^{-s}}{s^2 + 0.45s + 0.45^2} \tag{3.22}$$

A sampling period of T = 0.5 seconds is selected, a parametric compensating gain K_U = 0.001 (0.1%) is selected in order to apply Algorithm 1 to the system described by Equation 3.22.

Step 1 - Digitize the SISO LTI model using sampling period T: After digitizing Equation 3.22 using the selected sampling period T the following discrete SISO LTI system is generated:

$$G(z) = \frac{Y(z)}{U(z)} = \frac{0.0234186z + 0.021724}{z^2 - 1.753373z + 0.798516} z^{-2} \tag{3.23}$$

Step 2 - Define operating space of the model: Input-Output range - The operating ranges for input U and output Y are:

$$[0,1] = \{U \in \mathbb{R} : 0 \leq U \leq 1\}$$
$$[0,2] = \{Y \in \mathbb{R} : 0 \leq Y \leq 2\}$$

The system gain is 1 and is described by a second order system such as Equation (3.21). Because the system has overshoot, the range for the system output Y is selected equal to the system's input range.

Step 3 - Select fixed-point representation (I, F): Considering the selected data range (step 2) and the ad hoc data type (Section 3.3.4) a value of I=1 and a value of F=4 are selected. Table 3.9 shows an example of the minimum and maximum values from the original fixed-point representation to the ad hoc integer-only fixed-point representation.

Step 4 - Select Input-Output Scaling Gain K_S: Based on the information from Table 3.9 a value of K_S=10,000 is selected. This value allows to map a value of 1 into the integer representation equivalent of 10,000.

Table 3.9 Mapping between original floating-point values and fixed-point integer representation.

Values Mapping			
U		**Y**	
Original	**Abstraction**	**Original**	**Abstraction**
0.0000	0	0.0000	0
0.0001	1	0.0001	1
...	...	...	...
0.9999	9,999	1.9999	19,999
1.0000	10,000	2.0000	20,000

Step 5 - Convert the discrete SISO LTI model to fixed-point: The selected fixed point representation allows for 1 integer digit and 4 fractional digits. Taking Equation 3.23 into this format:

$$G(z) = \frac{Y(z)}{U(z)} = \frac{0.0234z + 0.0217}{z^2 - 1.7534z + 0.7985} z^{-2} \tag{3.24}$$

Step 6 - Verify non-zero fractional digits in coefficients a and b can be recovered: As can be seen in Equation 3.24 all the coefficients when put into the selected fixed-point representation have non-zero values. The reason for this step is to avoid eliminating coefficients altogether because they do not fit into the selected data type.

Step 7 - Calculate the value of the Coefficients Scaling Gain K_{ab}: To recover 4 fractional digits from the coefficients a value of 10,000 is required. The first value which is a power of 2 to allow to recover 4 fractional digits (hence it has to be in the order of the tens of thousands) is $2^{14} = 16,384$. This is the selected value for K_{ab}.

Step 8 - Recalculate coefficients using Equation 3.17, K_U, and Table 3.8.

Step 9 - Apply final gain value theorem (Equation 2.2).

Step 10 - Select the transfer functions with the highest and lowest gains from Step 9.

The previous 3 steps are clustered together. After applying the error compensations to Equation 3.24 and selecting the highest and lowest final gain values the following 2 transfer functions are obtained:

$$G_{Over}(z) = \frac{Y(z)}{U(z)} = \frac{0.0236z + 0.0219}{z^2 - 1.7553z + 0.7993} z^{-2} \tag{3.25}$$

$$G_{Under}(z) = \frac{Y(z)}{U(z)} = \frac{0.0234z + 0.0217}{z^2 - 1.7516z + 0.7979} z^{-2} \tag{3.26}$$

Equation 3.25 corresponds to the over approximation transfer function and Equation 3.26 corresponds to the under approximation transfer function.

Step 11 - Scale coefficients using K_{ab} gain and round off to integer values: The selected value of K_{ab} is 16,384 (Step 7). After scaling Equations 3.25 and 3.26 the following Equations are obtained:

$$G_{Over}(z) = \frac{Y(z)}{U(z)} = \frac{387z + 359}{16384z^2 - 28759z + 13096} z^{-2} \tag{3.27}$$

$$G_{Under}(z) = \frac{Y(z)}{U(z)} = \frac{383z + 356}{16384z^2 - 28698z + 13073} z^{-2} \tag{3.28}$$

Equations 3.27 (over approximation) and 3.28 (under approximation) consist of integer only data. These equations are the ones to be used to calculate the system's dynamics using Algorithm 3, the ad hoc data type, and integer data only. Figure 3.4 shows a 60 seconds simulation of the system and the abstraction.

Both the original discrete and the original discrete with fixed-point representation outputs (signals 2 and 3) are scaled up for comparison reasons with the abstraction. Scaling gain K_S is used to perform the scaling. Figure 3.4 shows the behaviour of the system abstraction consisting of the over and under approximations along with the original discrete-time system and the original discrete-time system with a fixed-point representation. The original discrete plant is encapsulated by the abstraction. As expected, the over approximation settles at a higher value than the original model and the under approximation settles at a lower value. The distance between the original system and the over and under approximations is determined by the amount of parametric compensation (K_U) mainly. Also, the sampling time along with the amount of fractional digits available play a role in this too: if the coefficients are small because of a small sampling period the fixed-point and scaling compensations can become a major contribution towards a final value of the abstraction coefficient.

Using the same dynamic system but with a parametric compensation K_U = 0.005 (0.5%), the system abstraction is described by Equations 3.29 and 3.30. Figure 3.5 shows the result when K_U is modified to a 0.5% value.

$$G_{Over}(z) = \frac{Y(z)}{U(z)} = \frac{388z + 360}{16384z^2 - 28874z + 13148} z^{-2} \tag{3.29}$$

$$G_{Under}(z) = \frac{Y(z)}{U(z)} = \frac{382z + 354}{16384z^2 - 28580z + 13020} z^{-2} \tag{3.30}$$

Equation 3.29 is the new *over approximation* and 3.30 is the new *under approximation*.

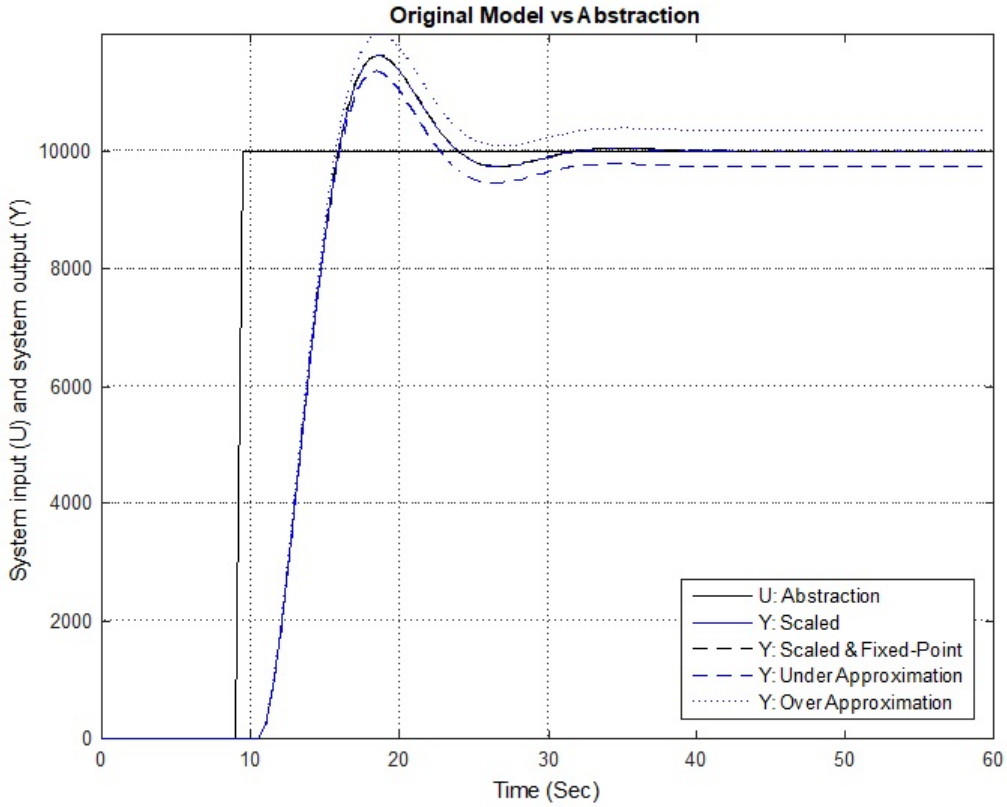

Fig. 3.4 Discrete-time system and discrete-time system abstraction response to a step input comparison. $K_U = 0.001$. K_S=10,000. K_S is used to scale the original system and original with fixed-point system responses (this is done for comparison purposes). The original input is a unit step, it is omitted in the comparison for scaling reasons.

The over approximation settles to a higher value and the under approximation to a lower value than the original system. This shows the effect the parametric compensation considered in this problem formulation has on the abstraction: the higher the parametric compensation the bigger the distance between the original system and the abstraction.

3.6 Final Remarks and Discussion

The system abstraction is suitable for a model checking environment implementation because it does not require the use of floating point data and the system dynamics can be calculated using recurrence equations. Dynamic control systems can be simulated using the proposed abstraction methodology. The example in Section 3.5.2 demonstrated the applicability of

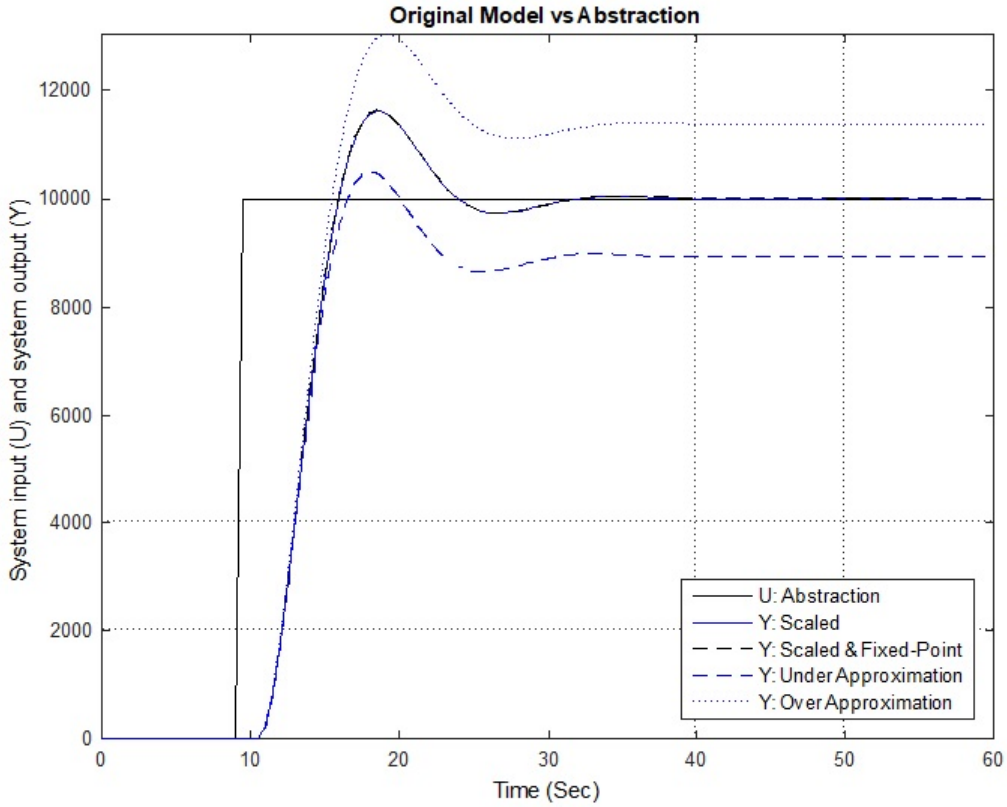

Fig. 3.5 Discrete-time system and discrete-time system abstraction response to a step input comparison. $K_U = 0.005$.

the methodology using one dynamic system, however the methodology is scalable and can be applied to a system described by more than one dynamic element. When using the methodology to generate a dynamic control system abstraction for verification and design purposes all the elements in the control system have to be considered. Algorithm 1 produces a system abstraction consisting of 2 discrete SISO LTI models per dynamic element. The system abstraction bounds the behaviour of the original system and does not require the use of floating point data. To compute the system response using the system abstraction Algorithm 3 has to be used.

The amount of parametric compensation is an important design decision when generating the abstraction because it could lead to an overcompensation in the modelling process. If the abstraction compensation is too conservative it may account for many modelling parametric errors but at the expense of hiding the true dynamics of the system in the process. From a control design point of view this could lead to an unstable closed-loop system when in reality the true system is stable. The purpose of generating the system abstraction is to implement a

dynamic control system in a model checking environment so a formal design and verification approach can be performed. For model checking memory related reasons it is important to consider the following factors which can affect the abstraction in its final fixed-point integer data representation before implementing it in a model checking environment:

1. Order of the system: Number of coefficients a and b in the recurrence equation.

2. System delay: Term n in Equation 3.1.

3. Sampling time of the system: Period T used to convert the system into a discrete form.

These factors become important considerations when using the abstraction in a model checking environment where memory becomes a limitation. The higher the number of coefficients (item 1) the more memory will be needed to recover dynamics. The higher the system delay (item 2) the bigger the registries to store past values of the systems inputs. The smaller the sampling period (item 3) the smaller the b coefficients magnitudes. A small coefficient may not fit the fixed-point representation or the rounding effect when limiting to the selected representation could add too much uncertainty thus changing the dynamics. Also, a smaller sampling period will require more iterations for a given running time scenario, hence more computational memory. It is important to consider these factors in order to obtain a good abstraction of the dynamic system.

The proposed abstraction methodology enables the use of discrete SISO LTI systems to calculate a dynamic system response without the use of floating-point or fixed-point to perform arithmetic operations. A fixed-point data representation which uses integer data is proposed for this purpose. This representation uses integer data arithmetic as well. By these means, the implementation of the abstraction becomes possible in a model checking environment where there are data type restrictions and floating-point data is not available or has limited use. The abstraction accounts for modelling errors and data-type rounding errors, generating bounds for the original system so that safety guarantees can be provided when using the abstraction for reasoning about the original system. Using the abstraction methodology, the next step is to implement a dynamic control system in a model checking environment and use the model checker to verify properties of the system such as performance requirements. The results to be obtained in model checking for the over approximation and the under approximations, thanks to the proposed abstraction methodology and the safety guarantees it accounts for, can be used to infer properties of the original system

Chapter 4

Control Performance Requirements Formal Verification

4.1 Overview

Modelling a dynamic control system is challenging from a model checking point of view due to data type restrictions. For the correct representation of control systems the use of floating-point data-type is desirable and no model checker fully supports it [108]. Most model checkers only support basic operations and integer values which presents a challenge when modelling systems such as PID controllers [153]. To overcome this data-type limitation different approaches are used:

- Generate the system response *a priori* and import it in the model checker [83]. Test scenarios have to be precomputed and the interaction among system components becomes limited restricting the search capabilities of model checking.
- Dynamics over-simplification to use integer data only with limited data range [81, 157]. Accurate transient behaviour is lost, traditional feedback loops become hard to implement.
- Disregard system dynamics [106, 108, 163, 165]. This limits the problem to a steady state behaviour. Traditional feedback loops cannot be modelled.
- Abstract dynamics as timing invariants [40]. The problem is solved as time-invariant restrictions instead of the original domain producing a change in coordinates of the system variables. Transitory behaviours are hard to interpret when projecting into the original domain coordinates.

The full adoption of model checking into standard software development practices, especially for safety-critical applications such as airborne systems, is an ongoing effort [54, 64, 112]. Standards and easy-to-follow practices are required to incorporate model checking into the development process for safety-critical control systems. Current efforts to develop standards still require expertise in model checking from the control engineer [16, 47, 53, 63, 86, 106, 107, 134, 141, 143].

Regarding the plant modelling aspect, discrete Linear-Time-Invariant (LTI) models have been proposed for modelling hybrid control systems using a model checking approach [74, 93, 152]. In this case the synthesized controller is from a symbolic nature - e.g. a state machine [65]. This limits the formal design and verification of current controller structures (e.g. PID). Therefore it is highly desirable to enable the use of common modelling practices for control systems in a model checking environment so that control engineers can exploit the benefits of model checking.

The methodology presented in Chapter 3 allows to model a dynamic system using discrete SISO LTI models without the need for floating-point arithmetic. By this means a dynamic control system can be modelled within a model checking environment that does not allow the use of floating-point numbers. The next step involves how to conduct the modelling process within the model checker. Modelling is driven by the type of properties required for the verification of the control system, e.g. high level control requirements. It is important to design the model checking automata in a way so that the control system requirements can be formulated as a property in the model checker, this is referred to as *design for verifiability*. The key contribution of this chapter is therefore presenting the underlying modelling methodology for a dynamic control system in a model checking environment using the abstraction methodology from Chapter 3. In this manner, control system performance requirements can be formally verified using the model checker. This will enable the design of digital PID controllers using model checking in Chapter 5 and the gain schedule design in Chapter 6.

This chapter is structured as follows: Section 4.2 presents the type of high level requirements to be verified. Section 4.3 presents the automata high and low level designs in order to enable requirements verification via model checking. Section 4.4 presents the process to verify the high level requirements using the model checker in the form of CTL queries formulation. Finally, Section 4.5 presents the full control problem to be addressed, a case study is extracted to show the applicability of the approach to verify high level control requirements.

4.2 High Level Requirements

Current approaches to closed-loop feedback control system are based on stability, performance, or both [44, 94, 121]. Despite the way the controller is designed, the verification procedure for the actual implementation in the form of software relies strongly on a testing phase [7, 86, 113]. The controller is designed, implemented, and then a series of testing stages take place. In the case of a safety-critical system the different levels of testing take place within simulation environments and the real system: embedded computer based controller and the real plant. For verification and validation activities, current software engineering practices include unit testing, integration testing, and acceptance testing [113]. The verification phase for safety-critical systems is still prone to human errors and requirements ambiguities. Model checking is an exhaustive verification technique and by using it during design and verification phases for safety-critical control systems, benefits such as test case generation, higher error-detection rates, and coverage increase, can be obtained thus increasing the safety of the end product.

Due to the complexity of a closed-loop system compliance verification phase, in this work we focus on performance requirements as the main driver for the control system design. This decision is also supported by the fact that usual control system requirements for a gain scheduling control scheme are provided in the form of performance features. Within this formulation and for the first time, high level performance requirements for a PID-type control system will be formally verified with the aid of model checking, thus demonstrating the underlying principles of verifiable design for control.

Figure 4.1 shows the most common performance requirements [121, 125]:

1. **Settling Time t_s**: Time required for the response to settle at a steady value. Time required to reach and stay within a specified range of the steady value. The criteria to determine this range is usually 2% or 5% of the settled value. In this work the selected criteria is that of 2%.

2. **Maximum Overshoot**: The difference between the maximum peak value and the steady state value. Usually expressed as a % of the change in steady state value response.

3. **Rise Time t_r**: Time required by the system to rise from 10% to 90% of its final value.

4. **Steady State Error e_{ss}**: Difference between the actual output and the desired output value (e.g. reference value) when time tends to infinite.

Model checking tools are very good at dealing with logic based requirements, safety requirements, and algorithmic requirements (e.g. verifying actions occur in a particular sequence or

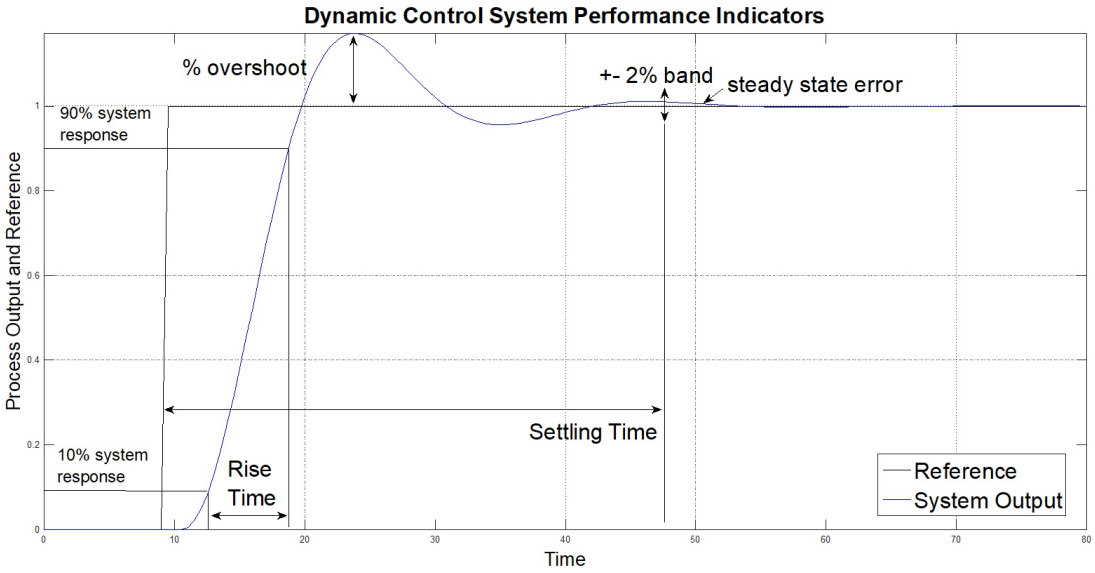

Fig. 4.1 Dynamic control system closed loop performance indicators: settling time, maximum overshoot, rise time, and steady state error.

after certain thresholds have been reached). Nonetheless and despite the fact that these are the most common performance requirements in control design, model checking tools are not aimed to address them [16, 80, 143]. Part of this limitation also resides on the notation of the formal language in the model checker and its ability to express complex requirements like the ones previously stated and the lack of knowledge in how to perform the translation [16, 143]. Therefore, it is desirable to increase the range of requirements that can be formally expressed and verified using model checking, thus improving design and verification stages in the controller development cycle. This thesis proposes a novel methodology to deal with the representation and verification of these requirements from a model checking perspective. The following sections explains how to perform the automata design to tackle this control problem formulation.

4.3 Design for Verifiability

The modelling approach in model checking is strongly driven by the type of properties to be verified about a given system. The problem formulation in this thesis is one of a closed loop control system and the properties under analysis are the high level control system performance requirements from Section 4.2. The design of the model checking automata must therefore enable the capturing of these requirements in the form of properties that the

model checker can understand. In this thesis a novel approach to capture such requirements using model checking by the means of automata is proposed.

The modelling approach is enabled by the ad hoc fixed-point data type presented in Section 3.3.4. Integer values are used to represent both the system input signals and the system output signals. The number of possible values available in the ad hoc data type is limited. For this reason, prior to addressing the reference tracking problem in a formal manner, a suitable operating space for the system's inputs and outputs must be selected. This has to be done in order to avoid truncating values, which could result in the loss of dynamic behaviour.

The operating space of each variable is the range that contains all the possible values that the given variable can take. The operating space of the system is the product of the operating spaces of the system's inputs and outputs.Figure 4.2 shows an example of a feedback control system with its relevant variables and their respective ranges. The ranges of the variables drive the selection of the data representation to be used. The controller reacts to the system

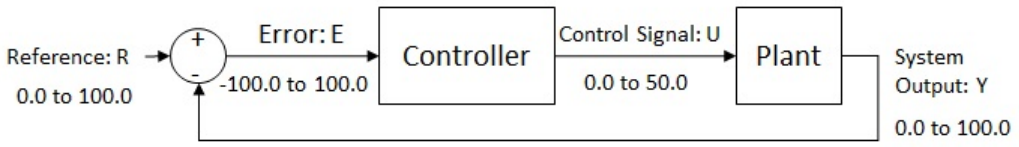

Fig. 4.2 Example of a simplified feedback control system consisting of a plant and a controller. The figure shows the most relevant control loop related variables and their respective operating ranges.

error and generates output U, which in turn is the input to the plant and generates system output Y. The control signal U drives the plant signal Y, which in turn means that there is a correspondence between the operating space of control signal U and the operating space of plant signal Y. In this example, the variables ranges which define the operating space of the control system, are given by:

$$
\begin{aligned}
[0.0, 50.0] &= \{U \in \mathbb{R} : 0.0 \leq U \leq 50.0\} \\
[0.0, 100.0] &= \{Y \in \mathbb{R} : 0.0 \leq Y \leq 100.0\} \\
[-100.0, 100.0] &= \{E \in \mathbb{R} : -100.0 \leq E \leq 100.0\} \\
[0.0, 100.0] &= \{R \in \mathbb{R} : 0.0 \leq R \leq 100.0\}
\end{aligned}
$$

Knowing the range of the variables allows to define how to map the real values into the ad hoc fixed-point representation using integer values only (Section 3.3.4). Disregarding how the mapping from the real values into the fixed-point representation is chosen to be,

the ad hoc data type has a maximum allowed operating space. Considering only the system output Y and the control signal U, using the full data range allowed by the ad hoc data type (16-bit signed integer) for both signals, the system operating space can be plotted in a X-Y coordinate plane. Figure 4.3 shows this operating space with control signal U in the X axis and controlled signal Y in the Y axis where both signals have a range and operating space given by:

$$[-32767, 32767] = \{U \in \mathbb{Z} : -32,767 \leq U \leq 32,767\}$$
$$[-32767, 32767] = \{Y \in \mathbb{Z} : -32,767 \leq Y \leq 32,767\}$$

Figure 4.3 shows the maximum operating space allowed by the ad hoc data type considering

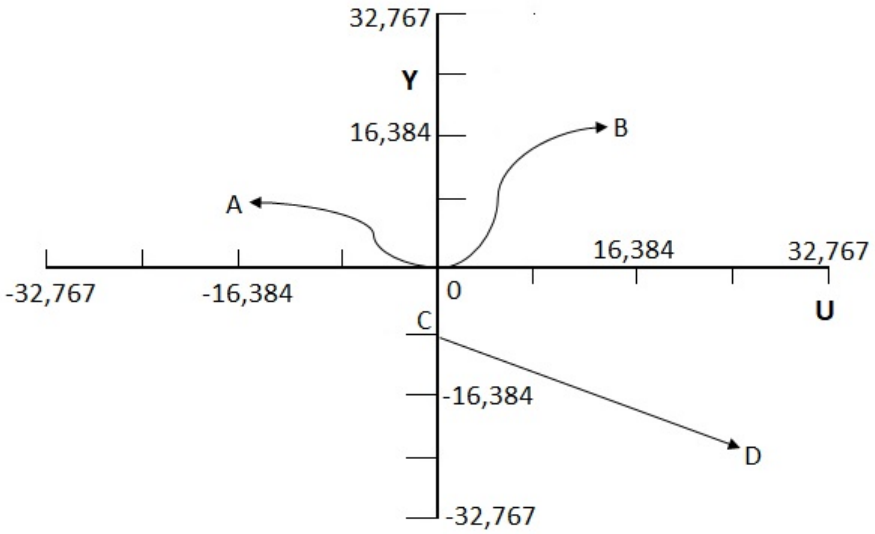

Fig. 4.3 Operating search space for an input (U) - output (Y) relationship using integer data only. Input and output range: -32,767 - 32,767. The control problem becomes a path search in *(U,Y)* coordinates - e.g. going from the origin to point A or point B, moving from point C to point D.

the system output and the control signal. Under this consideration, the control problem can be formulated and solved in the coordinate plane. The reference tracking control problem becomes a trajectory search in the coordinate plane: how to go from one point in the plane to another (e.g. point O to point A) in a certain amount of time and stay there indefinitely. In a similar manner, disturbance rejection can also be portrayed as a trajectory search with both origin and end points being the same point. The trajectory dynamics, which involve both the controller and the plant (how variables U and Y interact) are determined by the SISO LTI models presented in Chapter 3. How the trajectories will behave depends on the dynamics

portrayed by the SISO LTI models. In this thesis only reference tracking control will be addressed.

Depending on the variables ranges and their representation using the ad hoc data type, the operating space for the system can be reduced. Instead of using the full range allowed by the ad hoc data type, which covers the full coordinate plane in Figure 4.3, extracts of the coordinate plane are preferred because this will reduce the system operating space reducing the amount of effort when performing the trajectory search. Also, this limitation of the operating space means saving memory in the model checking implementation. As an example, Figure 4.4 shows an operating space for variables U and Y given by:

$$[0,10000] = \{U \in \mathbb{Z} : 0 \leq U \leq 10,000\}$$
$$[0,20000] = \{Y \in \mathbb{Z} : 0 \leq Y \leq 20,000\}$$

Time evolves in a discrete manner where every step represents a T amount of time in

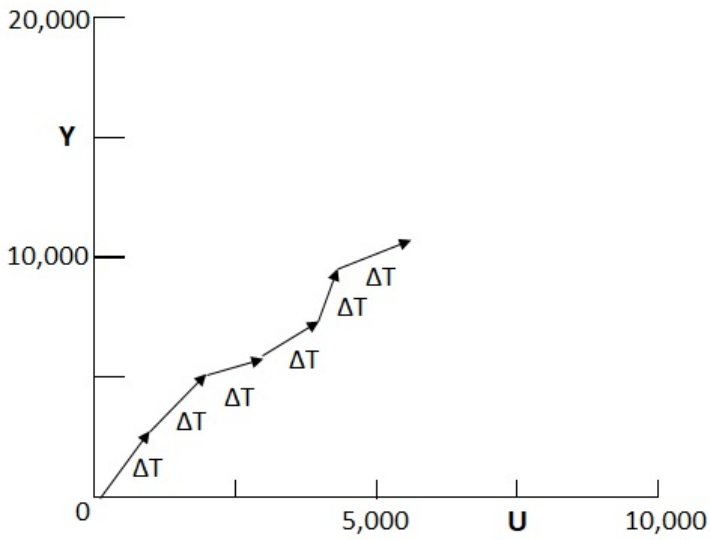

Fig. 4.4 Operating search space for an input (U) - output (Y) relationship using integer data only. Input range: 0-10,000. Output range: 0-20,000. Every arrow represents a ΔT amount of time elapsed. The reference tracking control problem involves moving the output Y from an initial point towards a final region by the means of changing the input U. Every ΔT both U and Y are updated using their respective dynamics.

the arrow direction. The purpose of defining an operating space is related to the type of requirements under analysis. In the reference tracking problem formulation for this thesis, high level control performance requirements need to be mapped into the trajectory search problem in the operating space. By doing so, the verification of high level requirements can

be performed along with the trajectory search. Figure 4.5 portrays the performance high level requirements from Section 4.2 in the example operating space.

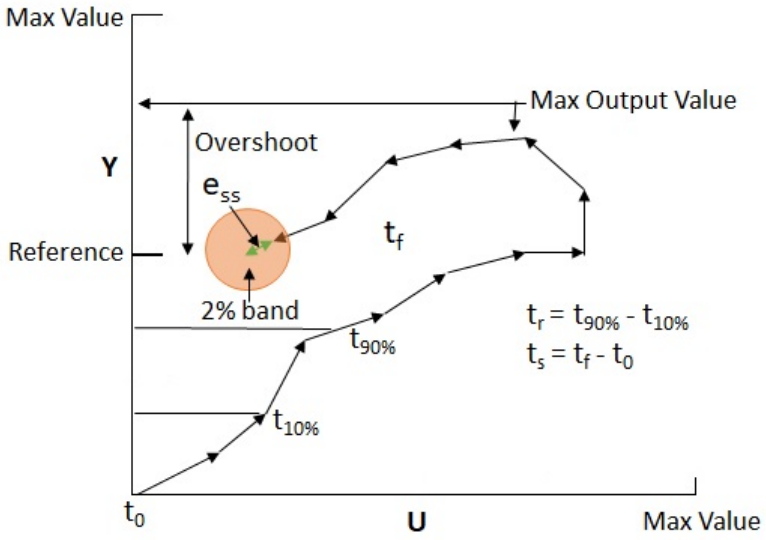

Fig. 4.5 Operating space for input signal U and output signal Y showing the control system performance indicators of interest. The reference tracking control problem becomes one of analysing the trajectory from the origin point to the reference point. The performance indicators become way-points in the trajectory.

The reference tracking control problem concerns the trajectory taken from the origin point to reach the reference point area and stay there indefinitely within the specified timing, steady state error, and overshoot requirements. Requirements become way-points in the trajectory and such trajectory must avoid a particular operating area related to the overshoot requirement.

Figure 4.6 shows a reference tracking problem formulated within a particular operating space. The process output Y is driven by input U from the origin to the reference area within a finite amount of time. Requirements are way-points in the trajectory. In this particular scenario, the maximum overshoot requirement is not met because the area related to output Y being equal to or higher than the maximum allowed value is included in the trajectory.

By formulating the control problem in this way, limiting the operating system space in order to perform a trajectory search within the space, the reference tracking problem can be addressed as a *reachability* problem in model checking. In order to bring this trajectory search into a model checker framework, the problem must be portrayed in a suitable way that the model checker can understand and work the problem around. In this thesis and for this purpose a set of automata are proposed. The automata will work within a defined operating space and process the system dynamics using discrete SISO LTI models as explained in

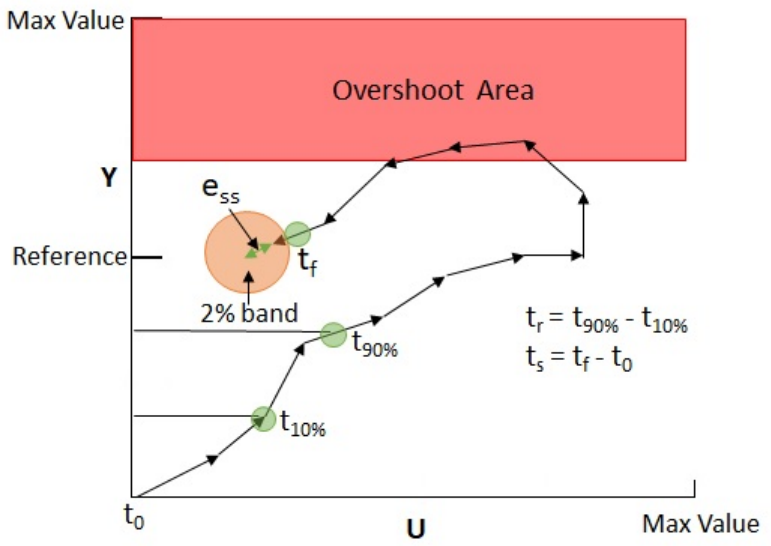

Fig. 4.6 Reference tracking control problem for input signal U and output signal Y with performance requirements as way-points in the trajectory.

Chapter 3, addressing the verification of high level performance requirements for a control system as a *reachability* problem by performing a trajectory search and analysis in the operating space. The following section explains the proposed automata to accomplish these goals.

4.3.1 Automata Design

In order to capture the features of interest for the control problem formulation the automata have to detect the behaviour of the system in a way that ensures the features are expressible in the model checker language. Considering a dynamic control system like the one in Figure 2.1 and high level control requirements 1-4 from Section 4.2, such requirements must be expressible in the form of properties. The labelling function must then tag states with such requirements so they can be detectable and used as expression in the model checker language.

The following automata are proposed to achieve this goal:

- The *Plant* automaton generates the process output and monitors it. Because requirements are related to the process output Y, it is in this automata where requirements are portrayed so they can be verified.
- The *Controller* automaton is in charge of generating the control action U.
- The *Observer* automaton synchronizes the correct execution of events between the *Controller* and *Plant* automata.

More detailed description is given below:

Plant

Figure 4.7 shows the automaton in charge of simulating the process to be controlled. The dynamics of the system are embedded in this automaton. Requirements are strongly correlated to output signal *Y*, which is the reason why particular states to capture requirements are generated. The automaton consists of the following states:

1. *Settled:* Initial state. An external event (change in reference) triggers a transition to the *Transient* state. After an event has been triggered, if the system output settles this state is reached. This state does not process the system dynamics.

2. *Transient:* Transitory state. Once a change in reference is triggered the process dynamics calculations begin here. Whilst the process has not settled and has not reached the 90% response level the process output is calculated here. If the process reaches a settling value a transition to *Settled* state is triggered.

3. *Rise Time:* Rise time state. When 90% of the response is reached the automata transitions here. While the process has not settled and has not reached an overshoot level the process output is calculated here. If the process reaches a settling value a transition to *Settled* state is triggered. If the process reaches an overshoot value a transition to *Overshoot* state is triggered.

4. *Overshoot:* If the process output reaches an overshoot value the automata transitions here. The overshoot threshold value is defined according to requirements. While the process has not settled the process output is calculated here. If the process reaches a settling value a transition to *Settled* state is triggered.

Controller

Figure 4.8 shows the automaton in charge of simulating the process controller. The dynamics of the controller are embedded in this automaton. The automaton consists of the following states:

1. *Settled:* Initial state. This state is similar to *Settled* in the *Plant* automaton. An external event (change in reference) triggers a transition to the *Transient* state. After an event has been triggered, if the controller output settles this state is reached. This state does not process the controller dynamics.

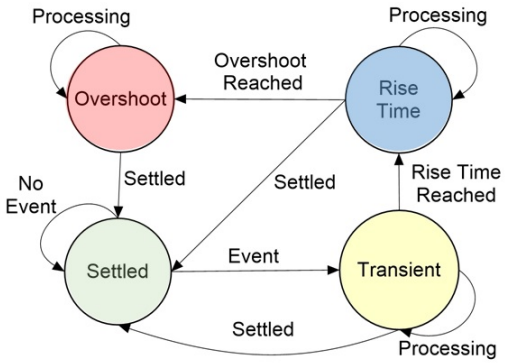

Fig. 4.7 Plant automaton. This automaton generates the output signal Y using a discrete SISO LTI model and monitors its behaviour to determine transitions between states. The high level performance requirements are processed in this automaton under its different states.

2. *Transient:* Transitory state. This state is similar to *Transient* in the *Plant* automaton. Once a change in reference is triggered the controller dynamics calculations begin here. While the controller output has not settled the controller output is calculated here. If the controller output reaches a settling value a transition to *Settled* state is triggered.

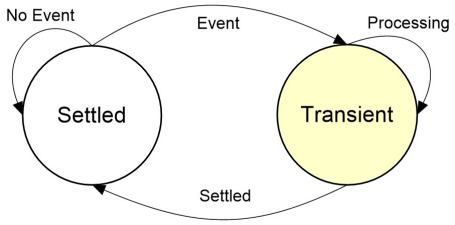

Fig. 4.8 Controller automaton. This automata generates the control signal U using a discrete SISO LTI model and monitors its behaviour to determine transitions between states. The controller automaton has less states than the *Plant* automaton because in this problem formulation the control signal is not under analysis, it is simple required for it to be generated.

Observer

Figure 4.9 shows the automaton in charge of coordinating the execution of the controller and the plant. This automaton monitors both the controller output U and the process output Y to determine when to finalize the simulation.

The automaton consists of the following states:

1. *Init:* Initial state. Initialize the models for simulation. Monitors the input reference signal in order to trigger an event. Once a change in reference has been detected a transition to *Transient* state is taken.

2. *Transient:* Transitory state. This is the state in charge of coordinating the execution of the *Plant* and *Controller* automata. It monitors both the process output signal Y and the controller signal U to determine when the process has reached a settling condition. Once both the controller and the plant have settled a transition is triggered to the *End* state.

3. *End:* Final state. After an event (change in reference) if an equilibrium condition is reached the dynamics process comes to a halt and no more transitions are allowed in any automaton.

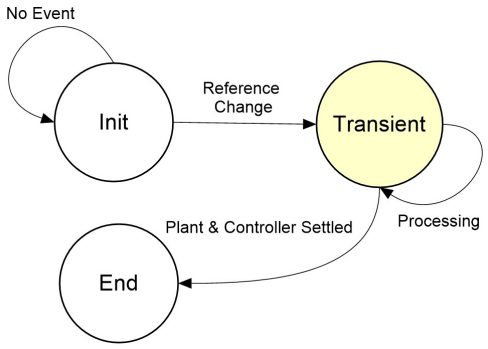

Fig. 4.9 Observer automaton. Automaton in charge of controlling the data flow between the controller and the plant. This automaton monitors the control signal U and output process signal Y to determine transitions between states.

The presented set of automata provide a working framework to address the high level control requirements verification problem in the reference tracking formulation. The actual automata implementation in a model checking environment depends on the model checker. Model checking tools vary depending on the type of problem they are aimed to solve. The tools design is driven by the type of systems to be modelled and the type of properties to be verified and whether these properties are expressible in the formal language used in the tool. Some representative examples are NuSMV [34], SPIN [76], Simulink Design Verifier (SDV) [109], and UPPAAL [13].

Simulink [109] is a popular tool for both modelling and development. Although, not all the Simulink functionalities are available in SDV to perform formal verification and the available formal verification blocks are mainly designed to deal with boolean-type problems and run-time errors (e.g. division by zero, infinite loops). NuSMV [34] is aimed mainly to verify synchronous systems which may represent a limitation when modelling a dynamic system. SPIN [76] is designed to verify models for distributed systems, focusing mainly on distributed software algorithmic behaviour. The model checker UPPAAL [13] is designed

to model systems as networks of timed-automata with integer variables, structured data types, clocks, and channel synchronization. Recent developments in the tool have enabled its use in industry-like case studies, showing that the tool is suitable to address these type of problems. UPPAAL offers a modelling environment which gives the user freedom to program tailor-made functionalities for the timed-automata.

UPPAAL [13] was selected here because of the freedom it provides to implement functionalities using a well-known programming language such as *C*. From a modelling point of view, this freedom makes the abstraction implementation easier. Also, the type of properties available in its formal language (details in Section 4.4) are a good fit to address the gain scheduling design and verification problem. The following section explains the UPPAAL implementation of the automata.

4.3.2 UPPAAL Automata

The particular implementation of the automata presented in Section 4.3.1 was carried out in the model checker UPPAAL. The automata presented in this section captures the required behaviour from those in Section 4.3.1.

Plant: UPPAAL

Figure 4.10 shows the automaton in charge of simulating the process under control. The dynamics of the system are embedded in this automaton. The *Plant_Calc* function is in charge of calculating the system dynamics given by a SISO LTI equation. This automata receives a message via channel *C* to enable the calculation of process output *Y*.

The automata consists of the following states:

1. *SS:* Initial state. Similar to *Settled* state depicted in Figure 4.7. An external event (change in reference) triggers a transition to the *TS* state. Once the system output settles within the defined % band this state is reached. This state does not process the system dynamics.

2. *TS:* Transitory State. Similar to *Transitory* state depicted in Figure 4.7. Once a change in reference is triggered the process dynamics calculations begin here. Whilst the process has not settled and has not reached the 10% response level the process output is calculated here. If the process reaches a settling value a transition to *STB* state is triggered.

3. *TP:* Rise time related state. When 10% of the response is reached the automata transitions here. The detection of the 10% of the response is required to calculate rise time. A time stamp is taken for the posterior calculation of rise time.

4. *RT:* Rise time state. Similar to *Rise Time* state depicted in Figure 4.7. When 90% of the response is reached the automata transitions here and it means that the response has reached the level required to calculate rise time. While the process has not reached the percentage band settling value and has not reached an overshoot level the process output is calculated here. If the process enters the percentage band settling value a transition to *STB* state is triggered. During the transition to the *STB* state the *settling time* is calculated. If the process reaches an overshoot value a transition to *OS* state is triggered.

5. *OS:* Overshoot state. If the process output reaches an overshoot value the automata transitions here. The overshoot threshold value is defined by the user according to the high level requirements of the particular problem to solve, it is a variable to be defined by the user. While the process output value remains equal to or above the overshoot threshold the process output is calculated here. If the process output drops below the overshoot threshold a transition to *TS* state is triggered.

6. *STB:* Settling state. Both *SS* and *STB* states comprise the behaviour of the *Settled* state depicted in Figure 4.7. If the process output reaches a settling value the automata transitions here. The process output is verified to stay within the settling percentage band for a pre-defined amount of time before deciding the process has indeed settled. If the process stays within the % band the pre-defined amount of time a transition to *SS* state is triggered. During the transition to the *SS* state the *steady state error* is calculated. If the process output goes outside the percentage band a transition to *RT* state is triggered.

Controller: UPPAAL

Figure 4.11 shows the automaton in charge of simulating the controller. The dynamics of the system controller are embedded in this automata. The *PI_Calc* function is in charge of calculating the controller output dynamic given by a SISO LTI equation. This automaton receives a message via channel *C* to enable the calculation of controller output *U*.

The automata consists of the following states:

1. *Settled:* Initial state. Similar to *Settled* state depicted in Figure 4.8. An external event (change in reference) triggers a transition to the *Transient* state. After an event has

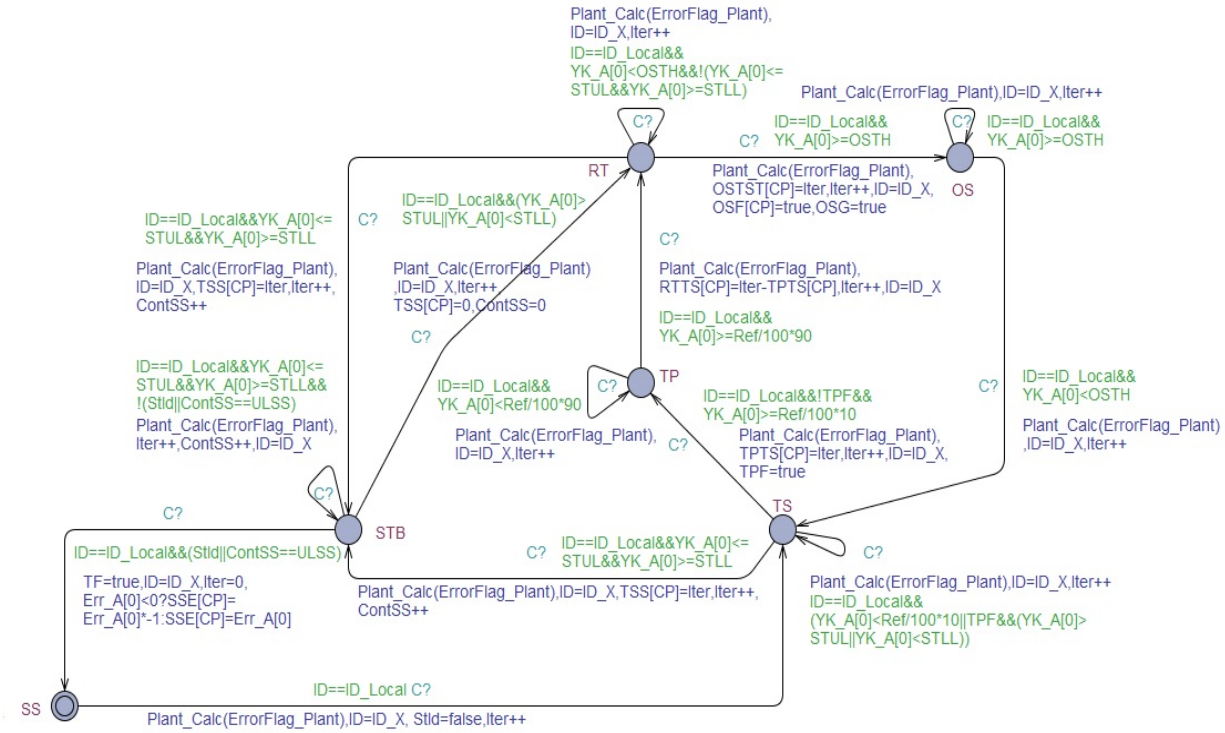

Fig. 4.10 UPPAAL implementation of the plant automaton. This automaton generates the output signal Y and monitors its behaviour to determine transitions between states.

been triggered, if the controller output settles this state is reached. This state does not process the controller dynamics.

2. *Transient:* Transitory state. Similar to *Transient* state depicted in Figure 4.8. Once a change in reference is triggered the controller dynamics calculations begin here. While the controller output has not settled the controller output is calculated here. If the controller output reaches a settling value a transition to *Settled* state is triggered.

Observer: UPPAAL

Figure 4.12 shows the automaton in charge of coordinating the correct execution of the controller and the plant. This automaton monitors both the controller output U and the process output Y to determine when to finalize the simulation. The coordination of the execution of the controllers and the plants is achieved by using synchronization channels C and D (one per system approximation). This automaton sends a message via channels C and D to enable the execution of either the *Plant* or the *Controller* automata for both the over and under approximations.

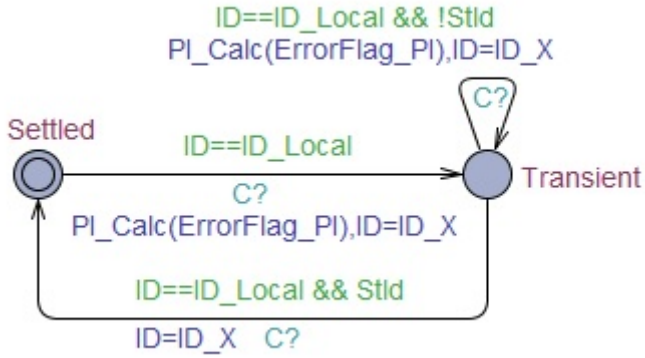

Fig. 4.11 UPPAAL implementation of the controller automaton. This automata generates the control signal U and monitors its behaviour to determine transitions between states.

The automaton consists of the following states:

1. *Init:* Initial state. Similar to *Init* state depicted in Figure 4.9. Initialize the models for simulation. Monitors the input reference signal in order to trigger an event. Once a change in reference has been detected a transition to *Sync* state is taken.

2. *Sync:* Transitory state. Similar to *Transient* state depicted in Figure 4.9. State in charge of coordinating the execution of the *Plant* and *Controller* automata. It monitors the process output signal Y, the controller signal U, and the elapsed time to determine when to finish the automata execution. The observer coordinates both the over and under approximations via communication channels. Once both the controller and the plant in both over and under approximations have settled and no time-out condition is detected a transition is triggered to the *End* state.

3. *End:* Final state. Similar to *End* state depicted in Figure 4.9. After an event (change in reference) if an equilibrium condition is reached the dynamics process comes to a halt and no more transitions are allowed in any automata.

4. *Timeout:* Fail-safe state. A pre-defined amount of time is given to the process to reach an equilibrium. In order to save memory and computations this condition is added. If the process does not reach an equilibrium within the allowed amount of time the automata transitions to this state.

This set of automata enables the implementation of a discrete control system using discrete SISO LTI models to portray the dynamics. By using the abstraction methodology presented in Chapter 3 and the automata in Section 4.3.2 the system can be implemented in

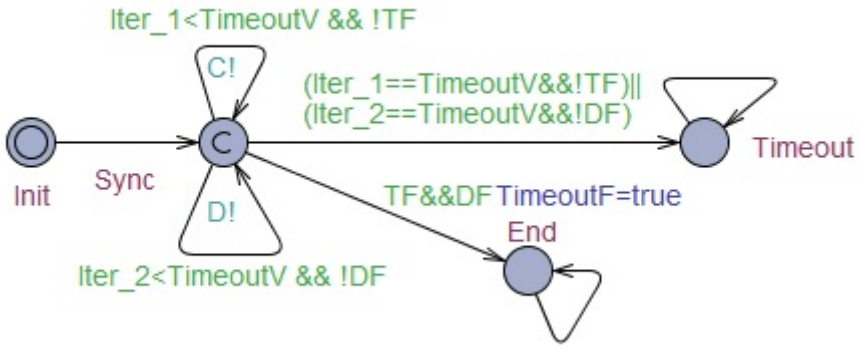

Fig. 4.12 UPPAAL implementation of the observer automaton. Automaton in charge of controlling the data flow between the controller and the plant. This automata monitors the control signal U, output process signal Y, and the elapsed time to determine transitions between states. It coordinates the execution between plants and controllers using communication channels C and D. One channel is used for the over approximation and the other one for the under approximation.

the model checker UPPAAL so that high level requirements can be formally verified. The following section explains how to perform such verification in the model checker.

4.4 Requirements Formulation for Verification

Modelling the control system is strongly driven by the type of features to be verified. In this problem formulation the features of interest are high level control system performance requirements. The implementation of the control system in the UPPAAL model checker allows to reason about the system using Computational Tree Logic (CTL). There are 3 types of properties available in UPPAAL:

1. *Reachability:* It is possible to reach a system state.
2. *Safety:* Something can never happen.
3. *Liveness:* Something will eventually happen.

The *Plant* automaton contains the necessary elements to verify the high level control performance requirements from Section 4.2. The states in the automaton are labelled with the variables related to performance requirements (e.g. settling time, rise time, overshoot, and steady state error) so they can be checked during the verification process. The verification of

high level performance requirements can be portrayed as a reachability problem - e.g. is it possible to reach a system state where all control system requirements are met?

When performing the verification of properties the UPPAAL model checker can return a *witness* or a *counter example* trace. A *witness* trace contains the actions that lead to a property being fulfilled and a *counter example* trace contains the actions that lead to a property not being fulfilled. Depending on how the verification is formulated is which type of trace can be generated.

Table 4.1 shows three ways to generate either a *witness* or *counter example* trace in UPPAAL, where:

1. *A:* Path quantifier - *For all computation paths*.
2. *E:* Path quantifier - *For some computation path*.
3. $<>$*: Eventually* temporal operator.
4. $[]$*: Globally* temporal operator.
5. φ*:* State formula, a logical expression that can be evaluated in the state.

Table 4.1 UPPAAL queries which can generate either a *witness* or a *counter example trace*.

Query Type	**Formula**	**Output**
Reachability	$E <> \varphi$	Witness Trace
Safety	$A[]\varphi$	Counter-example Trace
Liveness	$A <> \varphi$	Counter-example Trace

The control problem under verification as portrayed in Figure 4.6 can be formulated in the following way: *can the reference point be reached from the origin in a finite amount of steps whilst meeting all 4 performance requirements?*

Figure 4.13 shows a graphic description of the reachability problem. From an initial system configuration containing states *(Settled, Init, Settled)* can a final system configuration containing states *(Settled, End, Settled)* be reached visiting the system configurations *(Transient, Transient, Transient)*, and *(Transient, Transient, Rise Time)* in the process? This also means that any system configuration containing the *Overshoot* state must be avoided. The final system configuration containing states *(Settled, Init, Settled)* must be labelled with values which meet requirements for the verification to be successful. The desired trajectory in every automaton must then follow the blue arrows.

The requirements verification is thus performed using a push-button approach by querying the model checker using a *reachability* property such as:

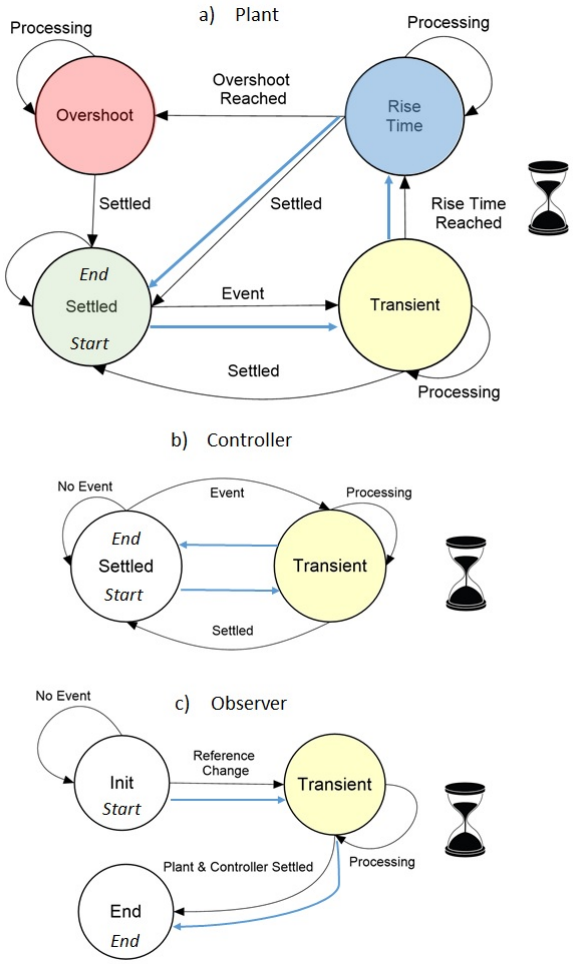

Fig. 4.13 Proposed automata: a) Plant automaton. b) Controller automaton. c) Observer automaton. Performance requirements verification portrayed as a reachability problem: can the states labelled as *End* be reached starting at the states labelled as *Start* visiting the states which follow the blue arrows trajectories in a finite amount of time?

$$E <> Observer.End \quad \text{and} \quad Plant.Settled \quad \text{and} \quad Controller.Settled \quad \text{and}$$
$$Plant.Overshoot \leq Requirement \quad \text{and} \quad Plant.RiseTime \leq Requirement \quad \text{and}$$
$$Plant.SettlingTime \leq Requirement \quad \text{and} \quad Plant.SSError \leq Requirement \quad (4.1)$$

If the verification process is successful the model checker will return *Pass* as a result and the *witness* trace to show this. If the verification is not successful the model checker will simply return *Fail* without a trace. In the *Fail* scenario in order to obtain information from

the model checker the path qualifier must be changed to *A* instead of *E* to use a *Liveness* query so that a *counter-example* trace is generated.

With the current framework, the model checker can only be used to verify that the trajectory ends in the desired region whilst meeting the performance requirements without performing any design or controller tuning. Dynamics are entirely dictated by the discrete SISO LTI models embedded in the automata. Nonetheless, this framework is necessary to address the controller design problem formulation within a model checker environment. The following section explains how to use the proposed automata along with the abstraction methodology from Chapter 3 to address the performance verification problem.

4.5 Case Study: Thrust Control Verification

Consider a commercial jet-engine where generated thrust is regulated using a PID-type controller with a gain scheduling scheme such as the one in Figure 2.2. The process dynamics will vary depending on the operating point: factors such as altitude and temperature generate a non-linear behaviour [149].

The control problem to solve is to find a control schedule that drives the system to meet high level requirements in every operating point. The non-linear nature of the process generates different dynamic behaviours throughout the operating points. Proving stability and conformity with requirements for a gain schedule scheme is hard to do analytically [127]. Also, and for safety reasons, the control scheme final software form undergoes an extensive verification and validation phase to make sure performance is met and to demonstrate and justify that the software is safe, certifiable, and to be trusted [51, 70]. A model checking approach can aid in this respect because the controller design can be done directly in the software domain, running simulations to find a controller which meets requirements while performing extensive testing. This problem formulation provides a good framework to show the proposed methodology to design and verify PID-type gain schedule controllers. The gain schedule design problem and the formal approach to address it will be presented in more detail in Chapter 6.

In order to fully address the gain schedule design and verification problem, the following tasks are required:

1. Abstract the control system dynamics.
2. Implement the control system abstraction in a model checking environment.
3. Verify high level performance requirements for the control system using model checking.

4. Perform the controller tuning process using model checking.

5. Generate and verify the control schedule using model checking.

Chapter 3 presented the dynamic system abstraction methodology to address item 1. The proposed automata from Section 4.3 address item 2 and Section 4.5.1 will present how to address item 3. Regarding items 4 and 5, Chapter 5 presents the methodology to tune a controller using model checking. Finally, 6 addresses item 5, the full design and verification methodology for the gain scheduling problem is presented.

4.5.1 Verification Problem Formulation

Before designing and verifying the schedule it is necessary to be able to verify high level performance requirements using model checking. In order to do so a feedback control loop must be able to be analysed by the model checker. The gain schedule problem will be addressed using various linear models to portray the non-linear nature of the system. From this formulation a linear model is extracted and selected as a starting point to apply the methodology for the formal verification procedure. Figure 4.14 shows the open loop dynamic response of the selected linear model which is part of the thrust control problem. The details about the full system dynamics will be provided in Chapter 6

The continuous linear model for the system in Figure 4.14 is given by a second order system (Equation 3.21). The parameters of the system equation are:

- System gain $K = 1$.
- System transport delay $\theta = 1$ sec.
- System natural frequency $\omega_n = 0.45$ rad/sec.
- System damping ratio $\zeta = 0.5$.

The continuous time system transfer function is given by:

$$G(s) = \frac{Y(s)}{U(s)} = \frac{0.45^2 e^{-s}}{s^2 + 0.45s + 0.45^2} \tag{4.2}$$

The selected controller is a parallel Proportional + Integral (PI) controller. In the continuous domain the controller equation is given by:

$$C(s) = \frac{U(s)}{E(s)} = K_P + \frac{K_I}{s} \tag{4.3}$$

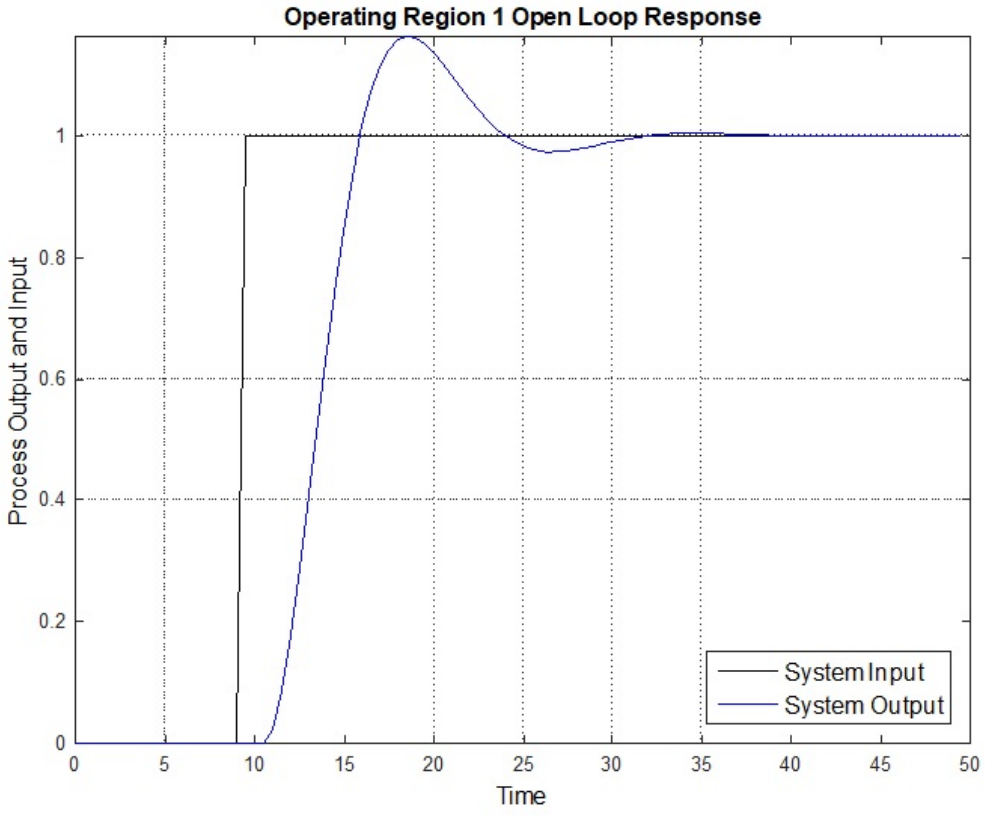

Fig. 4.14 Open loop response for the linear system part of the thrust control problem. The model corresponds to one of the operating regions in the thrust control problem.

where U(s) is the controller output, E(s) is the system error,K_P is the controller proportional gain, and K_I in the integral gain. The controller is discrete in nature because it is implemented in a computer-based system. It is common practice to perform the design in the discrete domain and do the conversion to the discrete domain using discrete equivalences (e.g. backward, Tustin) [1, 122]. Using the Tustin discrete equivalence with Equation 4.3 the discrete transfer function for the PI controller using a sampling period T is given by:

$$C(z^{-1}) = \frac{U(z^{-1})}{E(z^{-1})} = \frac{(K_P + K_I)T - K_I z^{-1}}{1 - z^{-1}} \tag{4.4}$$

In this case a PI structure has been chosen as the preferred controller along with the tustin continuous to discrete equivalence to digitize it. However, the controller structure and the discrete equivalence to address this problem is not restricted to this particular choice, the

methodology can be applied regardless of this choice. The next step is to generate the system abstraction and implement it in the model checker using the proposed automata.

4.5.2 System Abstraction

The abstraction is required for the implementation of the control system in the model checker UPPAAL. By applying Algorithm 1 to the system described by Equation 4.2 the system abstraction is generated, both the over and under approximation will be generated as a result of this process. The data type to be used in the model checker is as described in Section 3.3.4. This data type allows to use 5 digits for data representation and perform the necessary arithmetic operations to calculate the system dynamics using the recurrence equations. Considering the following configuration parameters for the abstraction:

- Sampling time T = 0.5 seconds.
- Parametric compensation gain $K_U = 0.0005$.

As a result of the application of Algorithm 1:

- Data-type representation format: 1 integer digit (I = 1) and 4 fractional digits (F = 4).
- K_S = 10,000.
- K_{ab} = 16,384.

The discrete integer-only SISO LTI models for the abstraction (over and under approximations) as described in Section 3.5.1 are given by:

$$G_{Over}(z) = \frac{Y(z)}{U(z)} = \frac{387z + 359}{16384z^2 - 28744z + 13089} z^{-2} \tag{4.5}$$

$$G_{Under}(z) = \frac{Y(z)}{U(z)} = \frac{383z + 356}{16384z^2 - 28710z + 13079} z^{-2} \tag{4.6}$$

Equation 4.5 is the abstraction *over approximation* and Equation 4.6 is the abstraction *under approximation*. The controller is discrete in nature, it is assumed that the controller implementation will be restricted to a fixed-point data type. For this reason the controller discrete model does not include error compensations. Considering this assumption, the controller transfer function only requires to be converted into the selected data format. To do so the controller coefficients from the discrete transfer function are scaled using gain K_{ab}. Using conventional tuning methods an initial configuration for the controller gains was generated. The initial controller gains are:

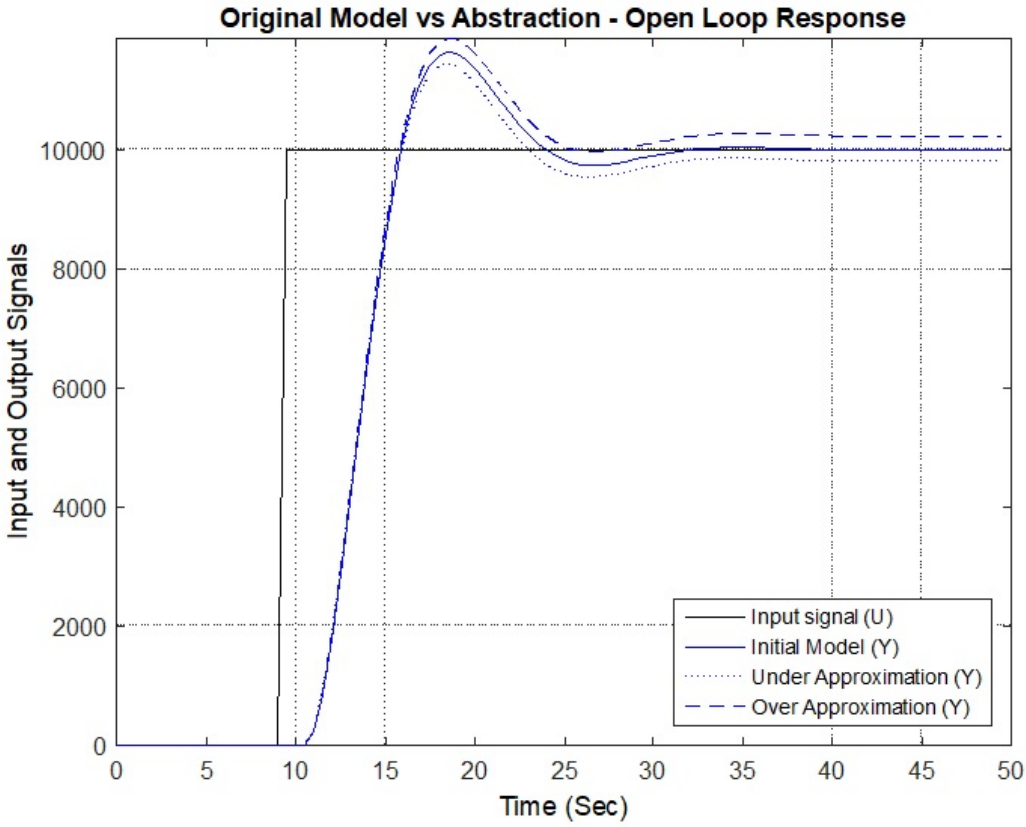

Fig. 4.15 Open loop response comparison for the system abstraction and the initial model for operating region 1 in the thrust control problem from Figure 4.14. The initial response is scaled up for comparison purposes (using $K_S = 10,000$).

- $K_P = 0.1392$.
- $K_I = 0.1496$.

Using Equation 4.4 with the controller gain values K_P and K_I, scaling gain K_{ab}, and sampling period T, the controller transfer function using the ad hoc data type is given by:

$$C(z^{-1}) = \frac{U(z^{-1})}{E(z^{-1})} = \frac{2,894 - 1,668z^{-1}}{16,384 - 16,384z^{-1}} \tag{4.7}$$

The abstraction implementation in the model checker UPPAAL consists of 5 automata:

- Two controllers: *Controller_OA* - over approximation controller. *Controller_UA* - under approximation controller. Both controllers have the same transfer function.

- Two plants: *Plant_OA* - over approximation plant. *Plant_UA* - under approximation plant. Each plant has its own transfer function corresponding to their respective dynamics.
- One observer: *Observer* - automata in charge of coordinating the interaction between controllers and plants.

Once the configuration of all the controllers and plants is done the verification of high level requirements can be performed using the model checker.

4.5.3 Verification Results

The type of requirements to be verified are those listed in Section 4.2. For the purpose of showing the applicability of the methodology, the following requirements are selected:

1. Maximum Overshoot % (OS) $\leq 13\%$.
2. Settling Time (ST) ≤ 40 seconds.
3. Rise Time (RT) ≤ 15 seconds.
4. Steady state error % (SSE) $\leq 1\%$.

The verification is performed by verifying properties on the model. Requirements are then expressed in the query language of the model checker in the form of properties. The verification is conducted by querying the model in a *push-button* fashion because the user simply has to click on the *verify* button with a property as the input to the model checker. The types of properties to be used for the verification are either a *reachability* or a *liveness* property (Table 4.1). The reachability property is used to generate a *witness* trace in case requirements are met and the *liveness* property to generate *counter example* trace in case requirements are not met.

$$\begin{aligned} E <> \mathit{Observer.End} \quad \text{and} \quad \mathit{Controller\_OA.Settled} \quad \text{and} \quad \mathit{Plant\_OA.Settled} \\ \text{and} \quad \mathit{Plant\_OA.OS} \leq 13\% \quad \text{and} \quad \mathit{Plant\_OA.ST} \leq 40 \quad \text{(seconds) and} \\ \mathit{Plant\_OA.RT} \leq 15 \quad \text{(seconds)} \quad \text{and} \quad \mathit{Plant\_OA.SSE} \leq 1\% \quad \text{and} \\ \mathit{Controller\_UA.Settled} \quad \text{and} \quad \mathit{Plant\_UA.Settled} \quad \text{and} \\ \mathit{Plant\_UA.OS} \leq 13\% \quad \text{and} \quad \mathit{Plant\_UA.ST} \leq 40 \quad \text{(seconds) and} \\ \mathit{Plant\_UA.RT} \leq 15 \quad \text{(seconds)} \quad \text{and} \quad \mathit{Plant\_UA.SSE} \leq 1\% \end{aligned} \tag{4.8}$$

Equation 4.8 shows the verification of requirements 1-4 using a *reachability* query in order to get a *witness* trace if requirements are met. The query can be read as: *there exists a path where the observer has reached a final state, both controllers have reached a settled state, both plants have reached a settled state, and in both plants overshoot is less than or equal to the specification, settling time is less than or equal to the specification, rise time is less than or equal to the specification, and steady state error is less than or equal to the specification.*

The result after running the query in Equation 4.8 is a *Fail*. For this reason the model checker does not return a trace. In order to get more information from the model checker, and not just a *Pass/Fail* answer, the *path quantifier* is changed from *E* to *A*. This changes the query into a *liveness* property, which returns a *counter example* trace when the property is not met. After modifying the query and re-running the verification a trace is returned showing why the verification fails: the over approximation fails to meet the overshoot requirement. Table 4.2 shows the results for the abstraction after running the verification, these results are cross-checked by running the simulation in the high fidelity simulator. The requirements values are shown for both the over approximation and the under approximation. Figure 4.16 shows the closed loop behaviour of the system abstraction. Highlighted in red is the overshoot area where the over approximation fails to meet the requirement.

As a sanity check and to corroborate the model checker is performing calculations correctly, the overshoot requirement is changed from 13% to 16% in the model. Given the information in Table 4.2 the expected result is *Pass*. After modifying the overshoot threshold in the model, the query is re-run and the result is a *Pass*. The use of the model checker and the generated traces for both the *Pass* and *Fail* scenarios will become fundamental in Chapters 5 and 6. In Chapter 5 they will be used to tune the controller and in Chapter 6 to find a suitable combination of controller tunings to generate the control schedule.

Table 4.2 Requirements verification results for the system abstraction.

Plant	**Requirement**			
	Overshoot (%)	**Settling Time (sec)**	**Rise Time (sec)**	**Steady State Error (%)**
Under Approximation	12.2	34	6	0.1
Over Approximation	15.74	35.5	6	0.12

If the abstraction meets high level requirements then the original system meets high level requirements. This is derived from the fact that using both *over-approximation* and the *under-approximation* in the model checker makes possible to infer properties of the original

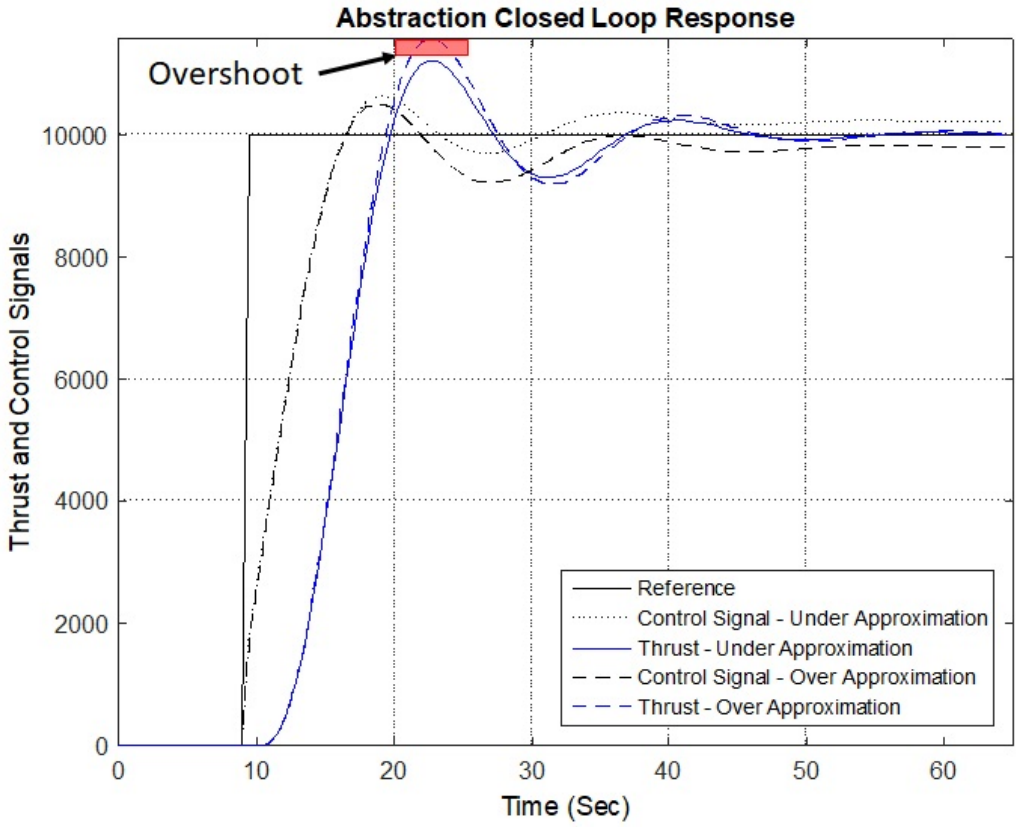

Fig. 4.16 Closed loop response of the system abstraction. Both over and under approximation responses are plotted. Highlighted in red is the area where the over approximation fails to meet the overshoot requirement.

system. Given the nature of the type of requirements to be checked, it can be reasoned that for some requirements it is only required that either the over or under approximation meets the requirement. For example, maximum overshoot is directly linked to the system gain, the *over-approximation* has a higher gain than the original system, therefore if the *over-approximation* meets this requirement the original system then is also guaranteed to meet the requirement. However, the proposed methodology in this thesis verifies that both approximations meet requirements.

The outcome when running the same scenario for the original system (given by Equation 4.2) is shown in Table 4.3. The original model also fails to meet the overshoot requirement. Figure 4.17 shows the closed loop behaviour of the original system. Highlighted in red is the overshoot area where the system fails to meet the requirement.

Figure 4.18 shows a comparison between the system abstraction and the original system closed loop response. The original system response is scaled using the selected value of

Table 4.3 Requirements verification results for the original system.

Requirement			
Overshoot (%)	**Settling Time (sec)**	**Rise Time (sec)**	**Steady State Error (%)**
13.93	35	6	0.05

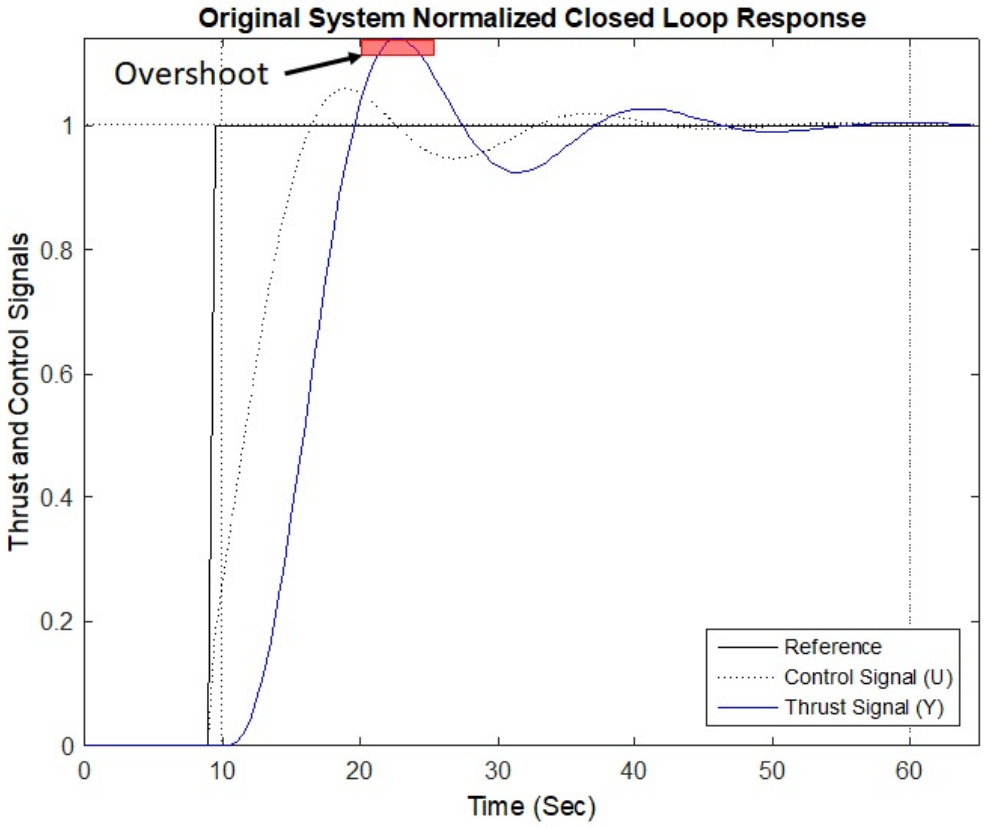

Fig. 4.17 Closed loop response of the original system. Highlighted in red is the area where the system fails to meet the overshoot requirement.

$K_S = 10,000$. The scaled system response is bounded by the system abstraction (over and under approximations).

4.5.4 Discussion

The obtained results show that the model checking approach allows to reason about the original system using the system abstraction and the timed-automata. The model checker returns a *Pass/Fail* answer to the query regarding the high level performance requirements.

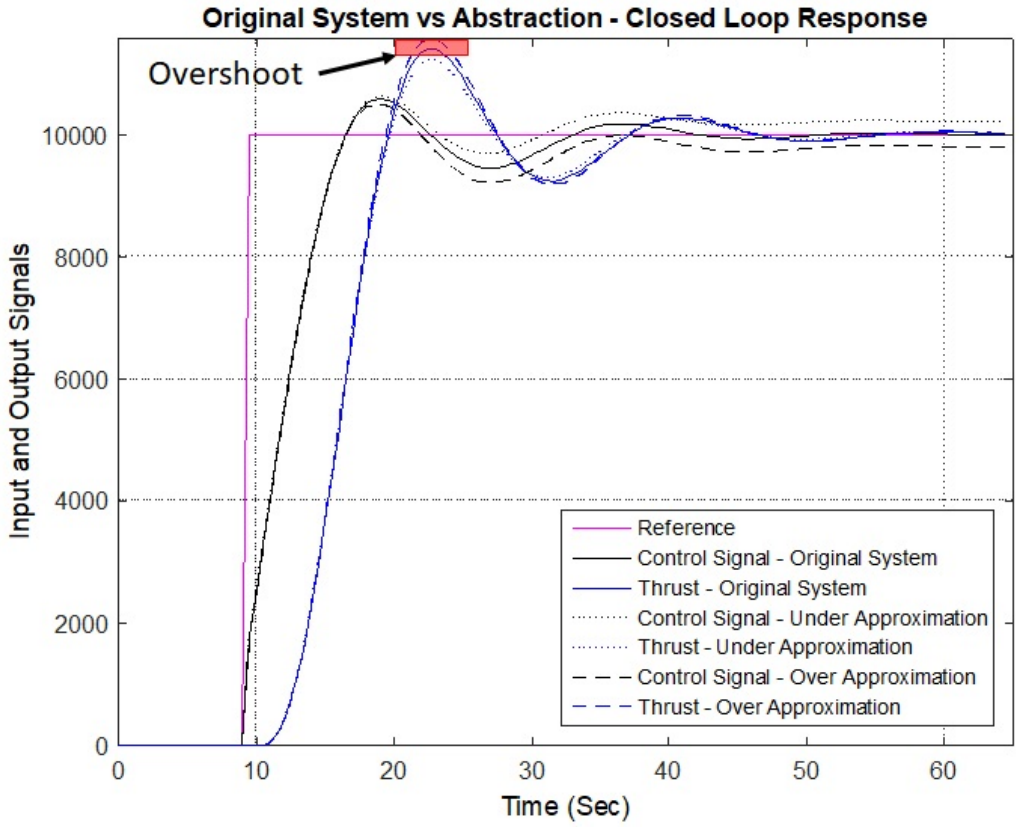

Fig. 4.18 Closed loop response of the system abstraction and the scaled original system. Highlighted in red is the area where the over approximation and the original system fail to meet the overshoot requirement. The original system response is scaled for comparison purposes (using $K_S = 10,000$).

Depending on the result of the query either a *witness* trace or a *counter-example* trace is generated which in turn returns information regarding the *Pass/Fail* result. The posteriori cross-check verification with the original system shows that the system behaviour is bounded by the system abstraction consisting of the *over approximation* and the *under approximation*. This in turn shows how model checking results of *over-approximations* and *under-approximations* combined can be used to infer properties of the original system, high level control performance requirements in this case.

The verification of the high level performance requirements fails because the overshoot requirement is not met. This is detected by the model checker in the *over approximation* where a value of 15.74% of overshoot is detected. Overshoot in the original system is 13.93% which also fails the requirement of maximum overshoot of 13%. Given the fact that the original system response is bounded by the system abstraction there can be cases

where a requirement is not met by the abstraction but indeed is met in the original system (e.g. a 14% maximum overshoot requirement in the previous case shows that). This is an important consideration related to the amount of parametric compensation and fixed-point representation. Nonetheless, if the abstraction meets requirements then the original system also meets requirements.

High level performance requirements are translated into a CTL formula so the model checker can understand them, which removes uncertainty and possible ambiguity during a typical verification and validation phase. So far, the exhaustive verification capabilities of the model checker have not been used because the control system as portrayed in the current automata framework does not include non-deterministic behaviour. Nonetheless it has been demonstrated that the approach is suitable to formally verify control system high level performance requirements using a model checker.

The proposed formal verification methodology presented in this chapter in combination with the abstraction methodology presented in Chapter 3 allow to address the controller design problem from a model checking point of view, which in turn will aid in the gain scheduling design problem (Chapter 6). The following chapter addresses the discrete PID-type controller tuning problem from a model checking point of view. The controller structure is presented in detail and how to digitize the controller transfer function is explained. This is necessary in order to perform the tuning process in the model checker, operating directly over controller gains. The automata presented in this chapter need to be updated to enable these capabilities of controller tuning. Using the updated automata an algorithm to systematically use the model checker in order to find a set of gains is presented: If the current controller tuning does not drive the system to meet requirements, the model checker can explore different tuning configurations in order to find if there exists a valid set of controller gains which drive the system into meeting requirements.

Chapter 5

Digital PID Controller Formal Design

5.1 Overview

The gain schedule control design and verification problem is the main focus of this thesis. In order to address the problem in its entirety the problem was broken down into individual problems which in turn are interconnected and have a dependency relationship. Providing a solution to each of these individual problems lays the foundations for the following problem.

A novel methodology to formally verify high level performance control requirements was presented in Chapter 4. A timed-automata framework was proposed to address the verification problem. The presented approach from Chapter 4 is designed to address the verification problem only. The controller structure is known and if modifications to the controller are needed in order to drive the system into meeting requirements, they must be performed outside the model checking environment. In order to provide a full formal solution to the gain scheduling design problem, the controller tuning problem must also be addressed in a formal manner.

By exploiting the model checking capabilities to perform exhaustive search within a defined operating space, the controller tuning problem can be formulated as a *reachability*/*safety* problem in a similar fashion to the performance requirements verification in Chapter 4. By augmenting the proposed timed-automata (Section 4.3) the PID-type controller gains can be tuned in order to drive the system to meet control performance requirements. In this way, performance requirements become a direct input for the model checker in a *correct-by-construction* fashion to find a possible solution to the problem. By addressing the tuning problem from a model checking perspective, a push-button approach is taken to perform the design and verification of the controller, providing a systematic way to tune the controller with minimum intervention from the user.

The process by which controller parameters are selected in order to meet certain requirements criteria (e.g. stability, performance) is called controller tuning. For a single variable feedback control loop there are many methods for control design and tuning. Among the most popular methods are [121, 167]:

1. Root locus pole placement.
2. Lead-lag compensation.
3. Ziegler-Nichols.

The first two methods are frequency domain approaches where the controller structure is constructed by shaping the closed-loop poles of the system. They rely on a model of the plant to be controlled. The Ziegler-Nichols method can be used with either a model of the plant or with an experimentally derived time-response of the system. This method is aimed at PID-type controllers (known structure). The Ziegler-Nichols rules provide a good guess for the controller parameters to make the closed loop system stable but in no way takes into account performance requirements [121, 167]. The usual approach is to generate a first set of controller gains and then a series of fine tunings are performed to the gains until requirements are met rather than generating the final set of gains in one attempt [121, 167]. In this chapter, a novel formal controller tuning methodology is proposed. From an initial controller tuning (e.g. such as one provided by the Ziegler-Nichols method) the fine-tuning procedure is addressed from a model checking point of view as a *reachability* and *safety* problem synthesising a controller which meets performance requirements.

This chapter is structured as follows: Section 5.2 presents the tuning problem formulation. Section 5.3 presents the updated timed-automata and the controller synthesis methodology. Finally, Section 5.4 presents a case study to demonstrate the applicability of the approach to formally synthesize a PID-type controller.

5.2 Problem Formulation

The following section presents the PID controller tuning problem to be solved in a formal manner. The type of controller and its structure is presented, which in turn drives the formal approach taken in model checking to address the controller tuning.

5.2.1 Discrete PID Controller

PID control is probably the most widely used type of control [59, 85, 101, 121, 122, 124]. The reason for this is its simple implementation either in analogue or digital form and the

fact that it only consists of 3 parameters for tuning (proportional, integral, and derivative gains). The effect of each of the controller gains is well known and has been widely studied, making this type of controller very robust for safety critical applications [6, 44, 59, 85, 101, 121, 122, 124].

There exist different type of PID configurations but it is usually a variation of its ideal form. The continuous transfer function for a PID in its ideal form is given by [101, 121]:

$$G_{I_{PID}}(s) = \frac{U(s)}{E(s)} = K_P(1 + \frac{1}{K_I s} + K_D s) \tag{5.1}$$

where:

- Controller output = U(s).
- System error = E(s).
- Proportional gain = K_P.
- Integral gain = $\frac{K_P}{K_I}$
- Derivative gain = $K_P K_D$

The transfer function of the PID controller can also be expressed in its parallel form. In this form each gain element is independent from each other. The transfer function for this configuration is given by [101]:

$$G_{P_{PID}}(s) = \frac{U(s)}{E(s)} = K_P + K_I \frac{1}{s} + K_D s \tag{5.2}$$

The derivative term improves the transient phase by providing a faster response on the system. On the other hand as a downside it can enhance the effect of noise in the process leading to possible instability. This effect also occurs when the process contains a transport delay [100, 101, 132]. This presents a challenge to select a suitable K_D value and for this reason the derivative term is often removed altogether. In safety critical applications (e.g. a commercial jet-engine) a PI controller is sufficient for most systems [101, 149]. A parallel PI controller configuration is thus selected for this problem formulation. The transfer function of a PI controller in its parallel formulation is given by:

$$G_{P_{PI}}(s) = \frac{U(s)}{E(s)} = K_P + K_I \frac{1}{s} \tag{5.3}$$

The control system considered in the problem formulation of this thesis is one of a continuous signal regulated by a computer-based controller (Figure 2.1). The controller is thus of

discrete nature which makes its implementation easier in the digital domain. Also, from a programming point of view it is expensive to have a numerical integration routine such as Runge-Kutta [125]. Discrete equivalences are preferred to digitize a controller [59, 122, 125]. The result from applying discrete equivalences are difference equations (Equation 3.2). Regarding the plant, a discrete representation is convenient for the same reason: difference equations are computationally less expensive than numerical integration. A discrete approach is convenient for modelling the control system as a whole (Figure 3.2) [1, 6].

There are 3 continuous to discrete equivalences to map the *S-plane* (continuous) to the *Z-plane* (discrete) using a sampling period T [59, 122]:

1. Forward rectangular rule: $s \longleftarrow \frac{z-1}{T}$.
2. Backward rectangular rule: $s \longleftarrow \frac{z-1}{Tz}$.
3. Trapezoid rule (Tustin method): $s \longleftarrow \frac{2}{T}\frac{z-1}{z+1}$.

Regarding the equivalences, it is worth mentioning some remarks [59, 122]:

1. Using the forward rectangular rule the discrete-time system is stable only if the continuous system is. It can lead to having a continuous stable system but discrete unstable.
2. Using the backward rectangular rule the discrete-time system is stable if the continuous system is. It can lead to having a discrete stable system but continuous unstable.
3. Using the trapezoid rule the discrete-time system is stable if and only if the continuous system is. It is the only equivalence that fully maps the stable region on the *S-plane* to the stable region in the *Z-plane*.

For these reasons the trapezoid rule is selected to convert the parallel PI continuous transfer function (Equation 5.3) to its discrete version. The resulting discrete transfer function for the parallel PI controller is given by:

$$G_{P_{PI}}(z) = \frac{U(z)}{E(z)} = \frac{(K_P + \frac{T}{2}K_I)z + (\frac{T}{2}K_I - K_P)}{z-1} = \frac{b_0 z + b_1}{z-1} \tag{5.4}$$

where:

- $b_0 = K_P + \frac{T}{2}K_I$.
- $b_1 = \frac{T}{2}K_I - K_P$.

In a generic way the tuning problem can be addressed as to find a set of values for K_P and K_I in order to drive the system into meeting design requirements. In this work the tuning will be performed directly over the K_P and K_I gains values. This means that coefficients b_0 and b_1 from Equation 5.4 have to be calculated in the model checker using the proposed data-type representation and its arithmetic (Section 3.3.4).

In this thesis, a model checking approach for tuning PID-type controllers is proposed. By the means of portraying the tuning problem as a *reachability* model checking problem, the model checker can then find a possible solution to the tuning problem. In the same manner as in Chapter 4 the verification of high level performance requirements was performed with a push-button approach, the controller tuning problem is addressed in the same way: the tuning procedure is performed by querying the model checker. The solution must then comply with the selected tuning criteria: high level control system performance requirements. The following section explains how the tuning problem is portrayed in the model checker in order to find a set of gains that drive the system into meeting requirements.

5.2.2 Controller Tuning: A Model Checking Formulation

In Chapter 4 a novel formal verification methodology of high level performance requirements for a control system was presented. In the Chapter 4 formulation the controller configuration was known and fixed, the controller tuning could not be modified. The verification consisted of observing the trajectory of the process in the operating space to determine whether or not the process dynamics met requirements. The reference tracking control problem is portrayed in Figure 4.5, Section 4.3. The trajectory followed by the process is determined by the process dynamics and its interaction with the controller in the pre-defined controller tuning. If the process failed to meet requirements the controller tuning had to be performed outside the model checking environment using additional methods. Here, a novel PID-type controller tuning methodology is presented, a formal approach to generate the controller gains which drive the system to meet requirements. In this manner, using high level performance requirements as a formal input to the model checker, the controller tuning will be generated by the means of the exhaustive verification capabilities of model checking.

To tune the PI controller in order to meet performance requirements requires finding a set of K_P, K_I gains which drive the process into the desired operating space area within the timing constraints while avoiding visiting certain operating space (e.g. overshoot area). Depending on the selected data representation format, the controller gains are also limited to an operating space much in the same way that the process and controller output are restricted to an operating space. If the controller does not meet requirements in its current tuning, the

task is to find if there exists another combination of gains that drive the process' trajectory in the desired way.

Figure 5.1 shows how starting from an initial controller configuration set of gains their values are modified in order find a new set of values which drive the process trajectory into meeting performance requirements. The tuning is performed by modifying each gain by the means of finding a Δ value by decreasing or increasing the gains in predefined step sizes. The operating space in the PI controller tuning formulation problem is the cross product

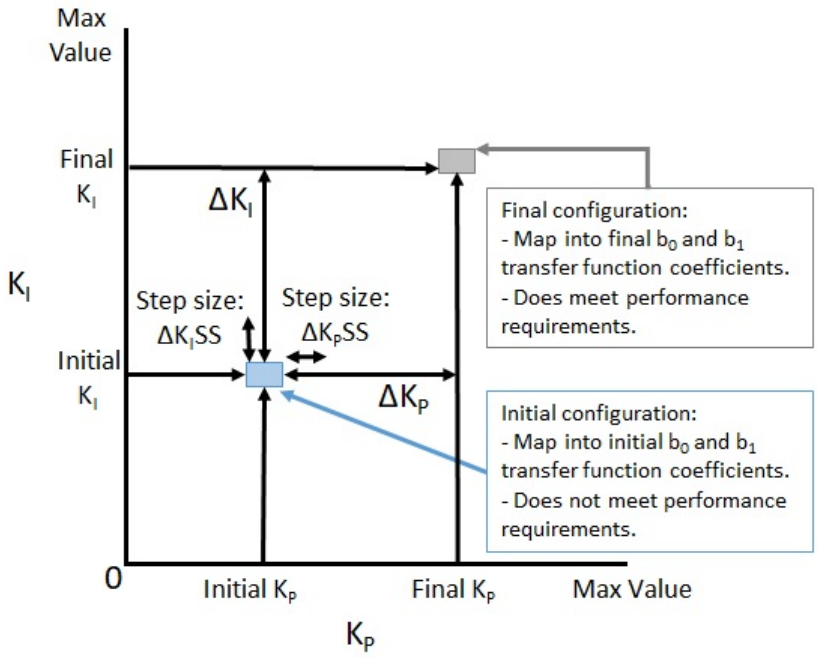

Fig. 5.1 Operating space for PI controller gains K_P and K_I. To tune the controller gains requires finding a different combination of gains which drive the system's dynamics into a trajectory that meets requirements. Starting from an initial set of gains a search is performed to find a possible solution to the control problem.

of the ranges of both gains. Ideally, the ranges are limited by the maximum allowed value according to the selected data representation using the ad hoc data type from Chapter 2. For practical reasons and to limit the search space thus preventing a possible state space explosion, boundaries are assigned for each gain in order to reduce the possible gain configurations.

Figure 5.2 shows a graphical representation of the search space for K_P and K_I gains with upper and lower boundaries. From the initial configuration, upper and lower bounds are imposed on both gains' ranges thus limiting the search space to the intersection of the two bounded areas. The resulting operating and search space corresponds to the intersection of the bounded areas for both K_P and K_I. It is in this operating space where the model checker

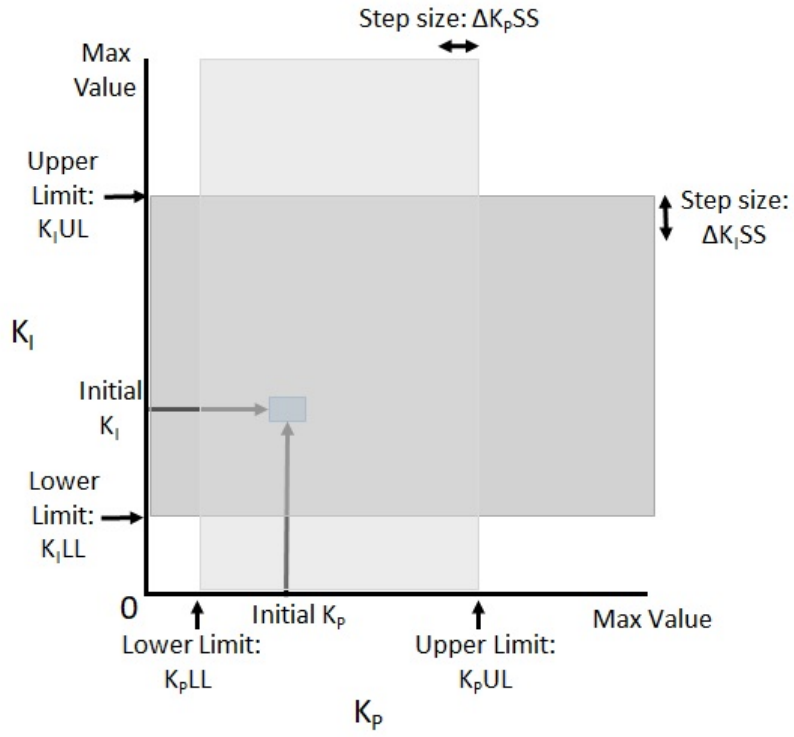

Fig. 5.2 Limited operating space for PI controller gains K_P and K_I. The search space for each gain is limited to a certain area with upper and lower boundaries. The available combination of controller gains are comprised in the intersection of the two areas. The model checker uses this bounded area as the search space to find a controller gains combination that drives the system into meeting requirements.

performs the exhaustive search. The tuning procedure requires the following variables to be configured before the search is performed:

- Initial K_P and K_I values.
- Minimum and maximum allowed values for both K_P and K_I gains.
- Step size for K_P and K_I gains. This is the resolution allowed when performing modifications in the initial gain values.
- Choosing how to perform the tuning: modifying K_P only, K_I only, or both gains.

Also, the following considerations regarding the tuning procedure as a search problem in the model checker have to be taken into account:

- Step size: having too small a step size may create memory issues in the actual implementation because of a state space explosion. Having too big a step size may limit the search in a way that a solution is not found when in fact there exists one.

- Number of gains to be modified: when modifying one gain only the search space is smaller and a solution could be found faster. Modifying both gains may result in running out of memory before finding the solution.
- The possibility that no actual solution exists.

These considerations are taken into account when configuring the model checker to perform the tuning procedure. The proposed tuning approach will consider a variable step size for each gain, configurable upper and lower boundaries for each gain to limit the search space, and the possibility of tuning either K_P, K_I, or both. Section 5.3.3 explains the tuning procedure in more detail.

The novel controller tuning approach needs to be added to the current automata framework from Chapter 4. Before verifying high level control requirements as explained in Chapter 4, the PI tuning feature will be added. The timed-automata need to be updated to include the controller gains search prior to the requirements verification, this also requires the calculation of the controller b_0 and b_1 coefficients according to the newly selected controller gains. The following section explains the necessary updates to the timed-automata to include these features.

5.3 Controller Synthesis Methodology

In order to tune the digital PI controller using the automata framework from Chapter 4 two features have to be included:

1. K_P and K_I gains selection.
2. b_0 and b_1 coefficients calculation for the PI transfer function.

The selection process to modify the controller gains is performed non-deterministically by the model checker before the requirements verification. The tuning procedure must address the following items:

1. Define K_P and K_I initial values.
2. Define K_P and K_I upper and lower limits.
3. Define K_P and K_I step size to perform the search.
4. Define if both gains are to be modified or only one of them, if only one, which one.

5. Enable the model checker to perform the search within the predefined boundaries and step sizes.

6. Calculate b_0 and b_1 coefficients for the PI discrete transfer function.

7. Perform the requirements verification in the same fashion as it was presented in Chapter 4: a *reachability* problem.

After modifying the initial gain values, the transfer function coefficients will be updated to reflect this change and finally the requirements verification will be performed. The model checking procedure for tuning the PI controller is presented in its entirety in Section 5.3.3. From the model checker point of view, the *reachability* query does no longer mean verifying the trajectory versus high level requirements only. The query is now also posing the question about the existence of a trajectory within the predefined boundaries of the gains by which the controller can drive the system in order to meet requirements. The controller is tuned simultaneously in a formal way aiding not only in the verification process but also in the design process. The following section explains the necessary automata updates to integrate the formal PI tuning procedure to the current automata framework.

5.3.1 Timed-Automata Update

In order to enable the features to solve the PI controller tuning problem in the previously proposed automata (Section 4.3.1), modifications are made to the *Observer* automata from Figure 4.9. Both the *Controller* (Figure 4.8) and *Plant* (Figure 4.7) automata remain without changes. In more detail:

Observer

Figure 5.3 shows the automata in charge of coordinating the execution of the controller and the plant. This automata monitors both the controller output U and the process output Y to determine when to finalize the simulation. A new state has been added so after the initialization process the tuning process takes place. This state will modify the initial K_P and K_I gains and calculate the new b_0 and b_1 coefficients for the controller.

The automaton consists of the following states:

1. *Init:* Initial state. Initialize the models for simulation. After the initialization the system transitions to the *Select Gains* state.

2. *Select Gains:* Gain selection state. In this state both K_P and K_I gains are modified and the new b_0 and b_1 coefficients for the controller are calculated. After the gain

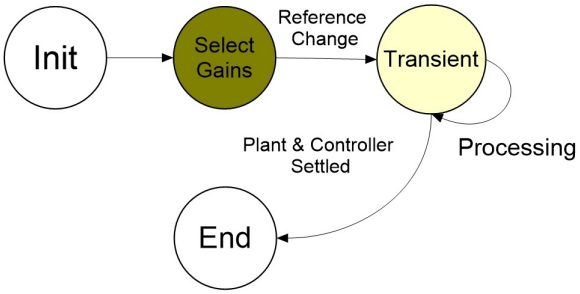

Fig. 5.3 Updated *Observer* automaton. Automaton in charge of the tuning procedure of the controller gains and controlling the data flow between the controller and the plant. This automaton monitors the control signal U and output process signal Y to determine transitions between states.

modifications have been performed, the input reference signal is monitored in order to trigger an event. Once a change in reference has been detected a transition to *Transient* state is taken.

3. *Transient:* Transitory state. This is the state in charge of coordinating the execution of the *Plant* and *Controller* automata. It monitors both the process output signal Y and the controller signal U to determine when the process has reached a settling condition. Once both the controller and the plant have settled a transition is triggered to the *End* state.

4. *End:* Final state. After an event (change in reference) if an equilibrium condition is reached the dynamics process comes to a halt and no more transitions are allowed in any automata.

The *Select Gains* state and its functionalities have to be included in the UPPAAL implementation. The UPPAAL implementation with the updated features is shown in Figure 5.4.

Observer: UPPAAL Implementation

Figure 5.4 shows the automaton in charge of coordinating the correct execution of the controller and the plant. The automaton modifies the values of the controller gains K_P and K_I in order to tune the controller. Once the gains have been modified the calculation of the controller's transfer function coefficients is done. This automata monitors both the controller output U and the process output Y to determine when to finalize the simulation. The coordination of the execution of the controllers and the plants is achieved by using synchronization channels C and D (one per system approximation). This automata sends a

message via channels *C* and *D* to enable the execution of either the *Plant* or the *Controller* automata for both the over and under approximations.

The automaton consists of the following states:

1. *KP:* Initial and K_P tuning state. This state functions partially as the *Init* state and the *Select Gains* state depicted in Figure 5.3. The tuning process is split into two states instead of one as in the automata from Figure 5.3. This state is dedicated to the selection of the K_P gain. The model checker non-deterministically selects a ΔK_P value to modify the initial K_P gain. The size of ΔK_P is determined by the $\Delta K_P SS$ size step and the $K_P UL$ upper and $K_P LL$ lower boundaries (Figure 5.2). This state can be bypassed to avoid modifying the K_P gain value. Once the gain modification has been performed a transition to *KI* state is taken.

2. *KI:* K_I tuning state. This state functions partially as the *Select Gains* state depicted in Figure 5.3. The model checker non-deterministically selects a ΔK_I value to modify the initial K_I gain. The size of ΔK_I is determined by the $\Delta K_I SS$ size step and the $K_I UL$ upper and $K_I LL$ lower boundaries (Figure 5.2). This state can be bypassed to avoid modifying the K_I gain value. Once the gain modifications have been performed function *CoCa* is called to calculate the new b_0 and b_1 coefficients. Input reference signal is monitored in order to trigger an event. Once a change in reference has been detected a transition to *Sync* state is taken.

3. *Sync:* Transitory state. Similar to *Transient* state depicted in Figure 5.3. State in charge of coordinating the execution of the *Plant* and *Controller* automata. It monitors the process output signal *Y*, the controller signal *U*, and the elapsed time to determine when to finish the automata execution. The observer coordinates both the over and under approximations via communication channels. Once both the controller and the plant in both over and under approximations have settled and no time-out condition is detected a transition is triggered to the *End* state.

4. *End:* Final state. Similar to *End* state depicted in Figure 5.3. After an event (change in reference) if an equilibrium condition is reached the dynamics process comes to a halt and no more transitions are allowed in any automata.

5. *Timeout:* Fail-safe state. A pre-defined amount of time is given to the process to reach an equilibrium. In order to save memory and computations, this condition is added. If the process does not reach an equilibrium within the allowed amount of time the automaton transitions to this state.

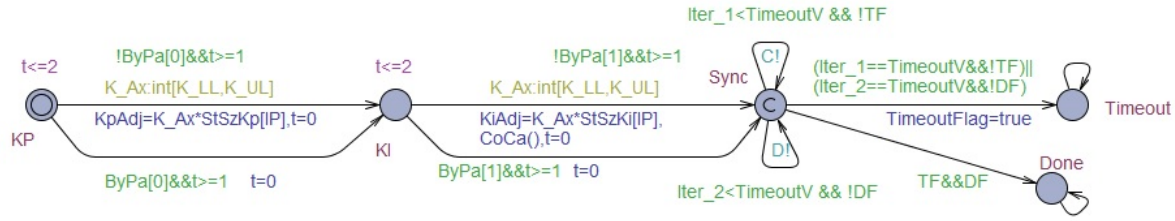

Fig. 5.4 UPPAAL implementation of the observer automaton. Automaton in charge of controlling the data flow between the controller and the plant. The controller tuning process has been incorporated. Gains K_P and K_I are modified in order to find a suitable combination which drives the process to meet requirements. This automaton monitors the control signal U, output process signal Y, and the elapsed time to determine transitions between states. It coordinates the execution between plants and controllers using communication channels C and D. One channel is used for the over approximation and the other one for the under approximation.

With these new features included in the automata the controller tuning process can take place. From a model checking point of view, the controller *design* process is similar to the previously presented requirements *verification* process in Section 4.4. The following section explains the required modifications to the model checker queries and the differences with the previous verification process.

5.3.2 Requirements Formulation for Design

The controller tuning problem is a search problem as portrayed in Figure 5.2 and can be formulated in the following way: *is there a set of gains K_P and K_I that can drive the system closed loop behaviour to meet the closed loop requirements?* The verification of the closed loop requirements is done in the same way as explained in Section 4.4: a trajectory search like the one in Figure 4.6. The gain search is performed verifying closed loop requirements.

The controller tuning design procedure is thus performed using a push-button approach by querying the model checker using a *reachability* property (Table 4.1) like Equation 4.1. The query is the same as the one presented during the *verification* phase. The reason for this is because the verification still takes place in order to decide if the design is successful or not. The main difference is that in this formulation the controller gains are modified before performing the verification.

If the property verification is successful the model checker will return *Pass* as a result. A *witness* trace will be generated to show how to obtain this result. As part of the *witness* trace the modification to the controller gains (Δ_{K_P} and Δ_{K_I}) will be returned by the model checker.

If the property verification is not successful the model checker will simply return *Fail* without a trace. This means that within the K_P and K_I search space there is not a combination that satisfies the requirements, a modification of the search space is necessary to systematically find a possible gain combination. The following section explains how to perform a systematic search in order to find a solution to the tuning design problem.

5.3.3 Controller Tuning Algorithm

Using the updated automata the controller tuning design problem can be addressed by performing a systematic search in the operating space of both K_P and K_I gains. Algorithm 4 presents the necessary steps to address the design and verification of the PI controller problem.

Algorithm 4: Controller formal design and verification procedure.

Input : Linear plant, Closed-loop performance requirements, Initial tuning: K_P and K_I, Fixed-Point integer-only representation for abstraction (I, F), Coefficients scaling gain K_{ab}.

Output : Tuned controller gains.

1. Using the selected fixed-point integer only representation and coefficients scaling gain K_{ab}, convert the initial K_P and K_I gains into the chosen representation.
2. Use the abstraction methodology (Chapter 3) to generate and implement the system abstraction in the model checker.
3. Bypass the gains modification states in the automata and perform an initial requirements verification (Section 4.4). If the verification is successful no tuning is necessary. If not, proceed to step 4.
4. Remove the bypass for the gain modification states.
5. Select a resolution (step size) for K_P and K_I gains.
6. Select upper and lower limits for the gain values search space.
7. Verify the system property (Equation 4.1) to find a new set of gains.
8. If no solution is found in the current search space, change resolution (step size), upper/lower limits for the gain values operating space, and/or restrict the search to a single gain. Go back to step 7, repeat.
9. If a solution is found, use Δ_{K_P} and Δ_{K_I} values returned by the model checker to calculate the new set of gains.

The objective of this design and verification process is to find a suitable set of gains for the controller which drive the closed-loop behaviour of the system to meet the given requirements. The process is iterative but it requires minimal intervention from the designer. A solution may be found in the first iteration but there is no guarantee for this to happen. Also, it may be the case that the given requirements cannot be met by the current controller

structure. If a solution exists the iterative process can eventually find it. The outcome after applying Algorithm 4 will be a set of controller gains which meets design requirements and has been formally verified. The following section shows how to apply the methodology using the design problem case study presented in Chapter 4.

5.4 Case Study: Thrust Control Design

The control design problem to be solved in this thesis is that of a gain scheduling control scheme using model checking to formally verify and design the schedule. In Section 4.5 a brief description of the necessary tasks to address in order to solve the main design problem as sub-problems was presented. So far items 1-4 have been addressed:

1. **Abstract the control system dynamics:** Chapter 3 presented the system dynamics abstraction methodology to be used to implement the control system in a model checking environment.

2. **Implement the control system abstraction in a model checking environment:** Chapter 4 presented how to use the system abstraction along with a set of proposed automata so that the abstraction can be implemented in a model checking environment.

3. **Verify high level performance requirements for the control system using model checking:** Chapter 4 presented how high level performance requirements for a control system can be formally verified by portraying the control problem as a *reachability* problem in model checking.

4. **Perform the controller tuning process using model checking:** Section 5.2 presented how to portray the tuning design problem in a similar fashion as the requirements verification problem: a *reachability* problem in model checking. Section 5.3 presented the full novel methodology for how to tune the controller using model checking.

By providing solutions to the aforementioned tasks a novel gain schedule formal design and verification methodology can be constructed; the full methodology will be presented in Chapter 6. In this section the applicability of the novel PI controller formal design and verification methodology is demonstrated (item 4). Taking the results in Chapter 4 Section 4.5.3 as a starting point, the PI controller tuning problem is addressed using the formal design and verification methodology presented in this chapter.

5.4.1 Requirements and Initial Conditions

The high level performance requirements for the control design and verification problem are as from Section 4.5.3:

1. Maximum Overshoot % (OS) $\leq 13\%$.
2. Settling Time (ST) ≤ 40 seconds.
3. Rise Time (RT) ≤ 15 seconds.
4. Steady state error % (SSE) $\leq 1\%$.

The system and its abstraction (recap from Section 4.5.2) are:

- Original system transfer function is given by Equation 4.2.
- Sampling time $T = 0.5$ seconds.
- Parametric compensation gain $K_U = 0.0005$.
- Data-type representation format: 1 integer digit ($I = 1$) and 4 fractional digits ($F = 4$).
- Input-Output scaling gain K_S = 10,000.
- Coefficients scaling gain K_{ab} = 16,384.
- Abstraction *over approximation* is given by Equation 4.5.
- Abstraction *under approximation* is given by Equation 4.6.

The initial configuration for the discrete PI controller was generated using the Ziegler-Nichols method:

- Controller discrete transfer function is given by Equation 5.4
- Initial proportional gain $K_P = 0.1392$.
- Initial integral gain $K_I = 0.1496$.

From this starting point Algorithm 4 is applied to design and verify the PI controller in order to meet high level performance requirements from Section 4.5.3. The query used to find a solution to the tuning problem is described by Equation 4.8.

5.4.2 Results and Discussion

From the obtained results in Section 4.5.3 it is known that the initial controller configuration fails to meet all the requirements: overshoot is not in compliance. Algorithm 4 is then used to find a solution to this control problem. The results are summarized in Table 5.1. From an initial K_P and K_I operating space selection the model checker is systematically used to find a suitable set of gains. Step 8 in Algorithm 4 is where the search is reconfigured in case no solution is found by the model checker with the current step sizes and boundaries. After every iteration it is the designer's arbitrary decision to select a different step size, boundaries, and to tune 1 or 2 gains. In this example, after every iteration, step sizes and boundaries were changed in order to find a solution.

Table 5.1 Controller gains operating space configurations and verification results. The initial values for every iteration are $K_P = 0.1392$ and $K_I = 0.1496$.

Parameter	Iteration 1		Iteration 2		Iteration 3	
	K_P	K_I	K_P	K_I	K_P	K_I
Step Size	0.0021	0.0021	0.0036	0.0036	0.0067	NA
Upper Limit	0.1926	0.2030	0.2231	0.2335	0.3070	NA
Lower Limit	0.0858	0.0961	0.0552	0.0656	0.0000	NA
Result	Fail		Fail		Pass	

Using Algorithm 4 provides a new set of gains that correctly drive the process into a trajectory which meets all the requirements. It was necessary to have 3 iterations in the search. The first two searches were performed modifying both K_P and K_I gain values. Both verifications returned negative results due to a memory issue: the search space was too big for the model checker to handle. Instead of increasing the step size to reduce the search space explosion even if it meant to increase the total range of the search, it was decided instead to just modify the proportional gain K_P. The model checker arrives at a solution. The initial and final tunings with their respective gain modifications are shown in Table 5.2.

Table 5.2 Initial and final gain values for the PI controller.

Parameter	**Initial**	**Final**	Δ
K_P	0.1392	0.1862	0.0470
K_I	0.1496	0.1496	0.0000

Table 5.3 shows the results for the abstraction after running the design and verification process. The requirements values are shown for both the over approximation and the under

approximation. The *witness* trace returned by the model checker provided the required modifications to the initial gain values. Figure 5.5 shows the closed loop behaviour of the system abstraction.

Table 5.3 Requirements design and verification results for the system abstraction.

Plant	**Requirement**			
	Overshoot (%)	**Settling Time (sec)**	**Rise Time (sec)**	**Steady State Error (%)**
Under Approximation	9.49	32	5.5	0.13
Over Approximation	12.96	33.5	5.5	0.08

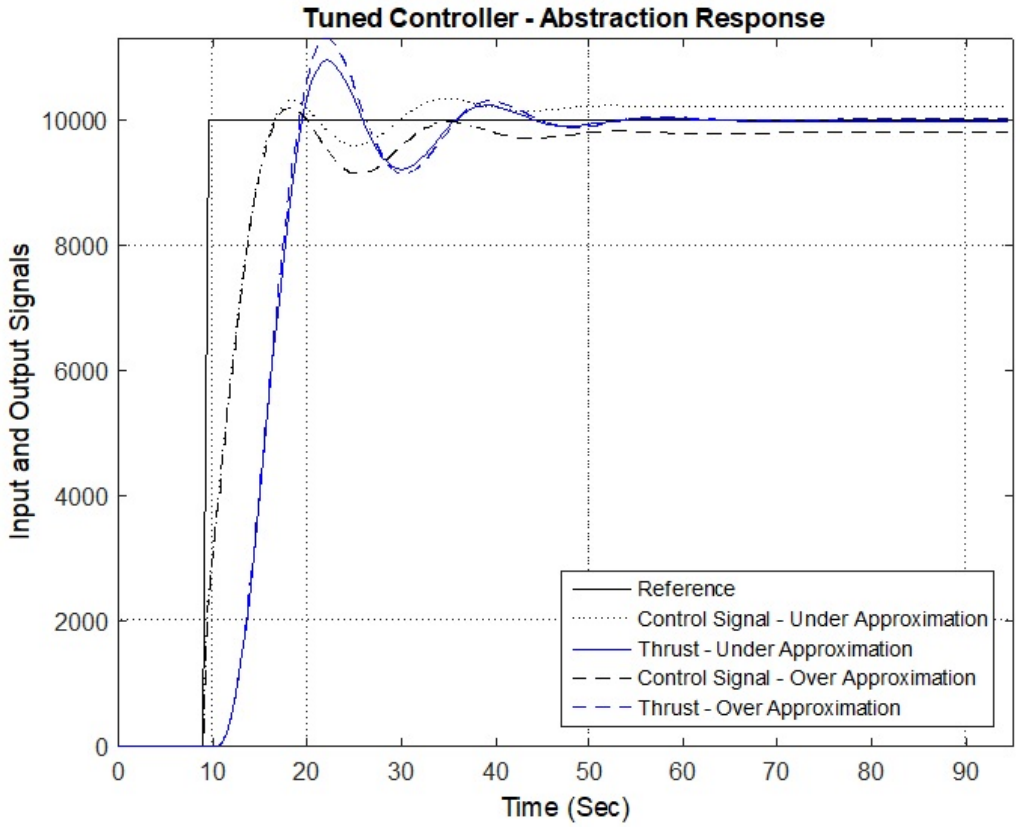

Fig. 5.5 Closed loop response of the system abstraction. Both over and under approximation meet all the requirements.

Both the under approximation and the over approximation meet all the requirements (Table 5.3). A comparison is performed between the abstraction and the original system. The

original system response is scaled-up for comparison purposes using input-output scaling gain $K_S = 10,000$. This comparison is shown in Figure 5.6. As expected the original system is bounded by the abstraction, which in turn means that the original system also meets the requirements.

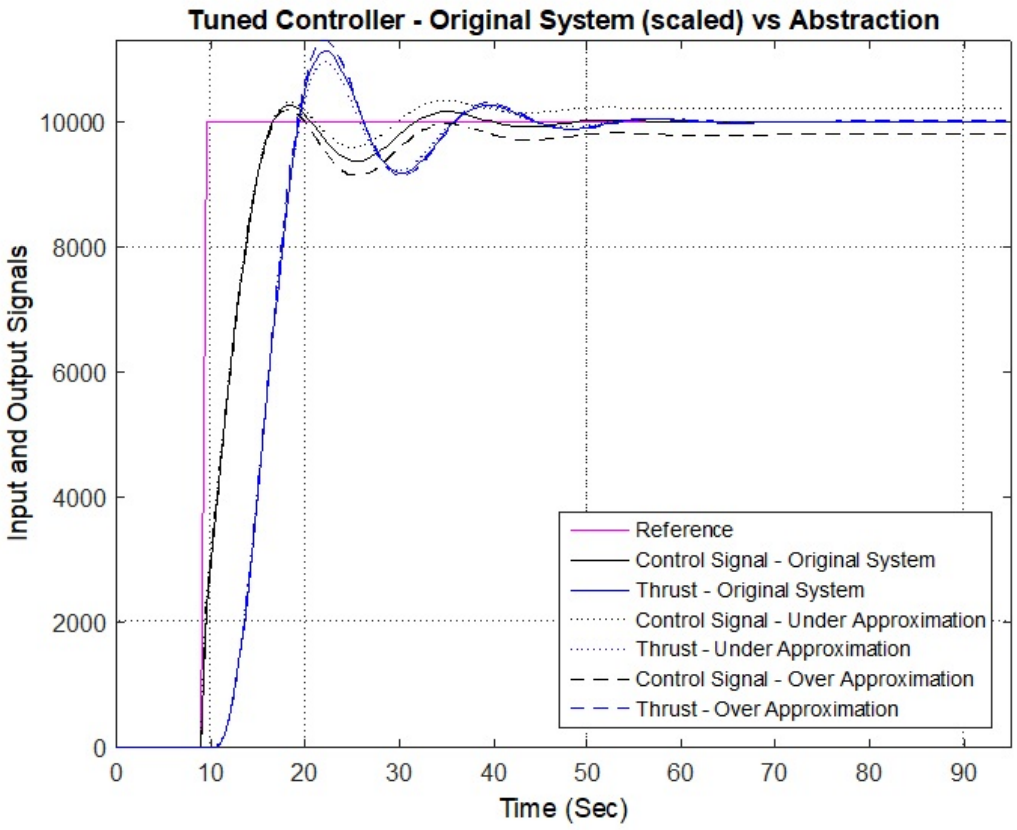

Fig. 5.6 Closed loop response of the system abstraction and the scaled original system. Both the abstraction (over and under approximations) and the original system meet the requirements as determined by the model checker. The original system response is scaled for comparison purposes (using $K_S = 10,000$).

The new gain values returned by the model checker make the system follow a trajectory which meets the desired requirements. Finally, a comparison is performed between the original system initial controller tuning and the final controller tuning. Table 5.4 shows the closed loop performance values for both controller configurations and Figure 5.7 shows the response for the system under both controller tuning configurations.

Table 5.4 Requirements comparison between initial and final controller tunings.

Tuning	**Requirement**			
	Overshoot (%)	**Settling Time (sec)**	**Rise Time (sec)**	**Steady State Error (%)**
Initial	13.93	35	6	0.05
Final	11.29	33	5.5	0.05

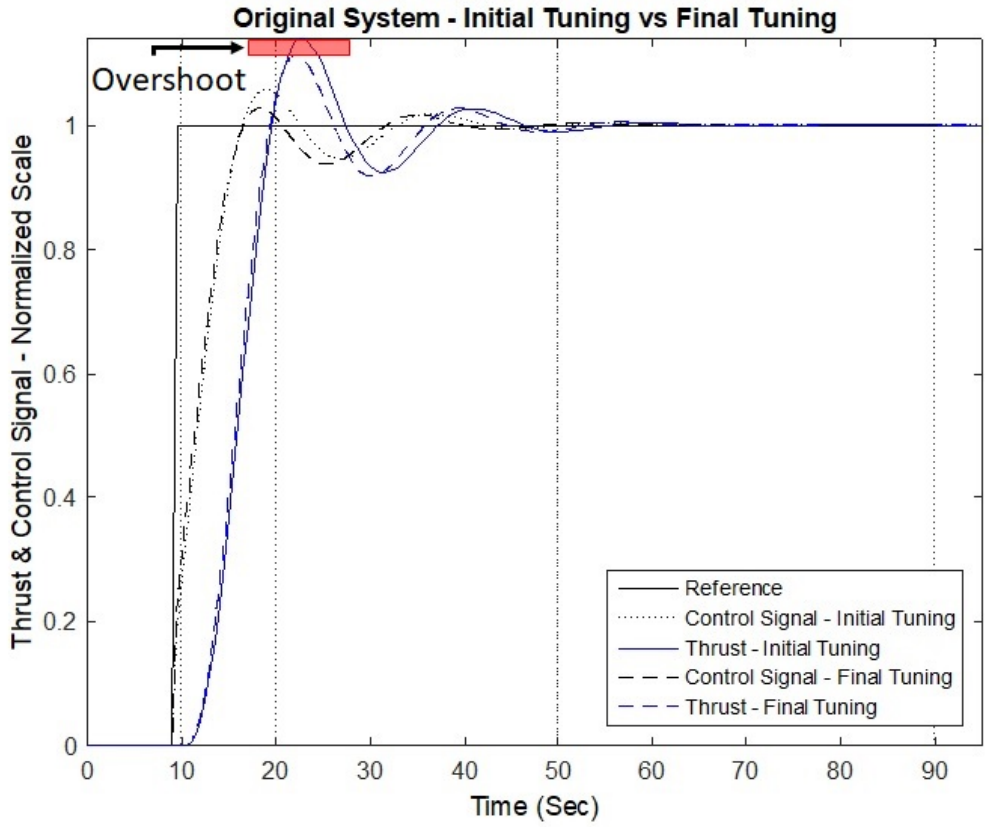

Fig. 5.7 Closed loop response for the original system using the initial PI controller gains and the final controller gains. Highlighted in red is the overshoot area where the initial controller tuning fails to meet the requirements.

Discussion

A novel PI controller tuning methodology has been presented in this chapter. The abstraction methodology (Chapter 3), and the verification approach for high level performance requirements (Chapter 4) set the building blocks to allow the formal design of the PI controller presented in this chapter. The methodology relies on model checking to perform the con-

troller tuning. By the means of approaching the PI tuning problem as a search problem in model checking, the controller gains can be modified so that high level control performance requirements are met. The methodology takes a controller configuration as a starting point, this is not a necessary requirement but it is desirable that this is the case. The reason for this is because if a solution to the tuning problem exists, the closer to a solution the initial configuration is the faster Algorithm 4 will converge. Starting from an initial configuration which is far from a solution will likely require more iterations over Algorithm 4 because the state space for the model checker to handle may be too big and memory runs out or indeed no solution exists in the current search space. However, and as demonstrated in the case study, Algorithm 4 can systematically be applied to find a solution to the problem.

Starting from an initial controller tuning that drives the system into a stable behaviour is desirable. This can be achieved by classical methods such as Ziegler-Nichols. Even if many methods for tuning a PID controller exists, the proposed methodology in this thesis takes a *correct-by-construction* formal approach where high level requirements are included in the tuning procedure from the start point.

The controller design and verification is performed using the discrete version of the controller, this version being the one that usually is implemented in the embedded system, hence the methodology considers the software implementation side of the design process. Verifying software compliance with requirements is a challenging task from a software development cycle point of view. Also, the safety of the software has to be proven as well, current practices rely strongly on extensive testing to comply with both assurance and safety regulations. By incorporating model checking into the design and verification framework of embedded PID-type controllers, testing practices can be improved by increasing testing coverage from early design stages and increasing the error detection rate as well.

The model checker allows to address the tuning problem with a push-button type of approach, requiring minimum intervention from the design engineer. This in turn can help reducing the gap between formal methods and industrial software practices for safety critical systems. The current framework allows to formally verify a closed loop control system versus performance requirements as well as to aid in the design process of the controller in case tuning the controller is needed.

The final design goal is to formally design and verify a gain schedule controller. The presented tuning methodology fits well within this problem because from an initial controller configuration the model checker can be used to find various controller tunings to systematically construct a schedule, requiring at most one initial controller tuning generated by classical methods. Taking the current framework as a starting point, the following chapter

shows the required additions in order to address the gain scheduling design and verification problem from a model checking design point of view.

Chapter 6

Digital PID Gain Scheduling Control Formal Design

6.1 Overview

Gain scheduling is a commonly used control scheme for non-linear processes and safety critical applications. It is appealing due to its simplicity compared to more advanced control methodologies [127, 149]. In safety-critical systems it is extensively used (e.g. commercial jet engines) and it is implemented in the form of embedded software [127, 149]. The usual approach to embedded control is to design analogue controllers and digitize them for implementation in a computer-based system (Fig. 2.1). Airworthiness certification requires evidence to show the correct behaviour of the system prior to operation, which in turn derives in a series of extensive and time consuming procedures to ensure safety [18, 43, 51]. However, current development and certification practices are prone to human error and requirements ambiguities [53, 106].

Demonstrating safety and requirements conformity for a gain scheduled controller is challenging from the design, verification, and implementation points of view. To guarantee safety and conformity with requirements, extensive testing is performed. It is estimated that current testing activities amount to approximately 30% to 50% of the total cost of a software project [7]. It is therefore desirable to find a new approach to verification and validation.

For the first time, using a *correct-by-design* approach a gain scheduled control scheme is formally designed and verified using model checking. The formal verification of control requirements and controller design is enabled by the proposed model abstraction methodology (Chapter 3), verification methodology (Chapter 4), and controller design methodology (Chapter 5). The proposed gain schedule design and verification methodology requires

minimum intervention from the control engineer because the schedule is incrementally constructed via a push-button approach enabled by the model checker. The end result consists of a gain schedule with the minimum number of controller tunings which satisfy high level performance requirements.

This chapter is structured as follows: Section 6.2 presents the discrete PID-type gain schedule problem formulation. Section 6.3 presents the new timed-automata to address this problem and presents the controller synthesis methodology. Finally, Section 6.4 presents the case study to show the applicability of the approach to formally synthesize a gain schedule PID-type control scheme and a discussion about the generated results.

6.2 Problem Formulation

To address the gain schedule design problem, this thesis proposes a methodology to systematically and incrementally construct a schedule to satisfy high level control performance requirements. The abstraction methodology presented in Chapter 3 allows to implement a dynamic control system in a model checking environment, allowing to formally verify high level control performance requirements as presented in Chapter 4. Chapter 5 presented a PID-type controller tuning method using model checking, taking as a formal design input the high level performance requirements into the model checker. By expanding the previous framework from Chapters 4 and 5, the necessary features to address the gain schedule design problem will be added. The following section explains in more detail the problem to be solved and how to formulate it from a model checking perspective.

6.2.1 Gain Scheduled PI Control

Gain scheduled control was developed as a solution to non-linear problems and it is an early attempt at adaptive control [94, 97]. The approach assumes a strong relationship between a measured variable which characterizes the operating conditions and the plant model. It is then possible to minimize the effect of parameter variation by changing the controller parameters accordingly. It can be considered as *open-loop adaptive control* because the performance modifications resulting from the change in controller parameters are not used in a feedback fashion to quantify the efficiency of parameter adaptation [94]. The implementation is usually a computer-based look-up table which gives the controller parameters for a given set of environment measurements. Due to its simplicity gain scheduling control is widely used, especially for safety critical applications [138].

As mentioned in Chapter 5, PID control is extensively used because of the number of parameters available for tuning (e.g. proportional, integral, and derivative gains) and the great body of knowledge on the individual controller actions. Furthermore, for critical applications in many cases a PI controller is the chosen configuration to avoid dealing with the possible side-effects of the derivative action [101]. A gain scheduling control scheme in combination with a PI controller structure is a versatile and easy to deploy option from a design and implementation point of view. The complexity resides in the number of operating points, compliance with high level requirements, stability, and the control system actual implementation [97, 127, 138].

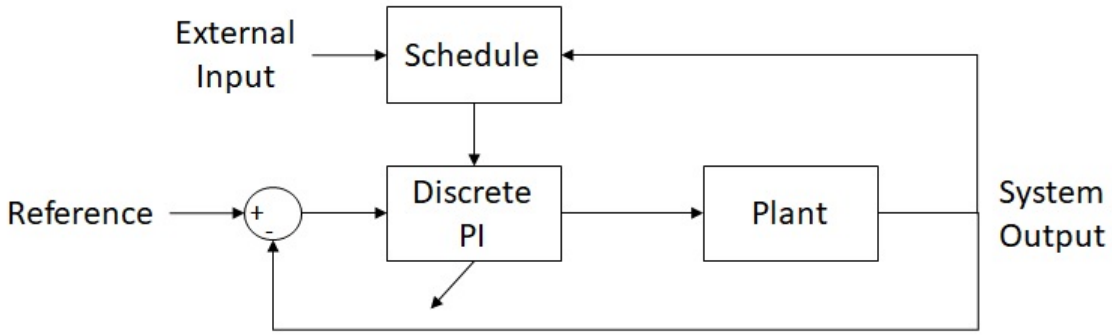

Fig. 6.1 Gain scheduled control scheme. The controller has a proportional+integral (PI) structure (Equation 5.4). The schedule is driven by a combination of external inputs and the controlled variable.

Designing a control gain schedule scheme and verifying its conformity with high level requirements is hard to do analytically. Also, the control system is to be implemented as an embedded computer system which means that the controller will be translated into software. An analytical design method usually does not consider this. Safety-critical systems, such as a jet-engine thrust controller, requires extensive testing in simulation environments to prove the system meets requirements and the software's safety. The proposed methodology, for the first time, uses high-level control performance requirements as a formal input during the design and verification phases of the control scheme. The controller design is performed using the same structure as its final embedded software form. By the means of running extensive simulations the controller design is done, thus performing extensive testing at the same time as the design is taking place, increasing coverage analysis early in the process.

Figure 6.1 shows the proposed gain schedule control scheme, where the control action is generated using a PI controller (Equation 5.4). The schedule in charge of modifying the controller gains is determined by a combination of external inputs and the controlled variable.

With this configuration in mind and the proposed model checking framework, the control problem must then be portrayed in a model checking fashion so it can be addressed.

6.2.2 Schedule Design: A Model Checking Formulation

The control problem to solve is one of finding a set of controller gains for every operating region in order to drive the system to meet performance requirements. In a similar fashion as described in Chapter 4, Section 4.3, the control problem becomes a *reachability* problem. Given a set of possible controller tunings, is there a combination of current gains that make the system trajectories for all the operating regions fall within the given requirements? If a new set of gains is required then it will be generated using the methodology presented in Chapter 5 thus completing the design and verification framework.

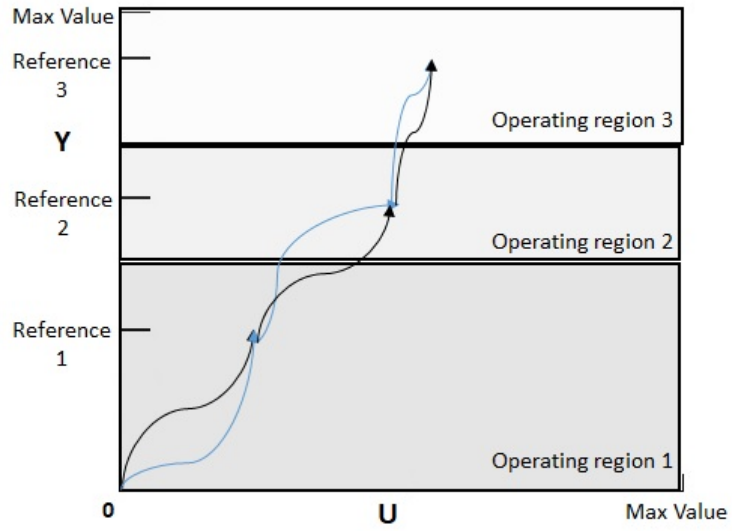

Fig. 6.2 Gain scheduled control problem portrayed as a *reachability* problem. Three different operating regions are presented. Each region has a different dynamic behaviour. Trajectories between reference points are given by the amount of possible schedule entries. In this case and for this example purpose only 2 possible trajectories are shown. The number of schedule entries will be generated as required.

Figure 6.2 shows how the gain schedule problem is presented as a *reachability* problem. In this example the operating space is partitioned into three different operating regions. Each operating region has different dynamics. Three different reference points are shown, one per operating region. How to move between those reference points is given by the selected controller tuning and each controller tuning is linked to a particular trajectory. For example purposes two different sets of trajectories are shown, meaning there are two possible controller tunings available.

Chapter 5 presented a novel model checking approach to perform PID controller tuning. The methodology from Chapter 5 is a push-button approach to the tuning problem where the solution is found by portraying the controller tuning problem as a model checking *reachability* problem. Following the same line of reasoning, in order to solve the gain schedule design problem, the problem is translated into a model checking *reachability* problem. To accomplish this the current timed-automata framework needs to be updated to include these features, the following section addresses the required updates.

6.3 Schedule Synthesis Methodology

In order to construct the control schedule containing the PI controller gains with the automata framework from Chapter 4 the following capabilities have to be added:

1. To have different system dynamics for the various operating regions.
2. To have different controller tunings and being able to switch between them.
3. Keep track of the operating region the system is operating on in order to change the system dynamics and the controller tuning if needed.

The first two items are addressed in their respective *Plant* and *Controller* automata. The dynamics for both the plant and the controller are included as arrays containing the coefficients. Instead of having a one-dimensional array with a single set of coefficients a multi-dimensional array is used instead. This multi-dimensional array contains the different sets of coefficients. To change the system dynamics or the controller dynamics means to point to a different entry in the array. Regarding item 3, because more than one operating region is to be included in the problem formulation, this means that the performance indicators have to be processed for all regions. This is done in the *Plant* automaton as explained in Section 4.3.2. In the same way that different dynamics are stored in multi-dimensional arrays, performance indicators will be stored in one-dimensional arrays instead of doing it in single-variable elements like in the original automaton (Section 4.3.2). Item 3 is the one that requires major changes in the *Observer* automaton, the following section explain such changes.

6.3.1 Timed-Automata Update

In order to capture the nature of the gain scheduling problem using the proposed modelling and abstraction methodology from Chapter 3, the automata must be able to change its dynamics depending on the operating region. Discrete SISO LTI models were selected to

portray the system dynamics (Section 3.2). In order to portray a non-linear dynamic system using linear models, a family of models are constructed to divide the non-linear problem into a subset of linear problems.

With this problem formulation, both the *Plant* and *Controller* automata need to be able to change its dynamics on demand. The *Observer* automaton is the one in charge of coordinating the execution of the controller and the plant, monitoring the simulation scenario. For this reason this automaton is where the updates to implement the required features are added in order to portray the schedule problem.

Observer

Figure 6.3 shows the automata in charge of coordinating the execution of the controller and the plant. This automata monitors both the controller output U and the process output Y to determine when to finalize the simulation. A new state has been added so that after the initialization process the tuning process takes place. This state will modify the initial K_P and K_I gains and calculate the new b_0 and b_1 coefficients for the controller.

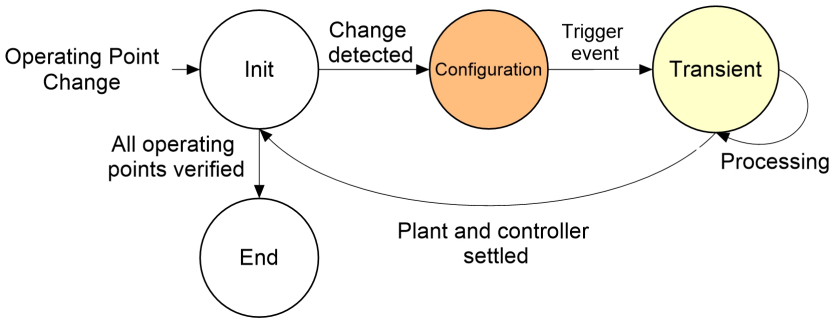

Fig. 6.3 Updated *Observer* automaton. Automaton in charge of controlling the data flow between the controller and the plant. This automaton keeps track of the changes in reference and operating regions and commands both the plant and controller automata to change their respective dynamics if required. This automata monitors the control signal U and output process signal Y to determine transitions between states.

The automaton consists of the following states:

1. *Init:* Initial state. Initialize the models for simulation. After initializing a transition to the *Configuration* state is taken. If all the required operating points have been processed and verified a transition to the *End* state is taken.

2. *Configuration:* Plant dynamics and controller tuning configuration state. After a change in operating point has been detected the automaton commands a change in plant dynamics if needed. If a change in controller configuration is required this

automaton commands the controller to do so. Among the possible controller tunings available the automaton non-deterministically chooses a tuning to control the process in that particular operating point. Once a change in dynamics and/or a controller tuning has been selected a transition to the *Transient* state is triggered.

3. *Transient:* Transitory state. This is the state in charge of coordinating the execution of the *Plant* and *Controller* automata. It monitors both the process output signal Y and the controller signal U to determine when the process has reached a settling condition. Once both the controller and the plant have settled a transition is triggered to the *Init* state.

4. *End:* Final state. After all the required operating points have been verified data processing comes to a halt and no more transitions are allowed in any automata.

The *Configuration* state and its functionalities have to be included in the UPPAAL implementation. The UPPAAL implementation with the updated features is shown in Figure 6.4.

Observer: UPPAAL Implementation

Figure 6.4 shows the automaton in charge of coordinating the correct execution of the controller and the plant. The automaton modifies the controller tuning by choosing among a pre-defined set of tunings in order to find a configuration which drives the process into a desired trajectory for every operating point. The automata modifies the system's dynamics when a change in operating region is detected. This automata monitors both the controller output U and the process output Y to determine when to change the operating region and to finalize the simulation. The coordination of the execution of the controllers and the plants is achieved by using synchronization channels C and D (one per system approximation). This automata sends a message via channels C and D to enable the execution of either the *Plant* or the *Controller* automata for both the over and under approximations.

The automaton consists of the following states:

1. *Init:* Initial state. Similar to *Init* state depicted in Figure 6.3. When an event is detected (change in reference) a transition to *OP1Tr* state is taken. During this transition the non-deterministic selection of the controller tuning is taken.

2. *OP1Tr:* Operating point 1 transient state. Operating point requirements verification. Once the process and controller have settled a transition to the *OP1SS* state is taken. Similar to *Transient* state depicted in Figure 6.3.

3. *OP1SS:* Operating point 1 steady state. This state functions as the *Init* state depicted in Figure 6.3 in the sense that the processing for a new operating point is performed here. Instead of returning to the initial state every operating point is split into transient and steady state sections. The reference signal is monitored and once a new event is detected (change in operating point) a transition to the next operating point transient state is taken. During this transition the non-deterministic selection of the controller tuning is taken.

4. *End:* Final state. Similar to *End* state depicted in Figure 6.3. After the last event (change in operating point) has been processed, if an equilibrium condition is reached the dynamics process comes to a halt and no more transitions are allowed in any automata.

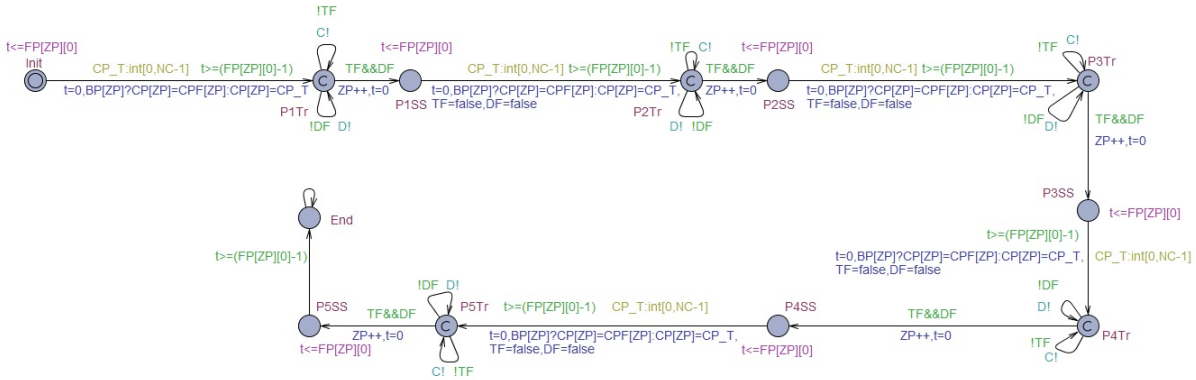

Fig. 6.4 UPPAAL implementation of the observer automaton. Automaton in charge of controlling the data flow between the controller and the plant. The processing of different operating regions, their respective dynamics, and different controller tunings has been incorporated. Every time a change in the operating point is detected the controller tuning can be non-deterministically selected. This automaton monitors the control signal U, and output process signal Y to determine transitions between states. It coordinates the execution between plants and controllers using communication channels C and D. One channel is used for the over approximation and the other one for the under approximation. This particular automaton processes 5 different operating points.

The automaton in Figure 6.4 is constructed to process 5 different operating points. This automaton is particular to this example where 5 different operating points are processed but it can easily be extended. Each operating point consists of a *transient state* and a *steady state*. Their functionality is similar in terms of transitions. Depending on the problem formulation more or less operating points can be included in the automaton. From a model checking point of view, the schedule *synthesis* process is similar to the previously presented PI *synthesis*

process in Section 5.3: the problem can be systematically addressed as a *reachability* problem. This automaton keeps track of which controller tuning is used in every operating point, if requirements are met in every operating point the automaton will contain the control schedule as part of its internal variables. With these new features included in the automaton the gain schedule design can take place in a formal manner. The following section explains the required queries to be used in the push-button approach within the model checker in order to fully enable the gain schedule formal design process.

6.3.2 Requirements Formulation for Design

The controller gain schedule design is a trajectory search problem as portrayed in Figure 6.2 and can be formulated in the following way: *is there a combination of controller tunings that can drive the system closed loop behaviour to meet the closed loop requirements for every operating point?* The verification of the closed loop requirements for each operating point is done in the same way as explained in Section 4.4: a trajectory search like the one in Figure 4.6.

The gain schedule design procedure is thus performed using a push-button approach by querying the model checker using a *reachability* property (Table 4.1) like Equation 4.1. The automata design splits the problem into each operating region performing the requirements verification independently as in the *verification* phase from Chapter 4. Therefore the query must contain the verification of all the operating points involved in the design problem. As mentioned in Section 6.3.1 the plant automaton will keep track of the requirements in an array form instead of single elements so that the query can be done using the locations in the array assigned for each operating point. The reachability query takes the following form:

$$
\begin{aligned}
&E <> Observer.End \quad \text{and} \\
&Plant.Overshoot[1] \leq Requirement \quad \text{and} \quad Plant.RiseTime[1] \leq Requirement \quad \text{and} \\
&Plant.SettlingTime[1] \leq Requirement \quad \text{and} \quad Plant.SSError[1] \leq Requirement \quad \text{and} \\
&Plant.Overshoot[2] \leq Requirement \quad \text{and} \quad Plant.RiseTime[2] \leq Requirement \quad \text{and} \\
&Plant.SettlingTime[2] \leq Requirement \quad \text{and} \quad Plant.SSError[2] \leq Requirement \quad \text{and} \quad \ldots \\
&Plant.Overshoot[n] \leq Requirement \quad \text{and} \quad Plant.RiseTime[n] \leq Requirement \quad \text{and} \\
&Plant.SettlingTime[n] \leq Requirement \quad \text{and} \quad Plant.SSError[n] \leq Requirement \qquad (6.1)
\end{aligned}
$$

Equation 6.1 is a *reachability* query which checks for all 4 requirements to be within limits for n operating points. The variables related to each individual operating region are

stored in arrays, hence the indexing to access them. In this format the verification can be done for particular operating points individually. If the property verification is successful the model checker will return *Pass* as a result and the *witness* trace to show how this result can be obtained. As part of the *witness* trace the model checker will return the schedule that drives the system to meet requirements, providing a solution to the control problem. If the property verification is not successful the model checker will simply return *Fail* without a trace. This means that within the available controller tunings there is not a combination that satisfies requirements for all operating points. In the *Fail* scenario in order to obtain information from the model checker the path qualifier must be changed to *A* instead of *E* to use a *Liveness* query so that a *counter-example* trace is generated.

Using the model checking framework presented in Chapter 5 to conduct the formal controller tuning procedure, and the model checking framework presented in this section, a novel methodology to formally design and verify a gain schedule control scheme is proposed. The following section explains how to use the full framework in a systematic way in order to find a solution to the gain schedule design problem.

6.3.3 Schedule Design Algorithm

The automata framework presented in Chapter 5 is exclusively designed to address the PI controller tuning problem. The automata framework from Section 6.3.1 addresses the gain schedule generation but does not solve the associated PI controller tuning problem. Using both sets of automata the full problem of the gain schedule design can be addressed in a formal manner with a model checking approach. The gain schedule generation could potentially use a different controller, or a different PID-type controller structure as well, this is not restricted to a PI controller design. In order to do so the associated controller automata, in both the controller tuning and schedule design formulations, would have to be adapted to deal with the new controller structure.

Figure 6.5 shows a block diagram of the proposed gain schedule formal design methodology using model checking. The model checking framework consists of the abstraction methodology, the PI controller tuning automata, and the gain schedule design automata. The abstraction methodology enables both sets of automata. The methodology takes a high fidelity model and high level requirements as inputs. The output is a formally designed gain schedule PI controller that meets the desired requirements. If requirements cannot possible be met the methodology will say why is this happening. The procedure for how to approach the design problem using the framework is presented as follows.

The objective of this design process is to generate a control schedule with the minimum necessary control tunings in order to meet requirements for all operating points. The schedule

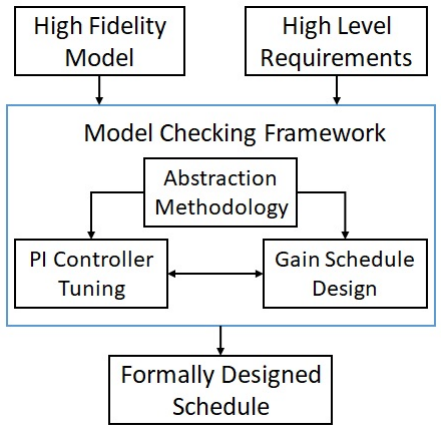

Fig. 6.5 Gain schedule formal design methodology. High fidelity model and high level requirements are taken into the model checking framework. The model checking framework is composed of the abstraction methodology which enables the formal PI controller tuning and the gain schedule design. Both the controller tuning and the gain schedule design automata are used in combination to solve the gain schedule design problem.

design is incremental. After the operating space has been split, an operating region is arbitrarily selected and the controller is tuned to meet requirements in that region. The model checker is then used to verify if requirements are met for all the other regions as well. If not, another region where requirements are not met is selected and the process is repeated. The number of available tunings increases and this also allows the model checker to use those tunings as options in order to meet requirements for the remainder of the operating points. The outcome after applying Algorithm 5 will be a control schedule which meets high level performance requirements and has been formally designed and verified. Step 10 in Algorithm 5 is added as a test coverage measure. Increasing coverage is a benefit from model checking and it is worth showing how this can be applied with the current design and verification framework. The following section presents the problem under analysis and how to use the proposed methodology to solve the control schedule design problem.

6.4 Case Study: Thrust Control Schedule Design

As an example to illustrate the proposed methodology, an aerospace related application is selected: gain scheduling jet-engine control. However, the methodology is not restricted to this type of application and can be applied to a generic embedded gain scheduling control approach.

Gain scheduling control is widely used due to the fact that it is relatively simple to implement, making it the usual control choice for commercial jet-engines. The implementation is

Algorithm 5: Gain schedule formal design methodology using model checking.

Input : High fidelity model, performance requirements.
Output : Formally verified control schedule.

1 Partition the operating space into M regions.
2 Generate a SISO LTI model for each of the M operating regions (3.2).
3 Use classical control methods (e.g. Ziegler-Nichols) for tuning the controller for operating region 1.
4 Use the abstraction methodology (Chapter 3) and generate the system abstraction for all the operating regions and the controller.
5 Use the automata framework (Section 6.3.1) to verify requirements for all M operating regions using the available controller tunings.
6 If requirements are met for all regions, cross-check in the original model.
7 If a region fails to meet requirements, use the PI controller tuning automata (Chapter 5, Algorithm 4) to find a suitable tuning for the operating region that fails. Use the last entry in the available controller tunings as a starting point for Algorithm 4.
8 Update gain schedule automata framework with the newly designed controller tuning.
9 Go back to step 5, repeat.
10 Perform an extensive verification with the final controller tunings and all the operating regions.

usually a computer-based look-up table which gives the controller parameters for a given set of environment measurements [94]. Jet-engine control is a complex mechanism because of the system restrictions to which the engine is subject to [135, 149]:

- Maximum fan speed.
- Maximum compressor speed.
- Maximum turbine temperature.
- Fan stall.
- Compressor stall.
- Maximum compressor discharge pressure.
- Minimum compressor discharge pressure.
- Lean burner blowout.
- Rich burner blowout.

The complexity of a jet-engine system requires a control approach which is also complex but one that can be broken-down into less-complex problems. Gain scheduling control meets the criteria and because it has been around for over 50 years is a well known control approach for avionics applications [97, 138]. Nonetheless the control software is also complex due to the fact that several functionalities reside in the same computer system creating dependencies and interactions among software modules which do not necessarily exist in the physical system. Therefore, for safety reasons the control software undergoes extensive verification and validation practices, certification requirements are extremely rigorous when it comes to safety-critical airborne software. For this reason a better approach to certification is desirable. The proposed model checking framework in this thesis fits well within this problem formulation and can aid in the design and verification process for such a critical piece of software. The applicability of the proposed gain schedule formal design methodology is presented in this section. A jet-engine thrust control gain schedule control design problem is used for this purpose.

6.4.1 Problem Formulation and Requirements

Consider a commercial jet-engine where generated thrust is regulated using a discrete PI controller with a gain scheduling scheme such as the one in Figure 6.1. The process dynamics will vary depending on the operating point: factors such as altitude and temperature generate a non-linear behaviour [149]. Figure 6.6 shows a possible behaviour of the control system which, without loss of generality, in this case consists of five operating regions ($M = 5$ - Algorithm 5). The control design problem to solve is to find a controller schedule that drives the system to meet high level requirements in every operating region. This problem formulation provides a good framework to show the proposed methodology to systematically design and verify a PI gain schedule controller in a formal manner.

System Dynamics

Taking the high fidelity model as a starting point the operating space is split into five operating regions. This is an arbitrary choice in order to show the applicability of the methodology. A continuous SISO LTI model per operating region is generated using linear identification techniques [121]. Table 6.1 shows the open loop continuous transfer functions for each operating region. Using these models and the high level performance requirements, the system abstraction will be generated in order to implement it in the model checker.

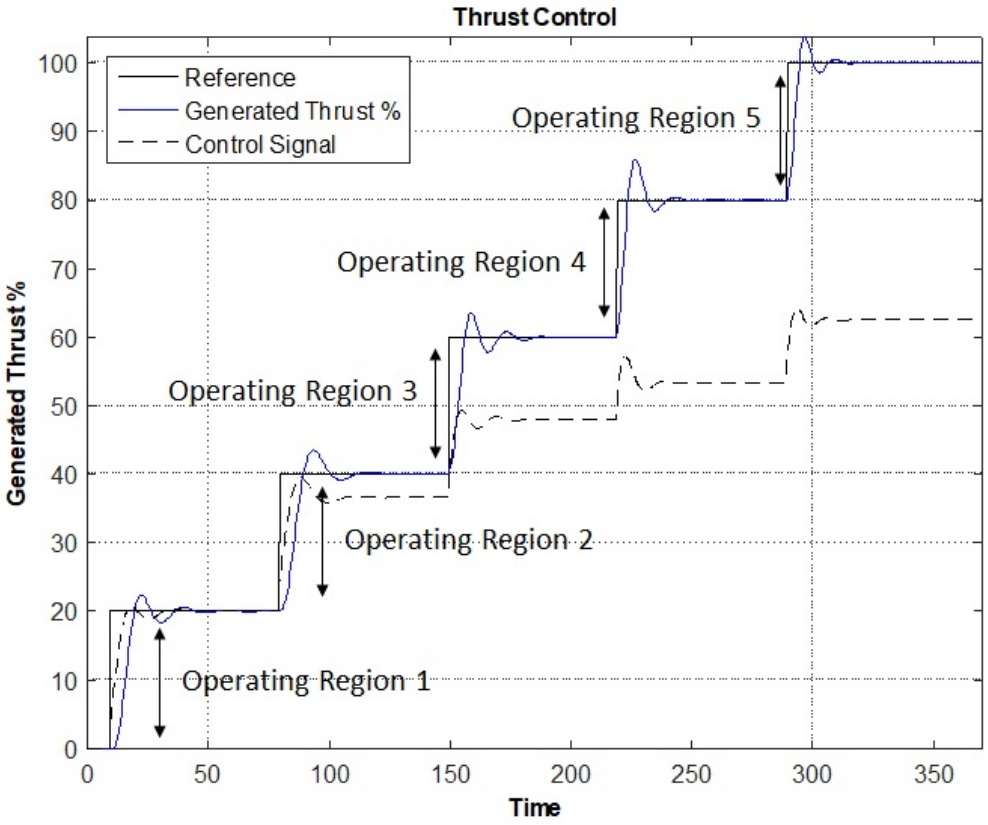

Fig. 6.6 Jet-engine thrust control. This is a possible behaviour of the control system consisting of five operating regions. Each operating region has a particular dynamic. In this scenario the same controller tuning is used for all the operating regions.

Table 6.1 Open loop continuous SISO LTI models for operating regions in the thrust control problem.

Region	Transfer Function
1	$G_{R1}(s) = \frac{Y(s)}{U(s)} = \frac{0.45^2 e^{-s}}{s^2+0.45s+0.45^2}$
2	$G_{R2}(s) = \frac{Y(s)}{U(s)} = \frac{(1.1)(0.45^2)e^{-s}}{s^2+(1.4)(0.45)s+0.45^2}$
3	$G_{R3}(s) = \frac{Y(s)}{U(s)} = \frac{(1.25)(0.55^2)e^{-s}}{s^2+0.55s+0.55^2}$
4	$G_{R4}(s) = \frac{Y(s)}{U(s)} = \frac{(1.5)(0.65^2)e^{-s}}{s^2+(1.4)(0.65)s+0.65^2}$
5	$G_{R5}(s) = \frac{Y(s)}{U(s)} = \frac{(1.6)(0.75^2)e^{-s}}{s^2+(1.2)(0.75)s+0.75^2}$

Requirements

The high level performance requirements for the control system, particularly selected to show the applicability of the methodology, are:

1. Regions 1 and 2: Maximum Overshoot % (OS) $\leq$ 13%.
2. Regions 3 - 5: Maximum Overshoot % (OS) $\leq$ 20%.
3. Settling Time (ST) $\leq$ 40 seconds.
4. Rise Time (RT) $\leq$ 15 seconds.
5. Steady state error % (SSE) $\leq$ 1%.

Depending on the system under analysis and the partition of the operating space, it is possible to have different requirements for different regions. In a complex system, like a jet-engine, there exists many constraints from which the control requirements are derived. These circumstances may allow to have different performance requirements for different operating regions. In this sense and for demonstrating purposes, different overshoot requirements were selected for different operating regions. This requirements relaxation could be a side effect of the number of design points for the control scheme: the system spends most of its lifetime in a certain operating point resulting in stricter requirements for that particular operating point.

Before generating the abstraction models for the over and under approximations, the configuration parameters for the abstraction must be selected.

Abstraction Configuration

The abstraction configuration parameters are:

- Original system transfer functions are given in Table 6.1.
- Sampling time T = 0.5 seconds.
- Parametric compensation gain $K_U = 0.0005$.
- Data-type representation format: 1 integer digit (I = 1) and 4 fractional digits (F = 4).
- Input-Output scaling gain K_S = 10,000.
- Coefficients scaling gain K_{ab} = 16,384.
- Model checker *Plant* state machines = 2 - under and over approximations.
- Model checker *Controller* state machines = 2 - under and over approximations.
- Model checker *Observer* state machines = 1.

Using the selected sampling time the discrete models are generated. Table 6.2 shows the discrete SISO LTI transfer functions from which the abstraction is generated. After applying the abstraction methodology, both over and under approximations are generated for each operating region. Table 6.3 shows the abstraction transfer functions using the ad hoc data-type representation which uses integer-data only. The discrete PI controller initial configuration

Table 6.2 Open loop discrete SISO LTI models for operating regions in the thrust control problem. Column 2 uses floating-point representation and column 3 the selected fixed-point representation.

Region	Transfer Function - Floating-Point	Transfer Function - Fixed-Point
1	$G_{R1}(z) = \frac{0.02342z+0.02172}{z^2-1.75337z+0.79851}z^{-2}$	$G_{R1_{fp}}(z) = \frac{0.0234z+0.0217}{z^2-1.7534z+0.7985}z^{-2}$
2	$G_{R2}(z) = \frac{0.02503z+0.02254}{z^2-1.68654z+0.72978}z^{-2}$	$G_{R2_{fp}}(z) = \frac{0.0250z+0.0225}{z^2-1.6865z+0.7298}z^{-2}$
3	$G_{R3}(z) = \frac{0.04294z+0.03917}{z^2-1.69386z+0.75957}z^{-2}$	$G_{R3_{fp}}(z) = \frac{0.0429z+0.0392}{z^2-1.6939z+0.7596}z^{-2}$
4	$G_{R4}(z) = \frac{0.06787z+0.058304}{z^2-1.55032z+0.63444}z^{-2}$	$G_{R4fp}(z) = \frac{0.0679z+0.0583}{z^2-1.5503z+0.6344}z^{-2}$
5	$G_{R5}(z) = \frac{0.09626z+0.08282}{z^2-1.525703z+0.63762}z^{-2}$	$G_{R5fp}(z) = \frac{0.0963z+0.0828}{z^2-1.5257z+0.6376}z^{-2}$

Table 6.3 System abstraction: integer-only discrete SISO LTI models for operating regions in the thrust control problem. Two models are generated per region: under approximation and over approximation.

Region	Under Approximation	Over Approximation
1	$G_{R1_{Under}}(z) = \frac{383z+356}{16384z^2-28710z+13079}z^{-2}$	$G_{R1_{Over}}(z) = \frac{387z+359}{16384z^2-28744z+13089}z^{-2}$
2	$G_{R2_{Under}}(z) = \frac{410z+367}{16384z^2-27615z+11954}z^{-2}$	$G_{R2_{Over}}(z) = \frac{413z+372}{16384z^2-27650z+11964}z^{-2}$
3	$G_{R3_{Under}}(z) = \frac{703z+642}{16384z^2-27738z+12442}z^{-2}$	$G_{R3_{Over}}(z) = \frac{708z+646}{16384z^2-27769z+12452}z^{-2}$
4	$G_{R4_{Under}}(z) = \frac{1111z+954}{16384z^2-25389z+10392}z^{-2}$	$G_{R4_{Over}}(z) = \frac{1116z+958}{16384z^2-25416z+10401}z^{-2}$
5	$G_{R5_{Under}}(z) = \frac{1576z+1357}{16384z^2-24981z+110445}z^{-2}$	$G_{R5_{Over}}(z) = \frac{1578z+1362}{16384z^2-25013z+10451}z^{-2}$

is given by:

- Controller discrete transfer function is given by Equation 5.4
- Initial proportional gain $K_P = 0.1862$.
- Initial integral gain $K_I = 0.1496$.

Using the selected data type representation and the abstraction configuration parameters, the PI controller transfer function is given by:

$$G_{P_{PI}}(z) = \frac{4276z - 3051}{16384z - 16384} \tag{6.2}$$

The problem which was solved in Chapter 5 is the one regarding the first operating region in the gain schedule scenario in this section. It is then expected that using the obtained PI tuning in Chapter 5 for region 1 meets requirements. From this baseline, Algorithm 5 is applied. The requirements verification is thus performed using a push-button approach by querying the model checker using either a *reachability* or a *liveness* property (Table 4.1). The reachability property is used to generate a *witness* trace in case requirements are met and the *liveness* property to generate a *counter-example* trace in case requirements are not met. The *liveness* property is also used to conduct the exhaustive verification at the end of Algorithm 5 to increase coverage.

$$
\begin{aligned}
E <> \mathit{Observer.End} \quad & \text{and} \\
& \mathit{Plant\_OA.OS}[1] \leq 13\% \quad \text{and} \quad \mathit{Plant\_OA.ST}[1] \leq 40 \quad \text{(seconds) and} \\
& \mathit{Plant\_OA.RT}[1] \leq 15 \quad \text{(seconds) and} \quad \mathit{Plant\_OA.SSE}[1] \leq 1\% \quad \text{and} \\
& \mathit{Plant\_UA.OS}[1] \leq 13\% \quad \text{and} \quad \mathit{Plant\_UA.ST}[1] \leq 40 \quad \text{(seconds) and} \\
& \quad \mathit{Plant\_UA.RT}[1] \leq 15 \quad \text{(seconds) and} \quad \mathit{Plant\_UA.SSE}[1] \leq 1\% \\
& \qquad \vdots \\
& \mathit{Plant\_OA.OS}[5] \leq 20\% \quad \text{and} \quad \mathit{Plant\_OA.ST}[5] \leq 40 \quad \text{(seconds) and} \\
& \mathit{Plant\_OA.RT}[5] \leq 15 \quad \text{(seconds) and} \quad \mathit{Plant\_OA.SSE}[5] \leq 1\% \quad \text{and} \\
& \mathit{Plant\_UA.OS}[5] \leq 20\% \quad \text{and} \quad \mathit{Plant\_UA.ST}[5] \leq 40 \quad \text{(seconds) and} \\
& \quad \mathit{Plant\_UA.RT}[5] \leq 15 \quad \text{(seconds) and} \quad \mathit{Plant\_UA.SSE}[5] \leq 1\% \qquad (6.3)
\end{aligned}
$$

Equation 6.3 shows an extract of the query used in the schedule design problem. Requirements must be verified for all 5 operating regions in both the under approximation and the over approximation. Requirements are indexed 1-5 referring to each operating region. The actual query contains the verification for all 5 operating regions (index 1 through 5).

6.4.2 Results and Discussion

Algorithm 5 was applied to solve the problem and generate a schedule that drives the system to meet requirements in every operating region. The starting point is the result of tuning a PI controller for region 1 in Chapter 5. As expected region 1 already meets requirements. The first iteration of the algorithm is to verify the rest of the operating regions with the initial controller configuration.

Iteration 1

After using the model checker, it is found that the initial configuration fails to meet requirements in regions 2 and 4. This is confirmed with the original model. Table 6.4 shows the performance indicators for both the abstraction and the original model using the initial controller tuning. As expected region 1 meets requirements in both the abstraction and the original model since the initial controller tuning was particularly generated for this region.

Table 6.4 Performance indicators for the system abstraction and the original model using the initial tuning only. The requirements that failed are highlighted in bold.

Region	Overshoot (%)	Settling Time (sec)	Rise Time (sec)	Steady State Error (%)
Under Approximation				
1	9.49	32	5.5	0.13
2	**14.74**	20	6	0.15
3	17.35	19	4	0.15
4	**29.17**	12	3.5	0.02
5	17.46	10.5	3.5	0.02
Over Approximation				
1	12.96	33.5	5.5	0.08
2	18.13	28	6	0.1
3	18.46	19.5	4.5	0.01
4	28.82	12.5	3.5	0.02
5	18.87	10.5	3.5	0.02
Original				
1	11.29	33	5.5	0.05
2	16.456	27	6	0.1
3	17.771	19.5	4.5	0.1
4	29.153	16.5	3.5	0.1
5	18.399	10.5	3.5	0.1

Figure 6.7 shows the abstraction behaviour using the initial controller tuning. Figure 6.8 shows the original system behaviour using the initial controller tuning. Because both regions 2 and 4 failed to meet requirements, a new iteration is required. Therefore region 2 is selected to apply Algorithm 4 and find a new tuning that meets requirements. The new set of gains are $K_P = 0.2778$ and $K_I = 0.1496$. Only the proportional gain was modified from its original value. Table 6.5 shows the performance indicators for operating region 2 with the new set of gains for both the abstraction and the original model. With this new tuning the verification can take place again and a new iteration within Algorithm 5 is performed.

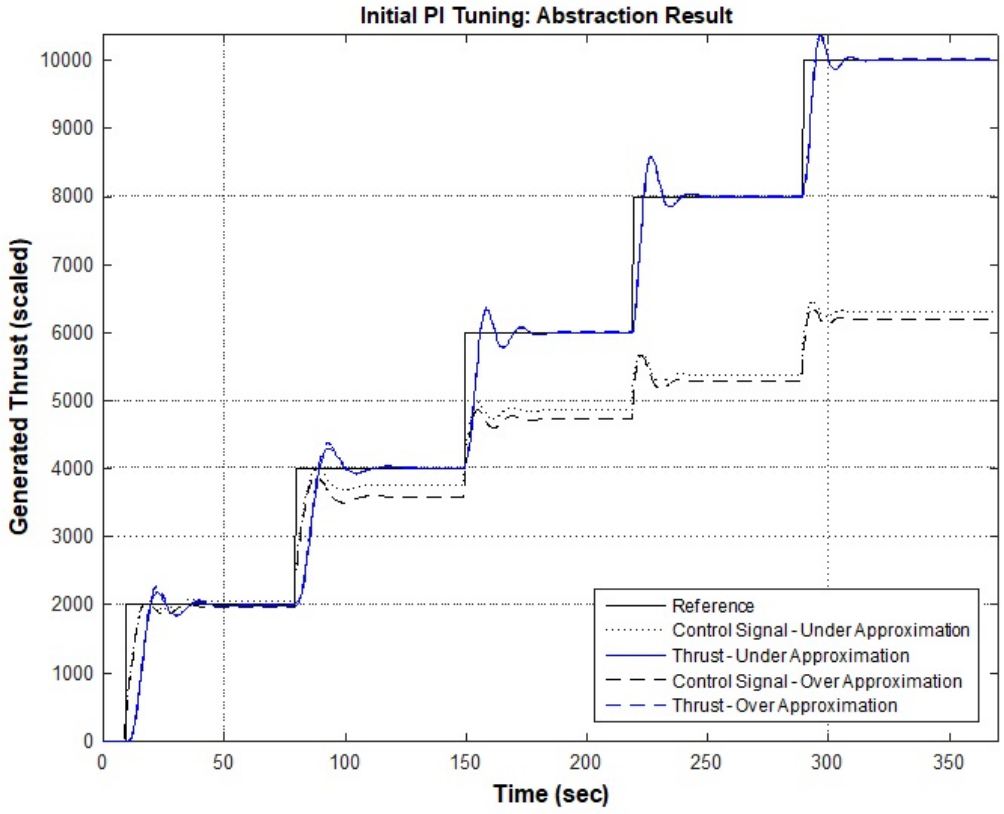

Fig. 6.7 Closed loop control system behaviour for the abstraction. Only one control tuning is available and used for the five operating regions. The performance indicators are listed in Table 6.4. The graph shows both the over and under approximations with their respective control signals.

Table 6.5 Performance indicators for the system abstraction and the original model for operating region 2 using the set of gains obtained with Algorithm 4.

Region	Overshoot (%)	Settling Time (sec)	Rise Time (sec)	Steady State Error (%)
Under Approximation				
2	9.86	26.5	6	0.04
Over Approximation				
2	13.0	27.5	6	0.09
Original				
2	11.36	27	5	0.05

Iteration 2

Since the new set of gains correspond to region 2, the model checker must select this as the default choice for this region. At this point only region 4 does not meet requirements. The

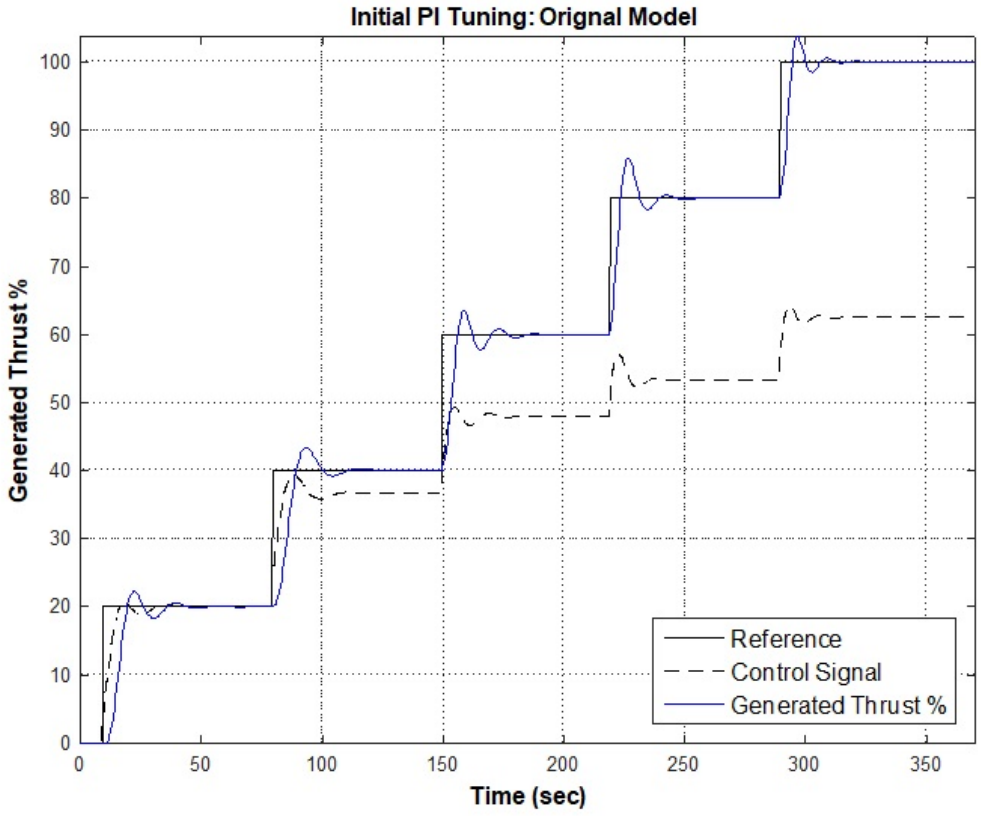

Fig. 6.8 Closed loop control system behaviour for the original system. Similar to Figure 6.7 only one control tuning is available and used for the five operating regions. The performance indicators are listed in Table 6.4. The graph shows generated thrust % and the control signal.

new set of gains are added to the search to see if a solution that includes region 4 is found. Table 6.6 shows the performance indicators for both the abstraction and the original model after performing a new search. The model checker fails to find a suitable combination which means that tuning 2 does not drive region 4 into meeting requirements.

Figure 6.9 shows the abstraction behaviour using the generated schedule consisting of two controller tunings. Figure 6.10 shows the original system behaviour using the generated schedule. Region 4 is still not compliant with requirements so Algorithm 4 is applied in order to find a new controller tuning. The new set of gains are $K_P = 0.1618$ and $K_I = 0.1191$. In this case both gains needed modification to drive the system to meet requirements. Table 6.7 shows the performance indicators for operating region 4 with the new set of gains for both the abstraction and the original model.

Table 6.6 Performance indicators for the system abstraction and the original model after 1 iteration. Two controller tunings are available for use. The requirements that failed are highlighted in bold.

Region	Tuning	Overshoot (%)	Settling Time (sec)	Rise Time (sec)	Steady State Error (%)
Under Approximation					
1	1	9.49	32	5.5	0.13
2	2	9.86	26.5	6	0.04
3	1	17.35	19	4	0.15
4	1	**29.17**	12	3.5	0.02
5	1	17.46	10.5	3.5	0.02
Over Approximation					
1	1	12.96	33.5	5.5	0.08
2	2	13.0	27.5	6	0.09
3	1	18.46	19.5	4.5	0.01
4	1	**28.82**	12.5	3.5	0.02
5	1	18.87	10.5	3.5	0.02
Original					
1	1	11.29	33	5.5	0.05
2	2	11.36	27	5	0.05
3	1	17.771	19.5	4.5	0.1
4	1	**29.153**	16.5	3.5	0.1
5	1	18.399	10.5	3.5	0.1

Table 6.7 Performance indicators for the system abstraction and the original model for operating region 4 using the set of gains obtained with Algorithm 4.

Region	Overshoot (%)	Settling Time (sec)	Rise Time (sec)	Steady State Error (%)
Under Approximation				
4	17.95	11.5	4	0.03
Over Approximation				
4	17.55	12	4	0.08
Original				
4	18.197	12	4	0.08

Iteration 3 - Final Schedule

The new controller tuning is added to the available options. Controllers have been designed for regions 2 and 4 and regions 1, 3, and 5 met requirements with the initial tuning. The

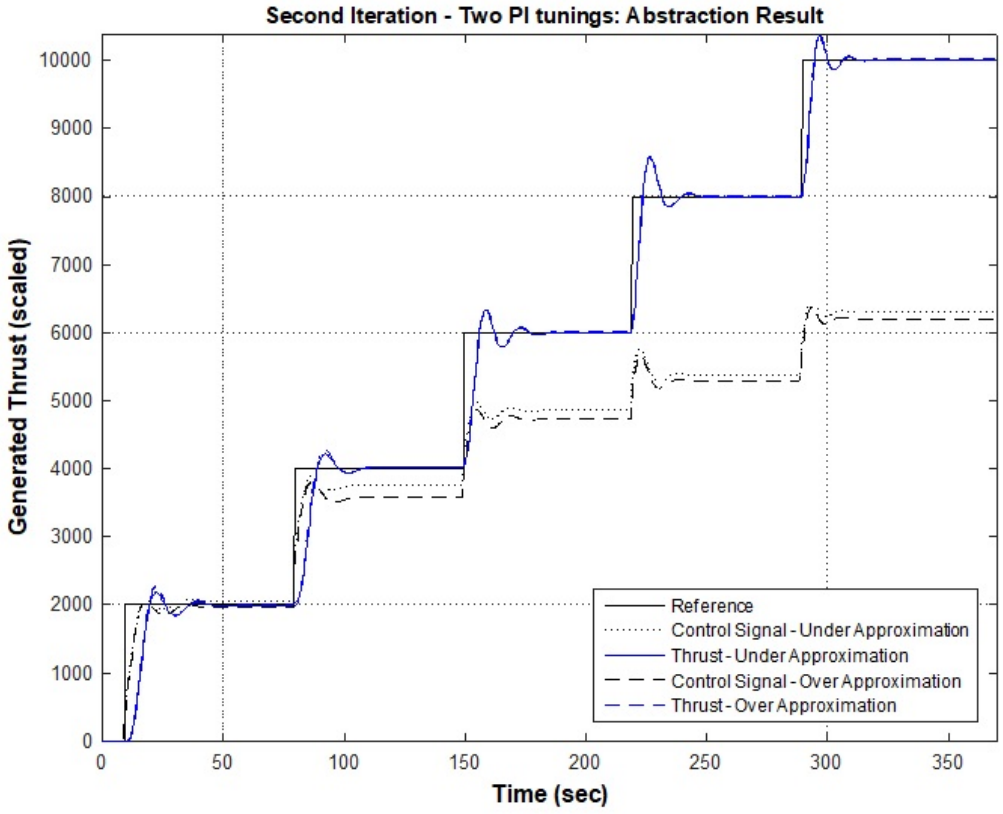

Fig. 6.9 Closed loop control system behaviour for the abstraction. Two control tunings are available. Tuning 1 is used for regions 1, 3, 4, and 5. Tuning 2 is used for region 2. The performance indicators are listed in Table 6.6. The graph shows both the over and under approximations with their respective control signals.

model checker must therefore provide this answer. After running the model checker this is confirmed and the final schedule is generated. This is confirmed using the original model. Table 6.8 shows the final schedule indicating which controller must be used in which region in order to meet requirements. Performance indicators for all the regions are shown. Figure 6.11 shows the abstraction behaviour using the final schedule consisting of three controller tunings. Figure 6.12 shows the original system behaviour using the generated schedule. A comparison is performed between the initial tuning and the final schedule. Figure 6.13 shows the behaviour of the original system using both the initial tuning (tuning 1) and the final schedule. As expected, because controller tunings were designed for those particular regions, regions 2 and 4 show a different behaviour whilst regions 1, 3, and 5 show the same behaviour.

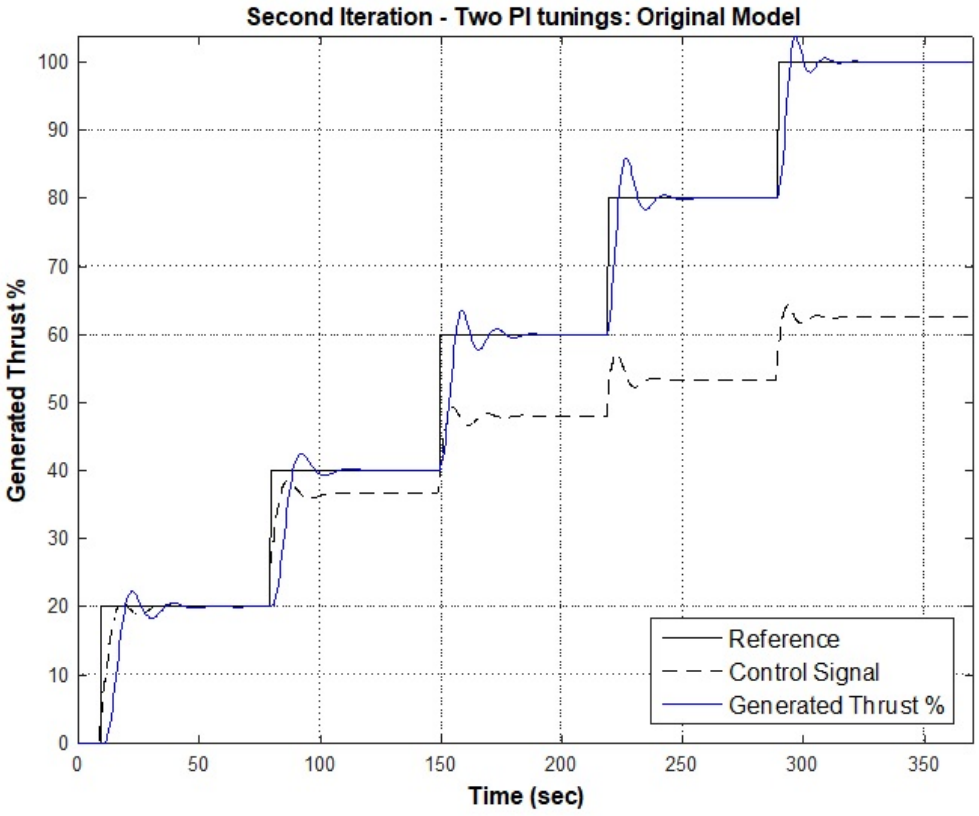

Fig. 6.10 Closed loop control system behaviour for the original system. Similar to Figure 6.9 two control tunings are available. Tuning 1 is used for regions 1, 3, 4, and 5. Tuning 2 is used for region 2. The performance indicators are listed in Table 6.6. The graph shows generated thrust % and the control signal.

Coverage Verification

The final step in Algorithm 5 is to perform a coverage verification using the final schedule. This step could be considered a sanity check: verify that if using any of the available tunings could drive the system out of requirements. Even if the tuning was not designed for a particular operating region it is worth performing this verification. The reason being that coverage is increased from a software point of view: more paths are executed. Also, the control software resides in a computer system that is not just dedicated to control purposes. The final product is a complex software system and it could be the case that for some external reasons to the control system itself, a different tuning may be chosen for a certain operating region even it if is not supposed to be used in that case.

Table 6.8 Performance indicators for the system abstraction and the original model after 2 iterations: Final control schedule. Three controller tunings are available for use.

Region	Tuning	Overshoot (%)	Settling Time (sec)	Rise Time (sec)	Steady State Error (%)
Under Approximation					
1	1	9.49	32	5.5	0.13
2	2	9.86	26.5	6	0.04
3	1	17.35	19	4	0.15
4	3	17.95	11.5	4	0.03
5	1	17.46	10.5	3.5	0.02
Over Approximation					
1	1	12.96	33.5	5.5	0.08
2	2	13.0	27.5	6	0.09
3	1	18.46	19.5	4.5	0.01
4	3	17.55	12	4	0.08
5	1	18.87	10.5	3.5	0.02
Original					
1	1	11.29	33	5.5	0.05
2	2	11.36	27	5	0.05
3	1	17.771	19.5	4.5	0.1
4	3	18.197	12	4	0.08
5	1	18.399	10.5	3.5	0.1

The *liveness* query is used allowing the model checker to choose any control tuning for any operating region. If the verification is successful it means that all the controller tunings are capable of driving the system into meeting requirements. This is not true for this case because it is already known that tuning 1 is not suitable for regions 2 and 4, so it is expected that this verification will fail. However, as a result of this failure a *counter-example* trace is returned showing why this is not the case. Table 6.9 shows the results from performing this verification.

The *counter-example* trace shows that using tuning 2 in operating region 5 makes the system fail to meet requirements in that region. Also it shows that using tuning 3 in regions 1, 2, and 3 also drive the system to meet requirements. Figure 6.14 shows the abstraction behaviour using the *counter-example* trace tunings. Figure 6.15 shows the original system behaviour using the *counter-example* trace tunings.

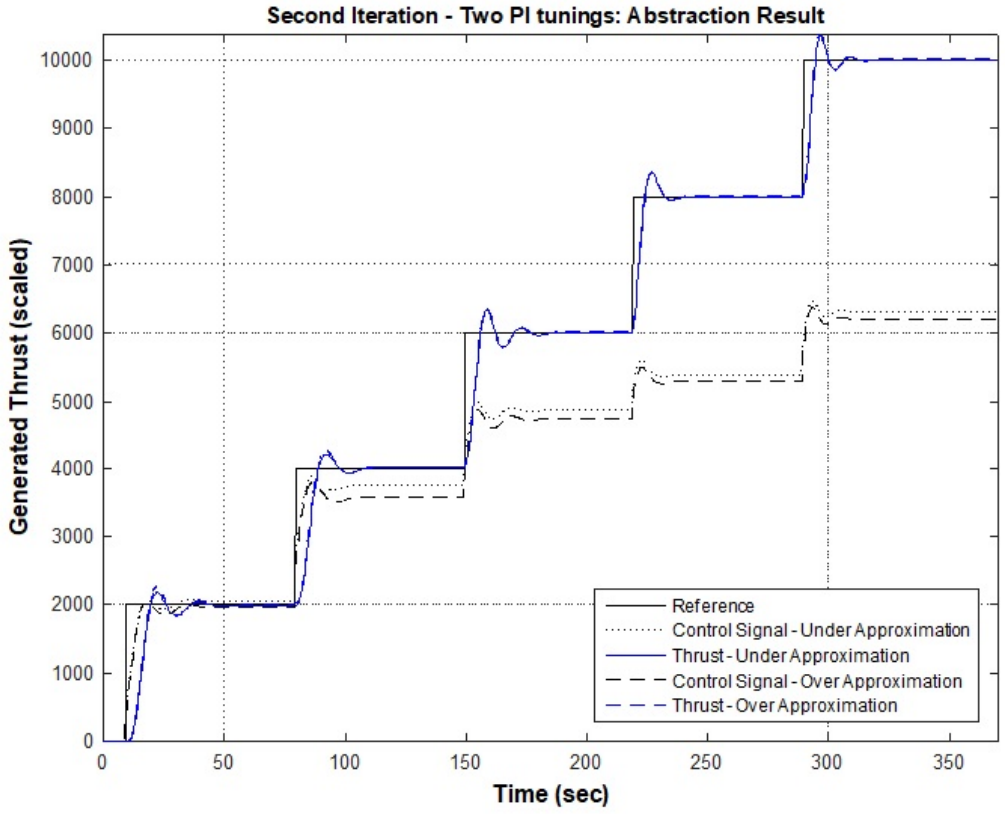

Fig. 6.11 Closed loop control system behaviour for the abstraction. Final schedule: Three control tunings are available. Tuning 1 is used for regions 1, 3, 4, and 5. Tuning 2 is used for region 2. Tuning 3 is used for region 4. The performance indicators are listed in Table 6.8. The graph shows both the over and under approximations with their respective control signals.

Discussion

The proposed novel methodology in this thesis delivers a control schedule that has been formally designed and verified with the aid of model checking, providing a working framework for the design and verification of safety-critical control software. High level performance requirements are taken as a formal input to the framework in order to design the control schedule. The timed-automata design is strongly driven by the high level requirements that the control problem requires. The verification of requirements is performed by the means of simulation using a well-known type of control models: discrete SISO LTI models. The abstraction methodology allows to use the same models as the original system in the model

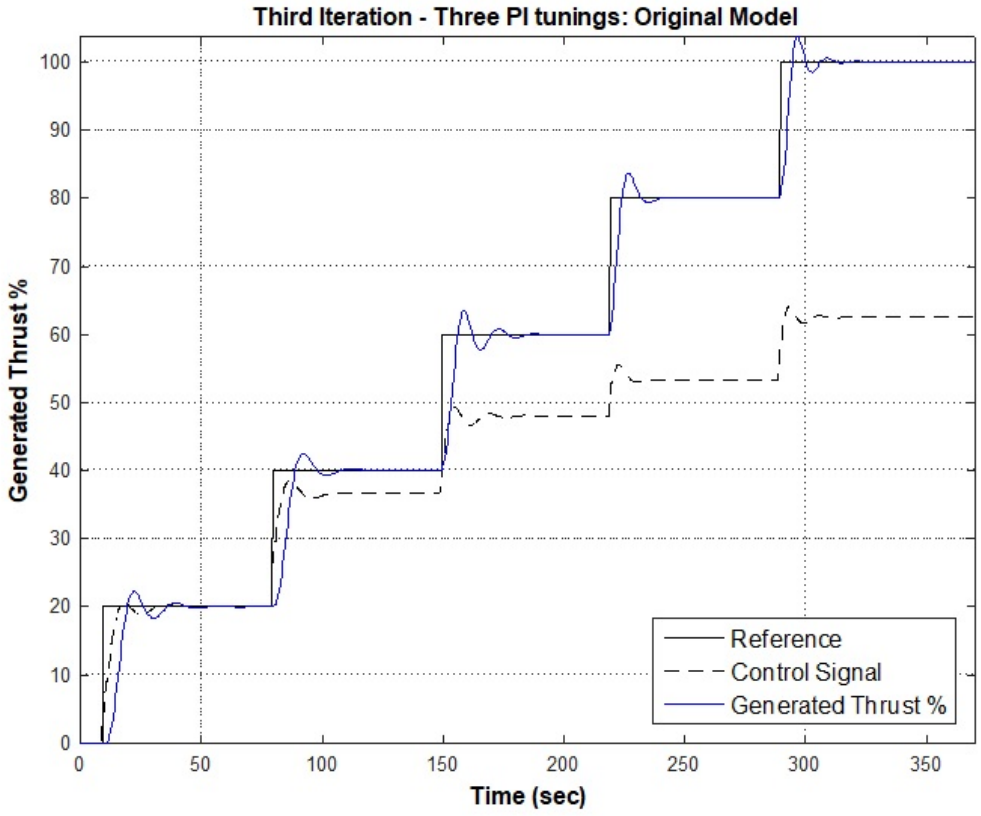

Fig. 6.12 Closed loop control system behaviour for the original system. Final schedule: similar to Figure 6.11 three control tunings are available. Tuning 1 is used for regions 1, 3, 4, and 5. Tuning 2 is used for region 2. Tuning 3 is used for region 4. The performance indicators are listed in Table 6.8. The graph shows generated thrust % and the control signal.

checker, if well a data-type restriction exists there is no need for changing operating space coordinates because the dynamics of the control system are recovered in the same domain.

The proposed abstraction methodology used for both the schedule design and the PI tuning problems provides good boundaries for the original system, allowing to reason about the original system with the abstraction in a sound way. Results are cross-checked with the original system confirming that the methodology is accurate in representing the original system behaviour. The behaviour of the original system is bounded in the abstraction by both the *over-approximation* and the *under-approximation*, which makes possible to infer properties about the original system using the model checker.

The model checking approach enables the use of formal methods to solve a control systems problem formulation which falls well within the current software development cycle for safety-critical embedded applications such as jet-engine thrust control. Coverage can

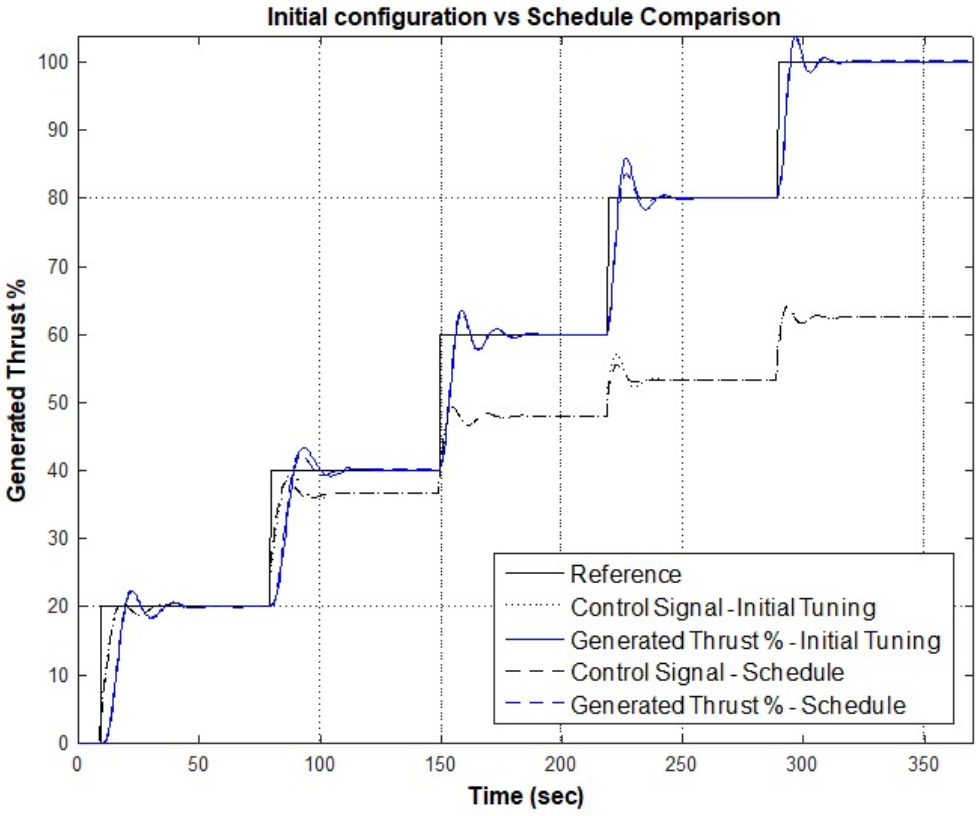

Fig. 6.13 Initial controller tuning versus final schedule. Final schedule: three control tunings are available. Tuning 1 is used for regions 1, 3, 4, and 5. Tuning 2 is used for region 2. Tuning 3 is used for region 4. The performance indicators for the final schedule are listed in Table 6.8. The performance indicators for the initial configuration are listed in Table 6.4. The graph shows generated thrust % and the control signal.

be benefited by the inclusion of this type of approach in the software development cycle by providing an increase by the means of exhaustive testing, also providing feedback when an scenario that drives the system out of requirements is found.

The proposed model checking framework uses a push-button type of approach for design and verification purposes in both the PI tuning and the gain schedule problem formulations, which in turn can help the software engineer to design a control schedule even if he is not familiar in control systems but knows model checking and formal methods. In a similar manner, the control systems engineer can benefit by the use of the novel methodology from a software design and verification point of view. Designing and verifying the compliance of a controller versus performance requirements and safety requirements nowadays involves the

Table 6.9 Performance indicators for the system abstraction and the original model performing the coverage verification. All three controller tunings from the final schedule are available for use. The requirements that failed are highlighted in bold.

Region	Tuning	Overshoot (%)	Settling Time (sec)	Rise Time (sec)	Steady State Error (%)
Under Approximation					
1	3	0	28	8	0.55
2	3	3.15	13	8	0.6
3	3	3.4	18	5	0.5
4	3	17.95	11.5	4	0.03
5	2	19.3	14	3	0.02
Over Approximation					
1	3	0.35	27.5	7.5	0.3
2	3	6.05	20	7.5	0.125
3	3	3.9	18.5	5	0.2
4	3	17.55	12	4	0.08
5	2	**20.55**	14	3	0.05
Original					
1	3	0	27	7.5	1.9
2	3	4.86	19	7.5	1.8
3	3	3.84	18.5	5	1.8
4	3	18.197	12	4	0.08
5	2	**20.15**	14	3	1.8

use and understanding of several software tools and practices, it is no longer a pure control design matter but one of software engineering as well.

The proposed novel methodology, by enabling the use of model checking in the design and verification processes of a well-known control system scheme, is a step towards closing the gap between safety-critical control software for airborne applications and the use of formal methods.

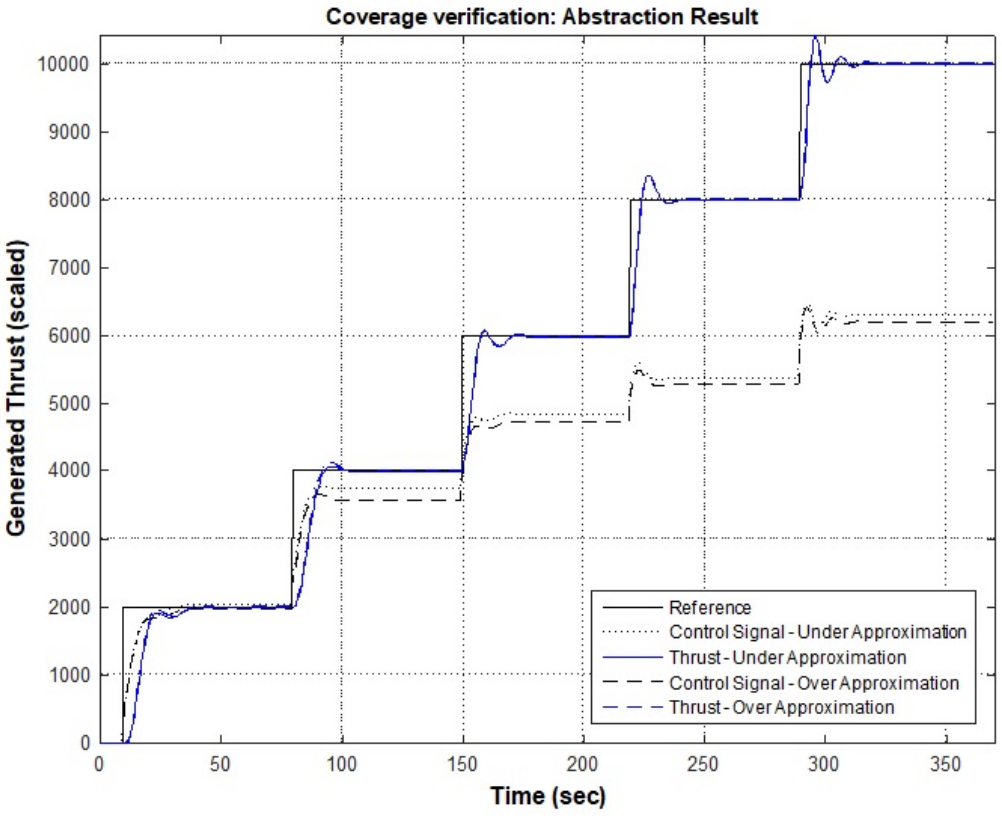

Fig. 6.14 Closed loop control system behaviour for the abstraction. *Counter-example* trace tunings were used in this case: Tuning 3 for regions 1, 2, 3, and 4. Tuning 2 for region 5. The performance indicators are listed in Table 6.9. The graph shows both the over and under approximations with their respective control signals.

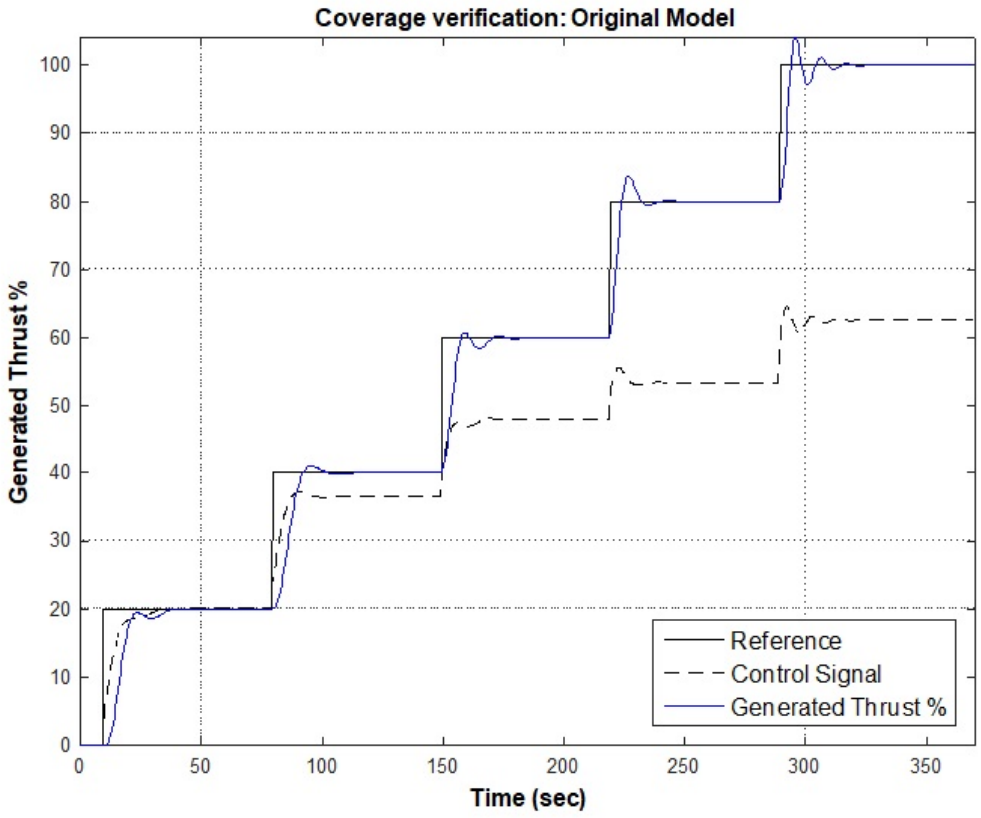

Fig. 6.15 Closed loop control system behaviour for the abstraction. *Counter-example* trace tunings were used in this case: Tuning 3 for regions 1, 2, 3, and 4. Tuning 2 for region 5. The performance indicators are listed in Table 6.9. The graph shows generated thrust % and the control signal.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The current development and verification processes of safety-critical control software use an *a posteriori* approach: the product is developed and tested iteratively until requirements are met. Exhaustive testing is required to ensure the safety of the product, which in turn takes a lot of effort and is a time consuming activity. The need for better, faster, and more reliable ways to verify control systems is impending: a new path to certification is needed. Formal methods and model checking can provide a new approach to development and verification, potentially benefiting the certification effort: an *a priori* approach. Potential applicabilities of formal methods for safety-critical systems have been shown in the past, but they usually deal with small scale problems, safety checks for the software itself or Boolean type of control. The applicability and feasibility of formal methods for gain scheduled control has yet to be addressed. In order to fully exploit the benefits of model checking, easy to follow practices have to be developed so that they can be easily integrated into the design and verification cycles.

The work presented in this thesis addressed the formal design and verification of a discrete PID gain scheduled control scheme, using an aerospace related example as a candidate application (e.g. jet-engine thrust control). A novel model checking design and verification framework was proposed to address the safety-critical control software problem. By using an *a priori* model checking framework, formality is introduced into the process for both requirements definition and system modelling. The methodology uses a known structure to represent the system dynamics: discrete SISO LTI. This being a common modelling structure makes the methodology easier to embrace by control engineers. Formal and easy to follow practices will help to develop safer and more reliable control systems. The proposed novel

methodology is a step towards closing the gap between safety-critical control software and the use of formal methods.

A dynamic system abstraction methodology was presented in Chapter 3. The purpose of the abstraction methodology is to recover the dynamic behaviour of a control system for its future implementation in a model checking environment. By omitting the necessity of floating-point data the abstraction can be implemented in a model checking environment using 16-bit signed integers only and in turn the *state-space explosion* problem is partially addressed by limiting the possible number of values that the system's inputs and outputs can take. The methodology compensates for modelling and rounding errors so that the original model behaviour can be properly captured. The final abstraction consists of a system *over-approximation* and a system *under-approximation* which provide bounds for the behaviour of the original model. The true behaviour is guaranteed to be bounded by the over and under approximations so that when using the abstraction in a model checking environment, the results can be used to infer properties about the original system.

The presented abstraction methodology allows to portray every element in a dynamic system independently; allowing to decouple the system components thus enabling the capability of analysing the controller and the plant independently. The closed loop behaviour can be then computed within the model checker so that the controller design can be addressed in a formal manner instead of redesigning the controller elsewhere. Because the control signal is independently generated, it can be analysed and manipulated if needed, which is impossible to do when the system is abstracted directly in its closed-loop form.

The translation of typical control system high-level performance requirements (e.g. maximum percentage overshoot, settling time, rise time, and steady state error) into a formal framework was demonstrated in the verification methodology in Chapter 4. Timed-automata were designed to target these type of requirements, enabling the formal verification using a temporal logic language (e.g. CTL). The automata in conjunction with the CTL properties used for verification enable the translation of high-level control performance requirements into a formal language. In this manner, the verification can be formulated as a *reachability* problem in model checking. A *design for verifiability* approach was used to construct the automata.

A formal PID controller tuning design and verification methodology was presented in Chapter 5. Using the abstraction methodology and the requirements verification methodology, the tuning problem was addressed as a *reachability* problem in model checking. To circumvent a possible *state-space problem* (e.g. the possible values for the controller gains can make the search space too big to handle) configurable upper and lower bounds for the gains are used. In this manner a systematic approach is used to find a possible set of gains

which solve the control problem. However, it is possible that the problem does not have a solution which the model checker will point out. The result after applying the methodology is a controller tuning that meets requirements and has been formally designed and verified, or that a controller is not achievable.

A formal gain schedule control design methodology was presented in Chapter 6.

By splitting the operating control space into several operating regions, the abstraction methodology allows to generate an abstraction per region.

Starting with an initial controller configuration (one single set of controller gains) the schedule is incrementally constructed until requirements are met for all regions. The schedule problem is addressed as a *reachability* problem in model checking. If requirements cannot be met for a given region the methodology points out which requirement fails. The result after applying the methodology is a control schedule which drives the system into meeting requirements and has been formally designed and verified. By addressing the design and verification problem in a formal environment such as a model checker, the intervention by the designer is greatly reduced because a *push-button* approach is taken. The controller tuning and the schedule design rely on this approach, where the designer can iterate over the algorithms making minimum adjustments when performing either the controller tuning or the schedule generation. This in turn reduces the required amount of expertise in formal methods and model checking from the designer.

The work in this thesis was presented and demonstrated using a safety-critical aerospace application: a jet-engine thrust control system. The design and verification of such a system currently relies mainly on extensive testing activities. In this respect and by the means of the proposed formal approach, the design and verification processes can be benefited by the inclusion of the methodology into the product design cycle. Potential benefits include the increase in testing coverage, earlier error detection, and an increase in safety because the controller is designed with a *correct-by-construction* approach.

7.2 Future Work

The scalability of the methodology needs to be evaluated. It was demonstrated that the methodology allows to address a gain schedule design problem in a formal manner, but the main limitation of model checking is still present: state-space explosion and memory capabilities. If the system consists of several operating regions (it was shown how to address the problem with 5 different operating regions) memory will inevitably become the main limitation. A divide and conquer approach may provide a good approach and instead of addressing the full operating space with a single set of automata the problem can

be decomposed into several smaller problems, aggregating the obtained results to provide a full solution to the bigger problem. The methodology presented a possible abstraction methodology, different levels of abstraction may be used as part of the divide and conquer approach in order to tackle the memory problem.

A trade-off analysis between the actual benefits from using the methodology in a real working development cycle, and the amount of effort required to implement it as part of the design cycle should be conducted. The acceptance and usability of the methodology requires training in the tools and methods, as well as a change in paradigm about the way design and verification is performed. The presented methodology will need to incorporate certain level of automation in order to generate and incorporate the system abstraction into the model checker. This will be required to make the transition into the model checking approach easier for the users, otherwise it will be harder to incorporate the methodology into a working environment.

The gain scheduling design and verification problem was addressed in this thesis considering the reference tracking case only. Disturbance rejection mode is yet to be analysed, as well as both modes working together. Also, optimization criteria can be included when using the model checker to generate a solution to either the controller tuning problem or the schedule design problem. As is, the model checker non-deterministically searches for a solution to the problem with the one objective of meeting requirements, if more than one solution exists there is no current criteria to explore them.

The presented methodology allows to consider the controller as a separate entity from the plant in the model checking environment. So far, the type of performance requirements that were considered for design and verification purposes are related to the plant. Constraints to the control signal and the monitoring of its behaviour can be added into the problem formulation as well.

The problem under analysis in this thesis was considered in an isolated manner. In reality, the control software resides in the same computer which handles several more functionalities (Figure 1.1) and has specific memory and timing constraints. By including more of these features, not only control performance requirements can be analysed, but fault modes, and the interaction of other control-related components and its overall effect in the system under analysis can be included. By including fault modes in the problem formulation, external factor boundaries and the type of damage these factors can do to the system can be analysed.

So far the problem was formulated so when the process switches operating regions a set of gains is selected and it does not switch whilst operating in that particular region. This assumption was made under the consideration that this is how the system behaves, but in reality the case may be different and the controller could switch gains or control loops due to

various circumstances. The effects of this possibility in the performance of the system need to be analysed. This is also related to the fault modes analysis

Another possible future line of work is to address the control design and verification problem using a different model-checking environment. Simulink Design Verifier (SDV) seems like a good option to continue exploring the use of formal methods in the design and verification of safety-critical dynamic control systems. Even if not all the Simulink functionalities are available to be analysed in a formal manner, the benefits of Simulink to deal with dynamic systems seem appealing. So far, most case studies regarding SDV and safety-critical applications deal with algorithmic verifications, run-time errors, coding errors, and boolean-type control laws, the applicability for aiding in the design and verification of a gain schedule control scheme is yet to be addressed with SDV.

References

[1] Abate, A., Bessa, I., Cattaruzza, D., Cordeiro, L., David, C., Kesseli, P., and Kroening, D. (2017). Sound and Automated Synthesis of Digital Stabilizing Controllers for Continuous Plants. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, HSCC '17, pages 197–206, New York, NY, USA. ACM.

[2] AlAttili, I., Houben, F., Igna, G., Michels, S., Zhu, F., and Vaandrager, F. (2009). Adaptive scheduling of data paths using UPPAAL-TiGa. *Proceedings of First Workshop on Quantitative Formal Methods: Theory and Applications, QFM 2009*.

[3] Antsaklis, P. J. (2000). A brief introduction to the theory and applications of hybrid systems. In *Proc IEEE, Special Issue on Hybrid Systems: Theory and Applications*, volume 88. CiteSeer.

[4] Aréchiga, N. and Krogh, B. (2014). Using verified control envelopes for safe controller design. In *American Control Conference (ACC), 2014*, pages 2918–2923. IEEE.

[5] Aréchiga, N., Loos, S. M., Platzer, A., and Krogh, B. H. (2012). Using theorem provers to guarantee closed-loop system properties. In *American Control Conference (ACC), 2012*, pages 3573–3580. IEEE.

[6] Åström, K. J. and Wittenmark, B. (2013). *Computer-controlled systems: theory and design*. Courier Corporation.

[7] Baier, C., Katoen, J.-P., and Larsen, K. G. (2008). *Principles of model checking*. MIT Press.

[8] Ball, T., Cook, B., Levin, V., and Rajamani, S. K. (2004). SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *International Conference on Integrated Formal Methods*, pages 1–20. Springer.

[9] Balluchi, A., Benvenuti, L., Di Benedetto, M. D., Pinello, C., and Sangiovanni-Vincentelli, A. L. (2000). Automotive engine control and hybrid systems: Challenges and opportunities. *Proceedings of the IEEE*, 88(7):888–912.

[10] Behbahani, A., Adibhatla, S., and Rauche, C. (2009). Integrated model-based controls and PHM for improving turbine engine performance, reliability, and cost. In *45th AIAA Joint Propulsion Conference*.

[11] Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K. G., and Lime, D. (2006). UPPAAL-TiGa: Timed games for everyone. In *Nordic Workshop on Programming Theory (NWPT'06)*.

[12] Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K. G., and Lime, D. (2007). UPPAAL-TiGa: Time for playing games! In *CAV*, volume 4590, pages 121–125. Springer.

[13] Behrmann, G., David, A., and Larsen, K. (2004). A tutorial on UPPAAL. *Formal methods for the design of real-time systems*, pages 33–35.

[14] Behrmann, G., Larsen, K. G., and Rasmussen, J. I. (2005). Optimal scheduling using priced timed automata. *ACM SIGMETRICS Performance Evaluation Review*, 32(4):34–40.

[15] Bennett, S. (2002). Control and the digital computer: The early years. *IFAC Proceedings Volumes*, 35(1):237–242.

[16] Bennion, M. and Habli, I. (2014). A candid industrial evaluation of formal software verification using model checking. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 175–184. ACM.

[17] Bertoli, P., Cimatti, A., Slaney, J., and Thiébaux, S. (2002). Solving power supply restoration problems with planning via symbolic model checking. In *Proceedings of the 15th European Conference on Artificial Intelligence*, pages 576–580. IOS Press.

[18] Bhattacharyya, S., Cofer, D., Musliner, D., Mueller, J., and Engstrom, E. (2015). Certification considerations for adaptive systems. In *Unmanned Aircraft Systems (ICUAS), 2015 International Conference on*, pages 270–279. IEEE.

[19] Black, H. S. (1934). Stabilized feed-back amplifiers. *Electrical Engineering*, 53(1):114–120.

[20] Bode, H. W. (1945). *Network analysis and feedback amplifier design*. Bell Telephone Laboratory series. Van Nostrand, New York, NY.

[21] Boehm, B. W. and DeMarco, T. (1997). Software risk management. *IEEE software*, 14(3):17.

[22] Bouyer, P., Cassez, F., Fleury, E., and Larsen, K. G. (2004). Optimal strategies in priced timed game automata. In *FSTTCS*, volume 4, pages 148–160. Springer.

[23] Bowen, J. and Stavridou, V. (1993). Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209.

[24] Bozzano, M., Cimatti, A., Gario, M., and Tonetta, S. (2013). A Formal Framework for the Specification, Verification and Synthesis of Diagnosers. *AAAI (Late-Breaking Developments)*, 13:17.

[25] Branicky, M. S., Borkar, V. S., and Mitter, S. K. (1998). A unified framework for hybrid control: Model and optimal control theory. *IEEE transactions on automatic control*, 43(1):31–45.

[26] Brat, G., Drusinsky, D., Giannakopoulou, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Venet, A., Visser, W., and Washington, R. (2004). Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2):167–198.

[27] Broy, M. (2006). Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 33–42. ACM.

[28] Bulychev, P., David, A., Larsen, K. G., Mikučionis, M., Poulsen, D. B., Legay, A., and Wang, Z. (2012). UPPAAL-SMC: Statistical model checking for priced timed automata. *Proceedings 10th Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL*.

[29] Butler, R. W., Caldwell, J. L., Carreno, V. A., Holloway, C. M., Miner, P. S., and Di Vito, B. L. (1995). NASA Langley's research and technology-transfer program in formal methods. In *Computer Assurance, 1995. COMPASS'95. Systems Integrity, Software Safety and Process Security. Proceedings of the Tenth Annual Conference on*, pages 135–149. IEEE.

[30] Cassandras, C. G. and Lafortune, S. (2009). *Introduction to discrete event systems*. Springer Science & Business Media.

[31] Cassez, F., David, A., Fleury, E., Larsen, K. G., and Lime, D. (2005). Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, volume 5, pages 66–80. Springer.

[32] Cassez, F., Jessen, J. J., Larsen, K. G., Raskin, J.-F., and Reynier, P.-A. (2009). Automatic synthesis of robust and optimal controllers-an industrial case study. In *HSCC*, volume 9, pages 90–104. Springer.

[33] Chow, T. S. (1978). Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187.

[34] Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri, M. (1999). NuSMV: A new symbolic model verifier. In *International conference on computer aided verification*, pages 495–499. Springer.

[35] Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model checking*. MIT press.

[36] Cofer, D. and Miller, S. P. (2014). Formal methods case studies for DO-333. Technical Report NASA/CR-2014-218244, NF1676L-18435, NASA Langley Research Center; Hampton, VA, United States.

[37] Craigen, D., Gerhart, S., and Ralston, T. (1995). Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98.

[38] Culley, D. E., Thomas, R., and Saus, J. (2007). Concepts for Distributed Engine Control. In *Proceedings of the 43th AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit*, pages 8–11.

[39] Cutts, S. (2002). A collaborative approach to the More Electric Aircraft. In *Power Electronics, Machines and Drives, 2002. International Conference on (Conf. Publ. No. 487)*, pages 223–228. IET.

[40] David, A., Grunnet, J. D., Jessen, J. J., Larsen, K. G., and Rasmussen, J. I. (2010). Application of model-checking technology to controller synthesis. In *International Symposium on Formal Methods for Components and Objects*, pages 336–351. Springer.

[41] David, A., Larsen, K. G., Legay, A., Mikučionis, M., and Poulsen, D. B. (2015). UPPAAL SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415.

[42] Deligiannis, V. and Manesis, S. (2007). A survey on automata-based methods for modelling and simulation of industrial systems. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 398–405. IEEE.

[43] Dodd, I. and Habli, I. (2012). Safety certification of airborne software: An empirical study. *Reliability Engineering & System Safety*, 98(1):7–23.

[44] Dorf, R. C. and Bishop, R. H. (2011). *Modern control systems*. Pearson.

[45] Dowson, M. (1997). The Ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84.

[46] Drusinsky, D., Michael, J. B., and Shing, M.-T. (2008). A framework for computer-aided validation. *Innovations in Systems and Software Engineering*, 4(2):161–168.

[47] Dwyer, M. B., Hatcliff, J., et al. (2003). Bogor: an extensible and highly-modular software model checking framework. *ACM SIGSOFT Software Engineering Notes*, 28(5):267–276.

[48] Enoiu, E. P., Sundmark, D., and Pettersson, P. (2013). Model-based test suite generation for function block diagrams using the UPPAAL model checker. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 158–167. IEEE.

[49] Evans, W. R. (1950). Control system synthesis by root locus method. *Transactions of the American Institute of Electrical Engineers*, 69(1):66–69.

[50] FAA (2005). Airworthness standards. *US Department of Transportation*, 25.

[51] FAA (2012). DO-178C. *Software considerations in airborne systems and equipment certification*.

[52] Faleiro, L. (2002). Power optimized aircraft. *Interavia Business and Technology*, 57(662):22–22.

[53] Fan, J., Jiao, J., Wu, W., and Zhao, T. (2015). A model-checking oriented modeling method for safety critical system. In *Reliability Systems Engineering (ICRSE), 2015 First International Conference on*, pages 1–6. IEEE.

[54] Fantechi, A. and Gnesi, S. (2011). On the adoption of model checking in safety-related software industry. In *International Conference on Computer Safety, Reliability, and Security*, pages 383–396. Springer.

[55] Fotouhi, A., Auger, D. J., Propp, K., and Longo, S. (2017). Accuracy versus simplicity in online battery model identification. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, PP(99):1–12.

[56] Fotouhi, A., Shateri, N., Auger, D. J., Longo, S., Propp, K., Purkayastha, R., and Wild, M. (2016). A MATLAB graphical user interface for battery design and simulation; from cell test data to real-world automotive simulation. In *2016 13th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, pages 1–6.

[57] Foukarakis, M., Leonidis, A., Antona, M., and Stephanidis, C. (2014). Combining Finite State Machine and Decision-Making Tools for Adaptable Robot Behavior. In *International Conference on Universal Access in Human-Computer Interaction*, pages 625–635. Springer.

[58] Franchetti, F., Sandryhaila, A., and Johnson, J. R. (2014). High assurance SPIRAL. In *SPIE Defense+ Security. International Society for Optics and Photonics*. SPIE.

[59] Franklin, G. F., Powell, J. D., Emami-Naeini, A., and Powell, J. D. (1994). *Feedback control of dynamic systems*, volume 3. Addison-Wesley Reading, MA.

[60] Garlapati, S. and Shukla, S. K. (2012). Formal verification of hierarchically distributed agent based protection scheme in smart grid. In *International SPIN Workshop on Model Checking of Software*, pages 137–154. Springer.

[61] Gerhart, S., Craigen, D., and Ralston, T. (1994). Experience with formal methods in critical systems. *IEEE Software*, 11(1):21–28.

[62] Giannakopoulou, D. (2010). "Fly Me to the Moon": Verification of Aerospace Systems. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 5–11. IEEE.

[63] Gibson, C., Bonnici, M., and Castet, J.-F. (2015). Model-based spacecraft fault management design & formal validation. In *Aerospace Conference, 2015 IEEE*, pages 1–12. IEEE.

[64] Gigante, G. and Pascarella, D. (2012). Formal Methods in Avionic Software Certification: The DO-178C Perspective. *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, pages 205–215.

[65] Girard, A., Pola, G., and Tabuada, P. (2010). Approximately bisimilar symbolic models for incrementally stable switched systems. *IEEE Transactions on Automatic Control*, 55(1):116–126.

[66] Gordon, R. (1998). A calculated look at fixed-point arithmetic. *Embedded Systems Programming*, 11(4):72–79.

[67] Grunnet, J. D., Bak, T., Bendtsen, J. D., and Ankersen, F. (2009). PAHSCTRL - A control synthesis toolbox for piecewise-affine hybrid systems. In *Control Conference (ECC), 2009 European*, pages 1161–1166. IEEE.

[68] Havelund, K. and Pressburger, T. (2000). Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381.

[69] Havelund, K. and Shankar, N. (1996). Experiments in theorem proving and model checking for protocol verification. In *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 662–681. Springer.

[70] Hawkins, R., Habli, I., Kelly, T., and McDermid, J. (2013). Assurance cases and prescriptive software safety certification: A comparative study. *Safety science*, 59:55–71.

[71] Hinchey, M. G. and Bowen, J. P. (2012). *Industrial-strength formal methods in practice*. Springer Science & Business Media.

[72] Hinton, A., Kwiatkowska, M., Norman, G., and Parker, D. (2006). Prism: A tool for automatic verification of probabilistic systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444. Springer.

[73] Hirst, M., McLoughlin, A., Norman, P., and Galloway, S. (2011). Demonstrating the more electric engine: a step towards the power optimised aircraft. *IET Electric Power Applications*, 5(1):3–13.

[74] Holaza, J., Takács, B., Kvasnica, M., and Di Cairano, S. (2015). Safety verification of implicitly defined MPC feedback laws. In *2015 European Control Conference (ECC)*, pages 2547–2552. IEEE.

[75] Holloway, C. M. (1997). Why engineers should consider formal methods. In *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE*, volume 1, pages 1–3. IEEE.

[76] Holzmann, G. J. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295.

[77] Irani, S., Singh, G., Shukla, S. K., and Gupta, R. K. (2005). An overview of the competitive and adversarial approaches to designing dynamic power management strategies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(12):1349–1361.

[78] Jaw, L. C. and Mattingly, J. D. (2009). *Aircraft Engine Controls: Design, System Analysis, and Health Monitoring*. AIAA, Reston.

[79] Jensen, S. C., Jenney, G. D., and Dawson, D. (2000). Flight test experience with an electromechanical actuator on the F-18 systems research aircraft. In *Proceedings on The 19th Digital Avionics Systems Conference*, volume 1, pages 2E3–1. IEEE.

[80] Jeppu, N. Y., Jeppu, Y., and Murthy, N. (2015). Arguing formally about flight control laws. In *Industrial Instrumentation and Control (ICIC), 2015 International Conference on*, pages 378–383. IEEE.

[81] Jessen, J. J., Rasmussen, J. I., Larsen, K. G., and David, A. (2007). Guided controller synthesis for climate controller using UPPAAL-TIGA. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 227–240. Springer.

[82] Jones, M. (1997). What really happened on mars rover pathfinder. *The Risks Digest*, 19(49):1–2.

[83] Jongerden, M., Haverkort, B., Bohnenkamp, H., and Katoen, J.-P. (2009). Maximizing system lifetime by battery scheduling. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 63–72. IEEE.

[84] Kamali, M., Dennis, L. A., McAree, O., Fisher, M., and Veres, S. M. (2017). Formal verification of autonomous vehicle platooning. *Science of Computer Programming*, 148:88 – 106. Special issue on Automated Verification of Critical Systems (AVoCS 2015).

[85] Kelly, R. and Moreno, J. (2001). Learning PID structures in an introductory course of automatic control. *IEEE Transactions on Education*, 44(4):373–376.

[86] Kim, K.-D. and Kumar, P. R. (2012). Cyber-physical systems: A perspective at the centennial. *Proceedings of the IEEE*, 100(Special Centennial Issue):1287–1308.

[87] Kornecki, A. J. and Zalewski, J. (2010). Hardware certification for real-time safety-critical systems: State of the art. *Annual Reviews in Control*, 34(1):163 – 174.

[88] Koutsoukos, X. D., Antsaklis, P. J., Stiver, J. A., and Lemmon, M. D. (2000). Supervisory control of hybrid systems. *Proceedings of the IEEE*, 88(7):1026–1049.

[89] Kröger, F. and Merz, S. (2008). *First-Order Linear Temporal Logic*, pages 153–179. Springer Berlin Heidelberg, Berlin, Heidelberg.

[90] Kulshreshtha, A. and Charrier, J. (2007). Electric actuation for flight and engine control: evolution and challenges. In *SAE-ACGSC meeting*, volume 99.

[91] Kumar, J. A. and Vasudevan, S. (2012). Verifying dynamic power management schemes using statistical model checking. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 579–584. IEEE.

[92] Kunze, M. and Weske, M. (2016). Sequential systems. In *Behavioural Models*, pages 39–79. Springer.

[93] Kwon, Y. and Kim, E. (2015). Bounded Model Checking of Hybrid Systems for Control. *IEEE Transactions on Automatic Control*, 60(11):2961–2976.

[94] Landau, I. D., Lozano, R., M'Saad, M., and Karimi, A. (1998). *Adaptive control*, volume 51. Springer Berlin.

[95] Larsen, K. G., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., and Romijn, J. (2001). As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In *CAV*, volume 2102, pages 493–505. Springer.

[96] Le Lann, G. (1997). An analysis of the Ariane 5 flight 501 failure - a system engineering perspective. In *Engineering of Computer-Based Systems, 1997. Proceedings., International Conference and Workshop on*, pages 339–346. IEEE.

[97] Leith, D. J. and Leithead, W. E. (2000). Survey of gain-scheduling analysis and design. *International Journal of Control*, 73(11):1001–1025.

[98] Lemmon, M. D., He, K. X., and Markovsky, I. (1999). Supervisory hybrid systems. *IEEE Control Systems*, 19(4):42–55.

[99] Leveson, N. G. and Turner, C. S. (1993). An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41.

[100] Levine, W. S. (1996). *The control handbook*. CRC press.

[101] Li, Y., Ang, K. H., and Chong, G. C. (2006). PID control system analysis and design. *IEEE Control Systems Magazine*, 26(1):32–41.

[102] Lindahl, M., Pettersson, P., and Yi, W. (2001). Formal design and analysis of a gear controller. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(3):353–368.

[103] Liu, X. and Smolka, S. A. (1998). Simple linear-time algorithms for minimal fixed points. In *International Colloquium on Automata, Languages, and Programming*, pages 53–66. Springer.

[104] Maler, O. (1997). *Hybrid and Real-Time Systems: International Workshop, HART'97, Grenoble, France, March 26-28, 1997, Proceedings*, volume 1201. Springer Science & Business Media.

[105] Maler, O., Pnueli, A., and Sifakis, J. (1995). On the synthesis of discrete controllers for timed systems. In *STACS 95*, pages 229–242. Springer.

[106] Markovski, J. (2013). An integrated systems engineering framework for supervisor synthesis, verification, and performance evaluation. In *European Control Conference (ECC)*, pages 650–657. IEEE.

[107] Markovski, J., van Beek, D. A., Theunissen, R. J., Jacobs, K. G., and Rooda, J. (2010). A state-based framework for supervisory control synthesis and verification. In *Decision and Control (CDC), 2010 49th IEEE Conference on*, pages 3481–3486. IEEE.

[108] Martin, L. K., Schatalov, M., Hagner, M., Goltz, U., and Maibaum, O. (2013). A methodology for model-based development and automated verification of software for aerospace systems. In *IEEE Aerospace Conference*, pages 1–19. IEEE.

[109] Mathworks (2013). Simulink Design Verifier.

[110] Moore, R. E. (1966). *Interval Analysis*, volume 4. Prentice-Hall Englewood Cliffs, NJ.

[111] Moore, R. E. (1979). *Methods and Applications of Interval Analysis*, volume 2. Society for Industrial and Applied Mathematics (SIAM), Philadelphia.

[112] Moy, Y., Ledinot, E., Delseny, H., Wiels, V., and Monate, B. (2013). Testing or formal verification: DO-178C alternatives and industrial experience. *IEEE Software*, 30(3):50–57.

[113] Munassar, N. M. A. and Govardhan, A. (2010). A comparison between five models of software engineering. *IJCSI International Journal of Computer Science Issues*, 7(5):94–101.

[114] Nerode, A. (1998). Special issue on hybrid control systems. *IEEE Transactions on Automatic Control*, 43(4).

[115] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73.

[116] Newman, R. (2004). The More Electric Engine Concept. In *SAE Technical Paper 2004-01-3128*. SAE International.

[117] Norman, G., Parker, D., Kwiatkowska, M., Shukla, S. K., and Gupta, R. K. (2002). Formal analysis and validation of continuous-time Markov chain based system level power management strategies. In *High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International*, pages 45–50. IEEE.

[118] Norman, P., Galloway, S., Burt, G., Hill, J., and Trainer, D. (2008). Evaluation of the dynamic interactions between aircraft gas turbine engine and electrical system. *4th IET International Conference on Power Electronics, Machines and Drives (PEMD 2008)*, pages 671–675.

[119] Norstrom, C., Wall, A., and Yi, W. (1999). Timed automata as task models for event-driven systems. In *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*, pages 182–189. IEEE.

[120] Nuzzo, P., Sangiovanni-Vincentelli, A. L., Bresolin, D., Geretti, L., and Villa, T. (2015). A platform-based design methodology with contracts and related tools for the design of cyber-physical systems. *Proceedings of the IEEE*, 103(11):2104–2132.

[121] Ogata, K. (1970). *Modern control engineering*. Prentice-Hall.

[122] Ogata, K. (1995). *Discrete-time control systems*, volume 2. Prentice Hall Englewood Cliffs, NJ.

[123] Ordóñez, P., Mills, A. R., Dodd, T. J., and Liu, J. (2017). Formal Verification of a Gain Scheduling Control Scheme. In *25th Mediterranean Conference on Control and Automation*. IEEE.

[124] Ortega, R. and Kelly, R. (1984). PID self-tuners: Some theoretical and practical aspects. *IEEE Transactions on Industrial Electronics*, 4:332–338.

[125] Pagetti, C., Saussié, D., Gratia, R., Noulard, E., and Siron, P. (2014). The ROSACE case study: From Simulink specification to multi/many-core execution. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 309–318. IEEE.

[126] Pajic, M., Mangharam, R., Sokolsky, O., Arney, D., Goldman, J., and Lee, I. (2014). Model-driven safety analysis of closed-loop medical systems. *IEEE Transactions on Industrial Informatics*, 10(1):3–16.

[127] Pakmehr, M., Fitzgerald, N., Feron, E. M., Shamma, J. S., and Behbahani, A. (2014). Gain scheduled control of gas turbine engines: Stability and verification. *Journal of Engineering for Gas Turbines and Power*, 136(3).

[128] Pasareanu, C. S., Mehlitz, P. C., Bushnell, D. H., Gundy-Burlet, K., Lowry, M., Person, S., and Pape, M. (2008). Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 15–26. ACM.

[129] Pecheur, C., Cimatti, A., and Cimatti, R. (2002). Formal verification of diagnosability via symbolic model checking. In *Workshop on Model Checking and Artificial Intelligence (MoChArt-2002), Lyon, France*.

[130] Pratt, V. (1995). Anatomy of the Pentium bug. *TAPSOFT'95: Theory and Practice of Software Development*, pages 97–107.

[131] Quan, T. T., Hoang, D. L., Nguyen, B. T., Nguyen, A. N., Tran, Q. D., Nguyen, P. H., Bui, T. H., Do, A. T., Huynh, L. V., Doan, N. T., et al. (2010). MAFSE: A model-based framework for software verification. In *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*, pages 150–156. IEEE.

[132] Quevedo, J. and Escobet, T. (2000). *Digital control: Past, present and future of PID control*. Elsevier Science Inc.

[133] Rajkumar, R. R., Lee, I., Sha, L., and Stankovic, J. (2010). Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference*, pages 731–736. ACM.

[134] Rodriguez-Navas, G. and Proenza, J. (2013). Using timed automata for modeling distributed systems with clocks: challenges and solutions. *IEEE Transactions on Software Engineering*, 39(6):857–868.

[135] Rolls-Royce (2015). *The jet engine*. John Wiley & Sons.

[136] Rosero, J., Ortega, J., Aldabas, E., and Romeral, L. (2007). Moving towards a more electric aircraft. *IEEE Aerospace and Electronic Systems Magazine*, 22(3):3–9.

[137] RTCA (2011). DO-333 formal methods supplement to DO-178C and DO-278A. *Techinal Report*.

[138] Rugh, W. J. and Shamma, J. S. (2000). Research on gain scheduling. *Automatica*, 36(10):1401–1425.

[139] Scherer, S., Lerda, F., and Clarke, E. M. (2005). Model checking of robotic control systems. In *Proceedings of the 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*.

[140] Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., and Bienmüller, T. (2017). Incremental bounded model checking for embedded software. *Formal Aspects of Computing*, 29(5):911–931.

[141] Sen, S. and Vangheluwe, H. (2006). Multi-domain physical system modeling and control based on meta-modeling and graph rewriting. In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*, pages 69–75. IEEE.

[142] Sesic, A., Dautovic, S., and Malbasa, V. (2008). Dynamic power management of a system with a two-priority request queue using probabilistic-model checking. *IEEE transactions on computer-aided design of integrated circuits and systems*, 27(2):403–407.

[143] Sexton, D., Gilhead, P., and Quadir, R. (2013). Practical experiences of using formal requirements and their role in an overall work-flow. *System Safety Conference incorporating the Cyber Security Conference 2013, 8th IET International*.

[144] Shmaliy, Y. (2007). *Continuous-time systems*. Springer Science & Business Media.

[145] Shukla, S. K. and Gupta, R. K. (2001). A model checking approach to evaluating system level dynamic power management policies for embedded systems. In *High-Level Design Validation and Test Workshop, 2001. Proceedings. Sixth IEEE International*, pages 53–57. IEEE.

[146] Silva, W., Bezerra, E., Winterholer, M., and Lettnin, D. (2013). Automatic property generation for formal verification applied to HDL-based design of an on-board computer for space applications. In *Test Workshop (LATW), 2013 14th Latin American*, pages 1–6. IEEE.

[147] Siminiceanu, R. I. and Ciardo, G. (2012). Symbolic model checking for avionics. *Formal Methods for Industrial Critical Systems: A Survey of Applications*, pages 85–112.

[148] Sobel, A. E. K. and Clarkson, M. R. (2002). Formal methods application: An empirical tale of software development. *IEEE Transactions on Software Engineering*, 28(3):308–320.

[149] Spang III, H. A. and Brown, H. (1999). Control of jet engines. *Control Engineering Practice*, 7(9):1043–1059.

[150] Sullivan, K. J., Socha, J., and Marchukov, M. (1997). Using formal methods to reason about architectural standards. In *Proceedings of the 19th international conference on Software engineering*, pages 503–513. ACM.

[151] Tabuada, P. (2009). *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media.

[152] Tabuada, P. and Pappas, G. J. (2006). Linear time logic control of discrete-time linear systems. *Automatic Control, IEEE Transactions on*, 51(12):1862–1877.

[153] Valkonen, J., Björkman, K., Frits, J., and Niemelä, I. (2010). Model checking methodology for verification of safety logics. *SIAS 2010 - The 6th International Conference on Safety of Industrial Automated Systems*.

[154] Visser, W., Pasareanu, C. S., and Khurshid, S. (2004). Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107.

[155] Walter, B., Hammes, J., Piechotta, M., and Rudolph, S. (2017). A Formalization Method to Process Structured Natural Language to Logic Expressions to Detect Redundant Specification and Test Statements. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pages 263–272.

[156] Wang, W., Qin, X., and Mishra, P. (2010a). Temperature-and energy-constrained scheduling in multitasking systems: A model checking approach. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pages 85–90. ACM.

[157] Wang, W., Qin, X., and Mishra, P. (2010b). Temperature-and energy-constrained scheduling in multitasking systems: a model checking approach. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 85–90. ACM.

[158] Whalen, M., Cofer, D., Miller, S., Krogh, B. H., and Storm, W. (2007). Integration of formal analysis into a model-based software development process. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 68–84. Springer.

[159] Wongpiromsarn, T., Topcu, U., and Murray, R. M. (2010). Receding horizon control for temporal logic specifications. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, pages 101–110. ACM.

[160] Wongpiromsarn, T., Topcu, U., and Murray, R. M. (2012). Receding horizon temporal logic planning. *IEEE Transactions on Automatic Control*, 57(11):2817–2830.

[161] Wongpiromsarn, T., Topcu, U., Ozay, N., Xu, H., and Murray, R. M. (2011). Tulip: a software toolbox for receding horizon temporal logic planning. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*, pages 313–314. ACM.

[162] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. (2009). Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):19–19.

[163] Xu, H. (2013). *Design, specification, and synthesis of aircraft electric power systems control logic*. PhD thesis, California Institute of Technology.

[164] Xu, H., Topcu, U., and Murray, R. M. (2012a). A case study on reactive protocols for aircraft electric power distribution. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pages 1124–1129. IEEE.

[165] Xu, H., Topcu, U., and Murray, R. M. (2012b). A case study on reactive protocols for aircraft electric power distribution. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pages 1124–1129. IEEE.

[166] Zeller, J., Lehtinen, B., and Merrill, W. (1982). The role of modern control theory in the design of controls for aircraft turbine engines. Technical Report NASA-TM-82815, E-1162, NAS 1.15:82815, NASA Lewis Research Center; Cleveland, OH, United States.

[167] Ziegler, J. G. and Nichols, N. B. (1942). Optimum settings for automatic controllers. *ASME*, 64(11).