University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Josip Babić

MODEL-BASED APPROACH TO REAL-TIME EMBEDDED CONTROL SYSTEMS DEVELOPMENT WITH LEGACY COMPONENTS INTEGRATION

DOCTORAL THESIS

Zagreb, 2014

University of Zagreb
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Josip Babić

MODEL-BASED APPROACH TO REAL-TIME EMBEDDED CONTROL SYSTEMS DEVELOPMENT WITH LEGACY COMPONENTS INTEGRATION

DOCTORAL THESIS

Supervisors:
Professor Ivan Petrović, PhD
Siniša Marijan, PhD

Zagreb, 2014

Sveučilište u Zagrebu
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Josip Babić

MODELSKI PRISTUP RAZVOJU UGRADBENIH RAČUNALNIH SUSTAVA UPRAVLJANJA ZA RAD U STVARNOM VREMENU UZ INTEGRACIJU NASLIJEĐENIH KOMPONENATA

DOKTORSKA DISERTACIJA

Mentori:
Prof. dr. sc. Ivan Petrović
Dr. sc. Siniša Marijan

Zagreb, 2014.

This doctoral thesis has been completed at the Department of Control and Computer Engineering, Faculty of Electrical Engineering and Computing, University of Zagreb, and at KONČAR Electrical Engineering Institute, Zagreb

Supervisor: Professor Ivan Petrović, PhD, and Siniša Marijan, PhD

The thesis has 175 pages.

Dissertation No.

To my family.

Acknowledgements

First of all, I would like to thank my two mentors. Professor Ivan Petrović has been with me since fourth year of my graduate study and has motivated and guided me in my scientific endeavours since. Siniša Marijan has mentored me during the first year of my employment at KONČAR – Electrical Engineering Institute and has continued to guide my professional development as head of Embedded Systems Department. His appointment as Member, and currently President, of the Managing Board of KONČAR – Electrical Engineering Institute was a great loss to the whole department, but I was lucky to still have him as my second mentor during doctoral studies.

I would like to thank all my colleagues at KONČAR – Electrical Engineering Institute for sharing their knowledge and experience and for making everyday professional challenges a pleasure. Special thanks goes to Mario Bilić who was my unofficial professional mentor, alongside Siniša Marijan. Besides teaching me a great deal, some of Mario's ideas have been directly implemented in this dissertation.

Most of all, I would like to thank my dear parents for all they have done for me. They taught me the value of hard work and discipline without which this dissertation would never be completed.

First Mentor

Ivan Petrović was born in Klobuk, Bosnia and Herzegovina in 1961. He received B.Sc., M.Sc. and Ph.D. degrees in electrical engineering from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER), Zagreb, Croatia, in 1983, 1989 and 1998, respectively.

For the first ten years after graduation he was with the Institute of Electrical Engineering of Končar Corporation in Zagreb, where he had been working as a research and development engineer for control and automation systems of electrical drives and industrial plants. From 1994 he has been working at the Department of Control and Computer Engineering at FER. In November 2009 he was promoted to Full Professor. He has actively participated as a collaborator or principal investigator on 30 national and 15 international scientific projects. Currently, he coordinates EU FP7 project "Centre of Research Excellence for Advanced Cooperative Systems" (ACROSS). He published 40 papers in scientific journals and more than 150 papers in proceedings of international conferences in the area of control engineering and automation applied to control mobile robots and vehicles, power systems, electromechanical systems and other technical systems.

Prof. Petrović is a member of IEEE, Croatian Academy of Engineering (HATZ), president of the Croatian Robotics Society, vice-president of the Technical committee on Robotics of the International Federation of Automatic Control (IFAC), executive committee member of the Federation of International Robot-soccer Association (FIRA), and a founding member of the iSpace Laboratory Network. He is also a member of the Croatian Society for Communications, Computing, Electronics, Measurements and Control (KoREMA) and Editor-in-Chief of the Automatika journal. He received the award "Professor Vratislav Bedjanič" in Ljubljana for outstanding M.Sc. thesis in 1990 and silver medal "Josip Lončar" from FER for outstanding Ph.D. thesis in 1998. For scientific achievements he received the award "Rikard Podhorsky" from the Croatian Academy of Engineering, "National Science Award of the Republic of Croatia" and the gold plaque "Josip Lončar" from FER in 2008, 2011 and 2013, respectively.

Ivan Petrović rođen je u Klobuku, Bosna i Hercegovina, 1961. godine. Diplomirao je, magistrirao i doktorirao u polju elektrotehnike na Sveučilištu u Zagrebu Fakultetu elektrotehnike i računarstva (FER), 1983., 1989. odnosno 1998. godine.

Prvih deset godina po završetku studija radio je na poslovima istraživanja i razvoja sustava upravljanja i automatizacije elektromotornih pogona i industrijskih postrojenja u Končar - Institutu za elektrotehniku. Od svibnja 1994. radi u Zavodu za automatiku i računalno inženjerstvo FER-a. U studenom 2009. godine izabran je u zvanje redovitog profesora. Sudjelovao je kao suradnik ili voditelj na 30 nacionalnih i 15 međunarodnih znanstvenih projekata. Trenutačno je koordinator EU FP7 projekta "Centre of Research Excellence for Advanced Cooperative Systems" (ACROSS). Objavio je 40 znanstvenih radova u časopisima i više od 150 znanstvenih radova u zbornicima skupova u području automatskog upravljanja i estimacije s primjenom u upravljanju mobilnim robotima i vozilima te energetskim, elektromehaničkim i drugim tehničkim sustavima.

Prof. Petrović član je stručne udruge IEEE, Akademije tehničkih znanosti Hrvatske (HATZ), predsjednik Hrvatskog društva za robotiku, dopredsjednik tehničkog odbora za robotiku međunarodne udruge IFAC, član izvršnog odbora međunarodne udruge FIRA te suutemeljitelj međunarodne udruge "The iSpace Laboratory Network". Član je i upravnog odbora Hrvatskog društva za komunikacije, računarstvo, elektroniku, mjerenja i automatiku (KoREMA) i glavni i odgovorni urednik časopisa Automatika. Godine 1990. primio je u Ljubljani nagradu "Prof. dr. Vratislav Bedjanič" za posebno istaknuti magistarski rad, 1998. srebrnu plaketu "Josip Lončar" FER-a za posebno istaknutu doktorsku disertaciju, a za znanstvena je postignuća dobio 2008. godine nagradu "Rikard Podhorsky" Akademije tehničkih znanosti Hrvatske, 2011. godine "Državnu nagradu za znanost" i 2013. godine zlatnu plaketu "Josip Lončar" FER-a.

Second Mentor

Siniša Marijan was born in Kutina, Croatia in 1960. He received B.Sc. and Ph.D. degrees in electrical engineering from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER), Zagreb, Croatia in 1985 and 2011, respectively.

Since 1984 he has been working at KONČAR – Electrical Engineering Institute, firstly on R&D activities of microprocessor based control systems for electrical drives and development of software for real-time embedded control systems, than as a project leader and Head of the Section for Embedded Systems responsible for complex projects. For more than 20 years he has been responsible for the development of SW and HW components of power converters and control systems applied in power generation units and rail vehicles. Developed solutions have been successfully applied in hydro and thermal power plants, locomotives, coaches, trams, trains and wind turbines. In 2011 he became a Member of the Managing Board, while nowadays he is the President of the Managing Board of KONČAR – Electrical Engineering Institute. In 2012 he was assigned research associate. He actively participated in the implementation of four research projects funded by state funds and is the author or coauthor of forty scientific papers and articles, and hundreds of internal surveys, studies, and reports.

Dr. Marijan has received several national and international awards for developed products, e.g. ARCA, INOVA, INVENTIKA, EUREKA and award of the Croatian chamber of economy for the best innovation.

Siniša Marijan rođen je u Kutini 1960. godine. Diplomirao je i doktorirao u polju elektrotehnike na Fakultetu elektrotehnike i računarstva (FER) Sveučilišta u Zagrebu, 1985. odnosno 2011. godine.

Od 1984. godine radi u KONČAR – Institutu za elektrotehniku d.d., najprije kao istraživač-suradnik na sustavima mikroprocesorskog upravljanja elektromotornim pogonima te razvoju softvera za rad u stvarnom vremenu, a potom kao voditelj složenih projekata i rukovoditelj Odjela za ugradbene računalne sustave. Dvadesetak godina bio je odgovoran za razvoj programskih i sklopovskih komponenata ugradbenih računalnih sustava koje se koriste u energetskim pretvaračima i sustavima upravljanja u elektroenergetici i tračničkim vozilima. Razvijena rješenja uspješno su primijenjena u hidroelektranama, termoelektranama, lokomotivama, vagonima, tramvajima, vlakovima i vjetroagregatima. U 2011. postao je član Uprave, a 2014. predsjednik Uprave KONČAR – Instituta za elektrotehniku. Godine 2012. izabran je u zvanje znanstvenog suradnika. Aktivno je sudjelovao u realizaciji četiri znanstvena projekta financiranih iz državnih fondova. Autor je ili koautor četrdesetak znanstvenih i stručnih radova te stotinjak internih elaborata, studija i izvještaja.

Dr. Marijan dobitnik je više priznanja za inovacije na izložbama "INOVA", "EUREKA", "ARCA", "INVENTIKA" te nagrade "Zlatna Kuna" koju dodjeljuje Hrvatska gospodarska komora.

Abstract

Embedded control systems software is continually gaining importance, it is becoming more complex and it often must comply with very rigid requirements. At the same time, legacy software components proven in practice are preferred in safety critical embedded control systems. Model-based development has emerged as an approach that can tackle the complexities of embedded control systems, but its' application can be hindered by established development procedures based on legacy components. Real-time properties validation in the context of model-based development is not well researched so this can also protract introduction of model-based techniques. This thesis proposes methods to facilitate transition from legacy development practices into modern model-based embedded control systems development. This goal is achieved by transferring knowledge and confidence condensed in legacy software components across the gap and by validating real-time properties of the embedded software.

Legacy software components are integrated into model-based development toolchain in a systematical and structured way. It has been shown that this approach provides flexibility in managing components and facilitates their reuse, that it provides highly customizable automated code generation, and that it enables linking of newly generated code with legacy object files.

Novel methods for real-time properties validation have been proposed that supplement existing functional model-based testing approaches. Software component real-time testing based on configuration space partitioning and on real-time testing pattern has been elaborated. The method itself has been thoroughly validated to establish confidence in the testing results which have shown to be consistent and reliable. All steps in the process can be (i) fully automated, (ii) partially automated with fine tuning of particular aspects, or (iii) performed completely manually. This enables full control of the tests on the one side and effortless regression testing of large number of components on the other side.

Real-time properties of complex control software structures can be validated by novel real-time integration testing method derived from the proposed component testing approach. Here, the model of the system under test is incorporated in the real-time testing pattern adapted for control algorithm testing. The executable code, generated from such model and executed on the target, provides validation of real-time properties either in an open-loop or in a closed control loop with model of the environment executed in real-time alongside the tested algorithm. It has been shown that this method represents a natural extension of the conventional functional processor-in-the-loop testing.

All the proposed methods have been validated in three case studies which describe two real-life embedded control system development projects.

Key words: legacy software components, real-time embedded control systems, model-based development, model-based testing.

Sažetak

Modelski pristup razvoju ugradbenih računalnih sustava upravljanja za rad u stvarnom vremenu uz integraciju naslijeđenih komponenata

Programska podrška ugradbenih računalnih sustava upravljanja postaje njihov sve važniji dio, sve je složenija i nameću joj se sve teži zahtjevi. Istovremeno, kod sigurnosno kritičnih ugradbenih sustava upravljanja prednost se daje korištenju naslijeđenih programskih komponenata čija je kvaliteta i pouzdanost potvrđena dugotrajnom eksploatacijom. Modelski razvoj programske podrške prikladan je za kompleksne ugradbene sustave, ali njegova primjena može biti otežana u okruženjima s uhodanim razvojnim postupcima zasnovanim na naslijeđenim komponentama. Vrednovanje vremenskih ograničenja u okviru modelskog razvoja nije dovoljno istraženo pa i to može biti kočnica njegove implementacije. U radu su predložene metode koje olakšavaju prijelaz s naslijeđenih razvojnih postupaka na moderan razvoj ugradbenih sustava upravljanja zasnovan na modelu.

Naslijeđene programske komponente sistematski i strukturirano su integrirane u MATLAB/Simulink skup programskih alata za modelski razvoj. U okviru istraživanja, izrađeno je GRAPlab proširenje ovog modelskog okruženje koje se sastoji od skupa Simulink komponenata i od programskih alata za automatsko generiranje programskog koda. GRAPlab komponente sastoje se od Simulink maskiranog atomskog podsustava, datoteke koja definira međuovisnosti parametara i inicijalizacijske funkcije. Maskirani atomski podsustavi su Simulink modeli izrađeni od izvornih Simulink komponenata te skriveni iza grafičkog simbola i dijaloga za parametriranje. Kod promjene parametara GRAPlab bloka, njegova inicijalizacijska funkcija na temelju definiranih međuovisnosti parametara modificira sadržaj modela te prilagođuje izgled grafičkog simbola i dijaloga za parametriranje. Svakoj GRAPlab komponenti pridružena je datoteka koja definira njezino mapiranje prilikom generiranja programskog koda za ciljni ugradbeni računalni sustav. Na ovaj način, omogućeno je generiranje ciljnog programskog koda iz GRAPlab modela korištenjem naslijeđenih programskih komponenata. U radu je pokazano kako GRAPlab pristup omogućuje fleksibilno upravljanje komponentama, olakšava njihovo ponovno korištenje, pruža prilagodljivo automatizirano generiranje programskog koda i omogućuje povezivanje novogeneriranog koda s naslijeđenim objektnim datotekama.

Nove metode vrednovanja vremenskih svojstava predložene su kao nadopuna postojećim pristupima funkcionalnom modelskom vrednovanju. Vrednovanje vremenskih svojstava programskih komponenata zasnovano je na particioniranju njihova konfiguracijskog prostora i na predlošku za vremensko vrednovanje. Particioniranje konfiguracijskog prostora komponenata provodi se pomoću metode klasifikacijskih stabala. Komponente na razini modelskog okruženja razlažu se na inačice, koje predstavljaju komponente na razini ciljnog sustava, a inačice se razlažu na konfiguracije, odnosno različite načine parametriranja inačice. Za svaku konfiguraciju inačice komponente, automatski se generira model korištenjem predloška za vrednovanje vremenskih svojstava. Predložak postavlja vrednovanu konfiguraciju na programsku zadaću nižeg prioriteta zajedno s generatorom promjenjivog procesorskog opterećenja, dok se sve moguće

konfiguracije dane inačice smještaju na programsku zadaću višeg prioriteta zajedno s nadzorom brzine odziva na prekide. Generiranjem programskog koda iz modela za vrednovanje i njegovim izvođenjem na ciljnom sustavu, mjere se vremena izvođenja i kašnjenje odziva na prekide te se nadzire funkcionalno ponašanje konfiguracije. Predloškom je osigurano prekidanje vrednovane konfiguracije u svakoj mogućoj točki njena izvođenja pa se tako tijekom vrednovanja provjerava „otpornost na prekide“. Sama metoda iscrpno je provjerena kako bi se steklo pouzdanje u njezine rezultate koji su se pokazali dosljednim i pouzdanim. Proces vrednovanja može biti (i) potpuno automatiziran, (ii) djelomično automatiziran uz podešavanje pojedinih aspekata ili (iii) proveden u potpunosti ručno. Time su istovremeno omogućeni potpuna kontrola nad procesom i jednostavno regresijsko vrednovanje velikog broja komponenata. Cijeli postupak vrednovanja iscrpno je dokumentiran automatskim postupkom generiranja izvještaja zasnovanim na LaTeXpredlošcima.

U radu je predloženo ublažavanje kompleksnosti vrednovanja složenih programskih struktura upravljanja razlaganjem vrednovanja u tri koraka. U prvom koraku provodi se vrednovanje u otvorenoj petlji. Ovaj korak može se dalje razložiti u dvije faze: u prvoj fazi vrednuje se funkcionalno ponašanje algoritma dok se u drugoj fazi vrednovani algoritam ugrađuje u predložak za vrednovanje vremenskih svojstava. Radi se o predlošku za vrednovanje vremenskih svojstava sličnom predlošku za vrednovanje programskih komponenata, ali prilagođenom vrednovanju upravljačkih algoritama. Drugi korak sastoji se od funkcionalnog vrednovanja kod kojega se programski kod vrednovanog sustava generiran iz modela izvodi na ciljnom sustavu u zatvorenoj upravljačkoj petlji s modelom okoline, odnosno procesa, izvođenom unutar simulacijskog okruženja na osobnom računalu. Vrednovanje vremenskih svojstava upravljačkog algoritma u zatvorenoj upravljačkoj petlji podržano je trećim korakom metode. Ovdje se model procesa prilagođuje za generiranje ciljnog programskog koda koji se potom izvodi na ciljnom sustavu. Na ovaj način omogućeno je zatvaranje upravljačke petlje na ciljnom sustavu i njezino izvođenje u stvarnom vremenu. Glavna ograničenja posljednje faze odnose se na kompromise koje je potrebno napraviti kako bi se model procesa prilagodio za generiranje ciljnog programskog koda te na ograničenja resursa ciljnog sustava koji mora moći paralelno izvoditi programski kod modela okoline i programski kod vrednovanog upravljačkog algoritma.

Sve predložene metode vrednovane su kroz tri studije slučaja koje opisuju dva projekta razvoja stvarnih ugradbenih sustava upravljanja. Metoda vrednovanja programskih komponenata provjerena je nizom eksperimenata nad naslijeđenim programskim komponentama namijenjenim kontroleru za digitalnu obradu signala TMS320F28335 tvrtke Texas Instruments. Ova je metoda primijenjena tijekom razvoja sigurnosne platforme za upravljanje pružnim prijelazima na vrednovanje programskih komponenata za C8051F580 mikrokontroler tvrtke Silicon Laboratories. Sigurnosna platforma mora zadovoljiti vrlo stroge SIL (engl. *Safety Integrity Level*)zahtjeve razine 4, a primjenom predložene metode vrednovanja komponenata značajno je pojednostavljen postupak certifikacije. Metoda vrednovanja upravljačkih algoritama primijenjena je na vrednovanje algoritma za praćenje točke maksimalne snage tijekom razvoja centralnog pretvarača za fotonaponske elektrane.

Ključne riječi: naslijeđene programske komponente, ugradbeni računalni upravljački sustavi za rad u stvarnom vremenu, modelski razvoj, modelsko vrednovanje.

Contents

Chapter 1

Introduction

Embedded systems are being used in an ever growing number of applications, from simplest toys to highly complex industrial, military and space systems. In [1], it is estimated that embedded processors account for more than 98% of all produced processors. Along with faster and more complex hardware, embedded software is gaining importance and makes up to 85% of the value of the entire embedded system, [2]. For example, the development of electronics in high end vehicles accounts for 40% of the total development cost and these systems contain more than 70 electronic control units (ECUs) and 2500 signals, [3]. Under market pressure the software must be produced quickly and as bug-free as possible. Driven by these two opposing requests, Model Based Development (MBD), or Model Driven Development (MDD), has emerged as the design approach of choice. As stated in [4], MBD is not just application of graphical domain-specific languages, i.e. "programming by drawing". A good definition of MBD is given in [5], and a part of it is reproduced in the following:

"In model-based development, the model is the central artifact and is used and systematically refined through the entire development process which is literally based on or centered around it."

Projects involving embedded systems are usually multidisciplinary, involving software, hardware and mechanical components, so model can act as a means of communication between different stakeholders. Also, model can be an instrument for separating responsibilities between domain experts. Model abstraction levels can isolate basic algorithms from implementation details such as fixed-point scaling or low-level drivers. It has been reported that usage of a graphical modeling language increases productivity by four to ten times, [2].

At the same time, a strong affinity for legacy software components exists in industry, accompanied by mistrust for new development processes. This is especially true in traditionally conservative industries that deal with safety critical applications. Legacy software components are attractive because of the effort already invested in their development and testing and because they have acquired a high level of confidence through prolonged exploitation, [6]. It is generally accepted that value and reliability of a software component are increased with number of its' applications, [7]. Legacy components can prove to be more efficient and more adequate for real-time applications than code automatically generated by modern MBD tools, [8].

When considering MBD of embedded applications it is necessary to take into account various specific characteristics of these systems. They are usually reactive, meaning that they

interact either with their environment or some larger system, and can contain continuous and/or discrete subsystems, [5]. Embedded systems control and/or monitor a specific device or function, they are self-starting and self-contained. If, besides functional requirements, the correctness of embedded system operation depends on its timeliness, they are said to have (soft or hard) real-time constraints, [9]. Many industrial embedded systems, to which results of this work are intended to be applied, fall into the hard real-time category with severe timing constraints.

This thesis tackles the development of real-time embedded software on two fronts. Firstly, an approach is presented that systematically integrates legacy software components into MBD workflow providing thus benefits of both approaches, a synergy between high reliability and modern development practices. Secondly, a set of model-based testing methods are employed to validate real-time properties of embedded software. Complex applications are usually built by combining simpler components so testing is performed on two levels. On the unit testing level, individual components are tested by applying test patterns and by using watchdogs and data partitioning techniques. On the integration testing level, control algorithms are validated in open- and in closed-loop tests with the environment simulated on the PC or on the embedded target.

1.1 Background

The research presented in this theses has been conducted during author's employment at KONČAR - Electrical Engineering Institute Inc. (KEEI) and in a real-life industrial environment that dictated its' direction and scope.

1.1.1 Application Domain

This thesis represents a follow up to [7], so its' results are intended to be applied to the same category of embedded control systems. These are low-volume safety critical hard real-time embedded control systems that should have long lifetime. A short explanation of these terms is given in the remainder of this subsection, while more details are presented in section 2.1.

Although embedded systems market is dominated by mass produced consumer products, there also exist market niches with specific low-volume embedded systems. For example, such systems can be found in military, avionic, medical, railway and other specific applications. Embedded systems considered in this thesis, specifically embedded systems found in railway and power systems, fall under this category. These systems usually support various control, protection, sequential, communication and diagnostic functions, but also must fulfil certain non-functional requirements such as extended temperature range, immunity to shocks and vibrations or certain level of electromagnetic compatibility. Applications tailored to the specific customer's needs impose platform approach because developing each product from scratch would be prohibitively expensive. Users of customized products are never completely satisfied and often introduce new requirements on the system. In case of embedded control systems, this in most cases leads to application program changes. For example, application program for

proprietary vehicle control unit in tram had 39 revisions and application program for proprietary traction control unit, also in tram, evolved to 22nd revision.

Temporal constraints represent non-functional requirements that impact software development the most. Safety critical hard real-time embedded systems are those that must meet their deadlines with no exceptions or otherwise system failure with possibly catastrophic consequences can occur.

Embedded control systems considered in this work should have long life cycle, up to several decades. Such systems are often called sustainment-dominated, in terms of technological (as opposed to environmental and business) sustainment that refers to activities necessary to:

- keep the system operational;
- continue to manufacture and field versions of the system that satisfy the original requirements;
- manufacture and field revised versions of the system that satisfy evolving requirements.

In [7], it has been shown that risk of embedded systems obsolescence can be minimized by developing them using modular hardware and software components that belong to proprietary product platform. In the same work, product platform is defined as:

"Product platform represents the collection of different assets (components) with well-defined interfaces what makes a solid basis for the development and production of similar products called product variants or members of the product family. Therefore, product family is based on common platform, but product variant has specific features according to the market or customer demands."

1.1.2 Integrated development environment

Legacy software development process considered in this thesis is based on a proprietary development environment consisting of: GRaphical Application Programming tool (GRAP), microcontroller specific project structure and proprietary Real-Time Operating System (RTOS), and PC-based service and debugging application. The graphical environment is used for the development of application programs (APs) that are to be executed on hard real-time systems and it supports several different architectures of microprocessors, microcontrollers and signal processors, [10, 11, 12]. Since recently, GRAP and proprietary service tool have been merged into a single application so they represent a truly integrated development environment (IDE). The in-house policy is to have full control over system software and application IDE so all elements of the development environment are proprietary solutions. Software architecture is depicted in Fig. 1.1. It consists of system software, development environment and control program. IDE comprises software component databases, graphical tools and development framework. The control program represents union of application program and system software, [7].

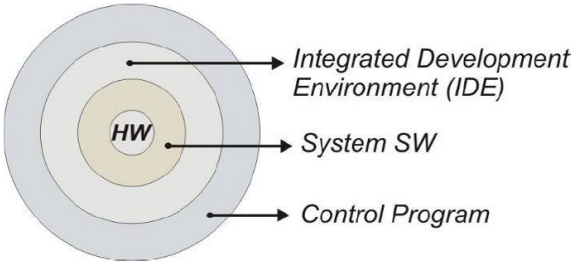


Figure 1.1: Software architecture

GRAP is used for constructing and editing APs and also for invoking compilation and linking tools. AP developed in GRAP consists of graphical program modules, so there is no need for textual coding. Modules are built by interconnecting programming elements represented by graphical symbols, similar to block diagrams. On a code level, each element is a macro program, hand-written in assembly language, carefully optimized and thoroughly tested. Macro library, as well as compilation and linking tools for the given hardware module (HWM), are included in a project structure. This is a folder structure that, besides library directory, contains folders with modules and application files. Successfully built AP is loaded onto the target, on which RTOS is running, via proprietary service and diagnostics software tool.

RTOSes used in proprietary embedded control systems consist of scheduler and diagnostic tools. Rate monotonic fixed priority scheduling is used, which means that task dispatcher schedules tasks during run-time according to priorities that have been defined in advance in such a way that tasks with shorter periods have higher priorities. Furthermore, task periods are in most cases harmonic, i.e. each task period is an exact integer multiple of the next shorter period. This ensures schedulability of up to 100% of processor load. However, the usage of all available processing resources is discouraged as this prevents changes that are regularly necessary during testing, commissioning, and during product life cycle, [7].

1.1.3 GRAP code generation

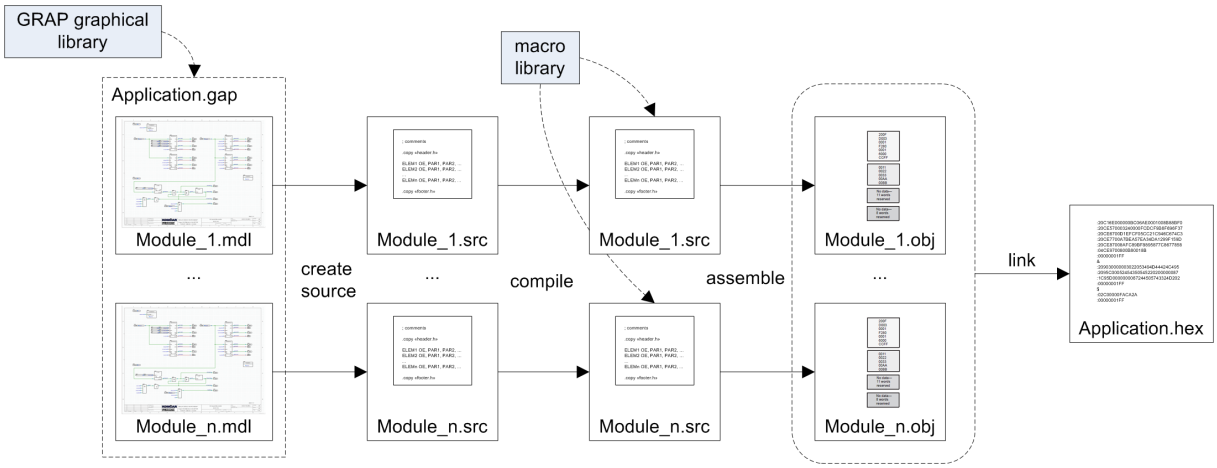


Figure 1.2: GRAP code generation

Generation of executable code from graphical application program using GRAP is illustrated in Fig. 1.2. Firstly, the graphical application program is built using programming element symbols from a graphical library. Application can be divided into an arbitrary number of program modules, which can then be reused. Each module has its own `.mdl` file, accidentally the same extension as for Simulink model files, but these two file types are not compatible.

Next, GRAP creates `.src` source files from graphical program modules. For every programming element in a module, a macro call with all the necessary arguments is placed in the source file. In the compilation step, these macro calls are expanded using macro library which results in a complete assembly listing of each module. By executing assembler on these files, Common Object File Format (COFF) files are generated that can be linked into executable `.hex` files.

This is an example of executable code generation for C2000 family of digital signal controllers (DSCs) from Texas Instruments (TI), but the code generation process is same or very similar for all the supported microprocessors, microcontrollers and signal processors.

1.1.4 Legacy Component Reuse

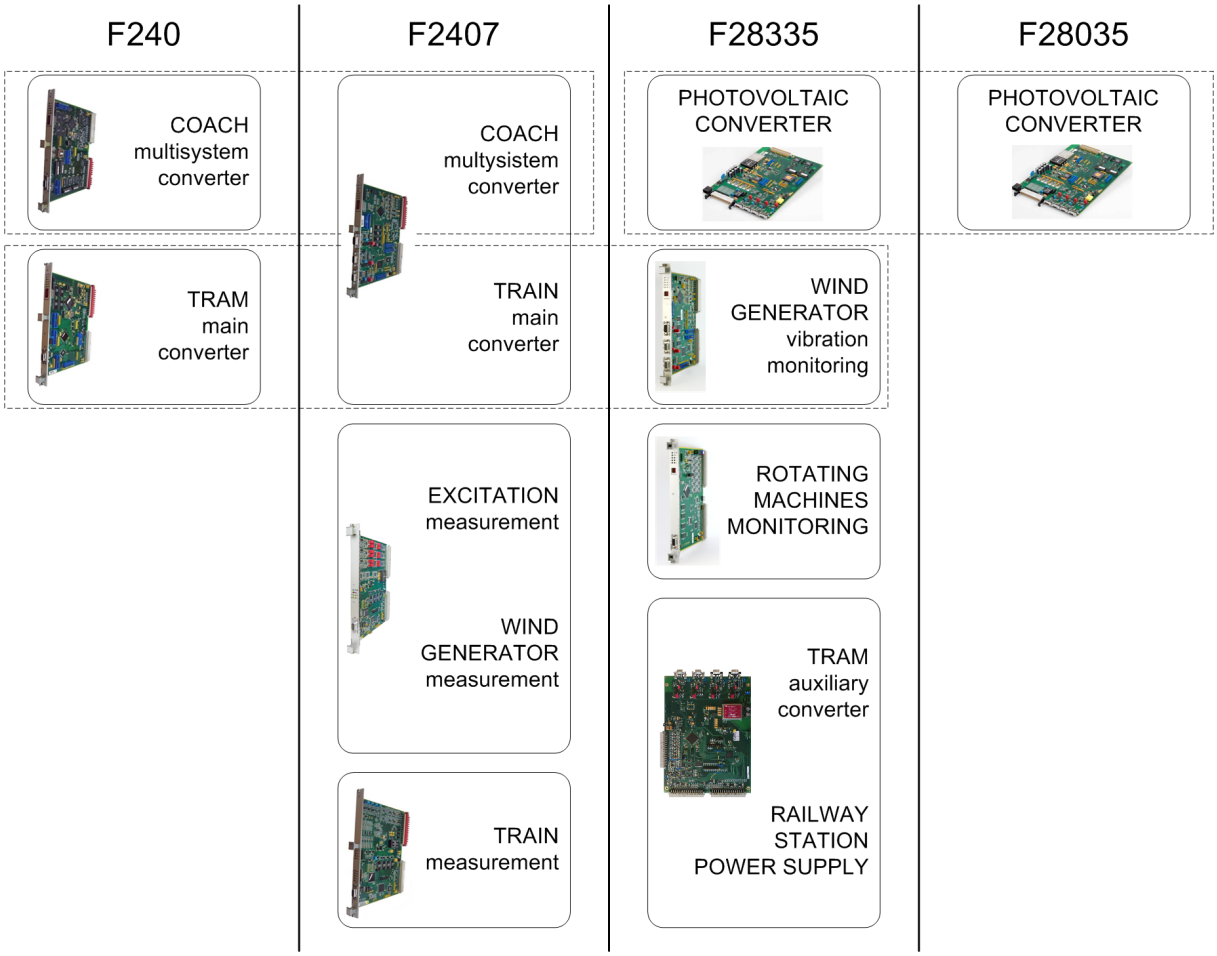


Figure 1.3: Legacy component inheritance

When developing new HWMs, based on already used or newly introduced microcontroller (MCU), existing atomic software components are inherited form previous projects whenever possible. In Fig. 1.3 a string of HWMs based on Texas Instruments' C2000 family of DSCs is shown together with short note about their application. A large number of atomic legacy components is inherited among these projects, e.g. a component for scaled summation of four signals and a component for filtration of logical signal shown in Fig. 1.4.

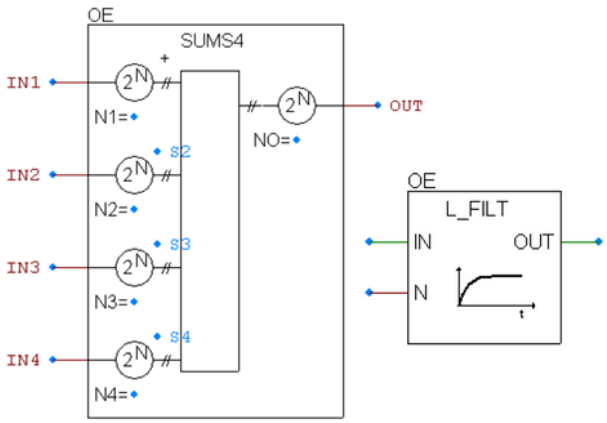


Figure 1.4: Inherited atomic components

1.1.5 Application Development Process

Application, as considered in this work, consists of one or more APs, each executed on a separate MCU/DSC based HWM, that together perform some functionality. The development of an application can be observed in several phases. Firstly, the project leader (PL) divides the functionality of the application between HWMs, defines interfaces between them and assigns the development of each application program to one application engineer (AE). AE divides his AP into functional modules (FMs), defines their interfaces and distributes their development to module developer (MD). Individual FM can't usually be thoroughly tested on their own so, after more or less partial testing by test engineers (TEs), they are handed to the responsible AE for integration into AP. When AP is successfully integrated, i.e. all FM are compiled and linked, limited amount of testing by TEs is performed on laboratory equipment. In order to fully test APs, they need to be integrated into final application by PL and run on the target hardware.

Depending on the complexity of the project, PL can play the role of AE, i.e. be responsible for one or more APs, and AE can play the role of MD, i.e. he can develop some or all modules of the AP he is responsible for. All through the development, PL, AEs and MDs are supported by system engineers (SEs). Their responsibility is to maintain development environments of individual HWMs and to develop new programming elements, if requested by AEs or MDs. The described application development process is illustrated by Fig. 1.5, development roles are depicted by light grey circles and software artifacts are represented by dark grey squares.

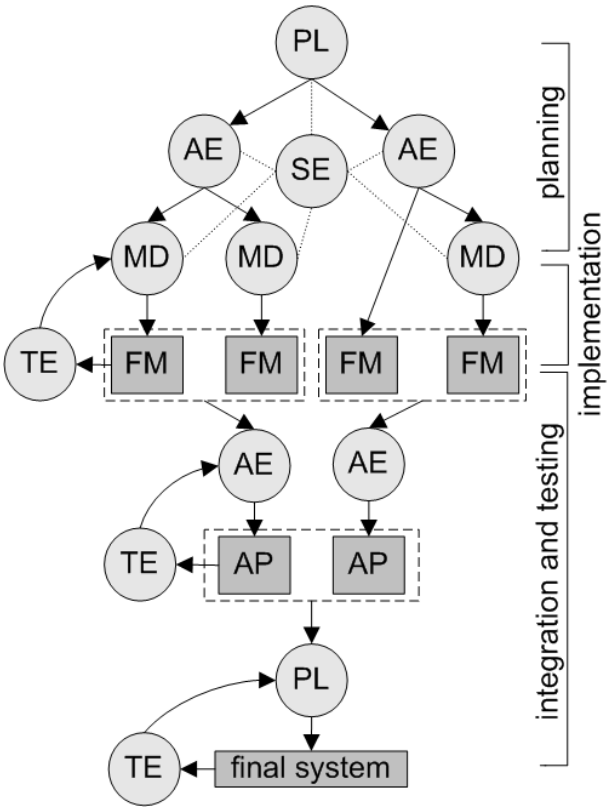


Figure 1.5: Existing software development process

Parallel to the application development, hardware development process is conducted. Proprietary hardware platform is a modular one, so existing modules are used where possible and new modules are developed if necessary, [7]. Hardware development is not in the scope of this thesis and will not be discussed further.

1.2 Motivation and Research Gaps

The motivation for the research stems from advantages and drawbacks of the existing control systems development process applied at KEEI. There existed a need to alleviate its' drawbacks by modernizing it while maintaining the advantages at the same time. Investigation of the available development methods and tools failed to provide suitable approach to achieve these goals.

1.2.1 Motivation: Advantages and Drawbacks of the Existing Process

Two main advantages of the described existing AP development process are graphical programming language and usage of legacy assembly macros as program building blocks. The first trait brings higher level of abstraction, in comparison with textual programming languages, which enables non-software engineers to implement their ideas in a target independent manner. Modularity is accomplished through the division of AP into FMs. As applications are built completely in graphical environment, without a single line of handwritten code, the very pro-

gram can serve as the documentation. Every functionality and parameter is visible in the block diagram scheme so that, when the AP is completed, it can simply be printed (in paper or PDF) and supplemented with minimal additional documentation.

Usage of assembly macros for code generation brings low-level control of all processes on the target, enables creation of highly optimized and thoroughly tested code chunks, [8] and introduces another level of modularity. By creating own code generation scheme, substantial knowledge is accumulated and kept in-house. System engineers with such knowledge have shown to be invaluable when it comes to supporting application engineers during application program development. Most of the macros have been inherited from previous projects where they have been in exploitation for years or decades and have acquired large amounts of working hours. This means that these components, besides knowledge and effort invested in them during their development, also bring great confidence obtained through real-world usage into each new project that inherits them.

The described existing application development process is flawed by two major drawbacks: by the absence of strong link between requirements management and proprietary IDE and by inability to perform complete functional tests of FMs before their integration into APs, and APs before their deployment into final system. The requirements on software artifacts, starting from individual programming elements up to full APs, are often informal or semi-formal, always in prose and in a separate document that has no automatic link to the respective artifact. This approach leads to misunderstandings that result in more development iterations than are necessary. As requirements evolve during development, the requirements document is often not updated and final design can significantly deviate from it. This can complicate product maintenance.

Algorithmic parts of FMs can be independently tested by replaying prerecorded signals at their inputs and recording and analyzing the results. The drawbacks of this kind of testing are:

- FM must be altered (input/output parts removed and signal playback added),
- closed control loops are broken,
- the tests must be performed on the target processor,
- testing process is unstructured so its' quality depends on skill and motivation of the test engineer,
- manual testing is time consuming and error prone.

Final system hardware configuration often isn't completely available during the development so testing is performed on available evaluation boards and laboratory hardware setups. These resources are scarce and can replicate the final system only to a limited extent. All this sums to the fact that complete and thorough software tests can be performed when most of the hardware and software is available, late in the development process when bugs are most expensive to correct.

If the development process was based on a model, than the model could serve as an executable specification. It would naturally evolve during development and there would be no

detachment of the design from the specification. Legacy software components would than have to be used for automatic code generation from the model in order to preserve knowledge invested in their development and confidence they have acquired during exploitation. Also, model based testing (MBT) could be conducted in order to find and fix mistakes sooner in the development. Real-time properties should be validated in the testing process.

1.2.2 Gap 1: Legacy Components and Model-Based Development

In the past, embedded systems were simpler and could often be replaced as a whole. This is often no longer possible so current trend is to "refurbish" embedded systems in place with updated software. This asks for in-depth knowledge of the embedded software maintenance, a field that is in [13] identified as lacking research. Studies show that the maintenance of existing software can often take up more than 60% of all the development efforts, [14]. In most cases, the maintenance of legacy software systems implies reuse of at least some of its' components. It is also interesting to use well understood and proven legacy software in new projects, [6, 15].

Integration of legacy software component with code automatically generated by modern MBD tools can be especially challenging. The integration asks for architectural compatibility of newly generated and legacy code. Unfortunately, most of the current code generators assume that the generated code is stand-alone so they don't make explicit the architectural choices made during code production and provide very limited support for affecting those choices. As a result, generated and legacy code must often be refactored during integration or special "glue" code must be developed, [16]. Requirements on automatically generated code from the perspective of its' integration with legacy hand code are given in [17]. The generated code must:

- not mangle variable or function names,
- allow for user defined function partitioning and function prototypes,
- call legacy functions,
- use the same variable declarations as the hand code,
- use the same C base types as the hand code.

Although importance of integrating legacy software components in model-based development of embedded control systems seems to be universally accepted, research on this topic, as well as its' support in commercial tools, is surprisingly scarce. If it is present, it exclusively refers to integration of legacy C and C++ code as in examples of literature cited above or in MATLAB's *Legacy Code Tool*. The few works that consider legacy assembly code are focused on its' automatic translation into some higher language, [18], or on extraction of original system requirements from the legacy assembly code in case of missing or faulty documentation, [14].

One solution to the "legacy assembly problem" could be to combine the two approaches: firstly to translate the assembly into a higher language, such as C, and than to use some of existing methods of integrating that code into MBD workflow. The loophole in this approach is that initial confidence in tried-and-proven legacy code is very likely lost during the process, defeating thus the purpose of the whole procedure.

1.2.3 Gap 2: Model-Based Real-Time Testing

When considering applying MBT on embedded real-time industrial systems, as presented in section 3, a number of shortcomings of present MBT methodologies arise:

- If state space explosion is to be avoided, formal methods must be applied only on high levels of abstraction [19, 20] where real-time properties of the system under test (SUT) are not modeled and cannot be verified.
- Some of the methods rely on automatically generated code in some high-level programming language, e.g. C code, [9, 21] which limits their application.
- Most of the MBT methods applied in the automotive industry are only applicable to Model-in-the-Loop (MiL) testing, [2, 22, 23, 24].
- Hardware-in-the-Loop (HiL) test approaches require expensive environment including real-time simulation tools and complex input/output hardware, [25].
- MBT approaches that do tackle real-time properties are mostly designed for telecommunications domain [26], and are not applicable to hybrid control systems.
- There exist real-time MBT procedures for automotive domain, but they are either based on discrete use-case models, and again not applicable to hybrid control systems [27, 28] or require two models, one for use case modeling and the other for modeling continuous behaviour [29].

In short, real-time constraints validation of embedded control systems that contain both continuous and discrete behaviours is not, to the authors knowledge, possible with current MBD methodologies proposed by the academia nor with available commercial MBD tools.

1.3 Contributions and Outline

1.3.1 Contributions

The three major contributions of this thesis are:

- Model-based approach to embedded control systems development that enables systematic integration of legacy software components.
- Method for testing software components in real-time embedded control systems that along with functional testing enables real-time constraints validation.
- Method for testing real-time constraints of complex software control structures in real-time embedded control systems.

Automatic code generation scheme presented in this thesis, and the GRAP Laboratory (GRAPlab) tool that implements it, systematically integrate legacy software components into MBD workflow inside MATLAB/Simulink environment. The approach makes no assumptions as

to programming language in which legacy software components are written so it could be some higher language, as C and C++, or even assembly, as shown in case studies presented in the thesis. Legacy software components are integrated into MBD toolchain without modifications so that no new software errors are introduced and high level of confidence in produced real-time code is preserved.

Proposed Model-Based Real-time Embedded System Testing (MoBREST) method's subset for testing software components MoBREST Component Testing (MoBREST-CT) is based on component configuration space partitioning and on real-time testing pattern (RTTP). Configuration space of the tested component is partitioned so that each parametrized implementation of the component presents one test case. Such component is than placed inside the RTTP to produce test model. This model is used twofold: firstly, it is analyzed and simulated to produce test vectors and test oracles and, secondly, executable test code is generated from it. Execution traces obtained by executing the test application on the embedded target are compared with the test oracle in order to validate component's functional correctness. At the same time, real-time constraints are checked: component's execution time is measured and its' behaviour with respect to interrupts is monitored. The tool that implements this method allows for full automation of the testing process. This can be very helpful when porting legacy development environment to new member of microcontroller family, e.g. when migrating form Texas Instruments' F2407 to F28335 or when switching form Motorola's 68000 to Freescale's Coldfire. Automatic regression testing of large number of components significantly reduces cost of such migrations.

Control algorithms can be considered as high-level software components and could as such be tested according to the proposed component real-time testing method. However, complete validation of complex software control structures often means that they need to be tested in a closed feedback loop which includes controlled plant or its' model. Considering real-time constraints introduces another level of complexity into such testing. Real-time algorithm testing method MoBREST Integration Testing (MoBREST-IT) presented in this thesis alleviates the complexity by braking down the testing process into three steps: real-time testing in an open-loop, non-real-time testing in a closed-loop, and closed loop real-time testing. Not all three steps must be conducted, e.g. for some applications it could suffice to conduct thorough functional testing on the model and validate timing constraints with open-loop real-time testing step of the proposed method.

The three contributions are placed in traditional V-model of software development in Fig. 1.6. GRAPlab automated code generation using legacy software components facilitates implementation of reliable embedded control systems while MoBREST-CT and MoBREST-IT methods enable real-time testing of these systems on component and integration level, respectively.

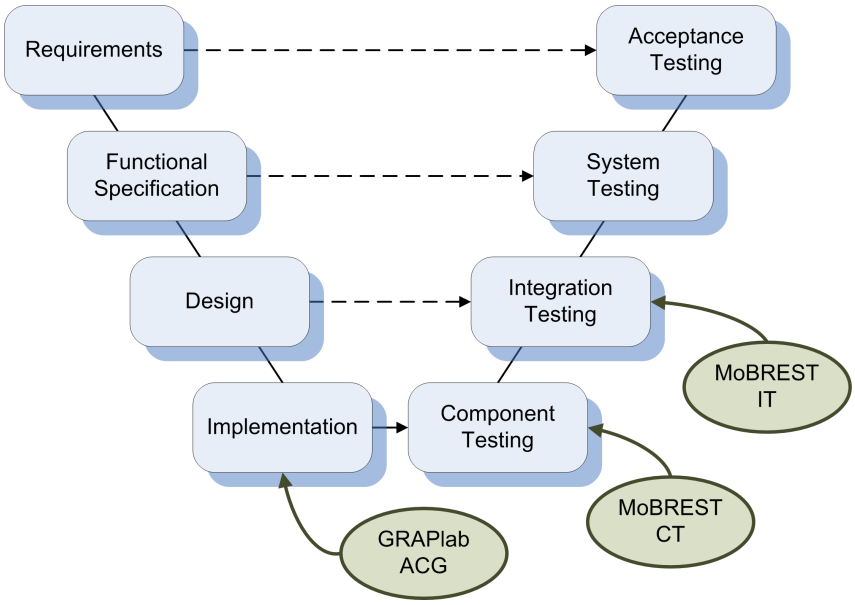


Figure 1.6: Contribution of the thesis in V-model

1.3.2 Outline of the Thesis

This chapter provides an introduction to the thesis. It outlines context in which the research was conducted by describing its' application domain and by introducing legacy tools and processes upon which it builds. Next, motivation for the research and research gaps that it fills are identified. The chapter ends by enumerating contributions of the thesis.

Chapter 2 lays foundations for the remainder of the thesis. First section deals with embedded control system: it explains what they are, explores current market trends, and stresses importance of processing resource management and of non-functional requirements they must fulfil. Second section introduces concepts of Component-Based Software Engineering (CBSE) and places proprietary legacy components in that context. Third section presents an overview of works on model-based software development from requirements through modeling, model transformations, and automated code generation to testing. The chapter ends with a short section that defines patterns in general and software test patterns in particular.

Chapter 3 elaborates the topic on MBT, opened in chapter 2. Firstly, it introduces taxonomy of software testing from the literature and expands it with regard to real-time testing of embedded control systems. Secondly, an overview of selected MBT approaches is presented with placement of each approach inside the expanded taxonomy. Thirdly, a gap in inspected MBT approaches is identified, the MoBREST method that fills it is shortly introduced and is also placed inside the taxonomy.

Chapter 4 deals with first major contribution of the thesis: the systematic and structured inclusion of legacy software components into MBD toolchain. Legacy components are assembly macros introduced in chapter 2 and MBD toolchain is selected to be based on MATLAB and Simulink. This commercial off-the-shelve (COTS) MBD environment is expanded with a *blockset* and a *toolbox* that enable automated code generation (ACG) with legacy software components integration.

Model-based real-time software component testing is the focus of chapter 5. It introduces main concepts behind the MoBREST method such as Classification Tree Method, component configuration space partitioning and real-time testing pattern. Automated test generation, execution, and documentation supported by the method are elaborated in this chapter.

Chapter 6 extends and adapts the MoBREST method for real-time integration testing of control algorithms. Complexity of such testing is alleviated by breaking it down into three steps: open-loop testing, closed-loop step-by-step testing and closed-loop real-time testing.

The first case study, described in chapter 7, deals with real-time testing of components for a Texas Instruments' digital signal microcontroller. The chapter starts by introducing the target controller, describing its' applications in proprietary control systems and experimental setup on which the study was conducted. Next, the RTTP thorough validation is conducted to ensure that all the real-time properties are indeed tested as intended. Finally, the testing process and the produced artifacts are presented on example of software component that performs Boolean operations.

Second case study presented in chapter 8 also deals with component testing, but here components for a Silicon Laboratories' microcontroller from 8051 series are tested. This microcontroller is the backbone of *Safety Platform*, developed as a central part of railway crossings (level crossing) control system. All its' components must satisfy strict safety criteria, including software components tested in scope of this case study. First part of the chapter introduces the Safety Platform while the second part presents specific aspects of component testing for this particular target system through illustrative examples. The chapter ends with an overview of test results for all the Safety Platform's tested software components.

Control algorithm real-time integration testing is showcased on the example of photovoltaic maximum power point tracking (MPPT) algorithm in last case study of the thesis presented in chapter 9. The chapter is structured in same way as the proposed testing method. Firstly, real-time testing of the MPPT algorithm implementation is conducted in an open-loop on the TI' signal controller. Secondly, the MPPT algorithm running on the target system is included in closed control loop with photovoltaic panels model simulated on a personal computer and non-real-time functional testing is performed in a step-by-step manner. Thirdly, target code is generated from the photovoltaic panels model and it is executed on the target system alongside tested MPPT algorithm, allowing thus real-time properties validation in a closed control loop.

The thesis ends with a summary, conclusions and outlook for future work presented in chapter 10.

Chapter 2

Fundamentals

In this chapter fundamentals of embedded systems, legacy software components, model-based development, and test patterns are laid out. Firstly, definitions regarding embedded systems, an overview of the embedded systems market, issues regarding processing resources and non-functional requirements are provided. Section 2.2 introduces concept of CBSE and describes legacy software components that are in the focus of research. In section 2.3 an overview of MBD is given with regard to requirements management (REQM), modeling, model transformations, ACG, and testing. Section 2.4 provides introduction to test patterns.

2.1 Embedded Systems

Embedded systems are electronic programmable sub-systems that are generally an integral part of a larger heterogeneous system. They differ from general purpose computers in that they are required to run without maintenance, can be intended to work in a spontaneous ad-hoc networks with emphasis on machine-to-machine communication, have a higher emphasis on fault tolerance and may have to compensate for failures, have stronger restrictions regarding the user interface, can have strict timing constraints, are reactive and continuously respond to incoming events and state changes, have restricted resources, and are often based on control theory. Most embedded systems are used for surveying and controlling physical processes, [13].

Real-time embedded computer systems can be defined as the ones that have a deadline, i.e. there is a time instant at which results of the computation should be available. According to the way deadline issues are handled, real-time embedded systems can be divided into soft, hard and firm real-time systems. Soft real-time systems can benefit from computation results even if deadline is missed because their timeliness requirements are based on average response time. For example, video conferencing system remains operational in case of timing constraints violation, although with reduced quality. In hard real-time systems deadline violations are not acceptable as they might cause significant functionality degradation or system failure. If timing constraints violations can cause catastrophic consequences with material and human losses, hard real-time systems are said to be safety-critical. Examples of safety critical hard real-time systems are control systems in car engines, railway crossings, process industry and medical equipment such as heart pacemakers. Firm real-time systems represent a mixture

of soft and hard real-time requirements. They allow for relaxation of timing constraints up to a certain level or within a particular time-frame. There also exist control systems where non-real time features are combined with real-time requirements of different levels of rigidity, [7].

Dynamical system which contains both discrete and continuous components is a hybrid system, [30], and its behaviour can be described by differential and difference equations. At the core of every embedded system usually there is a digital microcontroller, that represents discrete component of the system, but most embedded systems also contain continuous components. For example, analog units such as analog-to-digital and/or digital-to-analog converters can be present in the same chip as the central processing unit (CPU). Outside the microcontroller, analog electronic circuitry represents embedded system's interface to the physical environment. This mix of digital and analog world in embedded systems is the reason why they are often described and modelled as hybrid systems.

2.1.1 Market

According to [31], embedded systems account for about 98% of all computing devices, they are experiencing annual growth of more than 10% and over 40 billion embedded systems are expected to be in function worldwide by 2020. The same report states that the value added to the final product by the embedded system is often a few orders of magnitude higher than the cost of the embedded system alone. The results of this thesis could be applied to safety critical embedded control systems in the industrial, medical, automotive, aerospace/defence domains that accounted for 35% of world electronic production and 63% of European electronic production in 2012, as shown in Fig. 2.1.

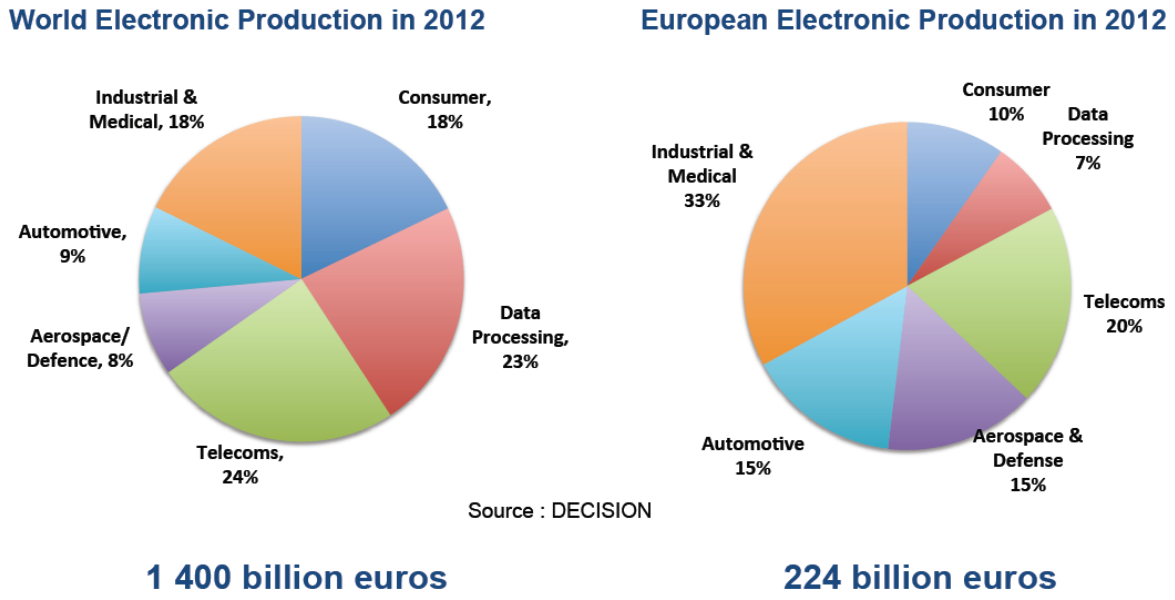


Figure 2.1: World and European electronic production per application domain in 2012, [32]

UBM's embedded market study [33] shows that embedded software consumes significantly larger amount of development effort than the embedded hardware, Fig. 2.2. Embedded soft-

ware is gaining on size and complexity. This can be seen in Fig. 2.3 that relates software size to annual volume and also shows the growth of the embedded software. Electronic control units in a new car are estimated to contain more than 100 million lines of code. Rising complexity and pervasiveness of embedded systems make software errors more common and potentially more dangerous. For example, between 1990 and 2000 about 40% of the half a million of recalled pacemakers were recalled due to firmware errors, [1].

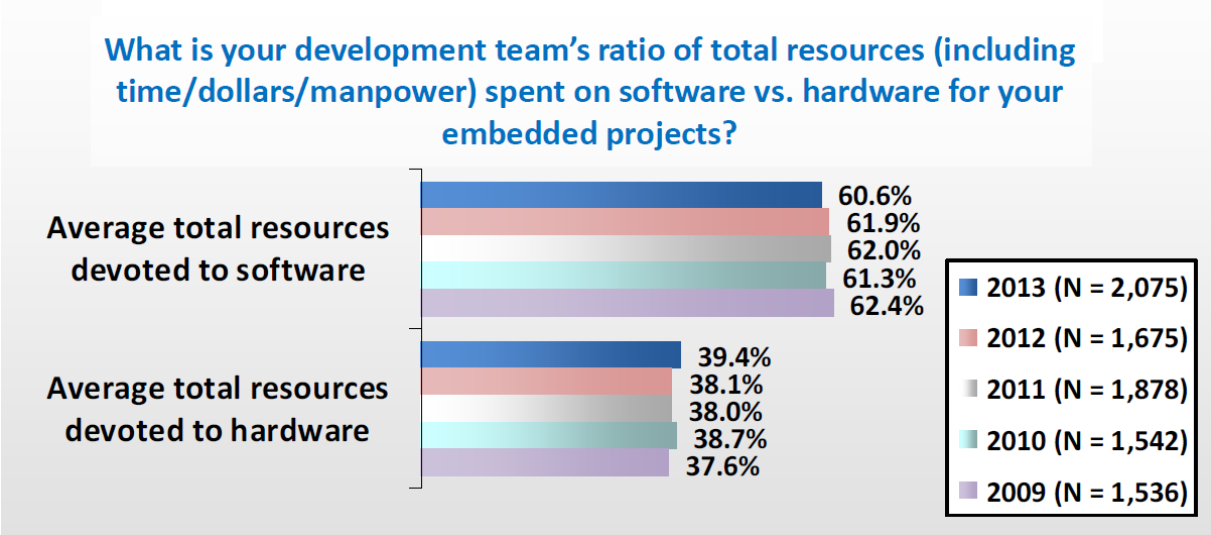


Figure 2.2: Ratio of development resources spent on embedded software and hardware, [33]

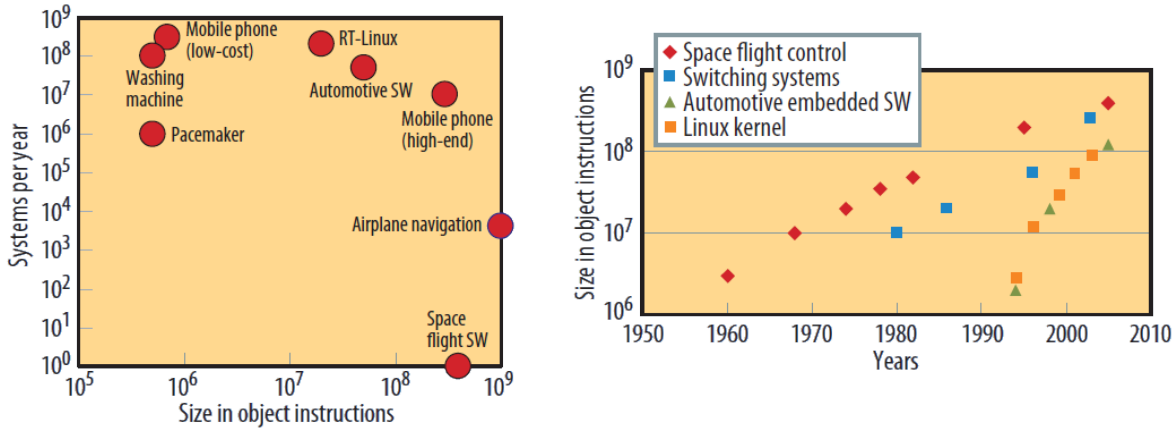


Figure 2.3: Embedded software size vs deployment (left) ant the rise of embedded software complexity (right), [1]

2.1.2 Non-functional Requirements

Requirements on a technical system can generally be divided into those that specify functionality of the system, the functional requirements, and all other requirements, the non-functional requirements. The literature more or less agrees on definition of functional requirements and in [34] two such definitions are cited:

- *"A statement of a piece of required functionality or a behavior that a system will exhibit under specific conditions."*
- *"A requirement that specifies an action that a system must be able to perform, without considering physical constraints; a requirement that specifies input/output behavior of a system."*

Timing requirements are not functional, but they can be viewed as behavioral. However, they are in literature mostly considered as performance related and thus classified under non-functional requirements.

Definitions of non-functional requirements show more discrepancies. For example, [34] cites definitions:

- *"Describe the nonbehavioral aspects of a system, capturing the properties and constraints under which a system must operate."*
- *"The required overall attributes of the system, including portability, reliability, efficiency, human engineering, testability, understandability, and modifiability."*
- *"A requirement that specifies system properties, such as environmental and implementation constraints, performance, platform dependencies, maintainability, extensibility, and reliability. A requirement that specifies physical constraints on a functional requirement."*
- *"Requirements which are not specifically concerned with the functionality of a system. They place restrictions on the product being developed and the development process, and they specify external constraints that the product must meet."*
- *"A requirement on a service that does not have a bearing on its functionality, but describes attributes, constraints, performance considerations, design, quality of service, environmental considerations, failure and recovery."*
- *"A description of a property or characteristic that a software system must exhibit or a constraint that it must respect, other than an observable system behavior."*

Non-functional requirements on embedded control systems considered in this thesis, section 1.1, can be grouped into:

- dependability – encompasses attributes of reliability, availability, safety, confidentiality, integrity and maintainability;
- environmental requests – such as storage and operational temperature and humidity ranges;
- immunity to shock and vibrations – especially important e.g. in railway applications;
- electromagnetic compatibility – refers to unintentional generation, propagation and reception of electromagnetic energy;
- resource-wise properties – like memory and power consumption;

- real-time properties.

According to embedded market study [33], a significant percentage of embedded projects include real-time and environment resistant capabilities as shown in Fig. 2.4. Non-functional requirements can significantly impact not only hardware but also software components development. Testing time can be especially extended due to type tests dictated by the non-functional requirements. Fig. 2.5 shows examples of such tests.

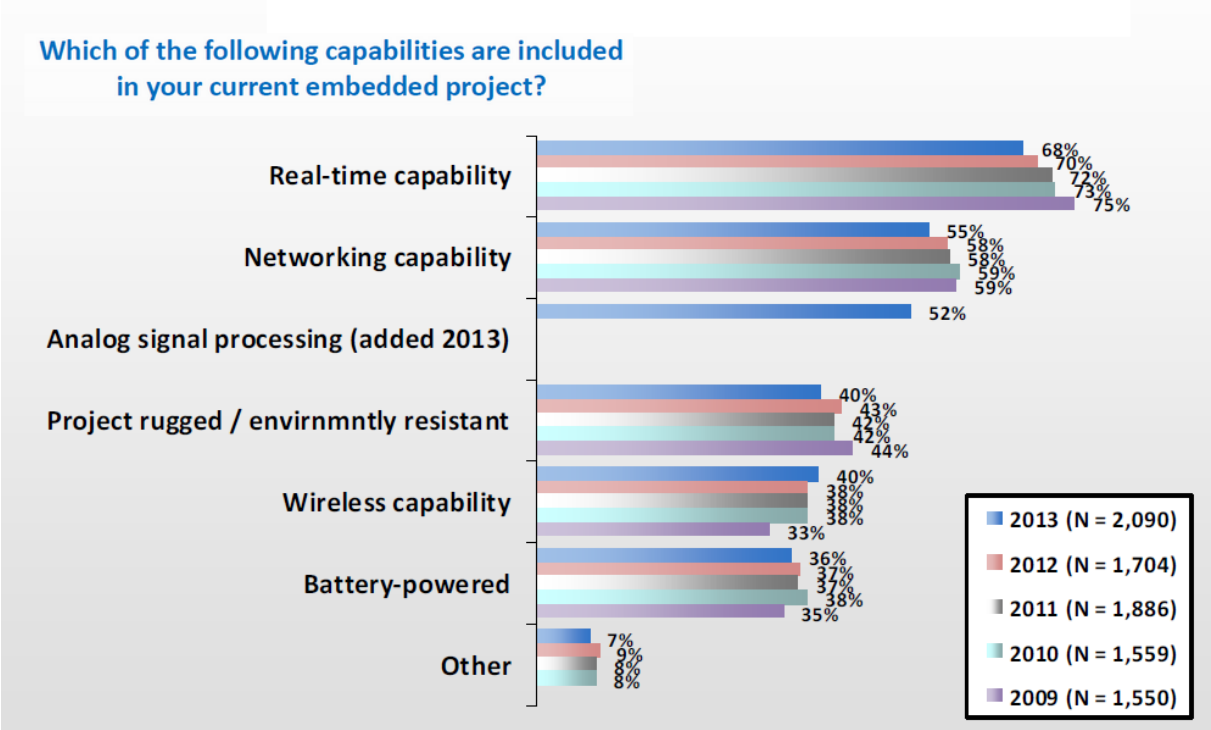


Figure 2.4: Capabilities included in embedded projects, [33]

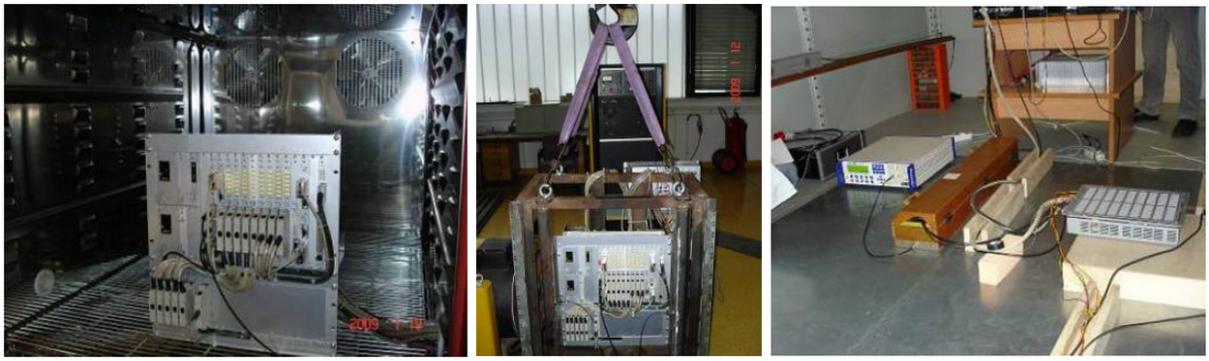


Figure 2.5: Environmental, shock and vibration immunity and electromagnetic compatibility tests of proprietary embedded control systems, [7]

Real-time properties encompass timing constraints and scheduling properties, [35]. Typical examples of timing constraints are, [2]:

- part of the code must be executed every T period of time,

- worst-case execution time of a part of the code is less than T,
- when event A finishes, event B must appear after T period of time.

Validation of timing constraints boils down to measurement of time periods, e.g. execution times of parts of the code.

Assessment of different scheduling strategies is outside the scope of this thesis and real-time embedded systems on which the research was conducted, section 1.1, utilize exclusively rate monotonic fixed priority scheduling. Thus, real-time scheduling properties considered in research include tested software behaviour with respect to interrupts: whether function of the code can be corrupted by interrupting its execution and if the code stalls interrupt servicing beyond given bound.

2.1.3 Processing Resources

Managing processing resources is key feature of every real-time system, and knowledge about task execution times is crucial in this process. Execution time analysis encompasses all methods and tools used in determining execution time of a program or its' parts. This analysis is complicated by the fact that execution times are not constant but vary with different probabilities of occurrence across some range of values. The changes are due to different input data, features of software architecture and implementation, type of processor and system architecture. Usually three types of execution time are taken into account: worst-case execution time (WCET), average-case execution time (ACET), and best-case execution time (BCET). For real-time control systems WCET is the most interesting in which case timing analysis is called WCET analysis, [7].

Timing analysis can be conducted by measurement, by static analysis or by hybrid methods. Measurements can be performed using emulators, logic analyzers, oscilloscopes, software profilers, operating system tools and other. Because it is very hard to identify the set of input data that causes WCET to occur, for hard real-time systems static timing analyses are preferred over measurement-based techniques. Static timing analysis is conducted in three steps:

1. program flow analysis – identification of all possible execution paths;
2. low-level analysis – gathering of execution times of program instructions;
3. calculation phase – estimating WCET by combining results from previous steps.

Static analysis is in most cases not appropriate for entire control software, so it is performed for parts of code and the results are used in schedulability analysis. Drawbacks of static analysis techniques are that they require significant effort of a skilled programmer, mapping between source and compiled code and hardware timing model. No matter which timing analysis techniques are used, WCET can in best case only be estimated so the WCET safe bound is usually set to a value higher than the one measured or calculated, [7].

Software architecture of embedded control systems considered in this work is component-based and application programs are composed by application engineers that are domain experts and not skilled programmers. This fact, along with the lack of timing models for used processors, is the reason for using measurement techniques in managing processing resources. The applied procedure can be broken down into steps, [7]:

1. The execution time of each atomic software component is measured and/or calculated during its development.

2. Application program is composed and downloaded to the target system where software tools are used for timing analysis. Real-time kernel based tools measure overall processor load while software components in the application program can be used to measure execution times of individual tasks or their parts.

3. If the measurements in the previous step come close to the processor utilization bound, than detailed hardware measurement are performed. These measurements are performed by logic analyzer using test points on the processor module housed in the complete system either in field or in the laboratory.

2.2 Legacy Software Components

Focus of this thesis is on reuse of proprietary legacy software components in new projects. This is a reoccurring problem in about 80% of all embedded projects according to UBM's embedded market study, Fig. 2.6.

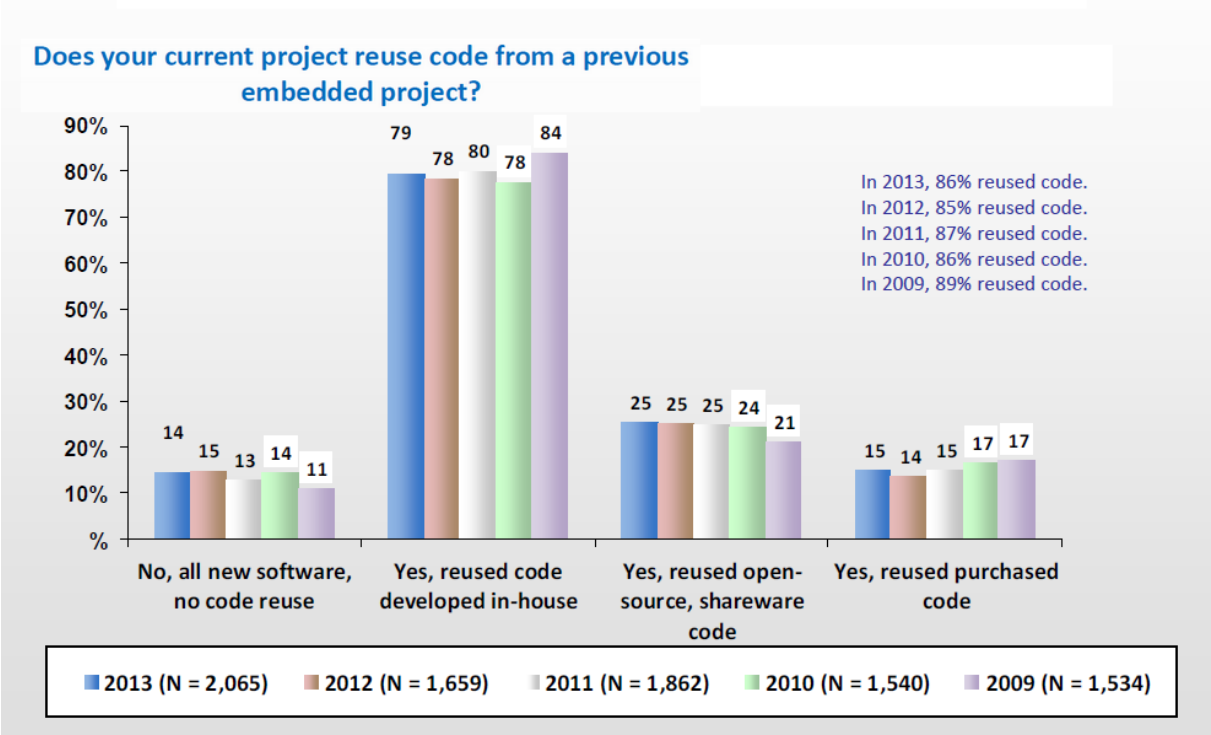


Figure 2.6: Reuse of code in embedded projects, [33]

2.2.1 Component-Based Software Engineering

This subsection places software components as defined and used in this thesis into broader context of CBSE paradigm. As already discussed in section 1.1, this work considers the same category of embedded control systems with the same type of software components as in [7], so in the following an updated discussion regarding CBSE from that thesis is reproduced.

Software Engineering Institute (SEI) provides very general definition of software component in [36]:

"A software component is an implementation, in software, of some functionality. It is reused as-is in different applications, and accessed via an application-programming interface. It may, but need not, be sold as a commercial product. A software component is generally implemented by and for a particular component technology."

Software components considered in this work are adequate for and have been extensively reused in different applications. They are accessed according to a set of rules, i.e. according to the specification of how components should interact with each other, which is generally accepted definition of application programming interface. Individual components could be sold as a commercial products, although until now only libraries of components have been sold. Component technology or framework includes run-time environment for components and other tools for designing, building, combining and deploying component and applications built from them, [36]. A run-time environment for considered components is provided by proprietary RTOSes while GRAP and proprietary service application represent tools for designing, building, combining and deploying components and application programs. As this short analysis shows, software component as considered in this work conform fully to the general definition from [36].

There exist more precise definitions of software components, one of which is given also by SEI in [37]:

"A component is an opaque implementation of functionality, subject to third party composition, and conformant with a component model."

The term *opaque implementation of functionality* refers to abstraction and information hiding and means that a component should for its consumer represent a black box. This implies that components should not be distributed in source code and should be executable on unknown target system. These conditions are not met by software components observed in this work because they are distributed as encrypted source code, should be compiled before execution and are target or domain dependant. Components in this thesis are composed by application engineers (system integrators) that are not members of the system engineering team that creates them so they are indeed "subject to third party composition". On the other hand, these components are proprietary solutions so there are no independent sources of alternative components, [7]. The last criterion in the above cited definition is what differentiates software components from conventional COTS software. Component model prescribes architectural design constraints by defining how components interact with each other so that component-based systems result in uniform, standard coordination schemes. COTS-based systems, in contrast, result in mash of product-specific interaction schemes, [37]. Software components considered in this research comply to a set of rules for their interconnection and data exchange so they fulfill the criterion.

Software components represent reusable software artifacts and reuse always brings benefits, no matter which definition of CBSE is employed. Main benefits are:

- reuse shortens development time and thus enables savings,
- extensions (components) of the existing system can be independently developed and deployed,
- component models can be designed to support properties important for specific application areas thus improving predictability of the deployed system, [37],
- the reliability and value of a component increase with the number of its applications, [7, 38].

Software component models based on uniform execution platform, like Microsoft's COM and Sun's Java Beans, enable composition of components according to their functional interfaces only. Non-functional properties of components are not considered so it is assumed that there exists unlimited virtual memory, that there are no timing constraints and that there is no interaction between components besides functional interaction, [39]. All these assumptions do not hold in case of real-time embedded systems. Computing resources in these systems are scarce, results must be produced on time and a component can influence execution of other components by consuming available processing time or by locking a peripheral unit. In [38], the importance of including non-functional properties in component model for real-time safety critical embedded systems is stressed and overview of available component models for embedded systems is given. Integration of non-functional, or extra-functional, properties into component model is an active area of research, [40, 41, 42], as well as testing of such properties in component-based systems, [38, 43].

Composition of components from independent sources is scarcely discussed in the literature on real-time embedded systems, [7]. AUTomotive Open System ARchitecture (AUTOSAR) framework, [44], is one of the rare attempts in that direction: applications are built from software components that deal with functional logic only, while the infrastructural services are provided by AUTOSAR component middleware. However, the component middleware is a layered software architecture only coarsely customizable by excluding unused layers with compile-time switches. This approach is impractical, error prone and suboptimal with respect to resource consumption. In [45], applying CBSE to AUTOSAR component middleware is proposed to alleviate its' shortcomings.

Components for real-time systems are mostly designed to be composable and executable only on proprietary product platforms so their portability is often limited. Usage of platform independent component model, such as function blocks defined in IEC 61131-3 standard, [46], or AUTOSAR's functional components, [44], can improve component portability.

2.2.2 Proprietary Legacy Components

Proprietary legacy software components exist on two hierarchical level. On the lower level there are basic atomic components, or program elements, each of which performs some specific function such as summation or logical *AND* function. On the higher hierarchical level there

are composed components, or program modules. Modules are created by composing atomic components in a graphical environment, as described in chapter 1, and they implement advanced functions like control structures or communication and I/O interfaces. An extensively commented example of graphical composed component is shown in Fig. 2.7.

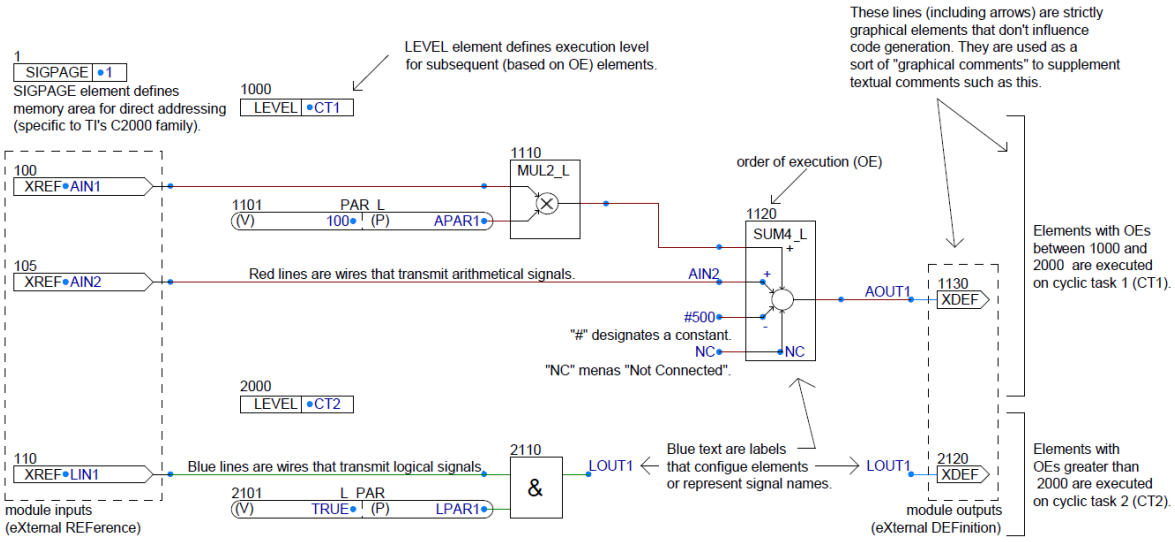


Figure 2.7: An example of graphical composed component

Legacy software components are organized in databases in graphical and code environments. Libraries in graphical environment contain atomic components' graphical symbols and graphical composed components. In code environment there are atomic components' source code library, composed components' source code library and composed components' relocatable object code library. Graphical processor as a part of the IDE provides interface between these two environments, [7]. The connection between component graphics and source code is illustrated in Fig. 2.8 on the example of composed component from Fig. 2.7.

Significant advantage of legacy software components is that their execution time is often known in advance. This is possible because software components are small and optimized and because rather simple processor architectures (with no caches, branch prediction or out-of-order execution) are used. Component execution time significantly depends on its' configuration and number of connected ports. Because components are configured and interconnected during application program development, without changes during run-time, the difference between WCET and BCET of a configured atomic software component is often less than few percent, [7].

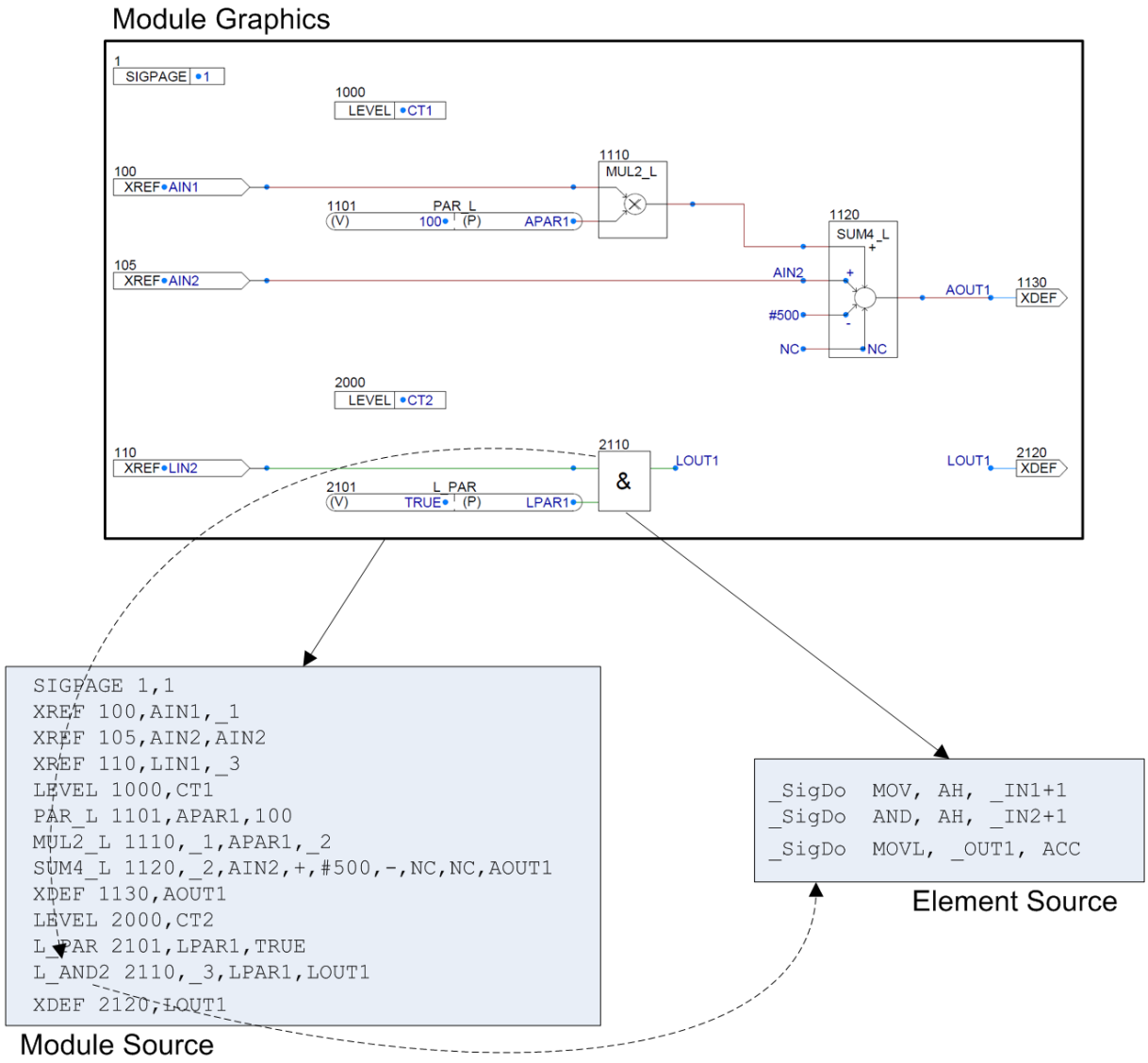


Figure 2.8: Graphical processor provides interface between graphical and code environments

2.3 Model-Based Development

MBD process is illustrated in Fig. 2.9. It starts with requirements elicitation and formalization, proceeds with modeling and code generation. Two way traceability between various development artifacts should be ensured – from requirements through model to code and test cases. Testing is performed throughout the development: model is checked against requirements and the code, executed on PC or on the target, is tested by comparing its' execution traces with simulation traces of the model.

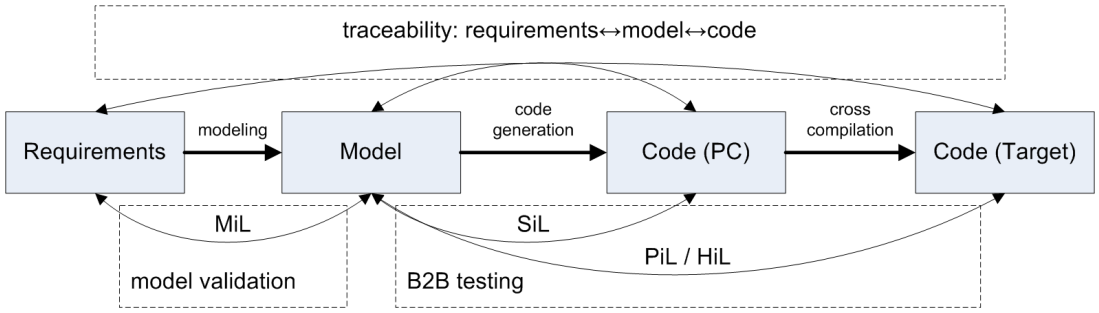


Figure 2.9: Model Based Development workflow

MBD encompasses thus a significant portion of development process. If development stages of embedded project from UBM's embedded market study [33] are adopted for software development, than MBD can cover, fully or partially, all stages shown in Fig. 2.10 except the *sending to production* stage.

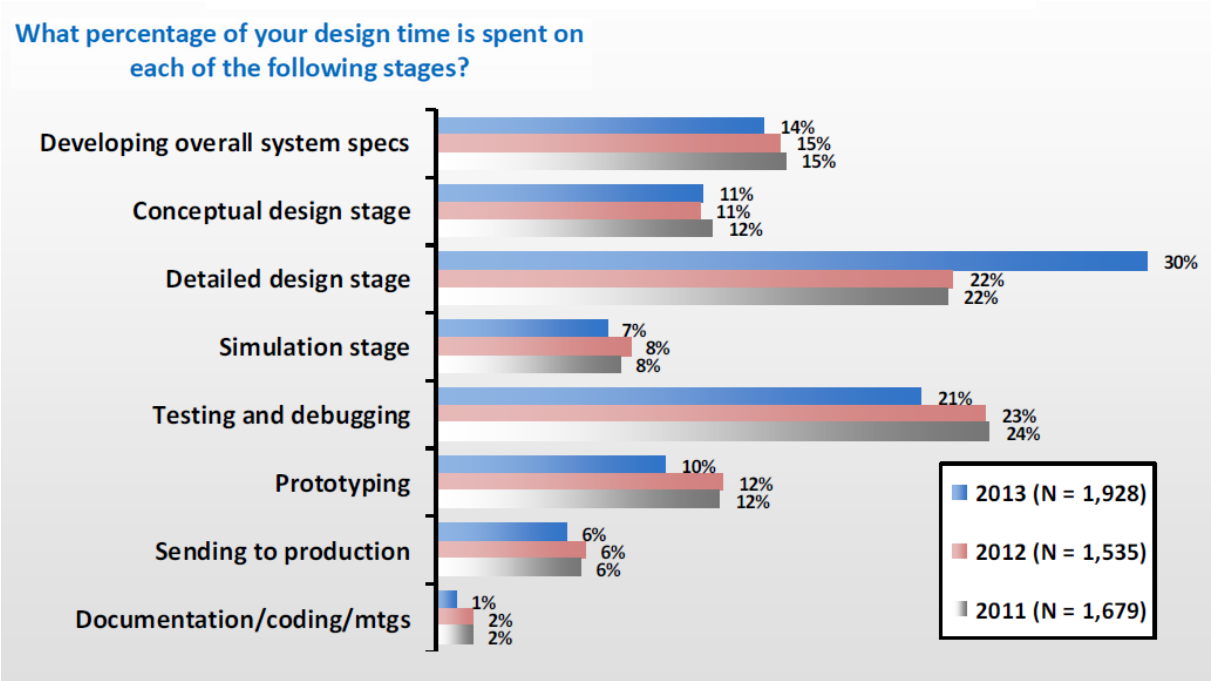


Figure 2.10: Percentage of time spent on design stages, [33]

2.3.1 Requirements

Requirements on software systems are often incomplete, ambiguous and even contradictory. Embedded systems are specific because, besides functional requisites, often a number of non-functional requirements and constraints must be satisfied. One of the goals of MBD is to make intended behavior explicit in the form of behavioral model. It has been reported in [47] that 2 to 6 times more requirement errors are discovered if system model is used during development and testing.

In [48], requirements traceability is achieved by defining four links among requirements and between requirements and other modeling artifacts: *derive* represents derivation of requirement

from another, *refine* indicates that an element is a refinement of a textual requirement, *satisfy* shows the satisfaction of requirement by design and *verify* links requirement with test case that verifies it.

Stronger link between requirements and testing is proposed in [49], where requirements are formalized by constructing *Timed Usage Models*, out of which tests are automatically generated. Motivation of this approach is to alleviate greatest drawbacks of current REQM methodologies: lack of systematical analysis and informal and ambiguous requirement description.

Requirements traceability is the focus of DARWIN4Req approach presented in [50]. This is a metamodel that establishes two-way links between three independent and heterogeneous models: requirement model, solution model and verification and validation model.

Tighter integration of REQM and design is achieved by AutoRAID extension to AutoFOCUS MBD development environment, [51]. Here, requirements are classified as use cases and architectural, modal or data type constraints and their stepwise refinement leads to consistent and complete *Requirements Engineering Product Model*. Elements of the design are produced from requirement model using *motivate* and *associate* functions. This approach ensures seamless transition from REQM to design and enables requirement analysis; a number of automatic consistency checks are described in the paper.

Direct generation of test cases from requirements written in natural language is proposed in [52]. SOLIMVA methodology automatically translates natural language requirements into Statechart models out of which test cases are generated. The method identifies test scenarios using combinatorial design techniques and input dictionary defined by the user. This approach is applicable to system and acceptance testing phases.

Non-functional requirements on embedded systems can be modeled and analysed in Unified Modeling Language (UML) profile Model and Analysis of Real-Time Embedded System (MARTE), [53]. It introduces time and resource models into UML, enables definition of hardware/software execution platform, provides allocation mechanism which associates application with platform, and supports quantitative analysis, e.g. of schedulability and performance, [54].

2.3.2 Modeling

In context of MBD, system design is performed by modeling. Various approaches to modeling methodology are available. In already mentioned AutoFOCUS environment, [55], separation of concerns is achieved by graphical modeling based on views: *Data Definition View* defines data types and basic functions as basis for further development, *System Structure View* describes system structure including components, interfaces and communication channels and *Behavior View* captures the behavior of each atomic component.

In order to generate implementation from the model, it must either be enriched with platform specific information or linked to separate platform model. First approach reduces potential for reuse so in [56, 57] separate modeling of functionality and HW/SW architecture is proposed. This way, independent and reusable functional and platform models are constructed and a set of rules, called model compiler, performs mapping between them. Similar approach called *Architecture Driven Development* presented in [58] consists of steps: 1) system architecture definition, 2) functional model construction, 3) HW/SW architecture definition, 4) mapping of

functional architecture to HW/SW architecture, and 5) code generation. If integrated model is used for code generation than modeling language becomes programming language [59].

Desired shift toward higher level of abstraction in MBD may be missing because many modeling languages are based on textual programming language concepts. With this motivation, [60] introduces *Model Integrated Computing* methodology where system development starts with domain specific modeling language definition, after which necessary tools are constructed and finally system design is conducted. This approach ensures that model artifacts represent domain elements, and not the code. Model interpreters that translate domain specific models to simulation, analysis and implementation models are crucial here.

The problem of modeling interactions of embedded systems with each other and their physical environment is tackled in [61]. The approach is based on Embedded Systems Modeling Language (ESMoL), a design environment that aligns control design, software modeling, code generation, and deployment. Cyberphysical systems are modelled on three levels: on physical layer, on platform layer and on computation/communication layer. A number of models are constructed or generated during proposed integration process: logical software architecture model defines functional block connections and data dependencies, hardware platform model defines computing nodes and communication networks, deployment model maps software components to computing nodes and data messages to communication ports, and timing model defines scheduling and timing constraints.

2.3.3 Model Transformations

Sometimes complexities of a system can be better managed through multiple models where each captures a different aspect of the system. In other cases, models can be refined or decomposed into other models. Automatic model transformations can help ensure consistency between multiple models in such situations. Transformations can be conducted by direct model manipulations, through intermediate representation or by using transformation languages. Direct manipulation of internal model representation is performed using standard procedural languages, but they lack abstraction so transformations are difficult to write and maintain [62]. Intermediate representation for model transformation can be some standard format, as XML, or a specific formalism, as *Synchronous Reactive Model of Computation* that enables definition of inherently correct transformations, [63]. Intermediate representation can also provide means for integration of components from different development stages, e.g. ESMoL in [61]. According to [62], domain specific languages for model transformations, e.g. *Atlas Transformation Language* and Graph Rewriting and Transformation (GReAT), represent the best alternative. GReAT treats models as graphs and performs transformations by linking input and output metamodel into unified metamodel and by defining transformations on the new metamodel, [60].

2.3.4 Automated Code Generation

A seamless transition from models to executable code is accomplished by ACG techniques. Code must be generated from the model consistently with limited use of a subset of the programming language that is considered to be safe and with limited use of well specified control

and data structures. It should comply with specified complexity measures and it must be maintainable, testable, stable, changeable and analyzable, [57].

As stated in [9], model-based code generators differ from conventional compilers in that for them both source and target languages are executable, i.e. it is possible to compare simulation results with code execution traces. Another difference is that modeling language semantics is often not explicitly defined but is instead embodied in the interpretation algorithms of the the simulator. Finally, direct transformation of hierarchical model structure to syntax tree of the target language is not possible; automatically generated code is based on a sequence of computation derived by analyzing data dependencies of the model. In the same work, the issue of confidence in code obtained through ACG is raised, similar as was for compilers when transition from assemblers to higher languages was taking place. Due to short development cycles and limited user base, validation of automated code generators by exploitation is not applicable. By certifying generators, fitness for purpose can be guarantied but only under strictly defined conditions of use. Third option is code generator testing that must be automated to be feasible because of great number of model variants and frequent new versions.

2.3.5 Testing

It has been reported in [2] that the cost of finding and fixing defects grows exponentially in the development cycle so it is important to start testing as soon as possible, Fig. 2.11. According to [64], up to 50% of the development effort of the critical systems is taken up by the testing process that can be reduced up to 50% by using MBT techniques. The term model based testing is used to describe all testing activities in the context of MBD projects. According to [65], MBT is cheaper, faster and almost as effective in terms of code coverage in comparison with traditional manual testing. As can be seen in Fig. 2.9, testing is deeply integrated in all phases of MBD. This means that MBT cannot be considered separate from the development process and every MBT methodology is in fact more or less MBD methodology.

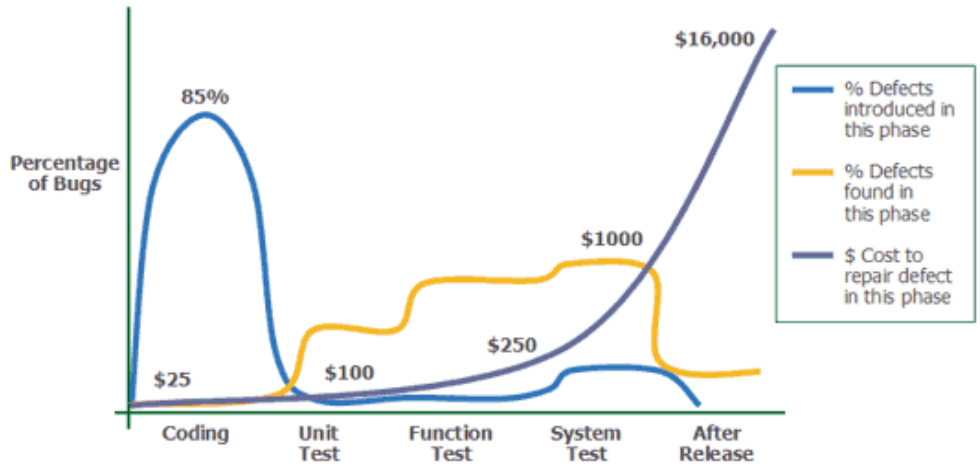


Figure 2.11: Cost of finding and fixing defects grows exponentially in the development cycle, [66]

A generic MBT process can be described in five steps: SUT model creation, definition

of test selection criteria, test case specifications creation, test suite generation and test case execution, [67]. Requirements on the MBT process are enumerated in [23]. Testing should be automated due to interdisciplinary and iterative nature of MBD. To facilitate reuse, tests should be transferable between integration levels and means to compare test results from various levels should be provided. Systematical test design is crucial to achieve satisfactory coverage without redundancy and to keep track of hundreds, or possibly thousands, of test cases. Tests must be readable to allow stakeholders from different domains to participate in testing process. Closed-loop or reactive testing, where test cases are dependent on system behavior, is preferable. To enable tests on target integration levels, timing constraints should be satisfied.

2.4 Test Patterns

Terms "pattern" and "pattern language" were firstly introduces in context of architecture as, [68]:

"The elements of this language are entities called patterns. Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

In software engineering, patterns are regarded as elements of reusability with pattern languages that defines how these elements can be combined, [26]. In contrast to software libraries, that provide predefined reusable elements, software patterns represent abstract solutions for generic problems. The abstraction keeps them customizable, but it means that they need to be instantiated before usage. Patterns also simplify system maintenance because it is easier to identify a known pattern than to investigate unfamiliar system structure.

When test suites for embedded systems are built from scratch, often inadequate methods are used and quality issues arise. This can be avoided by designing tests using test patterns. They capture test design knowledge in a canonical form for use in future similar contexts, [69]. Patterns enable test engineers to concentrate less on test design and more on the properties that need to be tested, they facilitate automation and make test cases readable. Because tests are performed with regard to specification it is opportune to use patterns from beginning of the project, starting with requirements elicitation, [26].

Chapter 3

Model-Based Testing

In this chapter a taxonomy of model-based embedded systems testing is presented and an overview of selected testing methods and tools from academia and from industry, together with their placement inside the taxonomy, is given. Analysis of the investigated MBT tools and methods reveals a gap, so a new testing method is proposed to fill it.

3.1 Model-Based Embedded Systems Testing Taxonomy

A comprehensive taxonomy for MBT is given in [67]. Three general classes: *Model*, *Test Generation*, and *Test Execution* are identified. These classes are divided into categories: the *Model* class categories are *Subject*, *Independence*,*Characteristics*, and *Paradigm*; *Test Generation* is split into *Selection Criteria* and *Technology*; while *Test Execution* is partitioned based on *Execution Options*. In [2], *Test Evaluation* class divided into *Specification* and *Technology* categories is added to the taxonomy.

The taxonomy has been further extended in this work. New classes and options have been emphasized in bold on the diagram in Fig. 3.1. Options used in the new MBT approach, introduced in section 3.3, are shown in frames on the same figure. Options designated by an arrow (↕) represent continuous ranges with border values provided. Mutually exclusive options are designated by "A/B/C" notation and straight lines connect discrete options that do not exclude each other.

3.1.1 Model

Model Subject category determines to which degree the model describes behaviour of the SUT and/or its environment, [67]. Model of the SUT encodes desired behaviour so it can be used as an oracle and its' structure can be utilized during test generation. Model of the environment restricts the set of all possible inputs to the SUT and in this way acts as a test selection criteria. One extreme is the model that fully describes the SUT but knows nothing about its environment so inputs to the SUT are not bounded by such model. Opposite is the model of the environment without information about the SUT. This model specifies all allowed inputs to the SUT but gives no information about expected outputs.

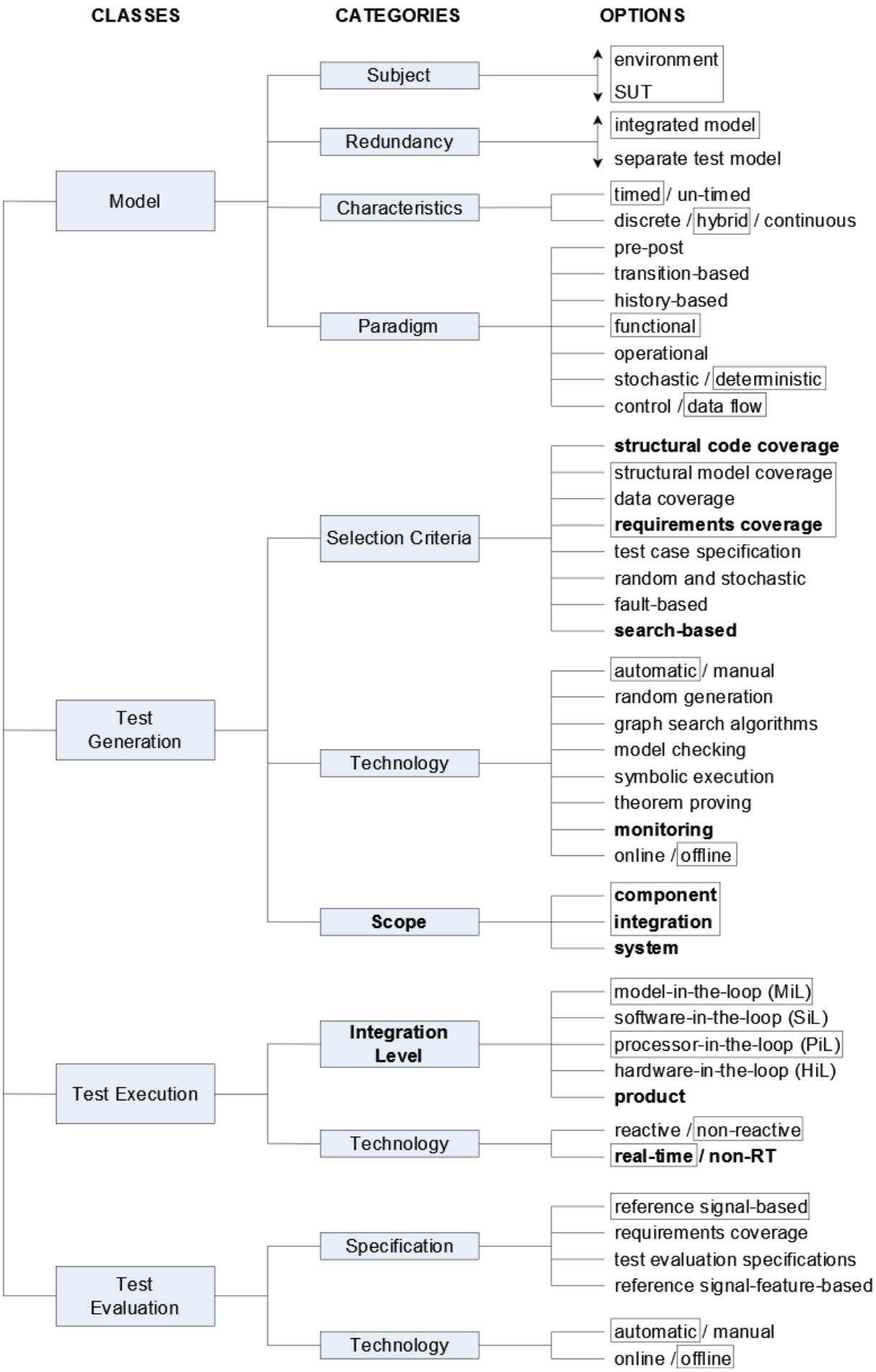


Figure 3.1: Diagram of the MBT taxonomy

During MBD, model can be used for testing and/or implementation so various levels of **redundancy** are possible. In an integrated model scenario, the same model is used for generating implementation and for testing so there is no redundancy. On the other extreme there exist two separate models: one for implementation and one for testing.

Model Characteristics refer to incorporation of timing issues and to the continuous or discrete nature of the model. Timing issues heavily complicate testing of real time systems because time represents additional degree of freedom. Much of the work in MBT is concerned with event-discrete systems, but most embedded systems exhibit continuous or hybrid behaviour. Hybrid systems behaviour consists of time continuous parts, where variable evaluations change with time, and time discrete parts, where events happen and variables are assigned values, [70].

Different modeling notations can be grouped into **Paradigms**:

- In **pre**/**post** (state-based) notations systems are modeled as a collections of variables that represent a snapshot of system's internal states. Operations described by *preconditions* and *postconditions* modify these variables.
- **Transition-based** notations are usually graphical arc-and-node notations that describe transitions between different states of the system, e.g. finite state machines and statecharts.
- **History-based** notations model the system by describing allowable traces of its behaviour over time.
- **Functional** notations describe a system as a collection of mathematical functions.
- **Operational** notations describe a system as a collection of parallel executable processes and are used for distributed systems and communication protocols.
- **Stochastic** notations describe a system by a probabilistic model. They are mostly used for modeling environment, e.g. Markov chains for usage modeling.
- **Data-flow** notations emphasize data flow, rather than **control flow**.

3.1.2 Test generation

Starting points for test case generation are system requirements, test objectives and test specification. Test generation approaches can be grouped according to selection criteria, generation technology and scope, [67]. Different methods can be combined to complement each other to achieve better coverage, [2].

Test selection criteria

Structural model coverage criteria exploit the structure of the model to generate test cases. Many of these criteria are adapted code coverage criteria and are based on control flow through model, same way as code coverage criteria are based on control flow through code. Modeling

notation determines which coverage criteria can be used, e.g. for transition based models some of possible coverage criteria are: all nodes, all transitions and all transition pairs. In [2, 67], only structural model coverage criteria are considered but here taxonomy is extended to include **structural code coverage** criteria. As shown in [71, 72, 73], code coverage criteria can be elegantly applied to MBT. Correlation between model and code coverage has been reported in [74]. Structural criteria are mostly used for generation of additional tests to achieve desired structural coverage after testing according to some other criteria has already been conducted, [75, 76].

Data coverage criteria help to choose test stimuli from large data space by partitioning it into equivalence classes. They are based on the uniformity hypothesis which states that it is enough to test the system with one value from each equivalence class because all elements of the class are "equivalent" in terms of their ability to detect failures. This method is often complemented with boundary analysis, where critical limits of data ranges are determined, [9].

Requirements coverage criteria can be applied if elements of the model can be explicitly associated with informal requirement on the SUT or if there exist formal requirements that can be automatically analysed, as in [54]. Testing should ensure and requirements coverage analysis should prove that requirements are sufficiently covered by test cases, e.g. that every requirement is checked at least once, [9].

Test case specification represents criteria for selecting test cases out of all possible model traces. Specification is written in some formal notation that can be the same as the one used for the SUT model. For example, in [47] tests can be specified by environment model or by a set of constraints. Popular test specification method for discrete systems is sequence-based specification (SBS), [77].

Random and stochastic criteria are usually applied to the environment models so that the generated tests follow an expected usage profile. Statistical usage model is usually combined with deterministic behavioral model so that the statistical model serves as selection criterion that chooses paths and the behavioral model generates oracles for the selected paths, [78, 79].

Search-based criteria also utilize environment model to generate test cases, but here they are traversed based on some search algorithm and not randomly. For example, in [80] Genetic Algorithms and (1+1) Evolutionary Algorithm are used together with four different heuristics and their combinations. Another example is presented in [20], where test cases are generated using Rapidly-exploring Random Tree probabilistic robotic motion planing algorithm guided by equidistribution-based coverage criteria.

Fault-based criteria rely on knowledge of typically occurring faults mostly described by a fault model.

Test generation technology

SUT model can be used for **manual** test case generation or it can partake in **automatic** test generation, if test specification is available. For example, environment model plus some additional constraints can be used as a test case specification and tests can be generated stochastically, by using graph search algorithms, by model checking, by symbolic execution or by theorem proving.

- **Random generation** can be applied to reactive systems for selecting input traces by randomly sampling the input space. Expected outputs are then obtained by applying input traces to SUT model. A random walk may be applied to model of the SUT or to the usage model. These methods are easy to implement but they take a lot of tests to reach satisfiable coverage.
- **Graph search algorithms** include node and arc coverage such as *Chinese Postman* algorithm.
- **Model checking** verifies or falsifies properties of a system. Test case specifications are formulated as reachability properties and model checker generates traces that reach desired states or fire certain transitions.
- **Symbolic execution** means executing a model with sets of input values, represented as constraints, instead with single values. This way symbolic traces are generated that must be instantiated before they can be applied to the SUT. Symbolic execution is guided by test case specification, similar as model checking.
- **Theorem proving** can be used for test case generation similar as in model checking where model checker is replaced with theorem prover. However, theorem provers are mostly used to check satisfiability of formulas that guard transitions in state-based models.
- Application sequences can be obtained by on-line **monitoring** of embedded systems and imported back into MBD environment for testing, [81].

Test cases can be generated online or offline. **Online** test generation is related to reactive test execution (see next subsection) useful for testing non-deterministic systems. Test generator can detect which path the SUT has taken and follow the same path in the model. **Offline** test case generation for non-deterministic systems is difficult and results in test cases that are not sequences but trees or graphs. There are many pragmatic advantages to offline test case generation for deterministic systems: existing test management tools can be used, regression testing is possible, test generation and execution are independent, test set minimization is possible, and tests are generated only once.

Test scope

Test scope category is not specific to MBT but to testing in general. In [2], scope is considered as one of the test dimensions. All other test dimensions are covered in the taxonomy: *Test Goal* dimension is covered by *Selection Criteria* category, *Test Abstraction* dimension is spread onto categories of the *Model* class, *Test Execution Platform* corresponds to *Integration Level* category and *Test Reactiveness* is covered by categories *Technology* of *Test Generation*, *Execution* and *Evaluation* classes. To avoid this sort of repetition, test dimensions are not considered here and test scope is introduced into the taxonomy.

At the scope of **component** or unit testing, the smallest testable part of an application is tested in isolation [82]. For example, a single function can be considered as a component in procedural programming, an entire class can be a component in object-oriented programming,

and in a modeling language based on block diagram notation, individual blocks are components. During **integration** testing, components are combined and tested as a subsystem, not yet as a whole system. Component integration testing exposes defects in interfaces and interaction between integrated components. **System** testing is the process of testing an integrated system to verify that it meets specified requirements, [82]. Integrated system consists of all hardware and software components that constitute final product.

3.1.3 Test execution

In [67], *Test Execution*, class has only *On/Offline* category with an option with the same name. This class is in [2] renamed into *Execution options* and has *MiL/SiL/PiL/HiL* and *Reactive/Non-reactive* options. Placing MiL, Software-in-the-Loop (SiL), Processor-in-the-Loop (PiL) and HiL under same option assumes that a test can be executed on only one level of integration and also omits final level of integration, here named *product*. Both taxonomies ignore possibility to execute test in non-real-time or in real-time environment. For these reasons the *Test Execution* class has been extended to include two categories as shown in Fig. 3.1.

Integration Level

The same tests can be repeated on various integration levels [23], or execution platforms [2], throughout development process. At the **Model-in-the-Loop** level, model of the SUT is tested in an open-loop or in a closed-loop with model of the environment. Various types of models can be tested this way, typically functional and implementational models. Functional models are quite abstract and don't consider aspects such as robustness or performance. During development, functional models are concretized into implementation models that include enough details (e.g. function encapsulation, fixed-point scaling, reuse) for manual or automatic code generation. MiL testing is functional testing, it takes place at early stages of development, and is performed inside a simulation environment.

Program code, typically C code, is on **Software-in-the-Loop** level tested in a simulated environment, [83]. Both simulated environment and SUT code are usually executed on the same machine, e.g. Windows or Linux based desktop computer. This is also functional testing whose goal is finding errors introduced by code generation.

Processor-in-the-Loop testing is similar to SiL testing with the difference that here code is executed on the target processor/controller. These tests reveal faults caused by the target compiler and target processor architecture. Testing on PiL level is important because this is the last integration level which allows for cheap and manageable debugging.

On **Hardware-in-the-Loop** level, software runs on final embedded device and its environment is emulated. Communication is performed through analog and digital interfaces of the device so these tests reveal faults in the device's low-level I/O services. HiL testing requires real-time behaviour of the environment so that the communication with the device is the same as in final application. Some advantages of HiL testing over testing on product level are: HiL tests are repeatable and can be automated and HiL emulators are usually much cheaper than test environments that encompasses entire final product, [84].

Some of the tests designed during previous integration levels can be performed on the final **product** (*car* in [23]). Drawbacks of testing on this level are: tests are performed late in development cycle and are expensive, arbitrary parameter variations are not possible, hardware faults are difficult to trigger and SUT reactions are difficult to observe because internal signals are often inaccessible.

Technology

Reactive/non-reactive test execution is related with online/offline test case generation, see section 3.1.2. During reactive testing signals from SUT or from the testing environment itself are used in online test case generation. This kind of testing is also called closed-loop testing, [85]. In non-reactive tests, the SUT responses have no influence on the test. Reactive testing can be more efficient because errors are handled when they happen, while a non-reactive test set can run for an extended period of time only to discover during post-processing that an error occurred soon after test started and that all subsequent data is worthless.

Tests can be executed in a **real-time** or **non-real-time** environment. In a non-real-time environment, system is tested against requirements that are not time related. Same or different tests can be executed in a real-time environment where real-time properties are checked. Real-time tests can be executed only on PiL, HiL or product integration levels because timing behaviour depends strongly on target architecture.

3.1.4 Test evaluation

Test evaluation is a process of analyzing SUT's output using test oracle and of deciding about the test result. An oracle may be some existing system, test specification or domain expert's knowledge [2].

Specification

Reference signal-based test assessment is performed by comparing the SUT responses with previously defined expected outcomes, or references. These signals may be constructed manually using a signal editor, [86], or they can be results of a simulation, [2].

Requirements coverage evaluation specification criteria aim at covering all informal SUT requirements with appropriate test evaluation scenarios, [21, 86, 87].

Test evaluation specification criterion uses the specification of expected SUT responses. Typically, test engineer defines test case specification for test generation together with evaluation specification, using same formal notation, [71, 87, 88].

Test evaluation based on **reference signal features** assesses the SUT behavior comparing its' responses partitioned into features with the previously specified reference values for those features. A signal feature, or property, is a formal description of certain defined attributes of a signal. It can be used to describe particular shapes of individual signals by providing means to address its' abstract characteristics, for example increase, step response characteristics and maximum are signal features [2]. Such an approach to test evaluation is supported by the Time

Partitioning Test (TPT) [22, 24] and in Model-in-the-Loop for Embedded System Test (MiLEST) [2] methods.

Technology

Automatic/manual test evaluation option relates to evaluation specification, while evaluation execution is presumed to be automatic. For example, in Simulink Verification and Validation, [86], manual specification is supported by defining assertion blocks that automatically evaluate tests during execution.

Online test evaluation happens during test execution and enables execution of reactive tests, [85]. **Offline** evaluation is performed after test execution using recorded SUT inputs and outputs.

3.2 Investigated model based testing Approaches

This section gives an overview of investigated MBT approaches present in industry and academia.

3.2.1 EmbeddedValidator

EmbeddedValidator is used to formally verify TargetLink, [89], design produced from MATLAB/ Simulink/Steteflow model by application of model checking techniques, [21, 90]. It enables robustness analysis, i.e. checks for range violations, automatic debugging, where traces are generated that lead system to user-defined state, and model certification, by application of patterns for formalizing temporal aspects of requirements. Model checking is performed at a high abstraction level, on a model without implementation details. If successful, it results in a so called *Reference-Model* that is used in subsequent MBD phases. Also, model checking requires consistent and unique requirements so any errors in requirements are detected early on.

3.2.2 Classification Tree Method-based approaches

Classification Tree Method (CTM), [91], is used extensively for structured test generation in automotive industry, [92, 93, 94, 95]. Typical application is construction of test vectors where SUT input domain is decomposed into classifications which are further partitioned into equivalence classes based on uniformity hypothesis. The classifications form columns of the combination table while rows of the table represent test steps. Abstract test sequences are specified by selecting classification combinations for each test step, the so called synchronization points. Extension of the method for the embedded systems, called Classification Tree Method for Embedded Systems (CTM/ES), [96], enables automatic instantiation of test sequences by introducing time tags and different interpolation functions for signal values between synchronization points.

MB^3T [93] (Model-Based Black Box Testing) approach establishes stronger link to requirements by formalizing them in the form of tests specified using requirements-based classification

tree (R-CT). Test sequences are generated from the SUT interface using model-based classification tree (M-CT). Requirements coverage is ensured by consistency check in two steps: structure of R-CT and M-CT is compared and requirements-based tests coverage by model-based tests is analyzed.

Gap between requirements and test design is also tackled in [95], where CTM/ES is extended for combined hardware and software functional testing. Constraint-based stimulus patterns applied to CTM enable definition of test sequences directly from requirements. Acceptance criteria, in the form of signal tolerance bands, and functional coverage criteria are also specified using CTM.

Test approach Model Test (MTest), [92], and commercial tool MTest Classic, [97], that implements it, are both based on CTM/ES method. They enable semi-automatic structured functional testing in earlier phases of MBD, namely on MiL and SiL phases. Various classification-tree coverage criteria enable automatic test sequence generation.

To a lesser extent, variants of the CTM are also used in other MBT approaches, e.g. [2, 9, 23].

3.2.3 Reactis Validation Tool

Reactis Validation Tool, [71, 88], is a commercially available tool for testing Simulink/Stateflow models of control systems. It comprises of three main components: Tester, Simulator and Validator. Reactis Tester enables automatic test case generation based on various model and code coverage criteria. It employs *guided simulation* approach that monitors signal values at each branching point in the model during simulation and than performs backward data-flow analysis to choose next input data set to reach untested model branches. Reactis Validator analyses the model with respect to requirements formalized in the form of assertions. It searches through the model using the same *guided simulation* approach and, if any of the assertions is violated, it produces execution trace that leads to the problem.

3.2.4 Simulink Verification and Validation

Simulink Verification and Validation (SL VV), [86], enables checks of compliance with modeling standards, requirements management, automatic test harness generation and component testing. Test harness generation automates interface configuration when isolating atomic subsystem or model referenced by larger system for separate testing and analysis. Test data is imported or created manually. Assertions can be assigned to individual test cases so verification of functional requirements can be performed during model simulation.

3.2.5 Simulink Design Verifier

Simulink Design Verifier (SL DV), [87], provides formal analysis of the model in order to generate test cases or prove properties of the model. Test cases are generated to satisfy chosen structural coverage criteria and can be additionally fine tuned by defining values that signals must assume at least once during test and/or by defining constraints on test signals. Model

properties are defined as logical expressions over signals and if they are violated, counter examples are generated. These are simulation traces that show the violation.

3.2.6 SystemTest

SystemTest, [98], is a MATLAB toolbox that contains templates of standard routines for testing MATLAB algorithms and Simulink models. It can use features of other MATLAB/Simulink extensions, e.g. structural model analysis by SL VV or test distribution by *Parallel Computing Toolbox*, [99]. Test vectors are generated by executing MATLAB expressions or stochastically. Results can be automatically evaluated if reference signals are available.

3.2.7 Time Partitioning Test

TPT is a method, [22, 23], and a commercial tool, [24, 100], for functional model-based testing of embedded control systems. It enables systematic and graphical modeling of test cases by hierarchical and parallel state machines with conditional branching. By representing similar test cases within the same model, redundancies are avoided and clear overview of large number of tests is provided. TPT tests are platform independent so they are easily transferred between development stages and facilitate regression testing. Real-time and reactive testing is supported. Evaluation of test results can be performed on-line by watchdogs modeled using same techniques as for test case modeling. A more powerful off-line test evaluation based on comparison with reference data can be implemented by Phyton scripts supplemented with specialized evaluation library.

TPT allows functional testing of MATLAB/Simulink and ASCET, [101], models and of C-code in a SiL environment. Tests can be executed via TPT's co-simulation environment FUSION in "Windows real-time" where TPT communicates with the target via controller area network (CAN) or local interconnect network (LIN) bus or via INCA, [101], or via CANape, [102], ECU measurement, calibration and diagnostics tools. If tests are translated to byte code, they can be executed on real-time hardware via TPT Virtual Machine (TPT-VM). The TPT-VM is implemented in ANSII-C and has response times in range of microseconds, [24].

3.2.8 MEval

In order to evaluate time-dependent signals in Back-to-Back (B2B) testing of embedded systems a concept for signal comparison and accompanying tool MEval are presented in [103, 104]. B2B is a testing approach where simulation model outputs are compared to measurements taken during system runs. Difference-matrix preprocessing algorithm implemented in this work allows independent evaluation of amplitude deviations and time shifts.

3.2.9 Model-in-the-Loop for Embedded System Test

MiLEST, [2, 105], is a test design methodology applicable during initial phases of design when no referent system responses are available, so no B2B testing is possible. Functional requirements are broken down to signal features and test harness, consisting of test data generator,

SUT, test specification and test control units, is automatically generated. Test evaluation can be automatically specified based on requirements specification and it boils down to signal feature detection. Evaluation is performed online so reactive testing is possible. Testing is in MiLEST supported by a rich and categorized library of test patterns.

3.2.10 Code Generation Tools Testing

An approach to testing of model-based code generator's optimization rules is presented in [9, 72]. It starts with formalization of optimization rules in the form of graph transformation rules. Input domain of the graph transformation rule is partitioned by CTM resulting in test model abstract descriptions. Specialized tool automatically instantiates test models from these descriptions. Code is generated from test models by applying tested code generator and test vectors are created based on structural model and code coverage criteria. Models and the generated code are than fed with the same test vectors and B2B validation of their outputs is performed.

3.2.11 Virtual Test Bed – Real Time extension

Virtual Test Bed – Real Time extension (VTB-RT) is an approach to HiL testing of power electronic systems based on open software and off-the shelf hardware, [25]. Virtual Test Bed (VTB) encompasses simulation of system dynamics and solid modeling of the system. It supports multiformalism, i.e. integration of different simulation environments by cosimulation or translation, and can perform distributed simulation on different platforms with different solvers and step lengths. VTB-RT is VTB's real time extension composed of (i) a Linux distribution that provides user interface and development tools, (ii) *Real-Time Application Interface* that modifies the kernel and enables task execution in real-time, and (iii) VTB solvers and simulation models. Real-time task has higher priority than the kernel and it handles real-time counter. Solvers are implemented as Linux processes that communicate with environment through data acquisition cards and with real-time task through real-time FIFO. Main limitation of VTB-RT is step length which is limited by processing power of the platform used for real-time simulation.

3.2.12 Sequence-based specification

SBS is a set of methods for rigorous software specification where system behaviour is described in terms of next externally observable response to to each sequence of external inputs, [77]. In [28, 78, 79], it has been shown that SBS can be combined with statistical methods to provide MBT approach. In these papers, Markov chain-based usage model is constructed from informal requirements and used for automatic test case generation, execution and evaluation. The approaches tackle discrete system behaviours and are therefore not applicable to hybrid control systems.

3.2.13 VETESS approach

Testing approach developed during Verification of Embedded systems for vehicles using automatic TESt generation from Specification (VETESS) project, [106, 107], utilizes a toolchain comprised of a mix of open-source and proprietary tools for generating functional tests out of Systems Modeling Language (SysML) model of the SUT. SysML is a UML profile for system engineering and in VETESS approach only its SysML4MBT subset is used. The proposed testing process is conducted in three steps: (i) SUT behavior is modeled in SysML4MBT, (ii) abstract test cases are automatically generated out of the functional model, and (iii) abstract test cases are concretized for execution on a specific platform. In [108], drawbacks of this approach are summarized: the tester needs to construct both SysML and MATLAB models, SysML4MBT model is based on discretization of the system and continuous aspects are taken into account late in the process during concretization, and adaptation layer must be updated for each model modification which complicates maintenance. The same work proposes to improve the approach by capturing real-time aspects in SysML4MBT model and by automatically using this information in the toolchain.

3.2.14 MOS

MOS is a tool for integrated model- and search-based testing of software for safety critical systems implemented in programmable logic controllers (PLCs) language Function Block Diagram (FBD), [109, 110]. Inputs to the test generators are the tested FBD and test coverage requirements defined by the user. The tool implements two complementing test generators. Model-based test generator firstly transforms the FBD under test to timed automata model and than uses UPPAAL model checker to explore the symbolic state space of that model and generate time-optimal traces based on a variation of the A*-algorithm. Search-based test generation is performed based on execution of C code generated from the tested FBD in a desktop computer environment that simulates PLC peripherals. Test data that satisfies modified condition/ decision coverage is generated using slightly modified hill climbing algorithm.

3.2.15 Hybrid systems Test Generation

Hybrid systems Test Generation (HTG) is a tool that implements formal model-based hybrid system testing method, [20]. Tested hybrid systems are here modeled by hybrid automata and tests are generated using Rapidly-exploring Random Tree algorithm, a probabilistic motion planning technique from robotics domain, which is guided by a novel test coverage criteria based on equidistribution degree of a set of states over the state space. The approach has been validated on a number of case studies from control systems domain, some with few hundreds of continuous variables, and on case studies concerning mixed signal electronic circuits.

3.2.16 ETAS RT2

RT2 is a commercial MiL (for ASCET and Simulink models) and SiL testing tool by ETAS Group, [111]. Tests are modeled by hierarchical automatons which support variations of states

and transitions reducing thus redundancies in test design. The tool supports both continuous and discrete signals and various ways of their definition and evaluation. Reactive testing and parallel execution of multiple automaton are also supported.

3.2.17 Safety Critical Application Development Environment Suite

Safety Critical Application Development Environment (SCADE) Suite is a model-based development environment for safety critical applications produced by Esterel Technologies, [112, 113]. It is based on Lustre, a formally defined, deterministic, and data-flow-based language for programming reactive systems. SCADE Suite also supports integration of state machines into data flow designs, automatic generation of certifiable C code, and graphical simulation based on the generated code. The suite incorporates several MBT tools:

- *SCADE Suite Model Test Coverage* enables analysis of thoroughness of high-level requirements-based test cases and assessment of the role of each test case in covering operator instances of the model, [114];
- *Design Verifier* verifies safety properties expressed in SCADE Suite and automatically generates counter examples in case of failure, [115, 116];
- *LabVIEW Gateway* integrates SCADE Suite with National Instruments LabView via its' software environment for configuring real-time testing applications VeriStand and enables thus HiL testing;
- *Timing and Stack Verifier* provides WCET and stack size analysis for a specific hardware platform.

3.3 Filling the gap

3.3.1 The Gap

Investigated approaches and novel MoBREST methodology have been placed in the proposed MBT taxonomy in tables 3.1, 3.2, and 3.3. Table 3.1 is concerned with *Model* class of the taxonomy, table 3.2 with *Test Generation* class, and table 3.3 with *Test Execution* and *Test Evaluation* classes.

Table 3.1: Positioning of investigated MBT approaches with respect to Model *class of the proposed taxonomy*

No.	Approach	Model			
		Subject	**Redundancy**	**Characteristics**	**Paradigm**
1	EmbeddedValidator, [21, 90]	environment and SUT	integrated	timed, hybrid	history based, functional, deterministic, data flow and transition-based
2	MTeset (CTM/ES), [92, 97]	environment and SUT	integrated	timed, hybrid	history based, functional, deterministic, data flow
3	Reactis Validator, [71, 88]	environment and SUT	integrated	timed, hybrid	history based, functional, deterministic, data flow
4	Reactis Tester, [71, 88]	environment and SUT	integrated	timed, hybrid	history based, functional, deterministic, data flow and transition-based
5	SL VV, [86]	environment and SUT	integrated	timed, hybrid	history based, functional, deterministic, data flow
6	SL DV, [87]	environment and SUT	integrated	timed, hybrid	history based, functional, deterministic, data flow

Table 3.1: Continued from previous page...

No.	Approach	Model			
		Subject	**Redundancy**	**Characteristics**	**Paradigm**
7	System Test, [98]	environment and SUT	integrated	timed, hybrid	history based, functional, deterministic, data flow
8	TPT, [22, 23]	environment	separate	timed, hybrid	transition-based
9	MEval, [103, 104]	not applicable (N/A)	N/A	N/A	N/A
10	MiLEST, [2, 105]	environment and SUT	integrated	timed, hybrid	history based, functional, deterministic, data flow and transition-based
11	Code Generation Tools Testing, [9, 72]	N/A	N/A	N/A	N/A
12	VTB-RT, [25]	SUT	integrated	timed, hybrid	N/A
13	SBS, [77, 28, 78, 79]	N/A	N/A	N/A	N/A
14	VETESS, [106, 107, 108]	SUT, environment	integrated	un-timed, discrete	transition-based, functional, deterministic
15	MOS, [109, 110]	SUT	separate	un-timed, discrete	transition-based, deterministic
16	HTG, [20]	SUT	integrated	un-timed, hybrid	transition-based

Table 3.1: Continued from previous page...

No.	Approach	Model			
		Subject	**Redundancy**	**Characteristics**	**Paradigm**
17	ETAS RT2, [111]	environment and SUT	separate	timed, hybrid	transition-based, history-based, functional, deterministic, data flow
18	SCADE Suite, [112, 113]	SUT	integrated	timed, hybrid	deterministic, data flow and transition-based
19	MoBREST	environment and SUT	integrated	timed, hybrid	history based, functional, deterministic, data flow

Table 3.2: Positioning of investigated MBT approaches with respect to Test Generation *class of the proposed taxonomy*

No.	Approach	Test Generation		
		Selection Criteria	**Technology**	**Scope**
1	EmbeddedValidator, [21, 90]	test case specification	automatic, offline, model checking	integration
2	MTeset (CTM/ES), [92, 97]	test case specification	manual, offline	system
3	Reactis Validator, [71, 88]	functional requirements coverage	offline	system
4	Reactis Tester, [71, 88]	structural code and model coverage, data coverage	offline	component, system
5	SL VV, [86]	structural model coverage	manual	component
6	SL DV, [87]	structural model coverage, data coverage, test case specification	automatic, offline, theorem prover	component
7	System Test, [98]	stochastic	automatic, offline	component, system
8	TPT, [22, 23]	test case specification, data coverage, requirements coverage	manual and automatic, offline and online	component, integration, system
9	MEval, [103, 104]	N/A	N/A	N/A
10	MiLEST, [2, 105]	data coverage, requirements coverage, test case specification	offline	component, integration

Table 3.2: Continued from previous page...

No.	Approach	Test Generation		
		Selection Criteria	**Technology**	**Scope**
11	Code Generation Tools Testing, [9, 72]	structural code coverage, structural model coverage, test case specification	automatic, offline	system
12	VTB-RT, [25]	N/A	N/A	integration, system
13	SBS, [77, 28, 78, 79]	test case specification, random and stochastic	automatic, offline	integration
14	VETESS, [106, 107, 108]	structural model coverage	automatic, offline	integration, system
15	MOS, [109, 110]	test case specification, structural code coverage	automatic, model checking	integration
16	HTG, [20]	structural model coverage, search-based	automatic, graph search algorithms, offline	integration
17	ETAS RT2, [111]	test case specification	manual/automatic, offline	integration
18	SCADE Suite, [112, 113]	structural model coverage, requirements coverage, test case specification	automatic and manual, model checking, offline	integration
19	MoBREST	data coverage, requirements coverage	automatic and manual, offline	component, integration

Table 3.3: Positioning of investigated MBT approaches with respect to Test Execution *and* Test Evaluation *classes of the proposed taxonomy*

No.	Approach	Test Execution		Test Evaluation	
		Integration Level	**Technology**	**Specification**	**Technology**
1	EmbeddedValidator, [21, 90]	MiL	non-reactive, non-real-time	requirements coverage	manual, offline
2	MTeset (CTM/ES), [92, 97]	MiL, SiL, PiL	non-reactive, non-real-time	reference signal-based	manual, offline
3	Reactis Validator, [71, 88]	MiL	non-reactive, non-real-time	test evaluation specifications	manual, offline
4	Reactis Tester, [71, 88]	MiL, SiL	non-reactive, non-real-time	reference signal-based	manual, offline
5	SL VV, [86]	MiL	non-reactive, non-real-time	requirements coverage	manual, online
6	SL DV, [87]	MiL, SiL	non-reactive, non-real-time	test evaluation specification	manual, online
7	System Test, [98]	MiL, SiL	non-reactive, non-real-time	reference signal-based	manual, offline
8	TPT, [22, 23]	MiL, SiL, PiL, HiL	reactive, real-time	reference signal feature-based	manual, online and offline
9	MEval, [103, 104]	MiL, SiL, PiL, HiL, product	non-reactive, non-real-time	reference signal-based	offline

Table 3.3: Continued from previous page...

No.	Approach	Test Execution		Test Evaluation	
		Integration Level	**Technology**	**Specification**	**Technology**
10	MiLEST, [2, 105]	MiL	reactive, non-real-time	reference signal-feature-based, requirements coverage, test evaluation specifications	automatic and manual, online
11	Code Generation Tools Testing, [9, 72]	SiL	non-reactive, non-real-time	reference signal-based	automatic, offline
12	VTB-RT, [25]	HiL	reactive, real-time	reference signal-based	online
13	SBS, [77, 28, 78, 79]	N/A	N/A	N/A	N/A
14	VETESS, [106, 107, 108]	SiL, PiL	non-reactive, non-real-time	reference signal-based	manual, offline
15	MOS, [109, 110]	MiL, SiL	non-reactive, non-real-time	reference signal-based	offline, automatic
16	HTG, [20]	MiL	non-reactive, non-real-time	test evaluation specifications	automatic
17	ETAS RT2, [111]	MiL, SiL	reactive, non-real-time	reference signal-based, test evaluation specification	automatic, online
18	SCADE Suite, [112, 113]	SiL, MiL, HiL	non-reactive, non-real-time	requirements coverage, test evaluation specification	manual, offline

Table 3.3: Continued from previous page...

No.	Approach	Test Execution		Test Evaluation	
		Integration Level	**Technology**	**Specification**	**Technology**
19	MoBREST	PiL	non-reactive, real-time	reference signal-based	automatic, offline

This analysis shows that there is very little support in investigated tools and methods for real-time Processor-in-the-Loop unit and integration testing of hybrid embedded control systems. The TPT approach comes closest to this goal, but it has three major drawbacks: (i) TPT-VM is written in C language and it is difficult to integrate it with custom RTOS written in assembly language, (ii) the tool has response time in microseconds but in real-time component testing nanosecond scale is often interesting, and (iii) the method does not support real-time integration testing in a closed loop. This gap is filled with novel MBT method MoBREST, shortly introduced in next subsection and elaborated in the chapters 5 through 9.

3.3.2 The Filling

The MoBREST is a novel approach to model-based development and testing of complex control algorithms for embedded systems with severe time constraints, e.g. regulation of power converters. The method assumes that SUTs are built by integrating individual well defined components using graphical block diagram based modeling language. MoBREST method covers two scopes of testing: component or unit testing and integration testing by its MoBREST-CT and MoBREST-IT subsets, respectively.

The method is implemented by tool with the same name inside MATLAB/Simulink environment. The MoBREST tool assumes existence of executable specification in the form of MATLAB/Simulink model. The model can be verified against requirements using model checking techniques, e.g. as in [21], or extensive functional testing, e.g. by applying test patterns as in [2] or by using TPT [22, 23, 24]. Verified functional model, called *Reference-Model* in [21], is then used as test oracle during PiL B2B testing.

Chapter 4

Model Based Development with Legacy Components Integration

Model Based Development process must be backed by appropriate tool support. The environment for MBD of real-time embedded control software should

- be flexible, modular, and extensible,
- provide graphical modeling environment,
- provide simulation capabilities, and
- enable integration of legacy software components.

The in-house development of entire model-based toolchain for the proprietary development environment introduced in section 1.1 would be exceedingly expensive, so integration of legacy tools and processes into an off-the-shelf environment has been evaluated. MATLAB tool family by MathWorks has been chosen because it fulfills most of the said requirements and also because:

- it is widely used in academia as well as in industry – according to [2], about 50% of functional behaviour of embedded systems is modeled using this environment,
- it is easily extensible by a variety of toolboxes,
- Simulink toolbox enables creation of graphical block diagrams and their simulation,
- powerful scripting language enables user-built extensions,
- there exists prior in-house experience with this family of products,
- in [5] it is shown how MATLAB can be used to create a complete MBD environment for industrial embedded systems.

MATLAB/Simulink falls short on the integration of legacy assembly software components so the environment has been extended by GRAPlab toolbox, developed in the scope of this research. The toolbox consists of Simulink blockset and an automatic code generation tool that

enables automatic generation of "GRAP code" out of Simulink models. In essence, Simulink has been used as a replacement for GRAP in building graphical modules, their compilation and linking into executable applications. The approach has been verified for C2000 family of Texas Instruments' DSCs and for 8051 family of Silicon Laboratories' microcontrollers, but can easily be expanded to other GRAP-supported architectures of microprocessors, microcontrollers and signal processors.

4.1 The blockset

Each block in GRAPlab Simulink blockset consists of:

- Simulink masked atomic subsystem,
- definition of parameter dependencies in the *dependencies file*, and
- initialization function.

Block masks are in Simulink's help documentation defined as: *"Masks are custom interfaces you can apply to Simulink blocks. A mask hides the user interface of the block, and instead displays a custom dialog control for specific parameters of the masked block".* Block masks cannot be saved separately from the block that it masks and it cannot be applied to more than one block, [117]. This drawback of the Simulink's block masking approach hinders mask reuse and is alleviated in GRAPlab toolbox by the system of parameter dependencies and initialization functions.

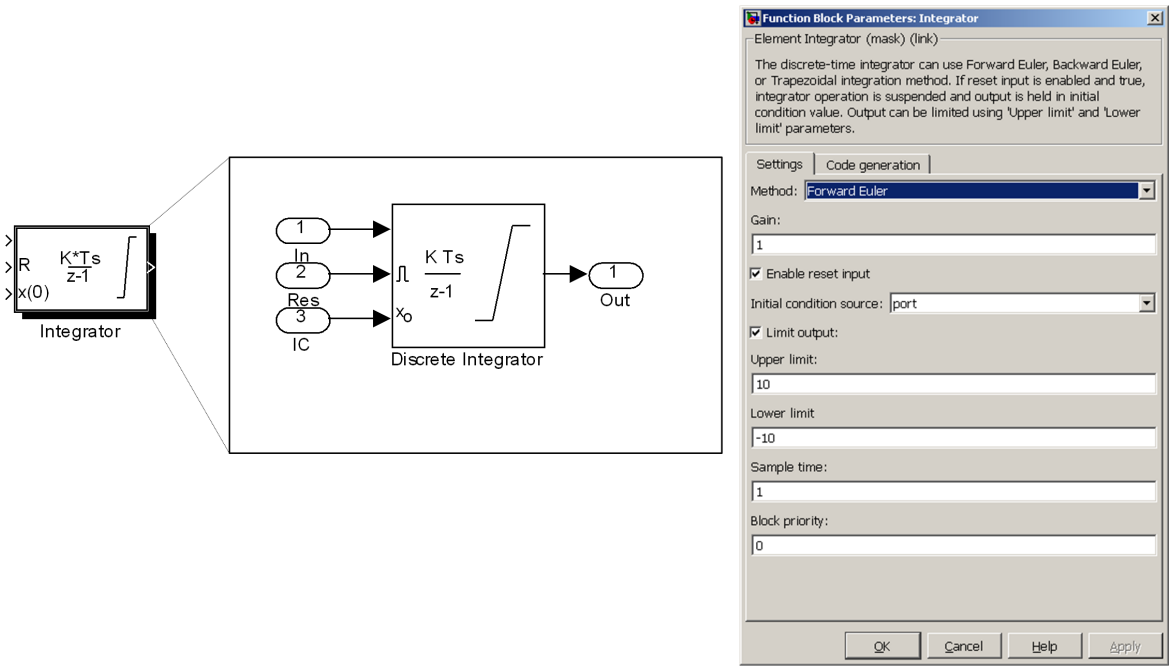


Figure 4.1: Integrator *block from GRAPlab blockset*

In GRAPlab blockset each block is a masked atomic subsystem that implements functionality of one or more GRAP elements in a way to emulate the corresponding GRAP block(s)

functionality as close as possible. This ensures that the model during simulation behaves much the same as the target code generated from it during execution on the target. Subsystem's settings are modified through dialog window whose layout is determined by the relations between different settings, as defined in dependency file. Based on the chosen settings, initialization function modifies the subsystem's content if necessary.

Some of the GRAPlab blocks are native Simulink built-in blocks wrapped inside masked atomic subsystems, for example *Integrator* blocks shown in Fig. 4.1. In such cases mask parameters map directly to underlying block's parameters and wrapping is necessary only because of structure of the code generation procedure.

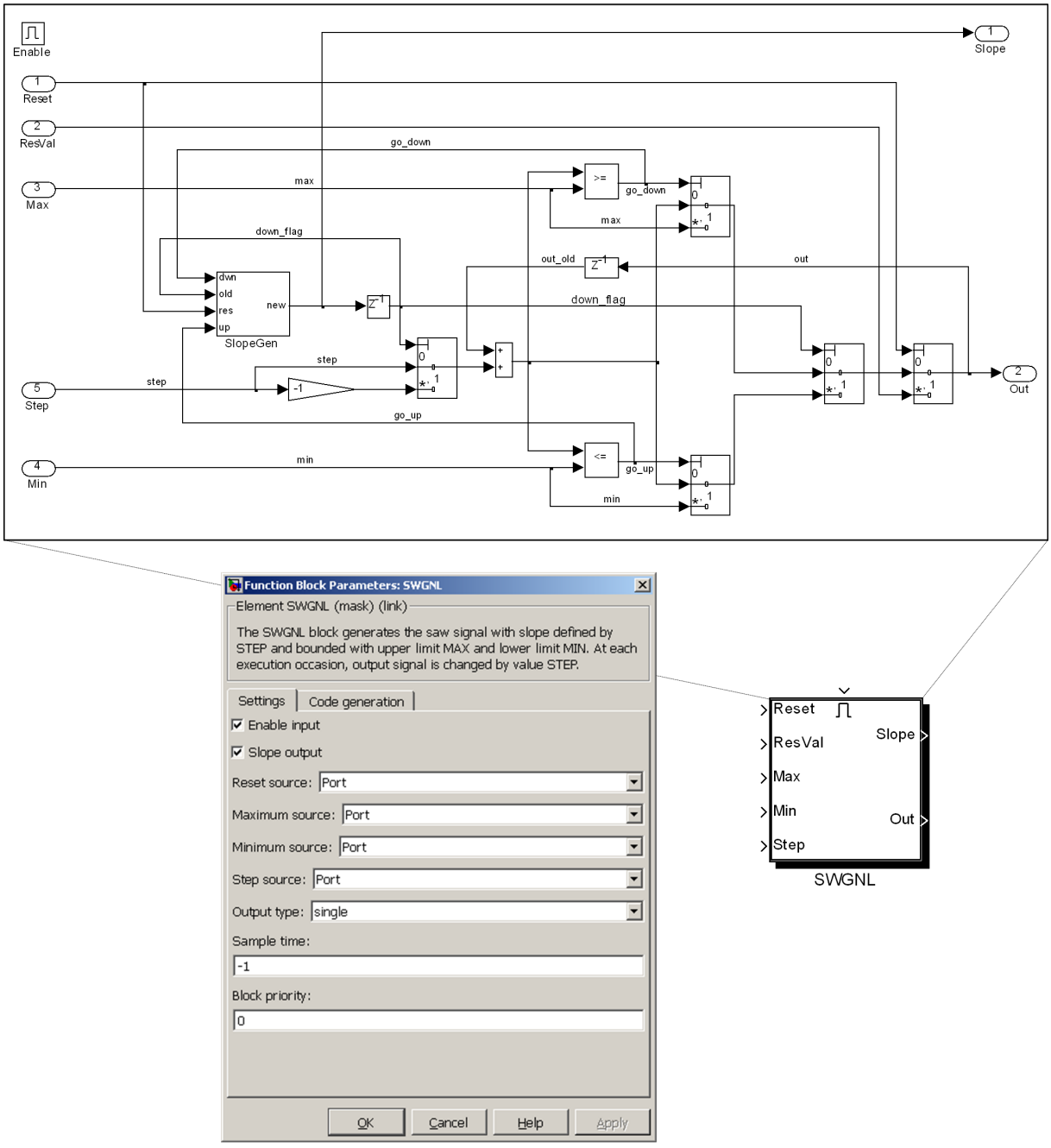


Figure 4.2: SWGNL *block from GRAPlab blockset*

Other blocks are implemented by a more complex composition of various Simulink built-in blocks. An example is given in Fig. 4.2. Here *SWGNL* block that generates saw signal is shown. User defined blocks can also be implemented by masked M-file S-function blocks, [118]. The two approaches were validated by comparing subsystem-type blocks MEMPLAY and RECORDER, and their functional M-file S-function counterparts MEMPLAYS and RECORDERS. The two blocks are used for playback of signals from memory and for recording signal waveforms into target memory, respectively. Although writing M-file block from scratch offers greater freedom in design, subsystem-type blocks have shown to be much faster during simulation so this was the preferred choice.

The contents of the block's subsystem as well as the layout of its' dialog window depend on the chosen settings. When a setting is changed a callback function is invoked that analyzes block's dependency file, adjusts dialog layout accordingly and modifies subsystem content if necessary. The structure of the dependency definitions is illustrated with equation 4-1. For example, if on the dialog in the figure 4.2 *Dialog* option is chosen from the *Maximum source* parameter's drop-down menu, than a *Maximum value* field is revealed on the dialog, as can be seen on Fig. 4.3. The two precondition-postcondition pairs that govern *Maximum value* field visibility based on *Maximum source* selection are given in equation 4-2. Here the *Value* field of *Maximum value* postcondition is not defined because it is not modified by the *Maximum source* setting.

$$
\begin{aligned}
&if PRECONDITION_1 \; than \; POSTCONDITION_1 \\
&if PRECONDITION_2 \; than \; POSTCONDITION_2 \\
&\qquad\qquad ... \\
&if PRECONDITION_N \; than \; POSTCONDITION_N \\
&\qquad \text{where} \\
&PRECONDITION_n.Name = ParameterName \\
&PRECONDITION_n.Value = ParameterValue(s) \\
&\qquad \text{and} \\
&POSTCONDITION_n.Name = ParameterName \\
&POSTCONDITION_n.Value = ParameterValue \\
&POSTCONDITION_n.Enabled =< 1/0 > \\
&POSTCONDITION_n.Visible =< 1/0 >
\end{aligned}
\tag{4-1}
$$

$$
\begin{aligned}
&PRECONDITION_1.Name = "Maximum\ source"\\
&PRECONDITION_1.Value = "Dialog"\\
&POSTCONDITION_1.Name = "Maximum\ value"\\
&POSTCONDITION_1.Enabled = 1\\
&POSTCONDITION_1.Visible = 1\\
&\qquad\qquad \text{and}\\
&PRECONDITION_2.Name = "Maximum\ source"\\
&PRECONDITION_2.Value = "NULL"\ or\ "Port"\\
&POSTCONDITION_2.Name = "Maximum\ value"\\
&POSTCONDITION_2.Enabled = 0\\
&POSTCONDITION_2.Visible = 0
\end{aligned}
\tag{4-2}
$$

Based on options selected from the dialog and/or set by the dependency rules, initialization function modifies block's subsystem content if necessary. For example, default settings and content of the *SWGNL* block are shown in Fig. 4.2. When settings are changed by the user, layout of the block's dialog is adjusted, subsystem content is modified and block's graphical symbol is updated. An example of impact of settings modification on SWGNL block's subsystem content, dialog and symbol is shown in figure 4.3. When *Maximum source* parameter value is changed from *Port* to *Dialog*, following changes to the block occur:

- *Max* block of the *Inport* type inside SWGNL block's subsystem is replaced by the *Constant* block with the same name, detail 1 on Fig. 4.3.
- *Maximum value* edit field on block dialog is shown, detail 2 on Fig. 4.3.
- Block's graphical symbol is updated, detail 3 on Fig. 4.3.

The approach with parameter dependency definitions enables reuse of block subsystems for different target systems. A block's subsystem can be defined in one Simulink (sub)library and than reused as a link in (sub)libraries corresponding to other target systems. To each instance of the block, a parameter dependency definition file is assigned that customizes dialog layout according to the target system. This way, any changes to the block's masked subsystem can be made on the original instance and are automatically propagated to all linked instances. An example can be seen in Fig. 4.4. In C2000 environment, *Summation* block can have 2, 3, or 4 inputs and data type can be chosen between *int16*, *int32* or *single*. On the other hand, in 8051 environment the same block can have only 2 inputs and data type is always *int16* so these parameters can't be modified.

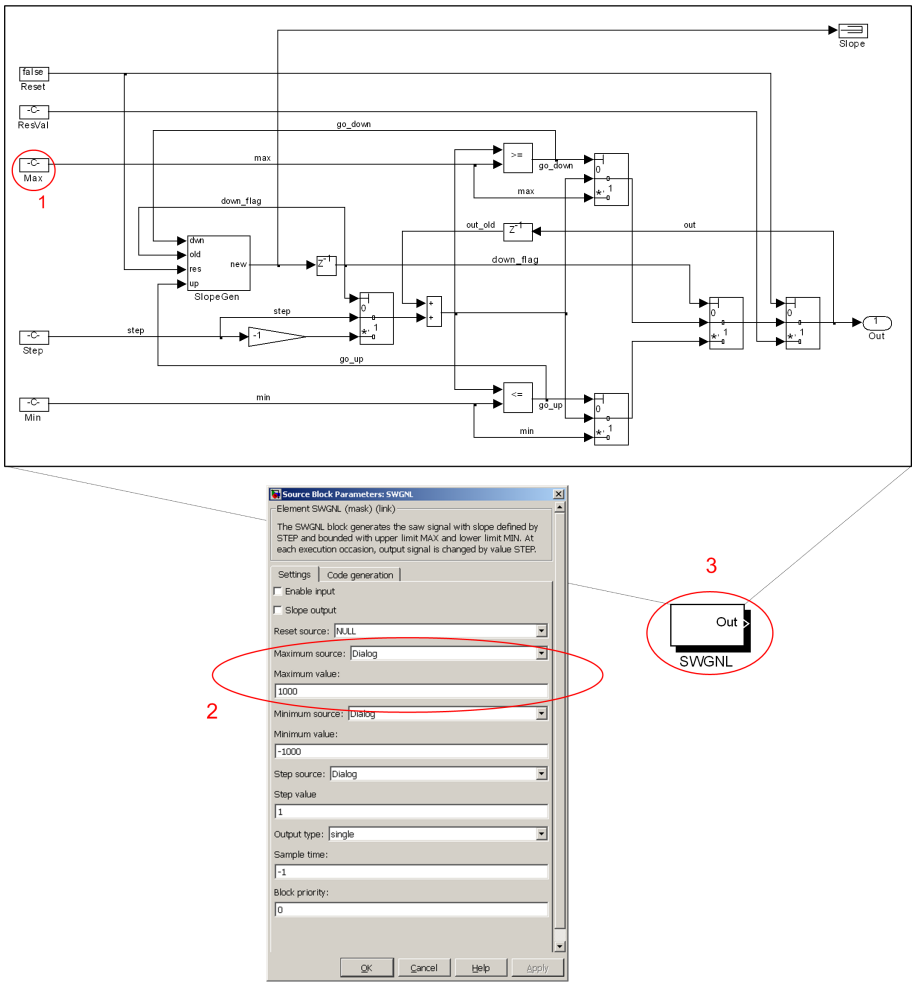


Figure 4.3: SWGNL *block with changed settings*

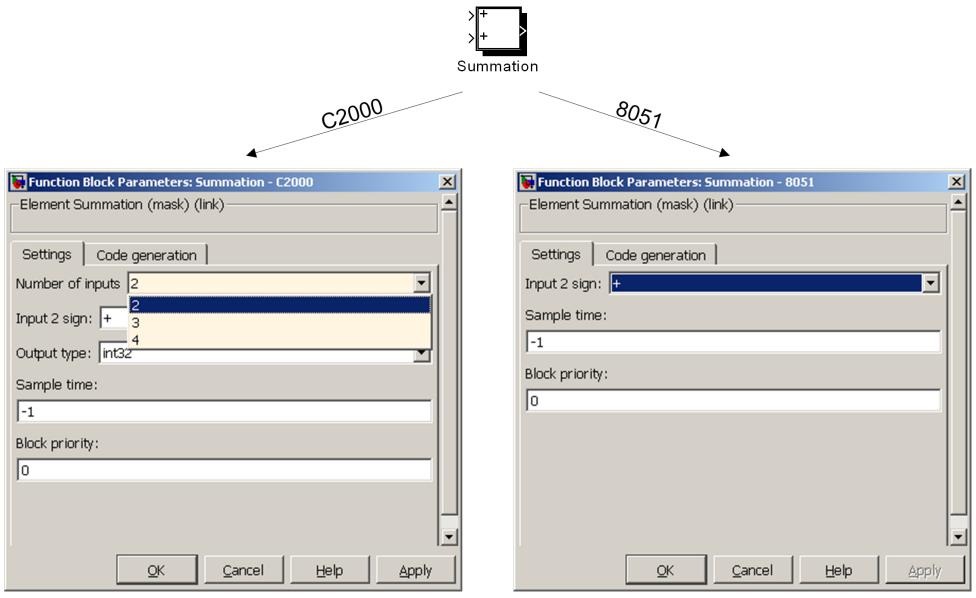


Figure 4.4: Dependency file enables block interface customization for different target systems

4.2 The model

Simulink models compatible with ACG procedure that integrates legacy software components can be constructed using blocks from the GRAPlab blockset. Code generation procedure introduces some additional recommendations and restrictions on the model:

- Fixed-step discrete solver should be used.
- Block and line names must not contain space characters and should be eight or less characters long.
- One-level deep partitioning of the model into atomic subsystems is supported.
- *Outport-Inport* block pairs that pass signals between subsystems must be named and have the same name.

GRAPlab blockset is designed to model GRAP legacy software components library so it doesn't contain continuous blocks. That is why it only makes sense to use discrete solvers in these models. Also, if simulation traces are to be compared with automatically generated code traces, than fixed-step solver should be used. Target application programs always execute with fixed task execution periods so fixed-step solver removes the need for time scaling and aligning of the two traces.

During code generation, target application signal names are constructed out of Simulink model's block or line names. Target application signal names in the proprietary environment cannot contain spaces and are at most eight characters long. If signal name is constructed from block/line name that is longer than eight characters, than the block/line name is truncated to eight characters. This could lead to ambiguities and errors due to multiply defined signal. For example, if two source blocks in Simulink model are named *InputVoltage1* and *InputVoltage2*, than both model signals will be mapped to application signal named *InputVol* and this will produce error during code generation.

Organization of the model into subsystems is supported only with atomic subsystems on root level of the model. This significantly simplifies mapping of the model structure onto target application structure during code generation: each atomic subsystem on model root level is mapped onto one composite component (program module) of the application program. Subsystems output and input signals using *Outport* and *Inport* block, respectively, but the connection between subsystems is implemented with lines on model's root level. On the other hand, GRAP application connects composite components using *external definition* and *external reference* atomic components that are associated by textual label, i.e. signal name. During code generation *Outport* and *Inport* blocks are mapped to *external definition* and *external reference* atomic components with textual labels same as respective block name. Thus, connected *Outport* and *Inport* blocks must have the same name.

If the code is to be generated from the whole model, than the whole model must/should comply to the stated restrictions/recommendations. An example of such models is given in Fig 4.5. This model imports previously measured acceleration waveform and integrates it twice to produce acceleration velocity and displacement. Calculated signals are compared to referent velocity and displacement waveforms, [119].

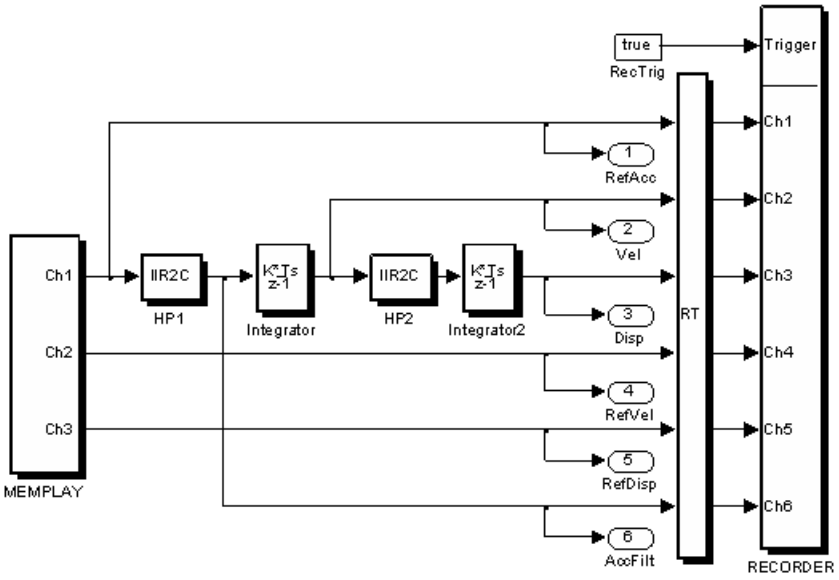


Figure 4.5: Simulink-GRAPlab model ready for automatic code generation

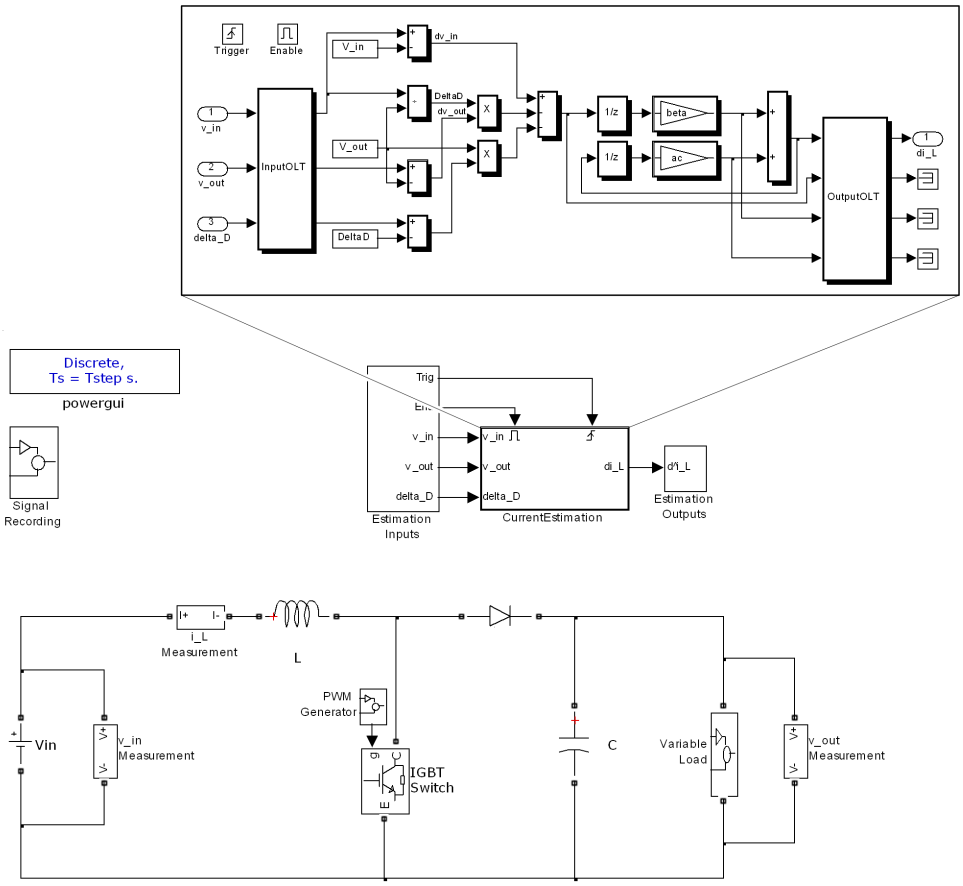


Figure 4.6: Simulink model with GRAPlab subsystem ready for code generation

The target code can also be generated from only a part of the Simulink model. In this case, the subsystem that is to partake in code generation must comply to GRAPlab's restrictions. This is shown on an example of DC-DC boost converter with inductor current estimator given in Fig. 4.6, [120]. The code is generated from the *CurrentEstimator* subsystem.

4.3 The code

Code generation is conducted in two steps: firstly Simulink model is translated into GRAP source files and secondly listing, object and executable code files are generated. The second step is preformed in the same manner as during code generation from GRAP (section 1.1), i.e. the same software tools are used with the difference that they are here invoked from inside MATLAB environment. The crucial step is thus the creation of source files out of Simulink model to feed the code generation software tools.

4.3.1 Simulink to GRAP conversion

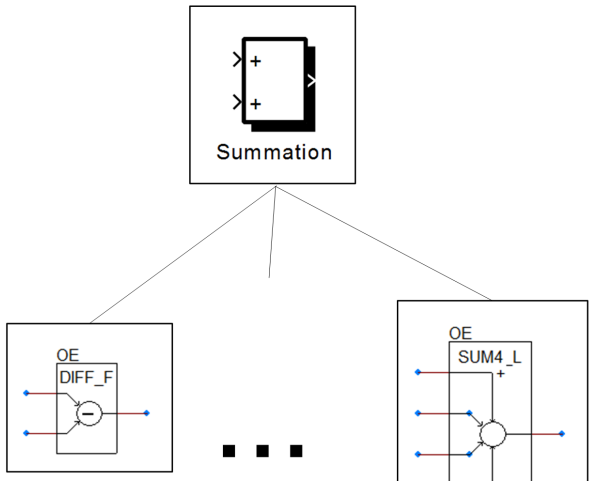
The conversion of GRAPlab-compatible Simulink model (or subsystem) into GRAP source files is performed in following steps:

1. **Get blocks** – Simulink model is analyzed and all information necessary for subsequent steps are extracted.
2. **Group blocks** – Blocks are grouped into program modules according to the model layout.
3. **Purge and order blocks** – Blocks that don't partake in code generation (e.g. *Scope* and *Terminator* blocks) are removed from the list and the remaining blocks are reordered according to GRAP composite component layout conventions.
4. **Resolve line names** – Line name resolution is necessary because GRAP passes signals between assembly macros by referencing them with signal names. If a signal in an application program is not named, GRAP assigns it a generic name. Simulink, on the other hand, connects blocks using tree-like line structure with branches and numerical designators. The conversion utility maps Simulink line with all its branches onto a signal that is named based on source or destination block name, parent line name or parent line designator.
5. **Construct macro calls** – A conversion function is assigned to each GRAPlab block that generates a macro call based on: block settings, block mappings definition and target system. In this step, such function is called for every GRAPlab block in the model.
6. **Write source files** – At the end of the conversion process, module source files and application definition file are created. These files are inputs for further code generation that results in executable files.

4.3.2 Block to component mapping

Each GRAPlab block has a mapping file that defines to which legacy component the block is mapped during code generation. Separate block mapping definition is provided for each supported embedded target. Mapping files are structured as parameter name-value pairs. If block parameters are the same as parameters of a mapping rule, than block is in generated code represented by atomic component defined by that mapping rule. Fig. 4.7 shows an examples of mapping definitions for *Summation* GRAPlab block. Based on the embedded target and the block parameters, this block can be mapped onto a number of atomic components. In the example, mapping rules for *DIFF_F* and *SUM4_L* atomic components are shown:

- *Summation* block is mapped onto *DIFF_F* atomic component if data type is "single", if it has two inputs, and if the second input has a minus sign.
- *Summation* block is mapped onto *SUM4_L* atomic component if data type is 32-bit integer, if it has three or four inputs, and if input signs for second, third and forth inputs are as stated in Fig. 4.7.




```
Mappings(1).Name = 'DIFF_F';
Mappings(1).Parameters(1).Name = 'OutType';
Mappings(1).Parameters(1).Value = {'single'};

Mappings(1).Parameters(2).Name = 'NumIn';
Mappings(1).Parameters(2).Value = {'2'};

Mappings(1).Parameters(3).Name = 'In2Sgn';
Mappings(1).Parameters(3).Value = {'-'};
```

```
Mappings(6).Name = 'SUM4_L';
Mappings(6).Parameters(1).Name = 'OutType';
Mappings(6).Parameters(1).Value = {'int32'};

Mappings(6).Parameters(2).Name = 'NumIn';
Mappings(6).Parameters(2).Value = {'3', '4'};
Mappings(6).Parameters(2).TestVals = {'all'};

Mappings(6).Parameters(3).Name = 'In2Sgn';
Mappings(6).Parameters(3).Value = {'+', '-'};
Mappings(6).Parameters(3).TestVals = {'all'};

Mappings(6).Parameters(4).Name = 'In3Sgn';
Mappings(6).Parameters(4).Value = {'+', '-'};
Mappings(6).Parameters(4).TestVals = {'all'};

Mappings(6).Parameters(5).Name = 'In4Sgn';
Mappings(6).Parameters(5).Value = {'+', '-', 'NULL'};
Mappings(6).Parameters(5).TestVals = {'all'};
```

Figure 4.7: Example of mappings definition for Summation *GRAPlab block*

Mappings definitions can have *TestVals* field that is used during automatic component testing and is explained in chapter 5.

4.3.3 Code generation customization

Mapping and dependencies definition files are specified on *Code generation* tab of the block's dialog window. These fields are set by default to point to respective blockset's definition files, but can be changed by the advanced user that wishes to customize block layout and/or its mapping during code generation. On the same tab, mapping can be forced to user specified value via *Force mapping* field. For example, *Summation* block that adds two 32-bit integer signals will by default be mapped to *SUM2_L* atomic component. By setting *Force mapping* field value to *SUM4_L*, user can force code generation procedure to implement this block with *SUM4_L* atomic component whose two inputs will be left unused. A callback function checks *Force mapping* field and reports error if invalid value is specified, e.g. if floating-point atomic component *SUM2_F* is forced for summation of integer signals.

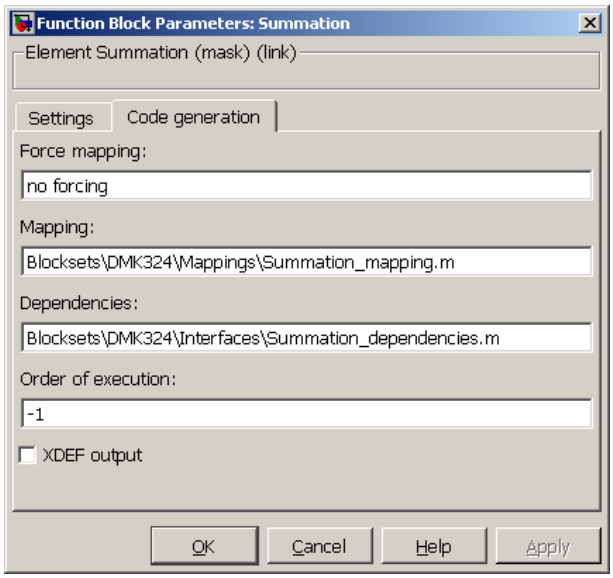


Figure 4.8: Code generation *tab of the GRAPlab block dialog window*

User can also manually specify execution order of the component(s) to which block is mapped via *Order of execution* field. Default value of "-1" means that execution order of the component(s) will be automatically determined by the code generation tool based on the sorted order of the Simulink block.

XDEF output checkbox causes all block's output signals to be "externally defined", i.e. to be visible to the user during generated code execution on the target system. Some of the signals will be externally defined by default, e.g. signals that cross subsystem boundaries and all signals connected to *Outport* blocks. The *XDEF output* option enables user to get better insight into generated code during its' execution on embedded target which is especially useful during application debugging.

Switching between automatic and manual execution order setting can be performed on a subsystem (i.e. composite component) level by placing the *ModConf* GRAPlab block inside the

subsystem. If *manual* is selected in the *Order of execution* drop-down menu, than during code generation *Order of execution* settings of all blocks in the subsystem are taken into account and every block displays the setting in an annotation shown beneath its' name, Fig. 4.9. If *automatic* option is selected, than *Order of execution* of all blocks in the subsystem is set to default value of "-1", annotations are not displayed and previous manual setting of each block is saved so that it could be restored by selecting *manual* option again.

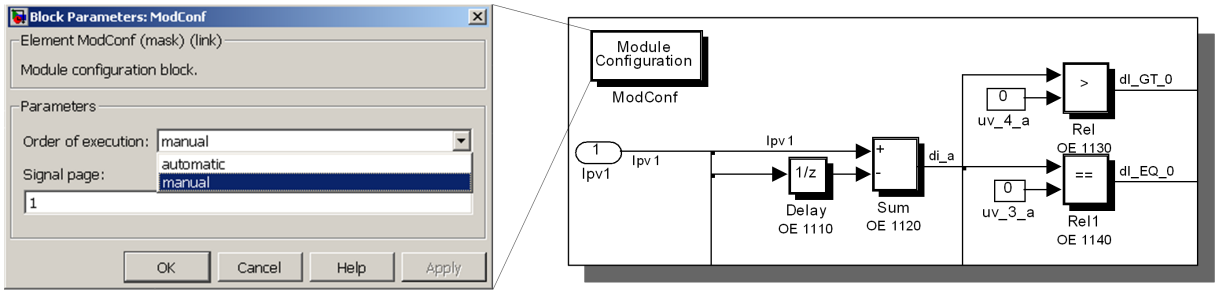


Figure 4.9: ModConf *GRAPlab block configures settings on a composite component level*

4.3.4 Mixing Simulink and GRAP

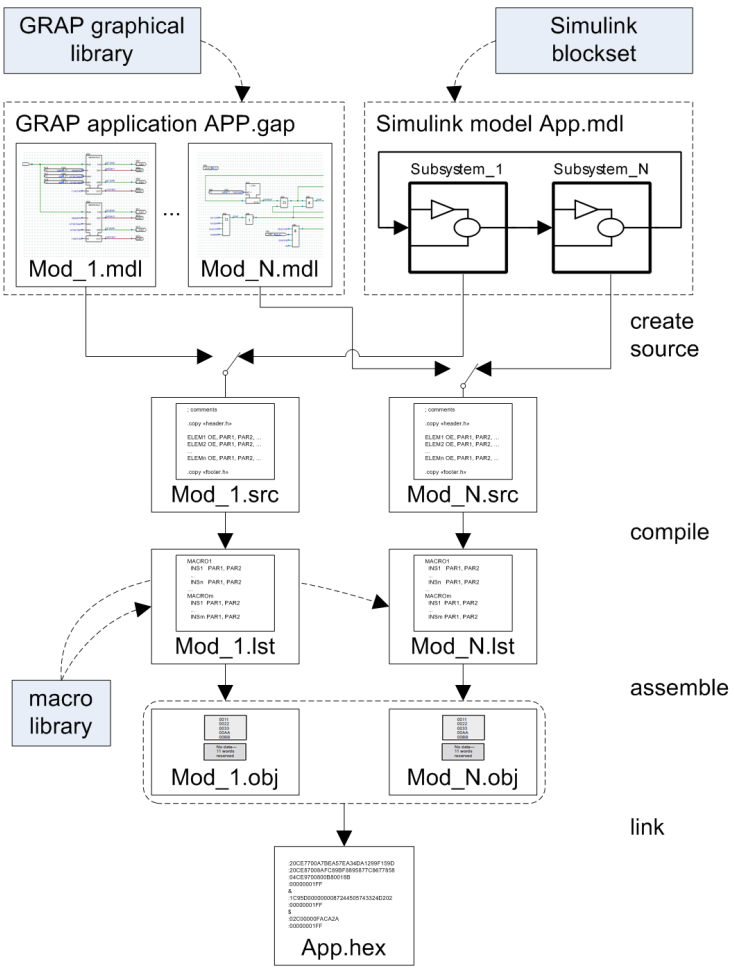


Figure 4.10: GRAP code generation from Simulink

Described code generation process results in a complete application program that can be loaded onto embedded target system and executed. Also, assembled object files corresponding to program modules designed in Simulink can be linked with object files generated in GRAP, Fig. 4.10. This way, algorithmically intense parts of application program can be developed in MATLAB/Simulink environment and linked to hardware-dependant input/output modules and legacy modules developed in GRAP.

Chapter 5

Component Real-Time Testing

This chapter elaborates the MoBREST Component Testing (MoBREST-CT) method. The real-time component testing is performed by combining non-real-time test cases from previous phases of development with the RTTP. Component test cases at non-real-time level are input signals, or test vectors, and expected output signals. Test vectors can be created in various ways, e.g. by using methods from section 3.2, and expected outputs, also called simulation traces, are obtained by component model simulation. The RTTP measures execution time of the component under test and validates its' behaviour in respect to interrupts. This way, the MoBREST-CT method represents extension to the available functional MBT.

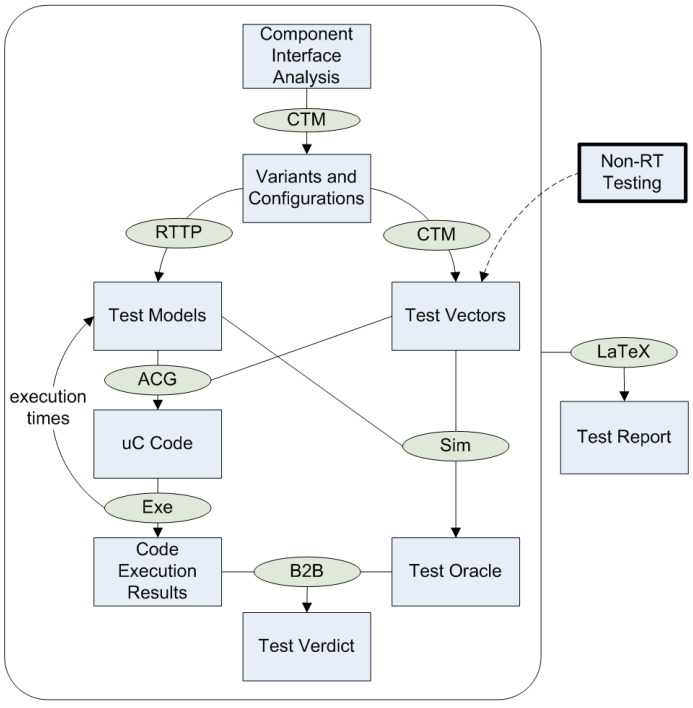


Figure 5.1: MoBREST component testing workflow

Workflow of the MoBREST-CT is shown in Fig. 5.1. Based on component interface analysis, its' mapping to component variants, or low-level atomic components, is determined. Each variant's configuration space is than partitioned to obtain all possible component variant con-

figurations. Decomposition of a component into variants and of a variant into configurations is done using CTM. Test vectors are imported from previous stages of functional MBT or they are automatically created by employing CTM for data partitioning. Configurations specification is combined with RTTP in automatic generation of test models. Embedded target executable code is, also automatically, generated from the test model, execution times of parts of code are measured by executing the generated code on the target and test duration is calculated based on these measurements. Test model is simulated to provide simulation traces used as test oracle which is compared to code execution traces to provide test verdict. All steps of component testing are automatically documented in a LaTeX-based test report. Each of the listed steps is explained in the remainder of this chapter.

5.1 Classification Tree Method

Classification Tree Method, already introduced in section 3.2, is used extensively by the MoBREST method so this section explains it in more details. It is an approach to black-box partition testing, [91], whose starting point is functional specification of the test object out of which test-relevant aspects are identified. Next, classification for each aspect and classes for each classification are formed. The resulting classes may be further classified. This partitioning of the test object's input domain is graphically represented by a classification tree. The tree acts as the head of a combination table inside which test cases are constructed by combining classes of different classifications. Test case design by CTM is a structured and systematic process which makes it easier to handle, understand and document.

An illustrative example of CTM application is taken from [91]. Test cases are generated for a computer vision system that determines sizes of different building blocks, Fig. 5.2 a). Test aspects are selected to be size, color and shape of blocks, Fig. 5.2 b).

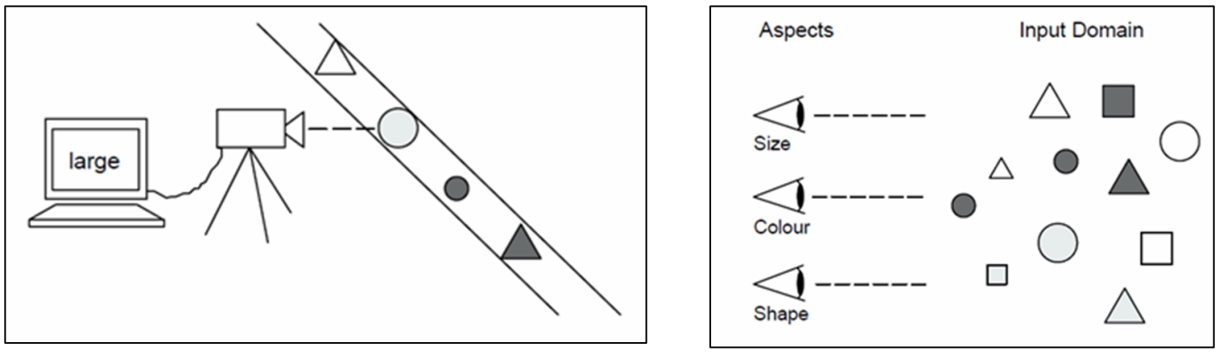


Figure 5.2: CTM example – a) computer vision system, b) aspects for classification, [91]

Size classification can be partitioned into *small* and *large* classes, *colour* classification can have *red*, *green* and *blue* classes and *shape* can be described by *circle*, *triangle* and *square* classes. In Fig. 5.3 the *triangle* class is further described with *shape of triangle* class partitioned into *equilateral*, *isosceles* and *scalene*. The classification tree obtained by described input domain partitioning represents head of the combination table out of which test cases are

selected. According to selections in the Fig. 5.3, computer vision system will be tested in three test cases with:

1. large red circle,
2. small green square and
3. small blue isosceles triangle.

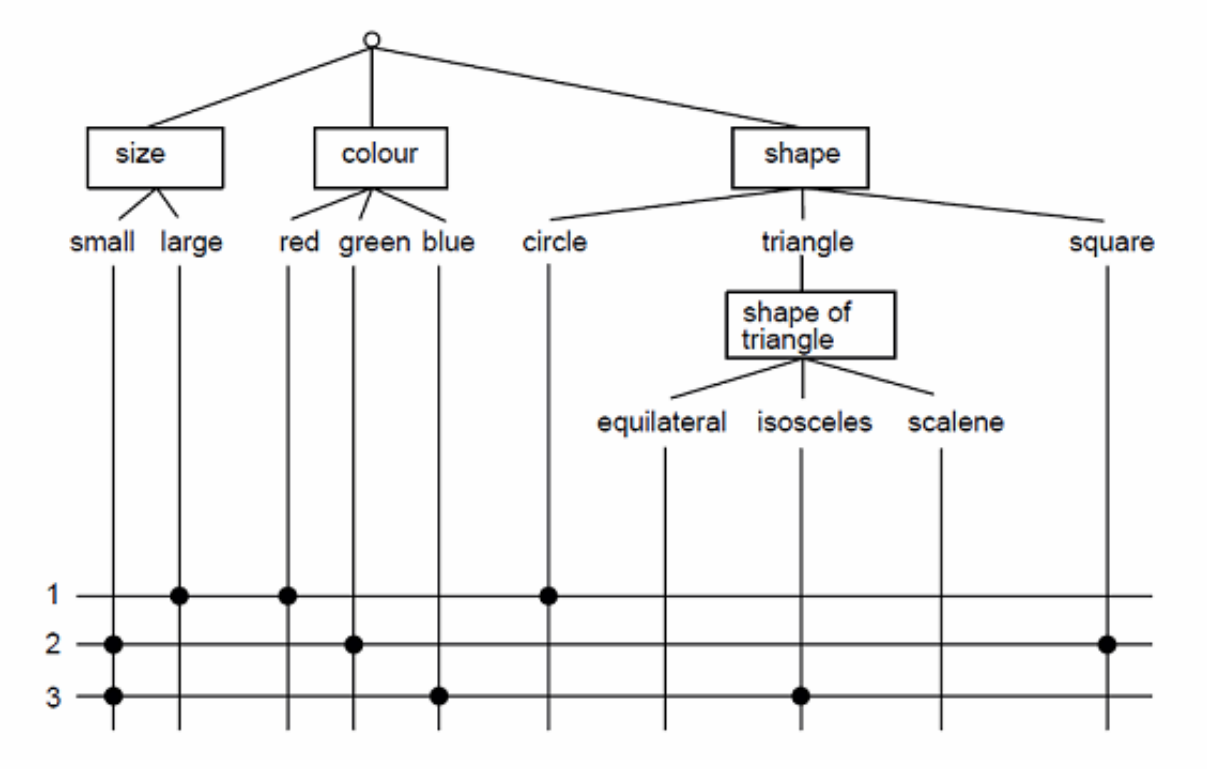


Figure 5.3: CTM example – classification tree, [91]

The selection of test cases or steps from the classification tree's combination table can be conducted based on different classification tree coverage criteria, [96]:

- The *minimum criterion* prescribes that every class in the classification tree must be selected in a least one test step. This criterion can be fulfilled with relatively small number of test steps but it usually achieves low error detection rate.
- The *maximum criterion* requires that every possible combination of classes is selected in at least one test step. Although this criterion leads to high error detection rate, combinatorial explosion in case of large number of classes makes it impractical.
- The *n-wise criterion* requires that every combination of n classes is selected in at least one test step. This criterion represents a compromise between minimum and maximum criteria. For example two-wise and three-wise criteria are often used.

5.2 Variants and Configurations

The first step of the MoBREST-CT method is analysis of the tested component's ACG mapping to target specific component implementations and configuration space partitioning for each of these low-level components. This step is performed automatically by applying CTM. In the remainder of the thesis, the following nomenclature is used in regard to component testing:

- **Component** is a target independent high-level software component that can during code generation be mapped into different low-level target dependant software components.
- **Component variant** or just **variant** is a low-level software component, a target specific implementation of the high-level *component*.
- **Component variant configuration**, **variant configuration** or just **configuration** represents parameters and connection layout of the *component variant*.

In this thesis, the MoBREST-CT method is applied to GRAPlab Simulink blocks utilized in ACG as presented in chapter 4. GRAPlab block can be mapped onto different atomic components (component variants) during automatic code generation based on block's and ACG procedure's settings. Each atomic component can be parametrized by textual labels and can have unconnected inputs and outputs. Fig. 5.4 shows an example of such partitioning for *Summation* GRAPlab block. Partitioning of this component is based on number of inputs, input signs and data type. Two variants are shown in the example: *DIFF_F* variant has only one configuration and *SUM4_L* variant can have a number of configurations that are distinguished by number of connected inputs and input sign labels.

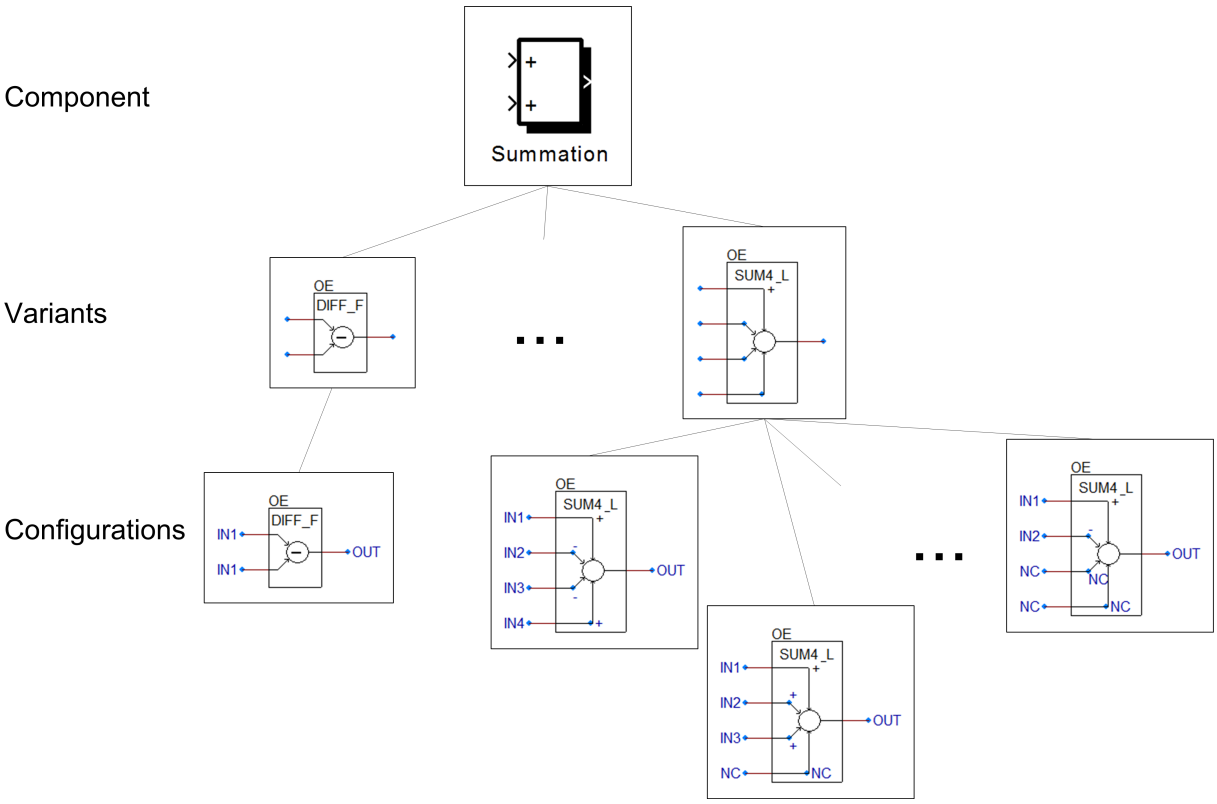


Figure 5.4: Summation *component partitioning into variants and configurations*

The introduced component testing nomenclature translates in case of GRAPlab component testing into:

- **Component** is a Simulink block, more specifically a block from GRAPlab blockset.
- **Component variant** is a legacy software component, i.e. GRAP programming element.
- **Component variant configuration** represents GRAP programming elements' parameters and port connectivity.

5.2.1 Automatic Partitioning of the Configuration Space

Configuration space of the tested software component is automatically partitioned to produce parameters classification tree. It defines sets of parameter values that are to be applied to the component during test case generation. This is a stepwise process based on component definition presented in chapter 4:

1. Classification tree is constructed based on all possible component parameters' values.
2. Classification tree editor (see section 5.7) dependency rules are created from component's dependency definition file.
3. Classification tree editor test case generation rules are created from component's mappings definition file.
4. Test cases are generated in classification tree editor and exported to M-files.
5. Test cases are imported into MATLAB and configurations are instantiated.

For example, *Summation* component has five parameters that partake in this process:

- *Number of inputs* (*NumIn*),
- *Input X sign* (*InXSign*) where *X* is 2, 3, or 4, and
- *Output type* (*OutType*).

These parameters and all their possible values are extracted from component's interface, as depicted by step **1** in Fig. 5.5, and used in automatic creation of the parameters classification tree.

In the second step, component's dependency definition file is read and dependencies are translated to classification tree editor's dependency rules. These rules are employed by the editor during automatic generation of test cases. For example, if two inputs are selected than *In3Sign* and *In4Sign* parameters must be set to *NULL* value. This is illustrated by step **2** in Fig. 5.5.

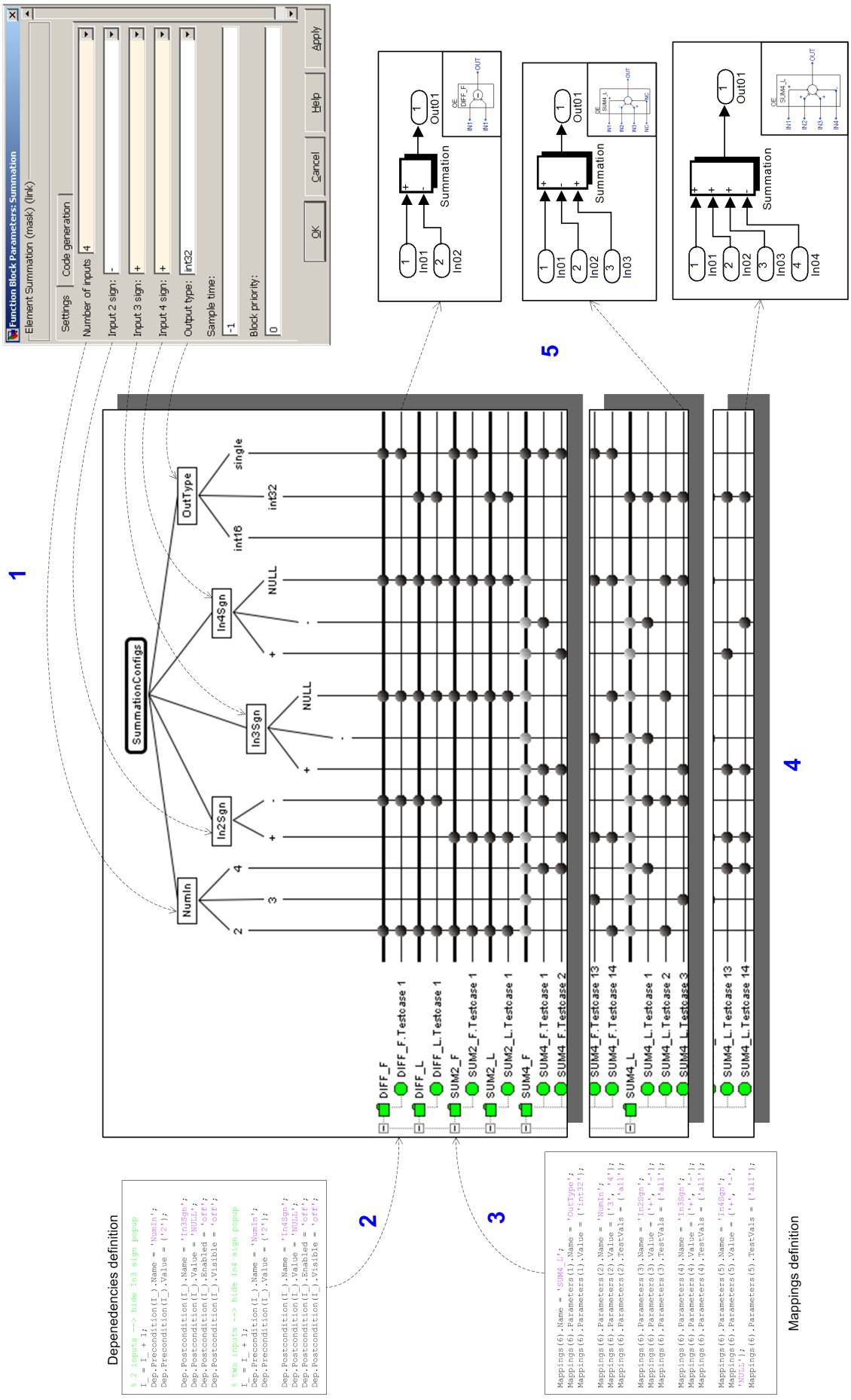


Figure 5.5: Summation *component parameter space partitioning process*

Besides dependency rules, classification tree editor needs test case generation rules for automatic test case creation. Generation rules are automatically constructed based on information obtained from component's mappings definition file. For each mapping, a variant is created and to each set of mappings' parameters, a variant configuration is assigned. Mapping definition determines each parameter's value or a set of values. If a mapping definition has *TesVals* field (see Fig. 4.7), than configurations are generated that cover all the stated parameter values. If the mapping definition doesn't have this field, than configurations are generated with values from the *Value* field. The difference between these two fields is that *Value* field is used for code generation and *TesVals* field, if present, is only used for configuration space partitioning. For example, the *Value* field of the *Sum4_L* mapping definition, Fig. 4.7, determines that *Summation* component is mapped onto *Sum4_L* variant if it has three or four inputs. The *TestVals* field of the same mapping definition determines, on the other hand, that during component testing every possible number of inputs will be checked. So a test case will be generated that will force mapping of the *Summation* component onto *Sum4_L* variant with only two connected inputs.

Classification tree, dependency and test generation rules automatically created by the MoBREST testing tool are imported into classification tree editor. Here, they can be inspected and edited in a graphical environment. The test cases, e.g. the ones shown in combination table on Fig. 5.5 in step **4**, are automatically created by the editor on a mouse click. The editor supports test case export to MATLAB format so that they can be imported and interpreted by the MoBREST tool. For each component variant configuration, a separate Simulink model file is created that contains the tested component with corresponding settings. This is illustrated on Fig. 5.5 with step **5** where, besides configured components, atomic components to which they map are shown.

5.3 Test Vectors

Test vector generation procedure is somewhat similar to the described configuration space partitioning. Here also partitioning based on classification tree method is performed, but input data ranges are partitioned in stead of configuration space. The process is conducted in the following steps:

1. Classification tree is constructed based on variant configuration input interface, i.e. number of inputs and their data types.

2. Test steps are generated inside classification tree editor to achieve selected classification tree coverage criterion.

3. Test vector specifications are imported into MATLAB and test signals are instantiated.

Test vector generation can be performed fully automatically or the tester can tweak individual steps manually. It is repeated for every configuration of every variant of the tested component.

In the first step, configuration's input interface is analyzed so that number of inputs and their data types are determined. This is done by enumerating *Inport* Simulink blocks inside GRAPlab masked subsystem block and by extracting their parameters. Besides data type,

specific values for each input are identified in this step, if they are defined by the *Description* parameter of the respective *Inport* block. Classification tree is constructed so that a classification is created for each input and classes for each classification are defined by partitioning its' data space into equivalence classes. For example, *DIFF_L* variant configuration of the *Summation* component has two inputs, so the classification tree in Fig.5.6 has two classifications, *In01* and *In02*. Each input's characteristic value represents a class, as well as value ranges between adjacent characteristic values. For example, if an input is of 16-bit integer data type and if 1000 is defined as its' specific values, than equivalence classes would be:

- -32768,
- between -32768 and 0,
- 0,
- between 0 and 1000,
- 1000,
- between 1000 and 32767, and
- 32767.

Data partitioning is based on the uniformity hypothesis according to which the test of a representative value from each equivalence class is assumed to be equivalent to any other value of that class. In other words this means that an error revealed by one value from the equivalence class should be detected by all other test cases of the same equivalence. For the given 16-bit integer input example this would mean that if component works correctly for input values -32768, -16384, 0, 500, 1000, 16384, and 32767, than it should work correctly for all values of that input. Equivalence partitioning is a heuristic method and the quality of produced tests depends on accuracy and detailedness of functional specification and of performed classification, [9].

Inside classification tree editor, test vector specification generation is conducted by employing test case generation rules generated by the MoBREST tool. Specification is exported to M-files, imported by the MoBREST tool and test vector waveforms are automatically instantiated. Selected equivalent classes of a classification, i.e. input, in the adjacent test steps are connected by a linear signal segment with predefined number of samples.

An example of the vector classification tree for the *DIFF_L* variant configuration of the *Summation* component is given in Fig. 5.6. Here, minimal classification tree coverage criteria is employed for test vector specification generation. Test vectors are instantiated by connecting selected classes in the combination table. In the example on Fig. 5.6 first input *In01* is constructed by connecting the most negative value (dashed green arrow 1) with value from "(0, 32767)" equivalent class (dashed green arrow 2), by connecting this value with zero (dashed green arrow 3) and so on. *In02* test vector is created in the same manner, i.e. by connecting classes of the *In02* classification of the vector classification tree, as suggested by red dotted arrows in the figure. By instantiating test vector specification defined with the classification tree and corresponding combination table, signal waveforms shown in the same figure are produced.

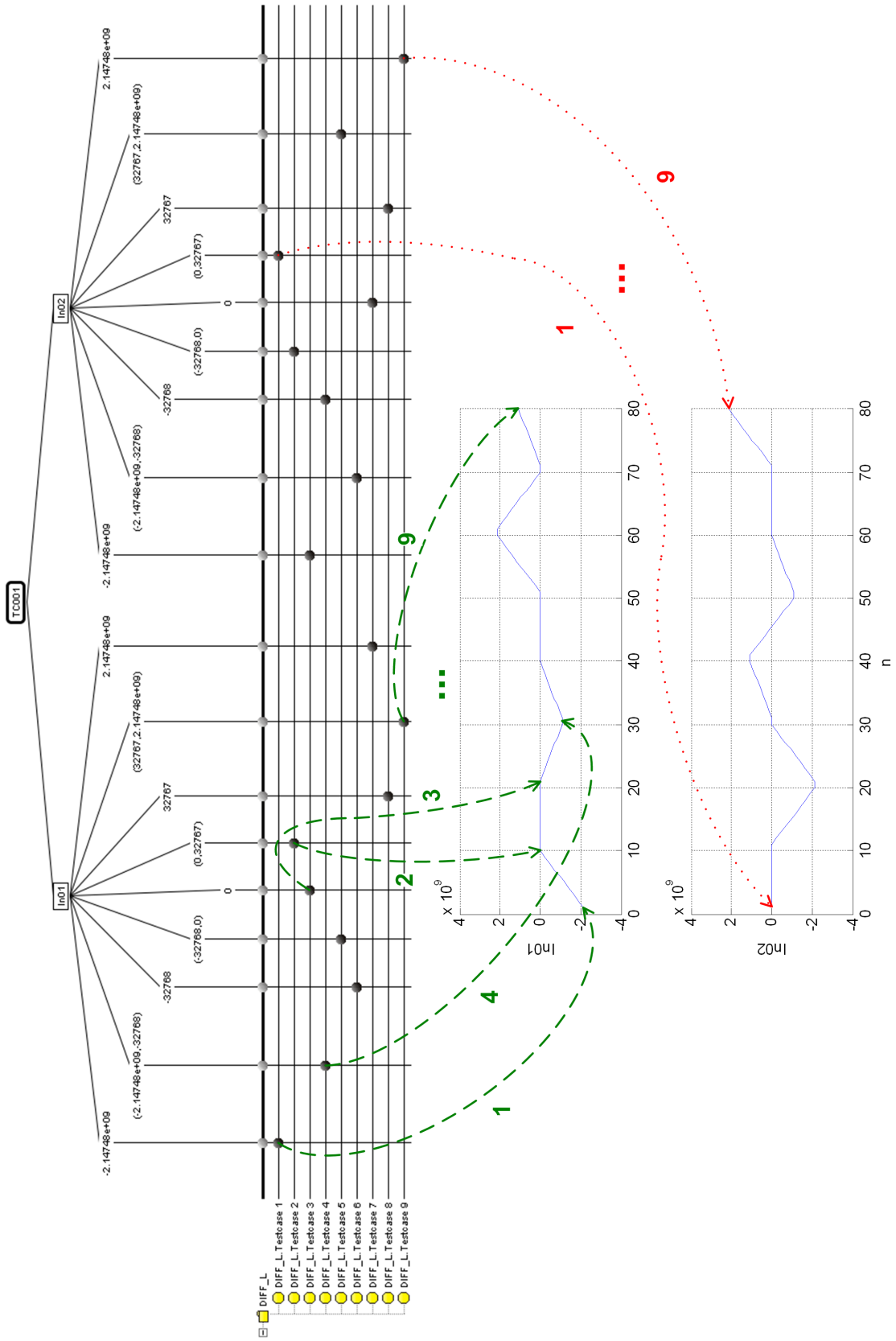


Figure 5.6: Input data partitioning and test vectors generation for DIFF_L *configuration*

5.4 Test Model

5.4.1 The Pattern

After test vector generation, the MoBREST-CT method proceeds to construction of test model. The model is generated fully automatically based on RTTP and on artifacts from preceding steps. Structure of the RTTP is shown in Fig. 5.7 a). An instance of the component for each configuration is created on the high priority cyclic task 1 (CT1). On the low priority cyclic task 2 (CT2), the tested instance of the component is placed alongside a variable CPU load generator. Variable load ensures that execution of the code of the configuration under test is interrupted at each possible point by CT1 when the code generated from the model is executed on the target system, Fig. 5.7 c). This kind of pattern is repeated with each component variant configuration on the CT2 so that one test model is generated for every configuration of the component variant.

The test pattern uses thus only two cyclic tasks with highest priority. Rest of the lower priority cyclic tasks present in the target system are available during test execution for additional monitoring or some other auxiliary functions. The pattern assumes that CT1 is not interrupted and that CT2 is interrupted only by CT1. This means that asynchronous tasks must either have lower priority than CT1 and CT2 or that they must be disabled during test execution.

Simulink implementation of the RTTP illustrated by Fig. 5.7 a) is shown on the Fig. 5.7 b). It contains six types of subsystems:

- *LoadBeg* and *LoadEnd* subsystems are specifications for generating target code used in execution time measurement. The code is inactive during regular test run and the subsystems have no effect during simulation.
- *IntWDog* subsystem is a specification for generating target code that monitors interrupt latency caused by the component under test during test application execution on the embedded target system. It has no function in simulation.
- *Conf_001* subsystem is reproduced for each component configuration, so that instantiated test model contains subsystems *Conf_001*, *Conf_002*, ..., *Conf_n*, where *n* is the number of possible variant configurations. The code generated from all these subsystems is executed on the high priority task so that it interrupts the execution of tested configuration's code.
- *Conf_CUT* subsystem houses component variant configuration that is being tested and the variable CPU load generator block. This block has no function during simulation but represents specification for corresponding code generation.
- *Log* subsystem records all test relevant signals in the model during simulation, same as the code generated from it does during test application execution on the target.

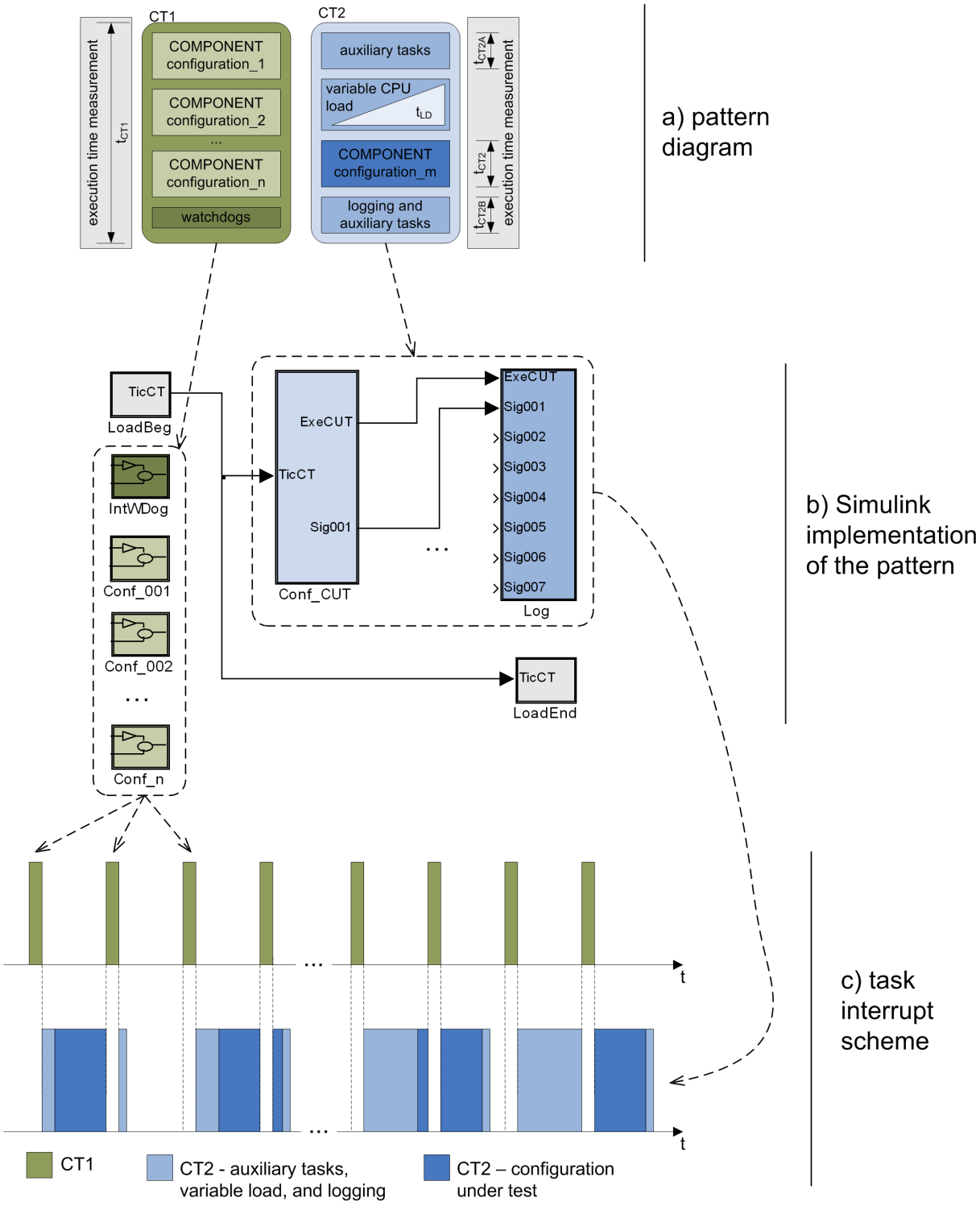


Figure 5.7: Mapping between a) RTTP diagram, b) Simulink implementation of the RTTP, and c) task interrupt scheme

Script with model parameters is also a part of the RTTP. This script, when instantiated, defines cyclic task execution periods, test run duration, and it assigns test vectors to the model.

5.4.2 Instantiation

Test models are instantiated by copying the pattern model, populating it with component under test and modifying it accordingly. Test models are parametrized by scripts which are instantiated by modifying a pattern script. Multiple sets of parameters, i.e. multiple parameter scripts, can be generated for a single test model.

Fig. 5.8 shows test model root view, the pattern is given on the left-hand side and the right-hand side provides example instance for the *SUM4_L* variant of the *Summation* component. The pattern is instantiated by reproducing *Conf_001* subsystem for each configuration of the respective component variant. In this example that means 14 instances of the *Conf_n* subsystem because *SUM4_L* variant of the *Summation* component has 14 possible configurations, see section 5.2. The remaining subsystems are populated and configured accordingly.

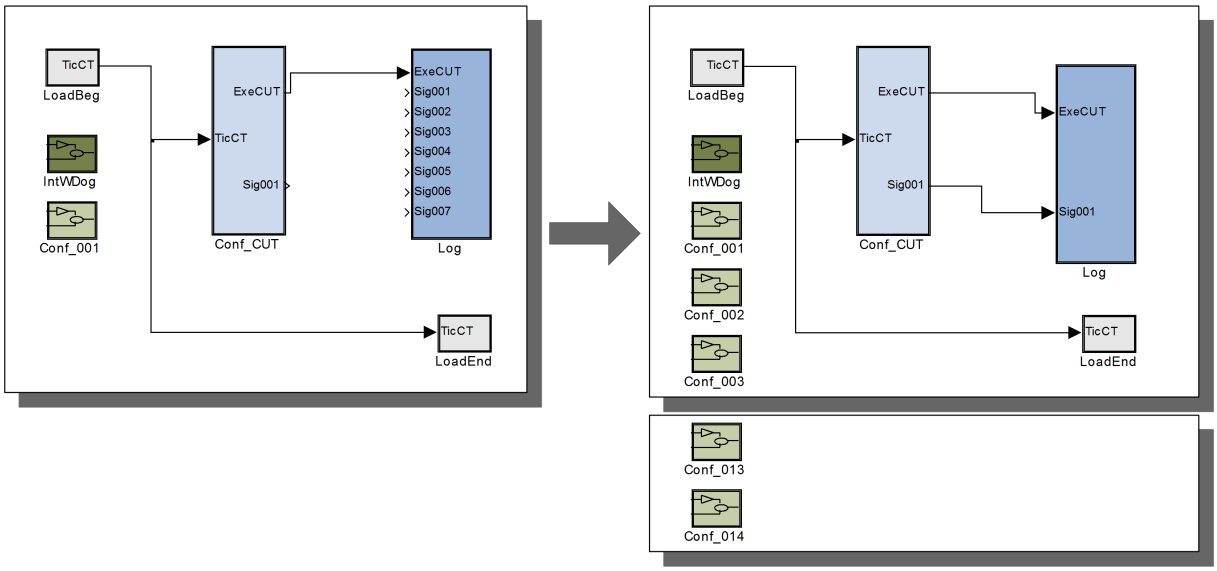


Figure 5.8: Test model root; pattern on the left and instantiated on the right

Tested configuration is placed inside *Conf_CUT* subsystem. *MEMPLAY* blocks reproduce signals from MATLAB workspace during simulation or from embedded target's memory during code execution. One of these blocks is connected to each of the configuration's inputs. Rest of the blocks inside *Conf_CUT* subsystem are used for real-time properties testing of the generated code and have no effect during simulation:

- *CpuLoad* – block generates variable CPU load which ensures that tested configuration is interrupted at each possible point;
- *set pin* and *clear pin* – blocks are used to control state of one of the target controller general-purpose output pins so that code execution can be monitored by oscilloscope;
- *TocCT2A* and *Peaks* – blocks, together with *LoadBeg* subsystem on the root model level, measure execution time of the code on the CT2 that executes before the tested configuration;
- *TicCUT* and *TocCUT* – blocks measure execution time of the tested component.

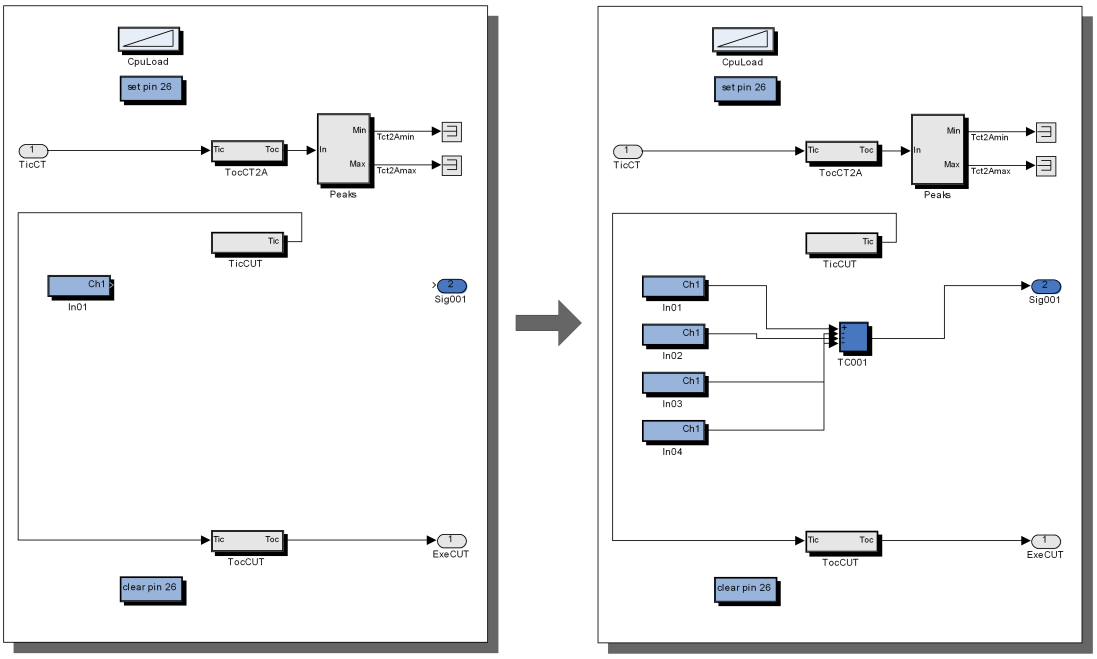


Figure 5.9: Conf_CUT *subsystem of the test model; pattern on the left and instantiated on the right*

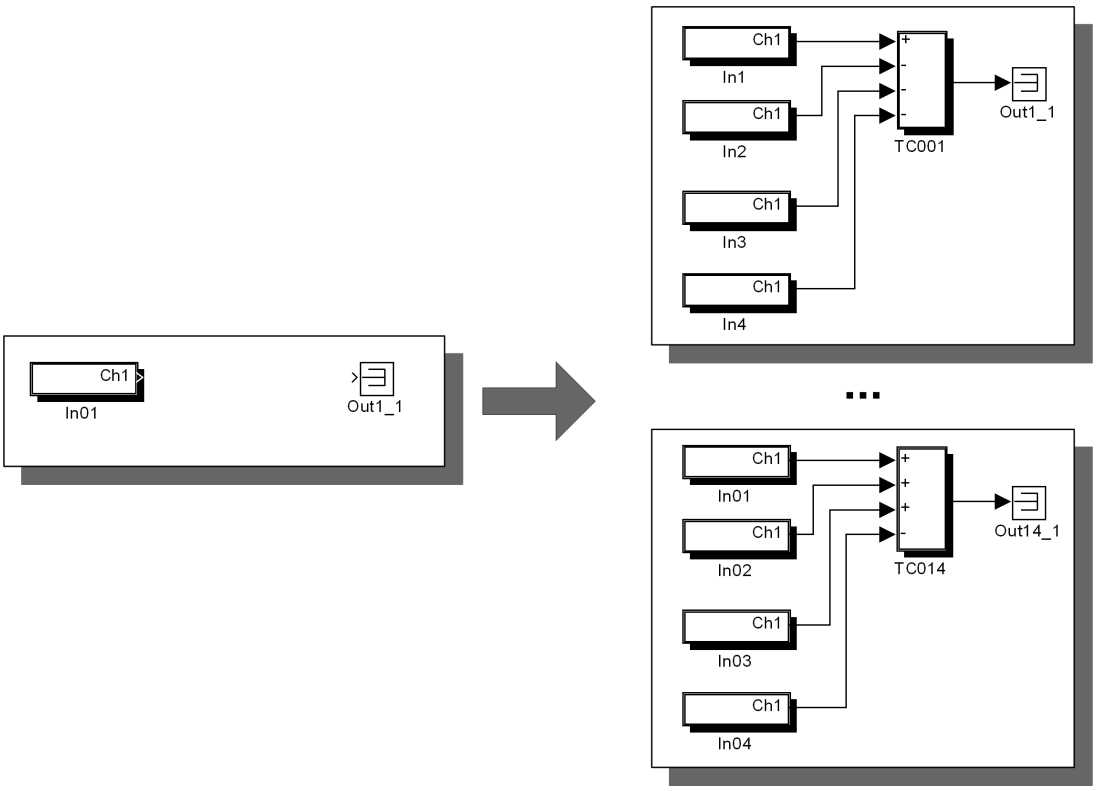


Figure 5.10: Conf_n *subsystems of the test model; pattern on the left and instantiated on the right*

All *Conf_n* subsystems' content is basically the same: *MEMPLAY* blocks configured to reproduce test vectors for the respective configuration and the configuration instance, Fig. 5.10.

Configuration instances on the CT1 are used only to interrupt the tested configuration so their outputs are not considered.

The *Log* subsystem is customized to record all outputs of the tested configuration, Fig. 5.11. This subsystem is somewhat cluttered as a result of Simulink's autorouting used during automatic model generation.

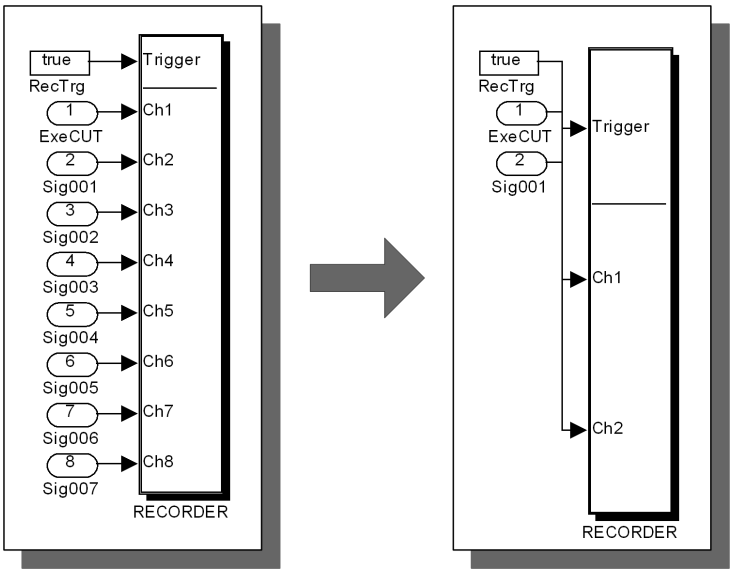


Figure 5.11: Log *subsystem of the test model; pattern on the left and instantiated on the right*

IntWDog subsystem is shown in Fig. 5.12. Target code corresponding to this subsystem is executed just after the code corresponding to *LoadBeg* subsystem. It starts by reading current value of the timer that generates cyclic interrupt. The example in Fig. 5.12 is for Texas Instruments' TMS320F28335 digital signal controller where cyclic tasks are led by general purpose CPU timer 2. This timer operates in downward counting mode, so the counter value is subtracted from the counter period to obtain interrupt delay measure. If the calculated value exceeds the threshold, than error flag is raised and the offending delay is remembered. By reading the state of the general purpose input-output pin 26, which is set only during execution of the tested component, it is ensured that only delays introduced by the component under test are detected.

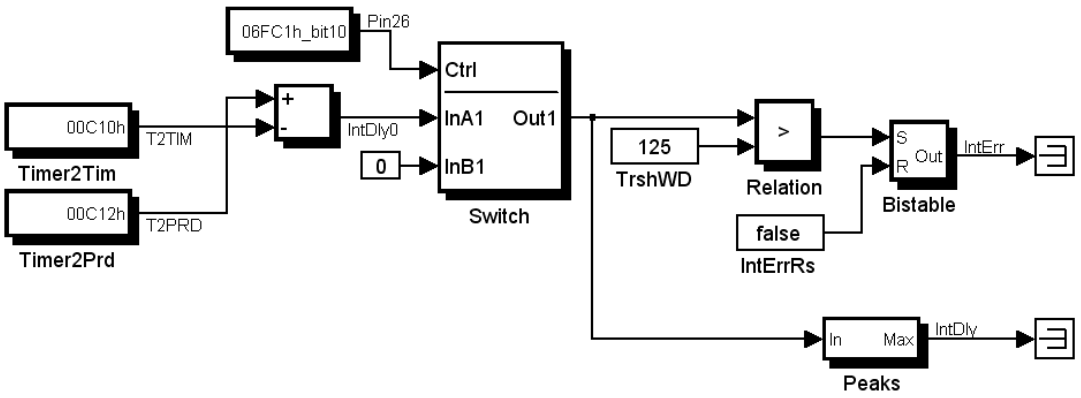
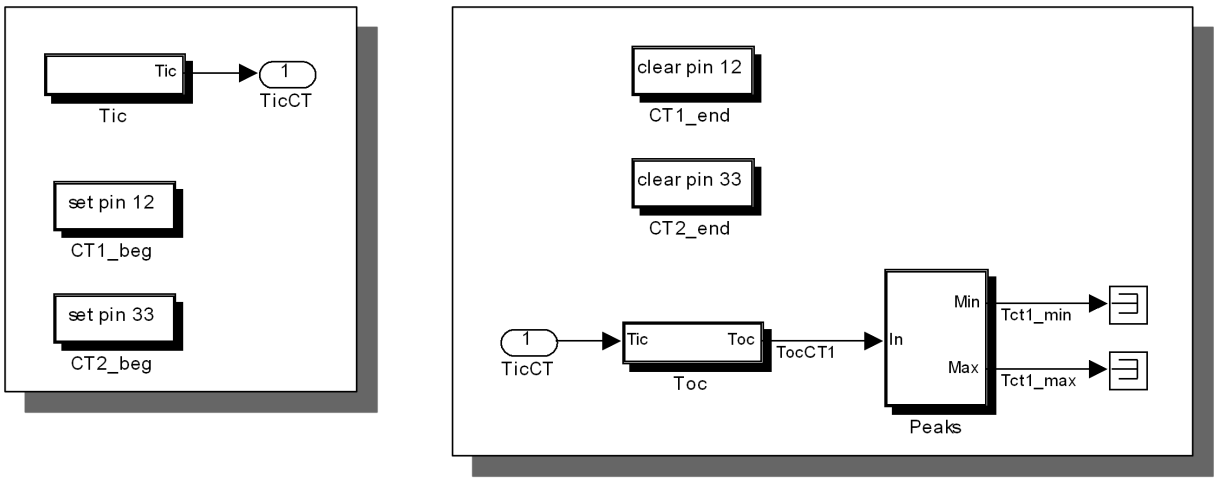


Figure 5.12: IntWDog *(interrupt watchdog) subsystem of the test model*

Content of the *LoadBeg* and *LoadEnd* subsystems for CT1 and CT2 execution time measurement and monitoring is shown in Fig. 5.13.



a) b)

Figure 5.13: Task execution time measurement subsystems a) LoadBeg *and b)* LoadEnd

5.5 Test Code

The test model is, together with test vectors, input to the automatic code generation procedure described in section 4.3. Generated executable code is automatically loaded onto the embedded target via serial connection and its' execution is initiated. All relevant signals, as specified by the *Log* subsystem of the test model, are recorded into embedded target's memory. Code execution traces are retrieved from the embedded target after the test run is over, again via serial connection.

Three versions of executable code are generated and executed during component real-time testing. The first two target applications are used to measure execution time of code on the two cyclic tasks. These are than used to calculate test duration of the actual test. The three target applications are generated from the same test model and they differ in the model parameters set utilized in code generation. The two parameter sets for task execution time measurement are generated during test model instantiation, while the third parameter set is constructed after task execution times are available.

5.5.1 Execution time measurement

Execution time is measured by the code represented in the test model with *Tic* and *Toc* blocks. The algorithm uses the same down counting CPU counter/timer that generates the cyclic interrupt. Execution time is measured in steps:

1. T_1: read value of the CPU counter/timer just before execution of the measured code.
2. Execute the measured code.

3. T_2: read value of the CPU counter/timer just after execution of the measured code.
4. $Diff_1 = T_1 - T_2$: Subtract the two counter/timer values. This difference represents execution time of the code plus the time spent on two reads of the counter/timer.
5. $Diff_2 = T_3 - T_4$: perform two subsequent counter/timer reads and subtract the values. This difference is a measure of the time needed for two reads of the counter/timer.
6. $Toc = Diff_1 - Diff_2$: calculate the measured code execution time.

GRAP implementation of the described algorithm is presented in Fig. 5.14 where execution time of the *L_RS* atomic component is measured.

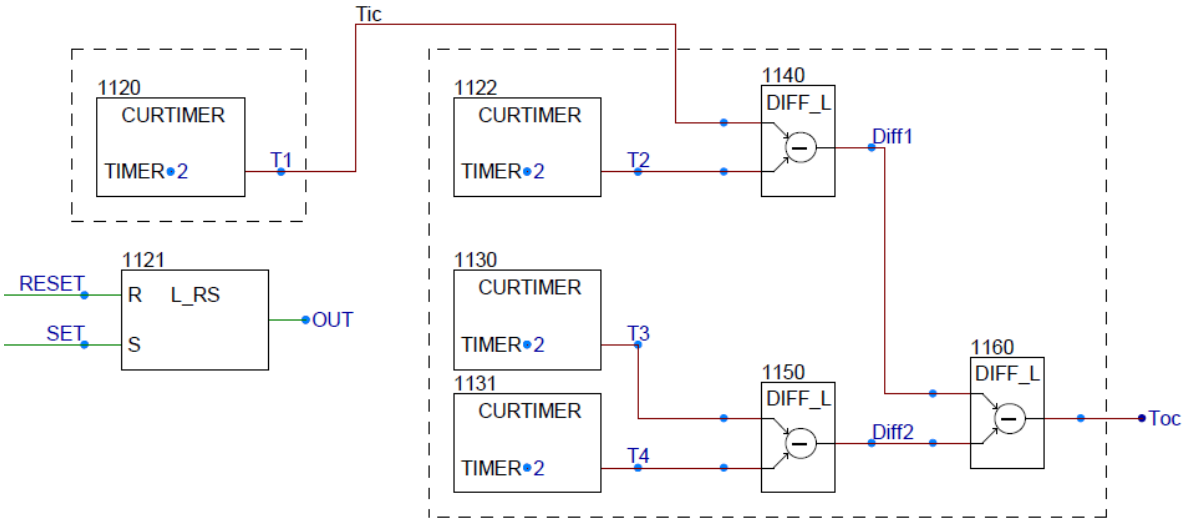


Figure 5.14: Execution time measurement implemented in GRAP

The first target application generated during component real-time testing is used to measure CT1's BCET $t_{CT1(min)}$ and WCET $t_{CT1(max)}$. This is achieved by placing *Tic* and *Toc* blocks from the *LoadBeg* and *LoadEnd* subsystems, respectively, on the CT1. Blocks intended to be executed on CT2 are placed on a task that is not executed during the measurement and are grayed out on the Fig. 5.15.

The second target application is generated from the same test model, but its parts are placed on different cyclic tasks by a separate model parameters script. In this case, parts of the code that should execute on CT1 are excluded from execution, and the code intended for CT2 is placed on CT1. This way all execution time measurements are performed with benchmarked code on uninterruptible CT1. Fig. 5.16 illustrates which parts of the model are executed during measurement of:

- BCET ($t_{CT2A(min)}$) and WCET ($t_{CT2A(max)}$) of the code executed on the same task as the tested configuration but before it;
- execution time of the tested component.

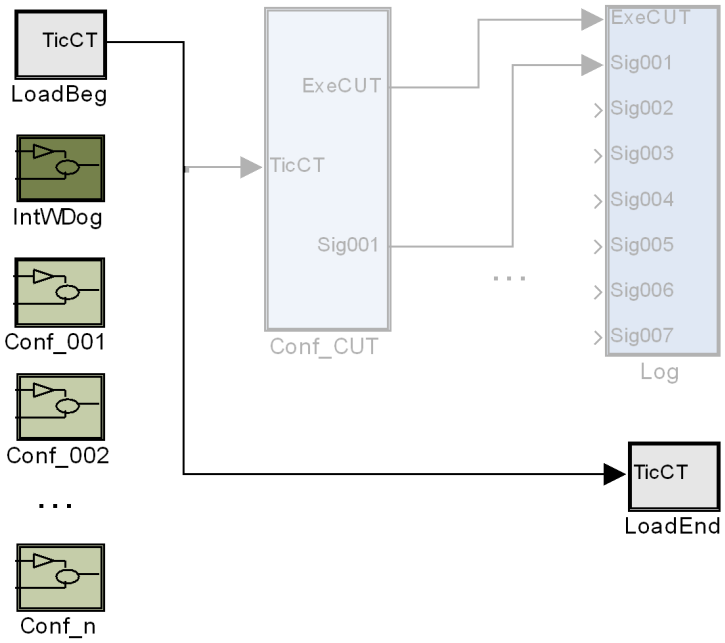


Figure 5.15: Measurement of the CT1 execution time

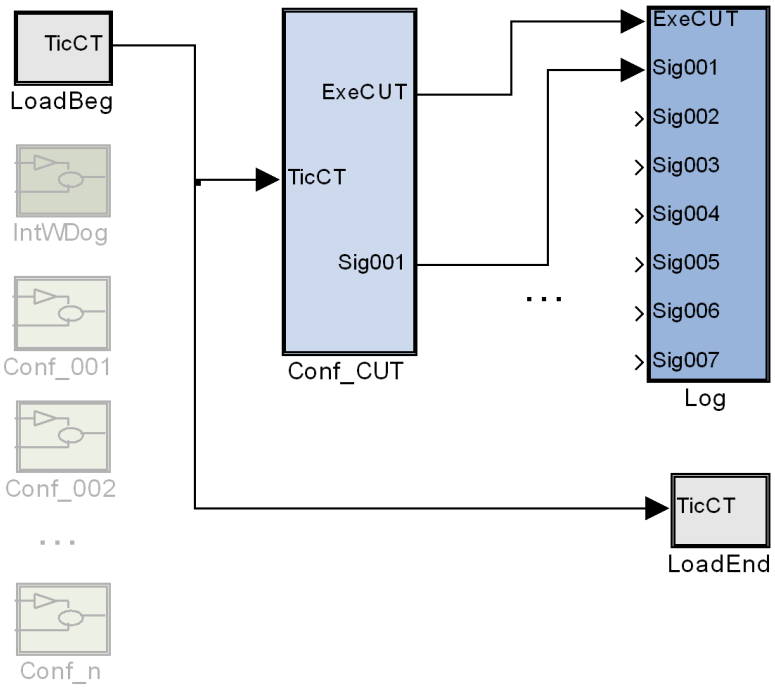


Figure 5.16: Measurement of the execution times of the CT2 code

5.5.2 The Test Run

Based on execution time measurements, test duration necessary to achieve 100% interrupt coverage of the tested configuration is calculated according to equation 5-1. The 100% interrupt

coverage means that the high priority CT1 tries to interrupt the tested component at every CPU cycle of its' execution. This coverage criteria is similar to the *"every interrupt at every task, module, object, or even every line"* criteria in [121], but it is applied on an even lower level.

$$
\begin{aligned}
t_{LD(min)} &= T_{CT1} - t_{CT1(max)} - t_{CT2A(max)} - t_{CT2(max)} \\
t_{LD(max)} &= T_{CT1} - t_{CT1(min)} - t_{CT2A(min)} \\
N_{exe} &= \frac{t_{LD(max)} - t_{LD(min)}}{T_{SYSCLK}}
\end{aligned}
\tag{5-1}
$$

Elements of the equation 5-1 are:

- T_{CT1} – is period of the high priority task CT1.
- $t_{CT1(min)}$ and $t_{CT1(max)}$ – are BCET and WCET of the code on CT1 task.
- $t_{CT2A(min)}$ and $t_{CT2A(max)}$ – are BCET and WCET of the code on CT2 task that precedes the code of the configuration under test.
- $t_{CT2(max)}$ – is the WCET of the code on the CT2 task.
- $t_{LD(min)}$ and $t_{LD(max)}$ – are minimal and maximal execution times taken by the variable CPU load generator.
- T_{SYSCLK} – is the period of one CPU cycle.
- N_{exe} – is the duration of the test execution in number of CPU cycles.

Some elements of the equation 5-1 are illustrated in Fig. 5.17.

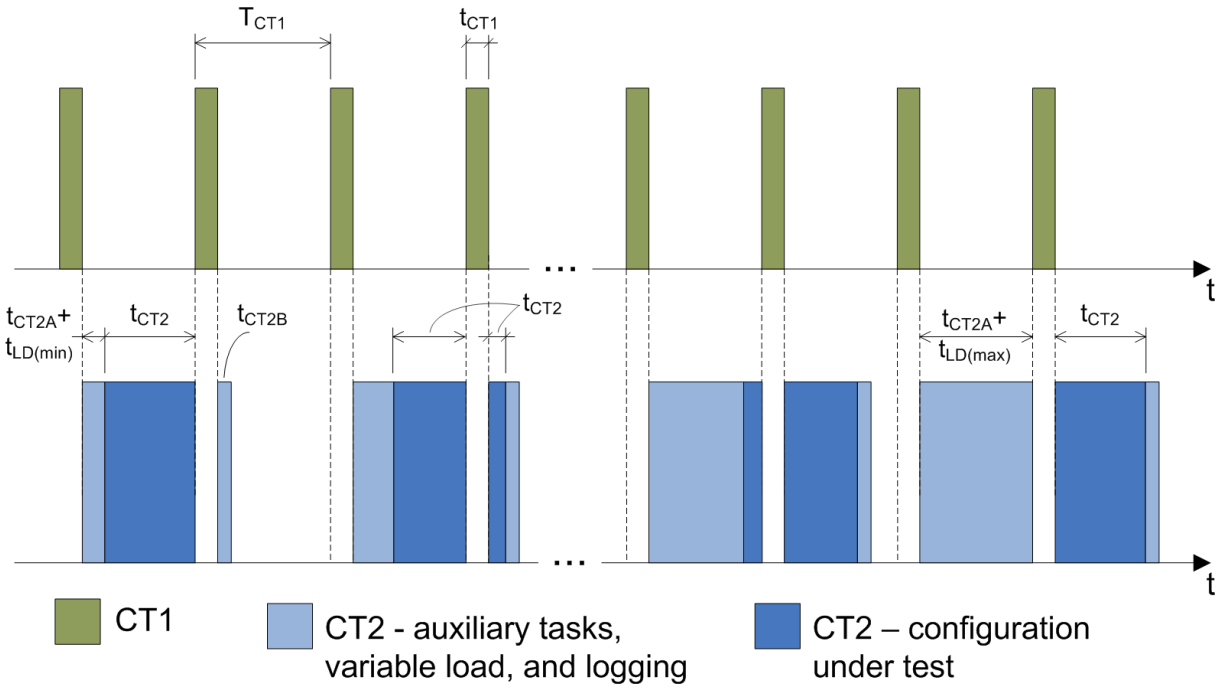


Figure 5.17: Task interrupt scheme

The basic idea behind equation 5-1 is to increase variable CPU load on the CT2 task by one processor cycle in each CT2 execution occasion. To achieve full interrupt coverage, the test run execution time must be long enough so that the variable CPU load increases from minimal $t_{LD(min)}$, where CT2 task ends just before next CT1 task starts, to maximal $t_{LD(max)}$, where CT2 task is interrupted by the next CT1 task just when variable CPU load code was executed and the tested component's code execution hasn't started yet. Between these two extremes, the CT1 task will try to interrupt the tested component's code execution in every processor cycle. Watchdog on the CT1 task monitors interrupt latency and detects any unallowed interrupt stalling by the code executed on CT2 task. Parts of the code that measure execution times are excluded from execution during actual test run, as illustrated in Fig. 5.18

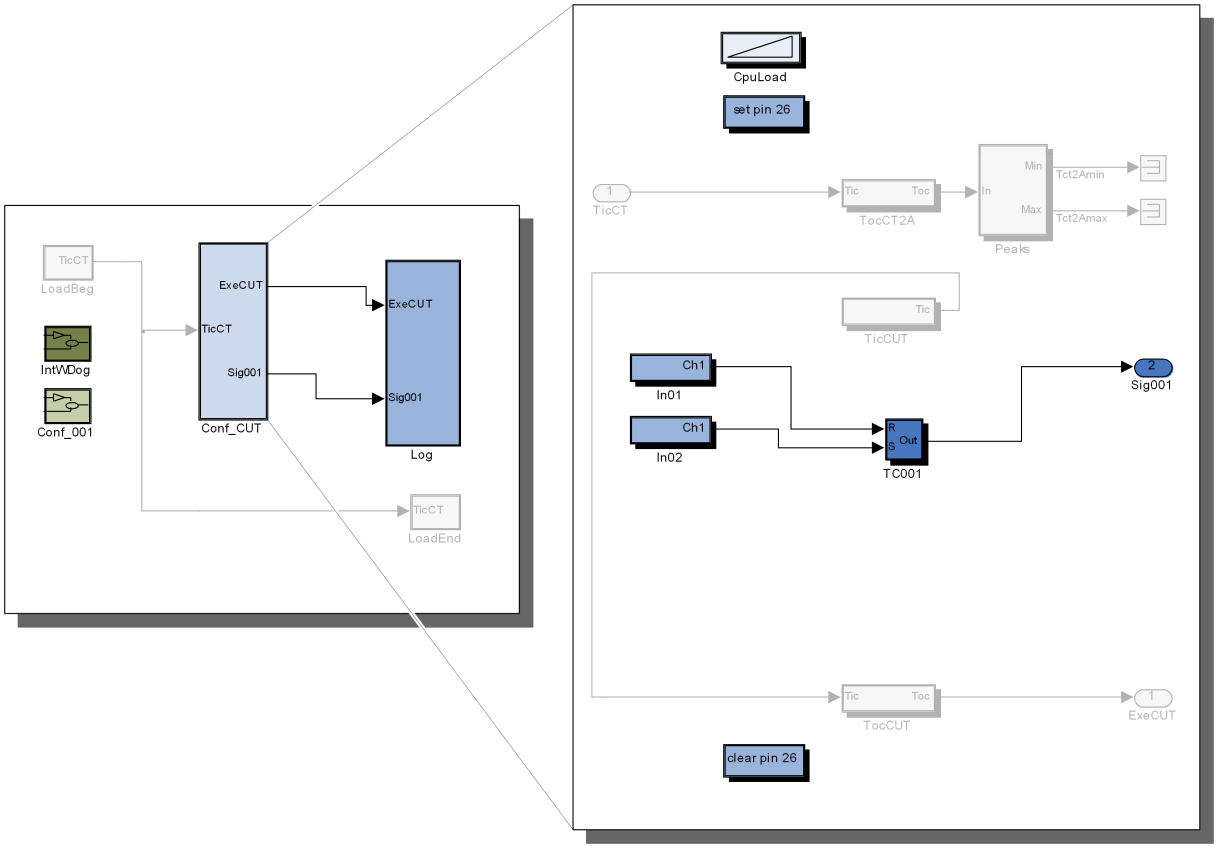


Figure 5.18: During the test run blocks used to measure execution time are excluded

Variable CPU load generator represented with *CpuLoad* block in Fig. 5.18 is implemented by atomic components (GRAP elements) with fixed execution time to avoid unintended jitter. The load rises from minimal to maximal values determined by *CpuLoad* parameters. It increments in each execution occasion by one CPU cycle. The "artificial" load is generated by atomic components:

- *WAIT* – stalls application execution for a set number of microseconds by repeating dummy writes to a memory location,
- *NOP255* – repeats one-cycle NOP (no operation) instruction for a set number of times, up to 255.

5.6 Verdict and Documentation

5.6.1 Test Verdict

Test verdict is produced by combining functional and real-time criteria. Functional component of the verdict is obtained by comparing test code execution traces to test model simulation traces, i.e. to the test oracle. If they are identical, than the component has passed the functional test. If discrepancies are present, the tester must inspect test results to decide whether the disparity can be tolerated or the test will be labeled as *failed*. An example of graphical representation of test results for low pass filter component from the safety platform environment, chapter 8, is given in Fig. 5.19. Test vectors are filter constant *KF* and input signal *In*, filter output is *SIG001*, and difference between code execution and simulation trace is $SIG001_{err}$. It can be seen that error signal assumes -1 value in two instances. This is a result of signal type conversions inside the referent GRAPlab block so the functional portion of the verdict test is manually set to *pass*.

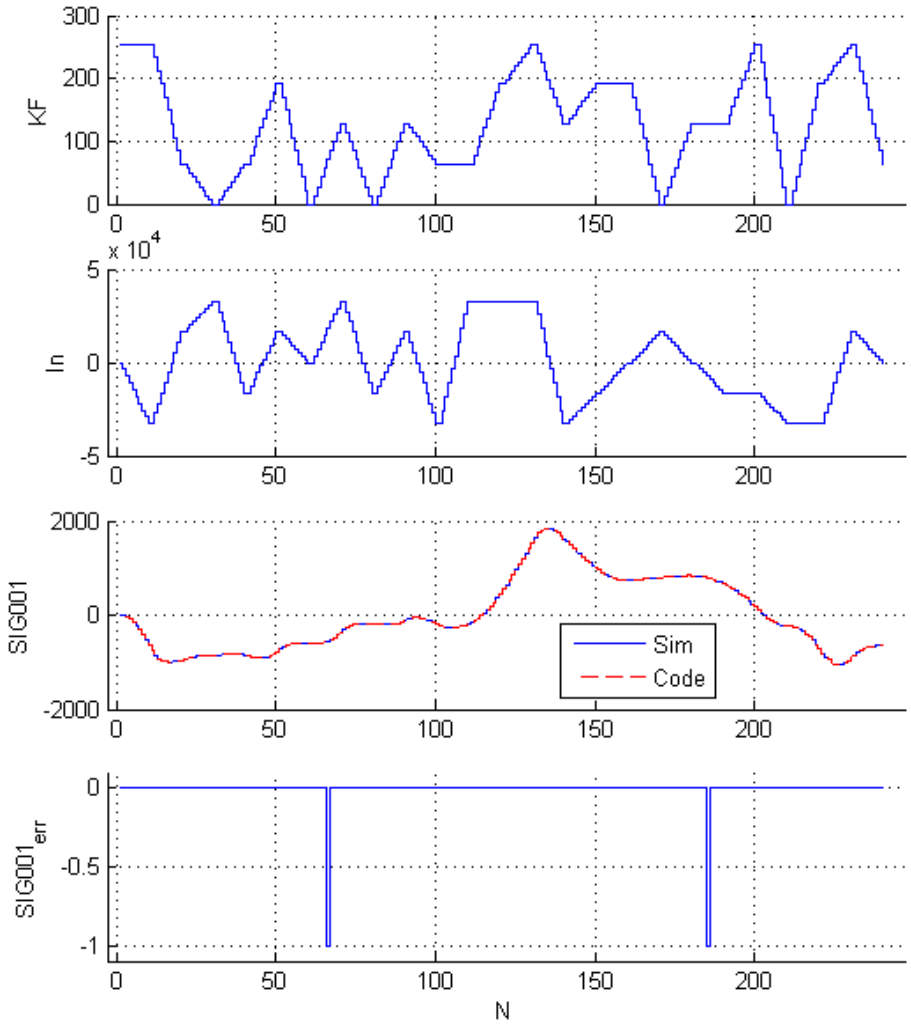


Figure 5.19: Test vectors, simulation and code execution traces, and error for safety platform's low pass filter atomic component FILTLPS

The described functional check also validates the component's immunity to interrupts, one of the real-time properties that need to be tested. For example, if the component uses some shared resources that are not properly handled by the context switching routines, it is expected that this shared resources will be corrupted when the component is interrupted by its' instance on a higher priority task and that this will result in invalid outputs. Verdicts for other two tested real-time properties, execution time and interrupt latency, are produced by comparing the measured values with the thresholds defined in requirements on the component. If measured values exceed limits, the test is assigned *fail* verdict.

5.6.2 Test Report

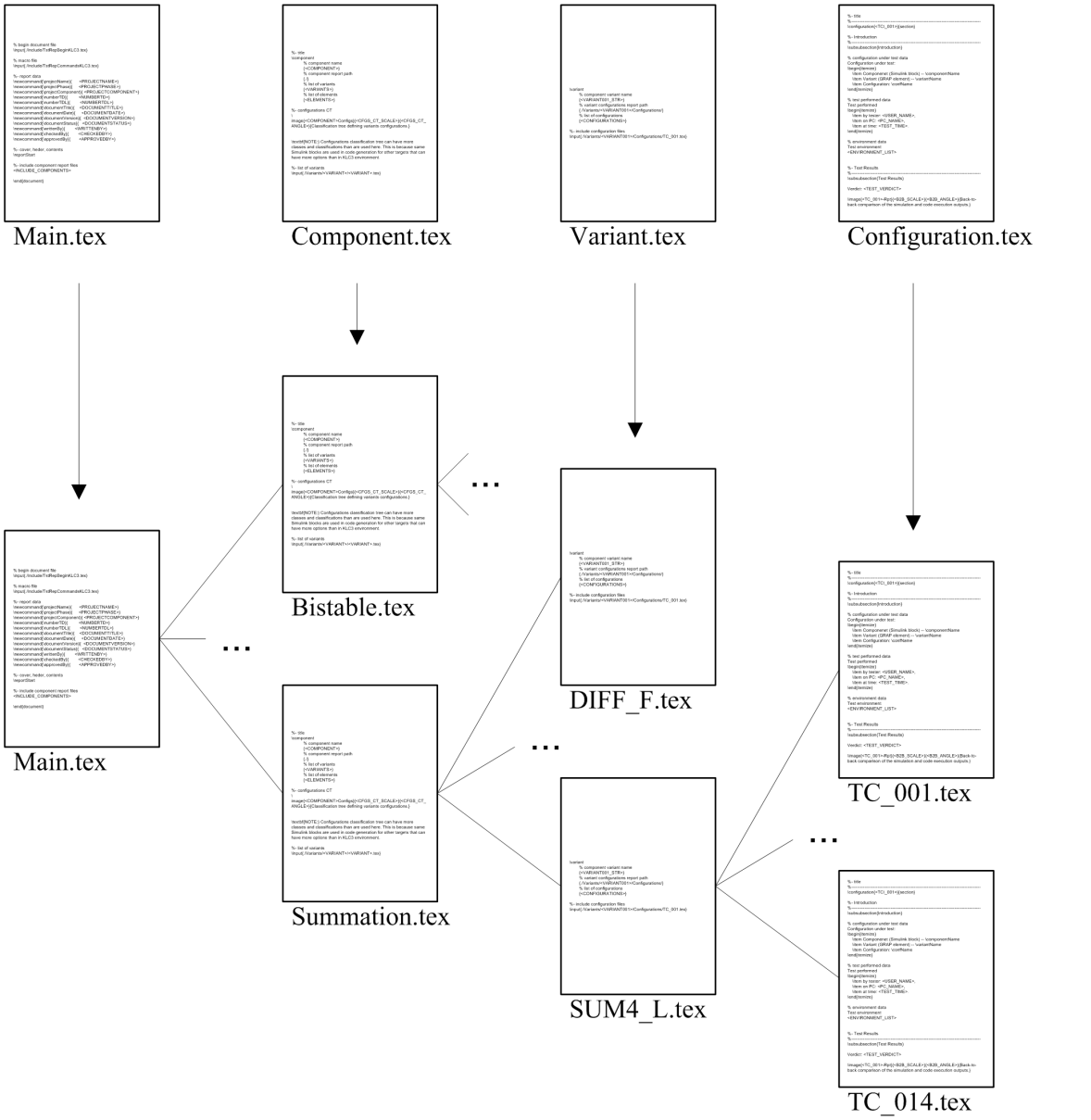


Figure 5.20: The structure of automatically generated LaTeX test report

A set of LaTeX commands and test report patterns has been developed to facilitate automatic test report generation. All classification trees, Simulink models, program code, and signal graphs produced during component testing are systematically documented. LaTeX files are created and compiled into PDF fully automatically. The structure of the test report follows closely the structure of the testing process, Fig. 5.20. The main test report file links reports for all tested components in which reports for all their variants are referenced. Variant test report is partitioned into separate reports for all its' configurations.

Test report for each component variant configuration contains sections:

- *Introduction* – section specifies what has been tested, testing environment, test date, name of the tester and other general information.
- *Test Results* – section provides test verdict, graphical signal representation and benchmark data.
- *Test Vectors* – section specifies parameters used during test vector generation and provides figures of vector classification tree and of vector waveforms.
- *Test Model* - diagrams of the root view of the test model and of all of its' subsystems are provided here as well as listing of the model parameters script.
- *Test Code* - section contains listings of all files produced by the automatic code generation from the Simulink model.

5.7 Infrastructure

5.7.1 Classification Tree Editor and Automation

Classification tree editor used in this work is a free version of *CTE XL*, [122], that doesn't support batch mode, unlike it's commercial variant *CTE XL Professional*. To automate tasks related to classification trees, scripting tool *AutoIt v3*, [123], is used. Another difference between free and professional versions of the *CTE XL* editor is that the free version doesn't support automatic test sequence generation. That is why sequences are automatically generated as sets of test cases where each test case represents one step of the sequence. The conversion of a set of test cases into a test sequence (i.e. test vector) is done when test cases are imported into MATLAB.

Interface of MoBREST tool to the CTE XL Editor, implemented by a number of MATLAB conversion functions and AutoIt scripts, is illustrated on Fig. 5.21. The classification tree layout, dependency rules, and test generation rules are automatically generated by MoBREST inside MATLAB in the form of MATLAB structures. These are than automatically translated into XML-based *.cte* file. Next, AutoIt script is invoked that opens CTE XL editor, loads the *.cte* file, performs test cases/sequence generation, and exports results to *.png* image file and a number of MATLAB *.m* files. The image file is later used in generating test reports and *.m* files are imported and processed by the M-functions: in case of component parameter space partitioning, the classification tree is translated into component variants and configurations, and, in case of

test vector generation, the tree is interpreted in terms of input signal waveforms. This way, free version of CTE XL is seamlessly integrated with MATLAB-based MoBREST tool.

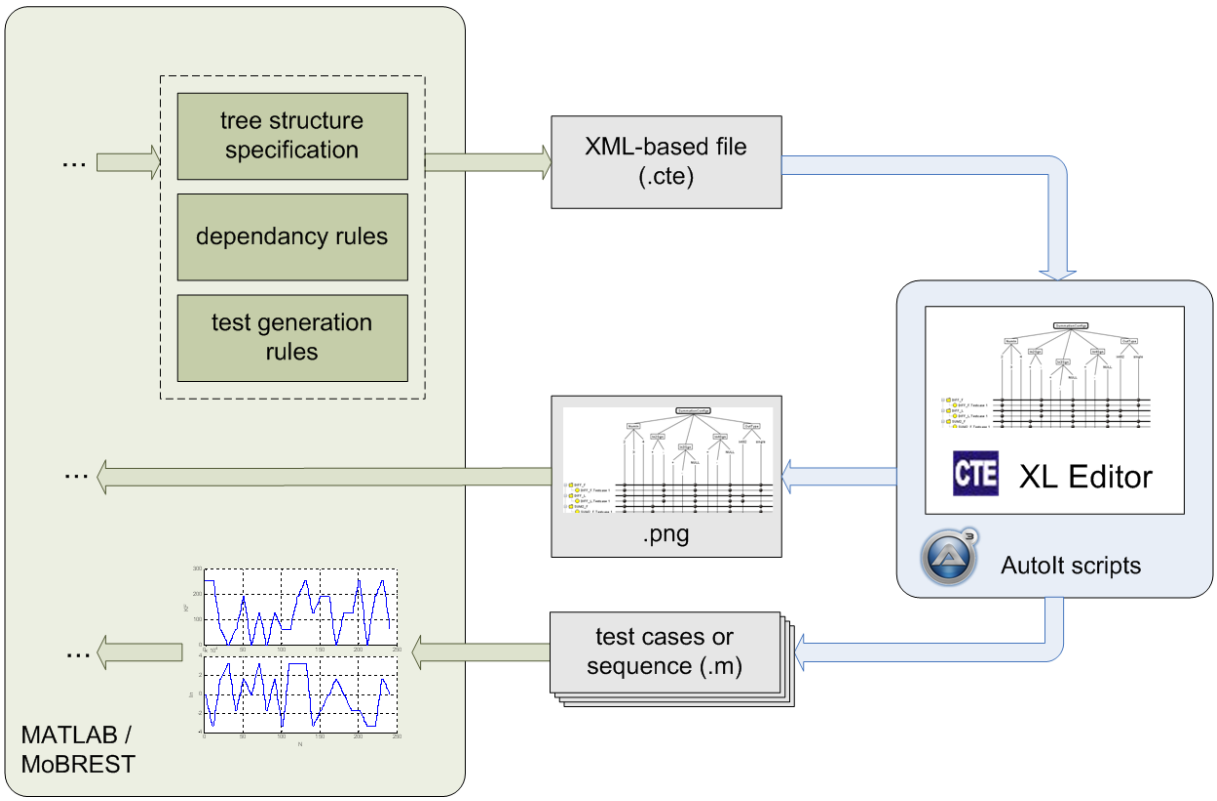


Figure 5.21: MATLAB/MoBREST to CTE XL interface via AutoIt scripting

5.7.2 Serial Communication

All the embedded targets from the intended application domain, as described in section 1.1, are equipped with serial RS-232 communication interface. In conventional development process this interface is used by the proprietary service and diagnostic software tool for tasks such as loading application programs, changing parameters, observing signals and retrieving recorded waveforms. In order to achieve seamless MBD environment, means for MoBREST tool to communicate with the embedded target from within MATLAB environment is necessary. With this motivation, a set of M-functions for communicating with the embedded target via RS-232 interface has been developed. They enable the MoBREST tool to:

- establish connection to the target and change baud rate,
- load the application program onto the target,
- retrieve symbols (parameters and signals) from the target,
- change parameters, and
- control memory recorder and retrieve its' records.

Chapter 6

Algorithm Real-Time Testing

Control systems testing is particularly demanding because of their tight coupling to the plant (i.e. environment). Timing constraints, if they are present, introduce an additional level of complexity into this kind of testing. MoBREST-IT method tackles the complexity of control systems integration real-time testing by elaborating processor-in-the-loop level testing into three steps:

- open-loop integration testing,
- closed-loop step-by-step integration testing, and
- closed-loop real-time integration testing.

Besides validation of functional algorithm properties, the proposed approach enables real-time properties testing by applying test patterns introduced in chapter 5. A SUT integrated from basic components can at this level be considered as a higher-level component and a variant of RTTP, presented in chapter 5, can be applied in order to validate its' real-time properties. Modifications of the pattern for control algorithm testing include:

- Due to the probable complexity of integrated component and limited computational resources of the embedded target, only one SUT variant is placed on CT1. In other words, multiple *Conf_n* subsystems in component pattern are replaced with one *SUT_CT1* subsystem. This subsystem contains one instance of the tested control algorithm model. The code it generates is executed on the higher priority task and it interrupts the tested algorithm implementation executed on the lower priority task.
- A new *Inputs* subsystem represents input interface of the RTTP. This subsystem also generates trigger signal that controls execution of SUT on higher priority task in order to achieve desired interrupt coverage.
- *Log* subsystem is replaced with *Outputs* subsystem, an output interface of the integration RTTP.

The modified RTTP model tailored for algorithm testing and root view of its' Simulink implementation are shown on Fig. 6.1 a) and b), respectively.

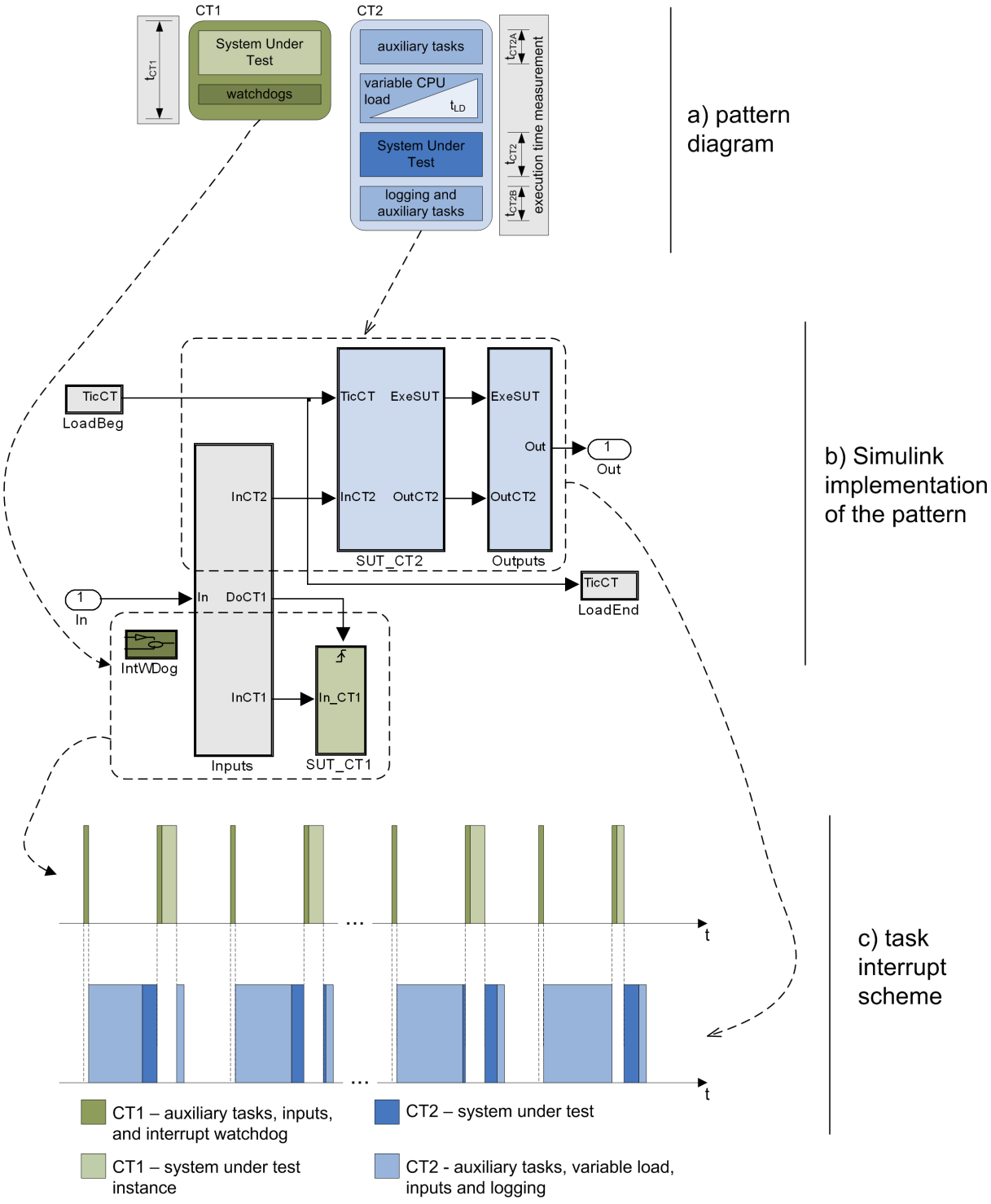


Figure 6.1: Mapping between a) integration RTTP diagram, b) its' Simulink implementation, and c) task interrupt scheme

Fig. 6.1 c) shows the distribution of parts of the code generated from the integration RTTP across cyclic tasks. SUT is executed on the lower priority CT2 together with variable CPU load generator which ensures that it is interrupted in each possible point by the higher priority CT1, same as in component testing. In order to maximize the probability of error detection, the SUT instances on CT1 and on CT2 are executed synchronously. This way when CT1 interrupts the

SUT on CT2, its' instance of the SUT is in the same state as the interrupted instance which is being tested. The synchronicity is achieved by placing the CT1 SUT instance into conditional subsystem that is executed on every second CT1 and by configuring CT2 period to be twice as long as CT1 period. Content of pattern's subsystems is shown in Fig. 6.2.

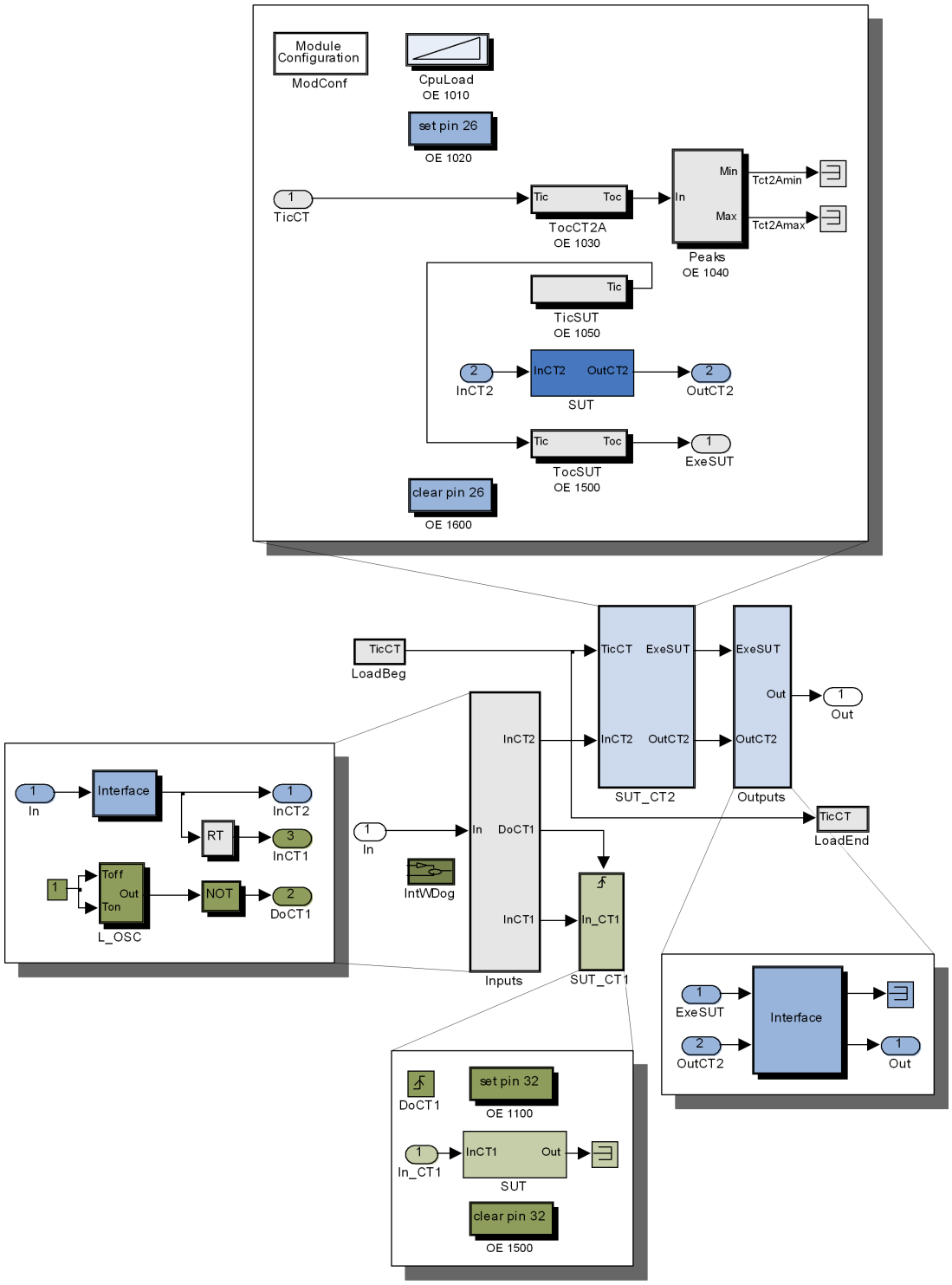


Figure 6.2: Subsystems of the Simulink implementation of the integration RTTP

6.1 Open-Loop Integration Testing

First step of the MoBREST-IT method is open-loop testing which is conducted in two phases. Firstly, only functional properties are checked to ensure that the generated code behaves same as the model. In the second phase the SUT is integrated into the pattern for real-time testing to validate its' real-time properties. Because functional properties are again validated during real-time testing, the first stage could be skipped. Purpose of the functional testing stage of open-loop testing is to identify, localize and correct any functional errors while the model is still relatively simple, i.e. before "cluttering" it with RTTP.

The diagram in Fig. 6.3 depicts algorithm open-loop testing and is applicable to both phases. Test vectors are generated in simulation environment or inherited from previous testing phases, such as functional testing on MiL integration level. Input vector generator can include model of the plant in which case a feedback can exist inside the model. Nevertheless, once the test vectors have been synthesized the generated code is tested in an open-loop.

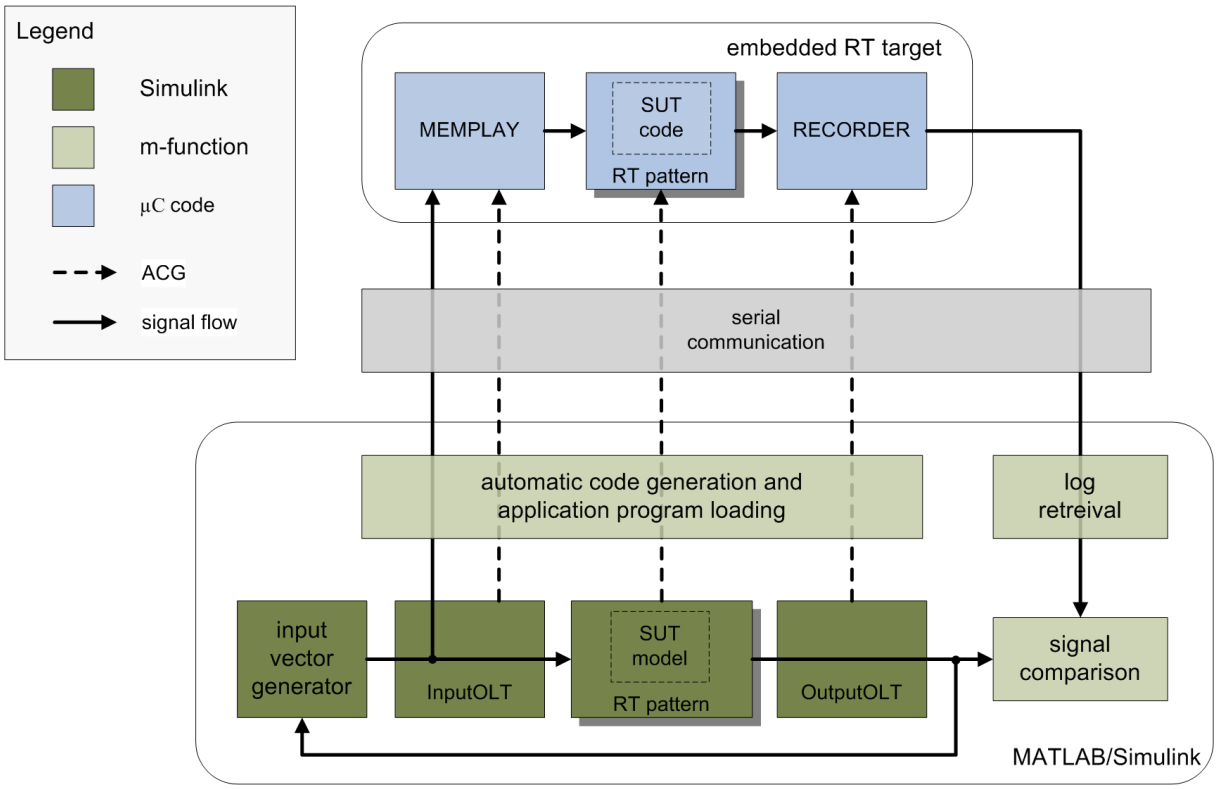


Figure 6.3: Open-loop integration testing.

Functional phase of the open-loop testing is performed in following steps:

1. SUT model is padded with *InputOLT* and *OutputOLT* MoBREST blocks that are responsible for off-line transfer of signals into and out of the generated code, respectively.

2. The test model is simulated to obtain referent simulation traces.

3. Code is automatically generated from the model. *MEMPLAY* element in Fig. 6.3 refers to reproduction of input vector from embedded target memory during code execution.

Also, all relevant SUT outputs are recorded to embedded target memory, as depicted by *RECORDER* element in the same figure.

4. Generated code is downloaded to the target through RS232 serial communication interface.
5. The code is executed, and execution traces are retrieved from the target memory.
6. Simulation and execution traces are compared inside the simulation environment and test verdict is produced.

Real-time phase of the open-loop testing follows similar procedure:

1. SUT model is integrated into RTTP. *Interface* blocks in *Inputs* and *Outputs* subsystems of the pattern are replaced with *InputOLT* and *OutputOLT* blocks, respectively.
2. The test model is simulated to obtain simulation traces.
3. The model is parametrized for CT1 execution time measurement, code is automatically generated, downloaded and executed, and measurement results are retrieved.
4. The model is parametrized for CT2 execution time measurement, code is automatically generated, downloaded and executed, and measurement results are retrieved. In this step SUT execution time is measured.
5. Parameters for the actual test run are calculated as explained in section 5.5. Test run code is automatically generated, downloaded and executed, and execution traces are retrieved.
6. Simulation and execution traces are compared inside the simulation environment, real-time properties measurements are checked against predefined limits, and test verdict is produced.

6.2 Closed-Loop Step-by-Step Integration Testing

The second step in in the MoBREST integration testing can be validation of the SUT in a closed-loop with simulated environment, as illustrated in Fig. 6.4. Here, models of the SUT and the environment are simulated inside the simulation environment while SUT code generated from the model is executed on the target. In each simulation step, input is fed to the SUT model and to the code, one simulation step and one target task execution are performed, and model outputs are compared with outputs of the code. Signal exchange between simulation environment and the target is controlled by *InputPiL* and *OutputPiL* Simulink blocks and corresponding atomic components *RS232IN* and *RS232OUT*. *OutputPiL* and *InputPiL* blocks are implemented using Real-Time Windows (RTW) Target MATLAB toolbox, [124]. Feedback can be closed with SUT model or code in the loop, depending on the *OutputPiL* element settings.

Closed-loop step-by-step testing can not be executed in real-time because the simulation environment is executed on plain desktop computer. The main benefit of this step is the validation of the SUT behavior inside a closed-loop after its' real-time properties have already been validated by open-loop integration testing.

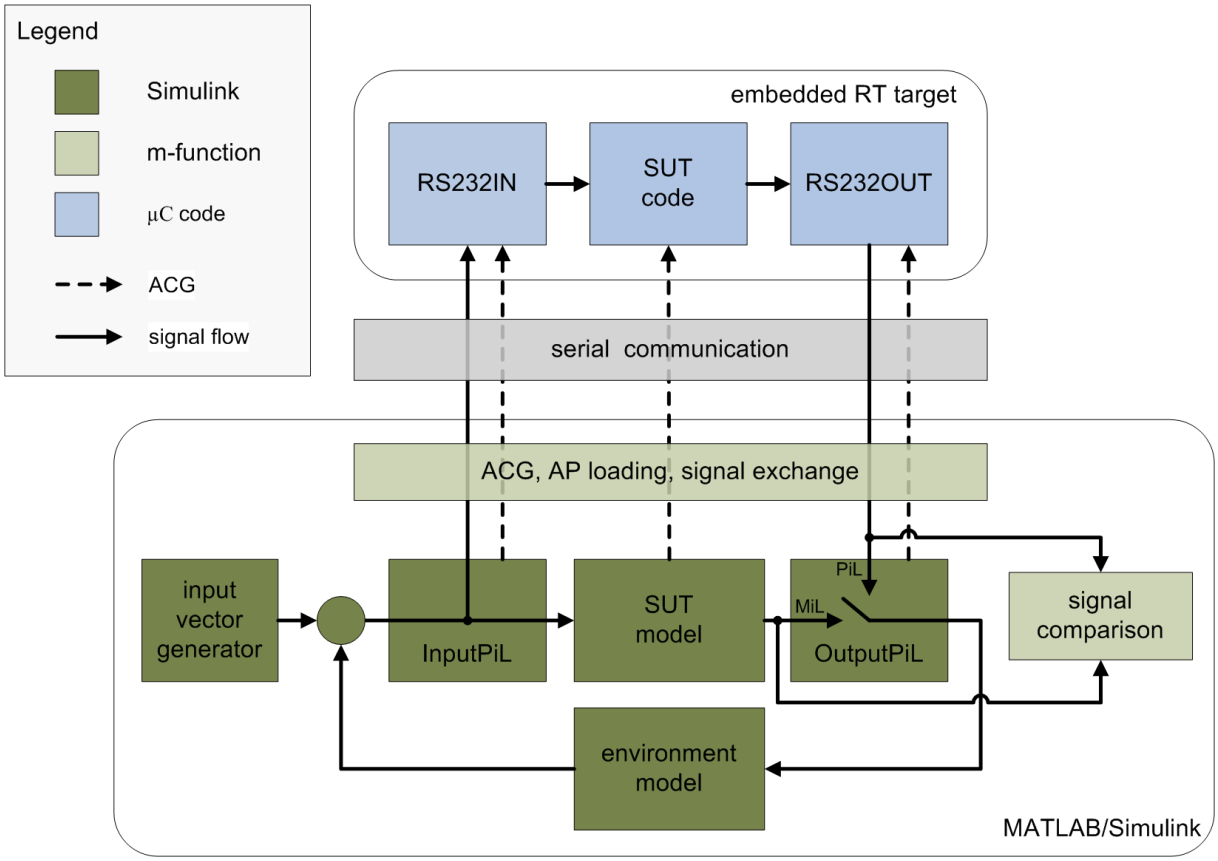


Figure 6.4: Testing in a closed-loop with simulated environment

6.3 Closed-Loop Real-Time Testing

If real-time properties of the tested algorithm need to be tested while it is running in a closed control loop, than there are three possibilities:

- A real-time simulation tool can be used, but this introduces the cost of the tool and of developing interface between the simulation tool and the embedded target system.
- Control algorithm can be tested in the plant or some laboratory setup that emulates it. This can be potentially hazardous, e.g. if power electronics or heavy machinery is controlled, and inconvenient, e.g. if the plant or the laboratory is dislocated from the testing team.
- Closed-loop real-time testing can be performed on the embedded target system without expensive real-time simulation tool and without access to the actual plant.

Closed-loop real-time testing is performed by placing the control system in a closed feedback loop with environment model running on the same target as the control algorithm. The preparation for this kind of scenario includes:

1. validation of the control algorithm code in open-loop integration testing and/or closed-loop step-by-step integration testing,
2. refinement of the environment functional model into implementational model adequate for ACG,
3. validation of the environment model's code in open-loop integration testing, and
4. real-time validation of the closed-loop containing SUT and environment code, as depicted in Fig. 6.5.

Here, the same inputs are fed to closed-loops in the simulation environment and on the target and relevant output signals are compared. SUT model can be placed in the integration RTTP if its' real-rime properties need to be validated in a closed-loop. This means that closed-loop testing on the embedded target can also be performed in two phases: functional phase, without RTTP, and real-time phase, if SUT is embedded in RTTP.

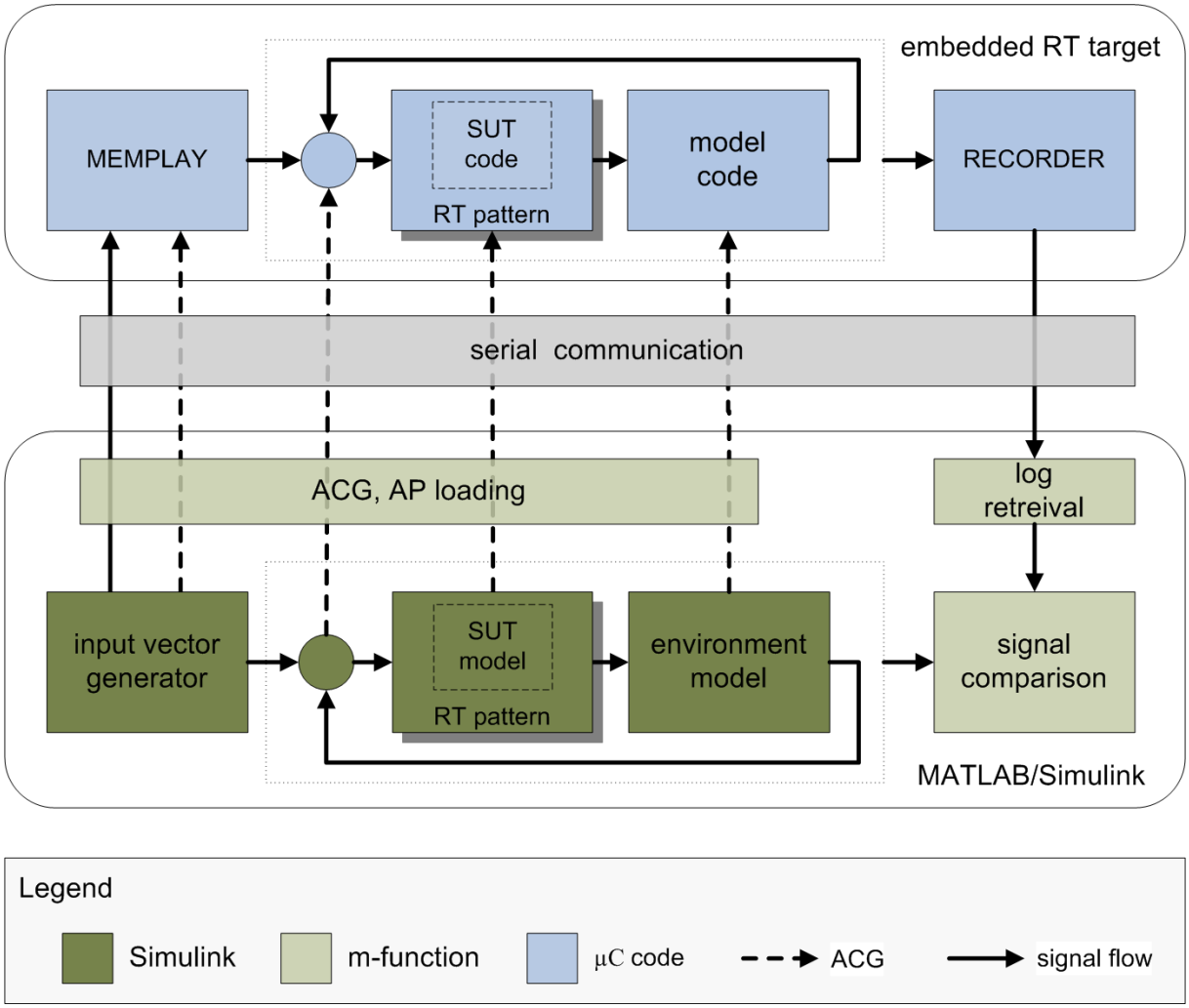


Figure 6.5: Real-time testing in a closed-loop with environment model's code in the loop

One application example of closed-loop real-time testing could be the case of tuning control algorithm parameters during commissioning of the plant where simulation environment is not

available. New parameters could than easily be checked by testing the control system in a loop with environment model running on the same target as the control algorithm. This approach assumes a target microcontroller powerful enough to simultaneously run control algorithm and model of the environment. If this is possible, application program should be structured so that the environment model is disabled during normal operation so it doesn't consume target's processing resources. During closed-loop real-time testing, environment model and control algorithm would execute while all other unrelated tasks would be suspended in order to free processing resources and stop all interaction with the physical environment.

Chapter 7

Case Study: Digital Signal Controller Component Testing

This chapter provides a case study of real-time software component testing for a digital signal controller. It starts by describing the embedded target, proceeds with validation of the RTTP, and ends by elaborating test process and by presenting test results for the *LogicalOperator* component.

7.1 Digital Signal Controller

The term "digital signal controller" designates a hybrid of microcontrollers and digital signal processors, [125]. DSCs thus incorporate features of microcontrollers, like fast interrupt response and control-oriented peripherals like pulse width modulation (PWM) units and watchdog timers, and of digital signal processors (DSPs), like single-cycle multiply–accumulate units, barrel shifters, and large accumulators. They are mostly used in motor control, power conversion, and sensor processing applications.

Proprietary digital signal controller hardware modules are based on Texas Instruments' C2000 family of microcontrollers and are used in a variety of applications ranging from power electronic control systems in electrical traction and power generation units to condition monitoring in wind turbines, rotating machinery and power transformers. Texas Instruments sometimes refers to these microcontrollers as "digital signal controllers" and sometimes as "real-time control microcontrollers". The former designation is used in this work.

7.1.1 The Controller

Texas Instruments' C2000 family of DSCs encompasses several lines of products, [126]:

- *C24x 16-bit Series* – are 16-bit controllers with CPU frequency up to 40MHz. These devices are either obsolete or not recommended for new design (NRND).
- *C28x Fixed-point Series* – are 32-bit fixed-point microcontrollers with working frequencies up to 150 MHz and a variety of integrated peripheral units.

- *C28x Delfino Floating-point Series* – microcontrollers from this series are equipped with a floating-point unit (FPU) and are available in dual-core (each core up to 200 MHz) and in single core (up to 300 MHz) versions.
- *C28x Piccolo Series* – are low cost 32-bit microcontrollers that have features of other two more advanced C28x series but work on lower frequencies and have smaller chip packages.
- *C28x + ARM Cortex-M3 Series* – integrates ARM Cortex-M3 core with C2000's 28x core on the same chip.

Members of C24x, Delfino and Piccolo DSC series are used in proprietary control systems. Real-time component testing is in this thesis showcased on the example of TMS320F28335 DSC, a single-core member of the Delfino series with following features, [127]:

- high performance 32-bit core with IEEE-754 single-precision floating-point unit, 16x16, 32x32 and double 16x16 multiply and accumulate, and Harvard architecture,
- integrated flash, ROM, single-access RAM, and boot ROM memory,
- programmable 16-bit or 32-bit external interface,
- up to eighteen PWM outputs, six capture inputs, and two quadrature encoder interfaces,
- 12-bit analog-to-digital converter with sixteen input channels,
- up to eighty eight general purpose input-output pins,
- serial peripheral interface (SPI), three serial communication interfaces (SCIs), two CAN interfaces, two multi-channel buffered serial port (McBSP) modules, inter-integrated circuit (I2C) interface.

7.1.2 Applications

Proprietary real-time monitoring and control systems based on C2000 DSCs come in two variants: modular systems with hardware modules mounted in a rack and integrated systems with fixed hardware configuration. C2000-based proprietary electronic modules are shown in 7.1. Systems containing these modules have thus far been used for:

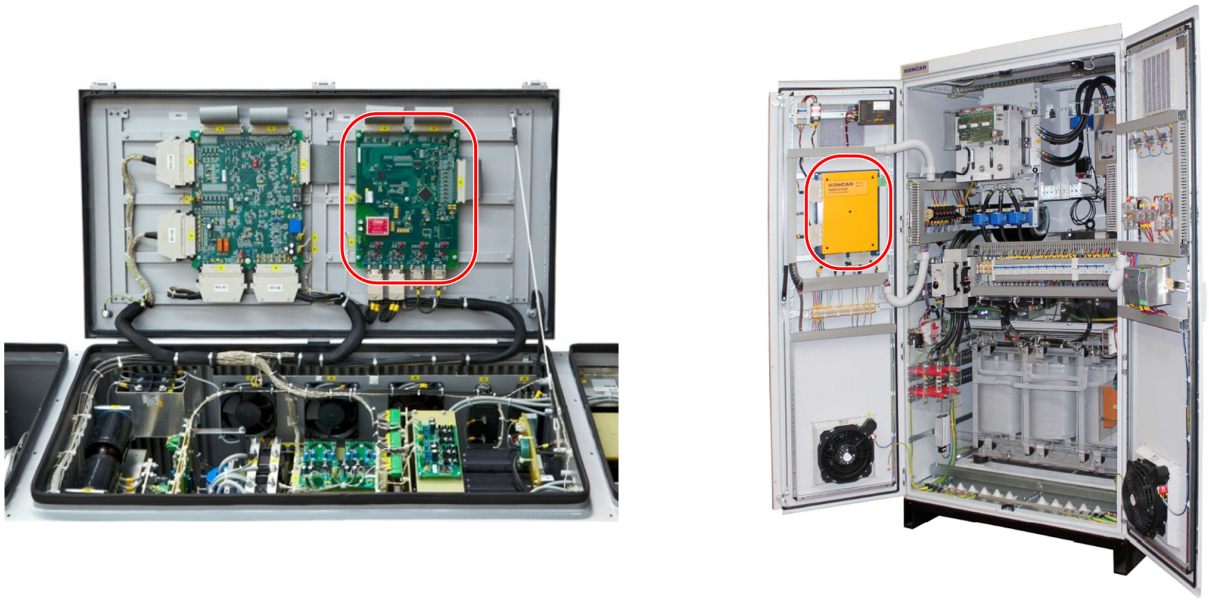
- control of multisystem static converters for power supply of railway passenger coaches,
- control of main propulsion converters and auxiliary power converters of trams and trains, Fig. 7.2,
- measurement of three phase voltage and current in generator excitation systems and in wind generators,
- condition monitoring of wind generators, rotating machinery, and power transformers.

F240 (C24x) F2407 (C24x) F28335 (C28x)

Figure 7.1: Hardware modules based on Texas Instruments' C2000 family of DSCs

Figure 7.2: Converter for main propulsion of low floor train with modular proprietary control system equipped with two electronic modules based on F2407 DSC

Short development time and flexibility are main advantages of the modular approach, but when smaller production costs and physical size are required, than integrated control systems can be a better solution. In Fig. 7.3 two such integrated control systems are show, both of which have evolved out of preexisting modular control systems.

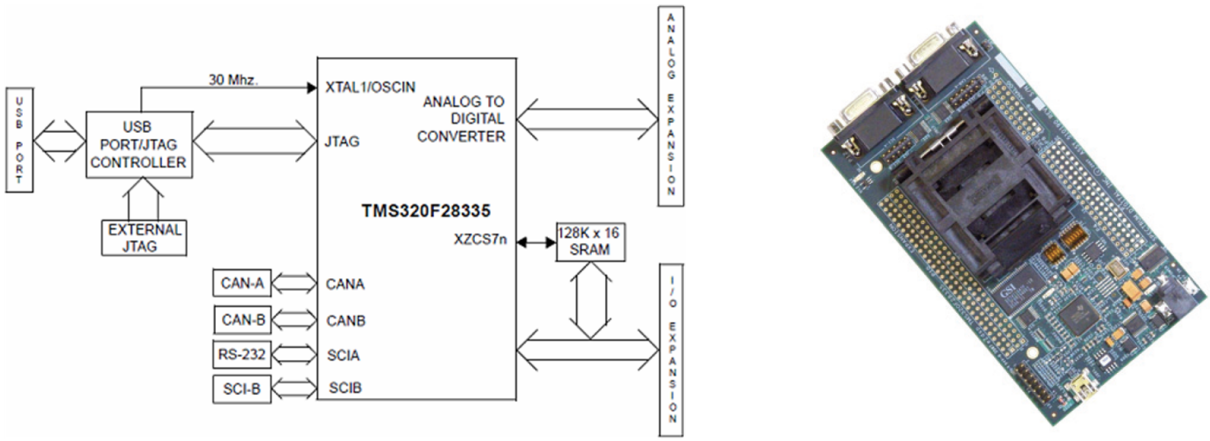


a) b)

Figure 7.3: Two power converters with proprietary integral control systems based on F28335 DSC: a) auxiliary power supply converter in tram and b) photovoltaic power converter

7.1.3 Experimental Target

Tests performed in the rest of this section are conducted on the TMS320F28335 DSC on an eZdsp28335 development board by Spectrum Digital, Inc. This board provides complete environment for the DSC including JTAG interface, external SRAM, two CAN interfaces, one RS232 and one SCI interface, analog interface and digital expansion interface. The board is equipped with 30 MHz oscillator and the DSC is configured to run with maximal 150 MHz system clock. Block diagram and photo of the development board are show in Fig. 7.4.



a) b)

Figure 7.4: a) Block diagram and b) photograph of the eZdsp28335 development board

The development board was chosen instead of proprietary hardware setups because of its

smaller dimensions which makes it more suitable for research.

7.2 Real-Time Test Pattern Validation

In this section the RTTP is checked to ensure that real-time properties are indeed tested as intended. In order to have confidence in test results, firstly confidence in test tools must be established. Mixed signal oscilloscope MSO614A from Agilent Technologies Inc. was used during these experiments.

7.2.1 Execution Time Measurement

Execution time measurement applied in RTTP is validated using model on Fig. 7.5. *Bistable* component and its' *L_RS* variant were chosen as subject of the measurement because this is a relatively simple component with fixed execution time. Validation is performed in three stages:

- Execution time measurement of the component is performed using *Tic* and *Toc* Simulink blocks, i.e. code to which they map. Software-based execution time measurement is described in section 5.5.
- One of general purpose input-output (GPIO) pins of the DSC is set during execution of one instance of the measured component. The duration of the positive pulse on this pin (T_1) represents time needed to set and clear the pin (T_{GPIO}) plus the execution time of the component (T_{CUT}).
- The same GPIO pin is set during execution of two instances of the measured component. The pulse duration here (T_2) corresponds to pin setting and clearing time (T_{GPIO}) plus execution time of two component instances ($2 \times T_{CUT}$).

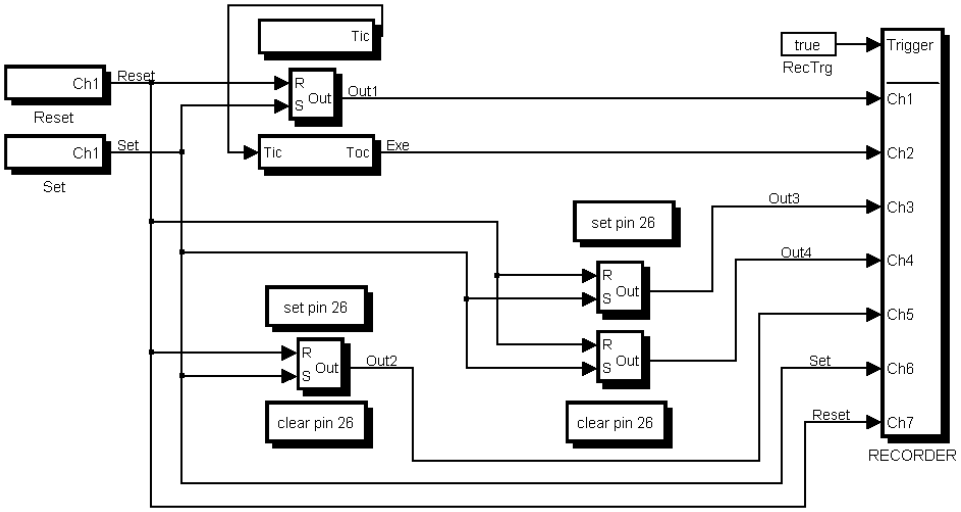


Figure 7.5: Model for validation of execution time measurement

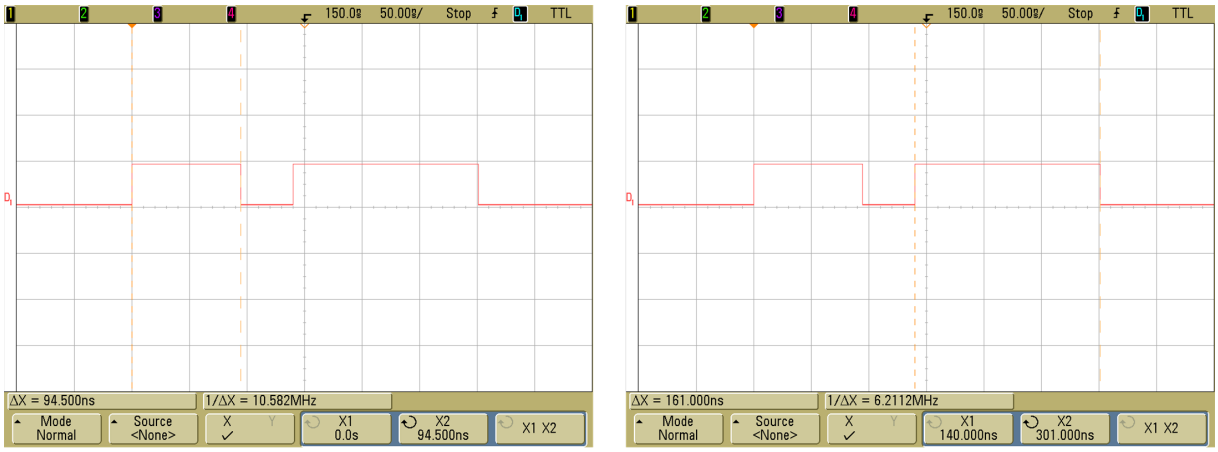
Execution time of the component is obtained by subtracting duration of the shorter pulse from duration of the longer one according to equation 7-1.

$$T_{CUT} = T_2 - T_1 = (2 \times T_{CUT} + T_{GPIO}) - (T_{CUT} + T_{GPIO}) \tag{7-1}$$

Executable code is generated from model in Fig. 7.5, loaded onto the evaluation board and GPIO pin of the DSC is monitored by oscilloscope. As expected, two pulses are observed: one shorter and one longer, Fig 7.6. According to the pulse duration measurements in Fig. 7.6, execution time of the *L_RS* variant of the *Bistable* component is:

$$\begin{aligned} T_{CUT} &= 161ns - 94.5ns = 66.5ns = 10\ CPU\ cycles \\ &(one\ cycle\ is\ 6.67ns\ at\ 150MHz\ system\ clock). \end{aligned} \tag{7-2}$$

Execution time is not measured by subtracting T_{GPIO} from $T_1 = T_{CUT} + T_{GPIO}$ because GPIO pin setting and clearing takes longer to perform if these two operations are adjacent than when there is some other code between them. This is a characteristic of the target DSC's pipeline structure and of its' *"write followed by read protection"*. By inserting additional pipeline cycles this protection ensures that if a write operation is followed by a read operation, the read value will be the same as the value written, [127].



a) b)

Figure 7.6: Execution time measurement of the L_RS *variant of the* Bistable *component by oscilloscope: a) one component instance and b) two instances*

Execution traces of the generated code are shown in Fig. 7.7. Here it can be seen that software execution time measurement by the RTTP (the *Exe* subplot) is exactly the same as hardware based measurement via oscilloscope, i.e. 10 CPU cycles.

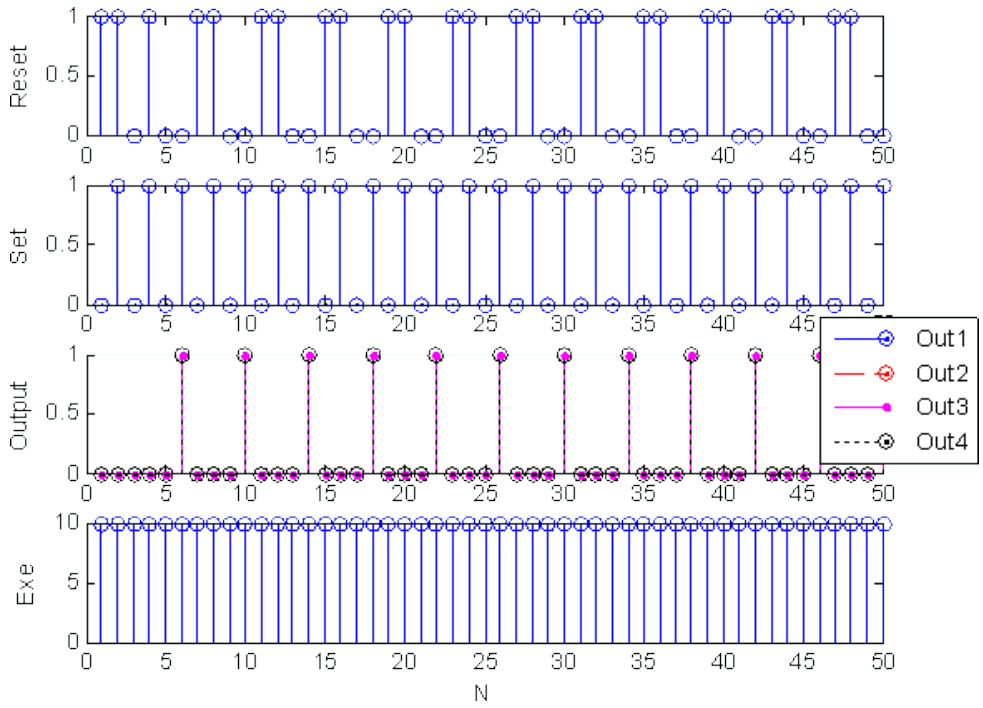


Figure 7.7: Execution traces of the code generated from the model in Fig. 7.5

7.2.2 Variable Load

The mechanism for achieving 100% interrupt coverage presented in sections 5.4 and 5.5 is validated in two steps. Firstly, variable CPU load generator is checked and, secondly, target application code parts' execution order is analyzed.

The variable CPU load block is validated using model in Fig. 7.8. The test is performed by measuring execution time of the target code corresponding to the *CpuLoad* component. Execution time measurement is performed by observing GPIO pin of the DSC using an oscilloscope.

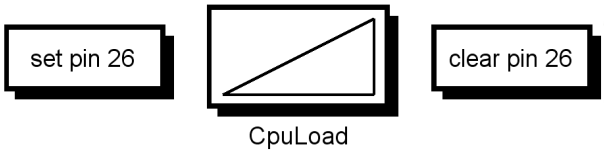


Figure 7.8: Simulink test model for variable CPU load generator

The component was configured to generate minimal load of 120 CPU cycles and maximal load of 247 CPU cycles. The oscilloscope was configured to continuous running mode and waveform of the signal on GPIO pin was recorded. As can be seen in Fig. 7.9, the minimal generated load was 827.5 ns and maximal load was 1.674 μs. If measured interval durations are divided with CPU cycle period and if 4 CPU cycles spent on GPIO pin handling (see Fig. 7.6) are subtracted, exactly the desired minimum and maximum CPU loads are obtained.

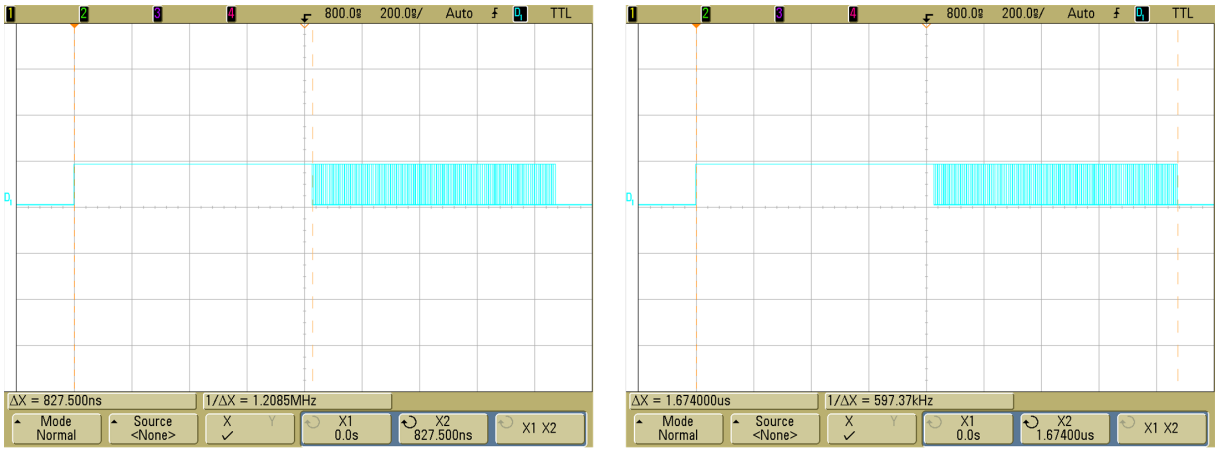


Figure 7.9: Variable CPU load generator test: a) minimal and b) maximal load measurement

A part of the waveform on the Fig. 7.9 is zoomed to check if the load increments by one CPU cycle, Fig. 7.10. A one nanosecond measurement jitter can be seen on this waveform.

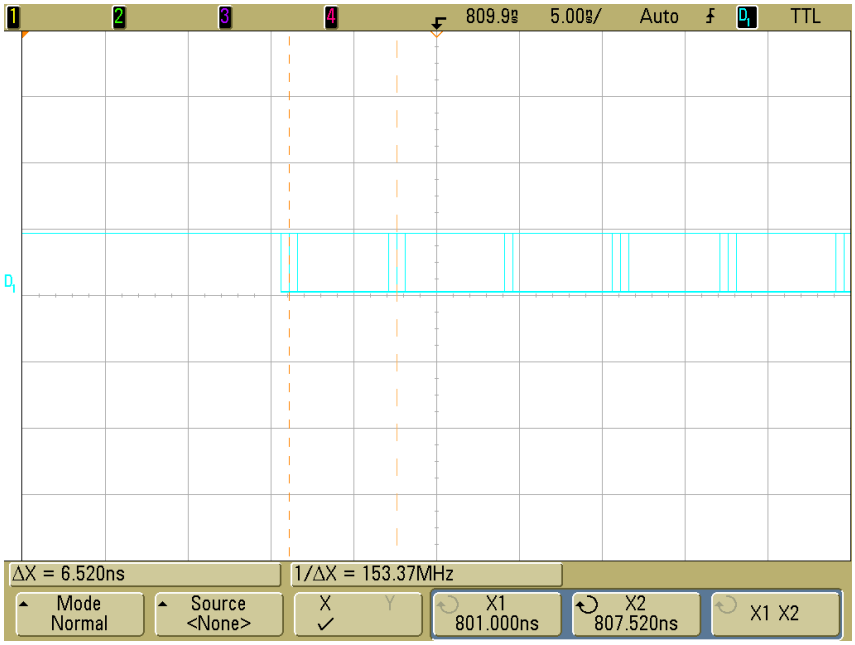


Figure 7.10: Variable CPU load increments by one CPU cycle

7.2.3 Interrupt Coverage

The second step in validation of the mechanism for achieving 100% interrupt coverage presented in sections 5.4 and 5.5 is target application code parts' execution order analysis. The application signals execution of its' parts via GPIO pins of the DSC. During execution of test application for the *L_RS* variant of the *Bistable* component, these pins have been monitored using oscilloscope. Three recorded waveforms are shown in figures 7.11, 7.12 and 7.13. Three signals have been recorded:

- *CT1* – duration of pulses correspond to execution time of the CT1 (see Fig. 5.15);

- *CUT* – duration of pulses correspond to execution time of the component under test (see Fig. 5.18);
- *CT2* – duration of pulses correspond to execution time of the CT2 (see Fig. 5.16).

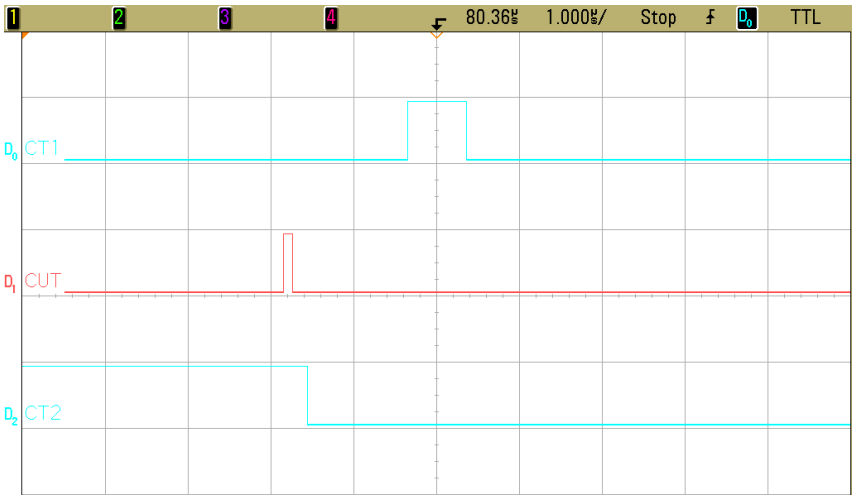


Figure 7.11: Task execution during real-time testing before the tested component starts being interrupted

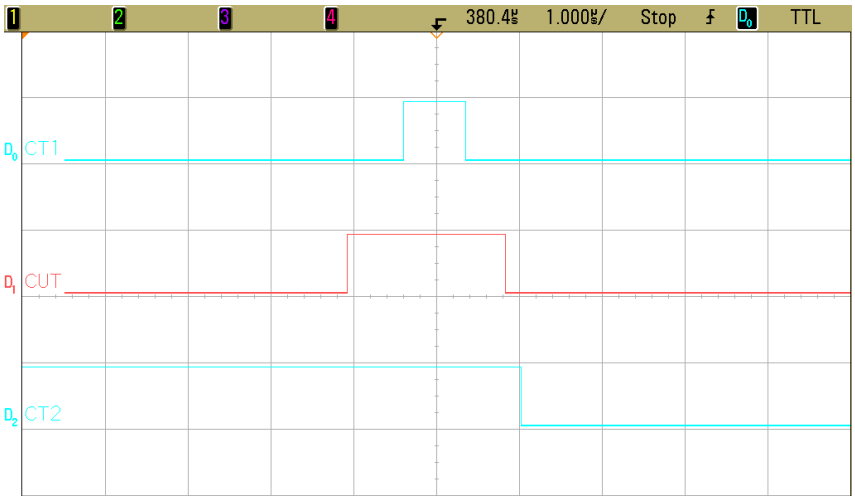


Figure 7.12: Task execution during real-time testing when the tested component is interrupted

Variable CPU load increases during test application execution by one CPU cycle from the minimal value, in which case tested code is executed before CT1 interrupts CT2 as in Fig. 7.11, to the maximal value, where tested code executes after CT1 has already interrupted CT2 as in Fig. 7.13. In-between these two extremes the CT1 interrupts tested component's execution in each possible point. This situation is shown in Fig. 7.12 where pulse duration of the *CUT* signal is extended for the duration of the interrupting task's *CT1* pulse duration.

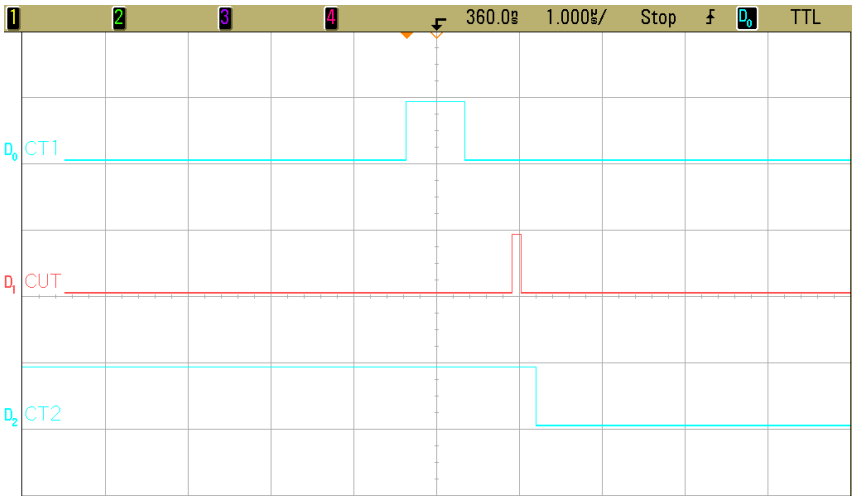


Figure 7.13: Task execution during real-time testing after the tested component has passed the "interrupt zone"

7.2.4 Interrupt Delay Detection

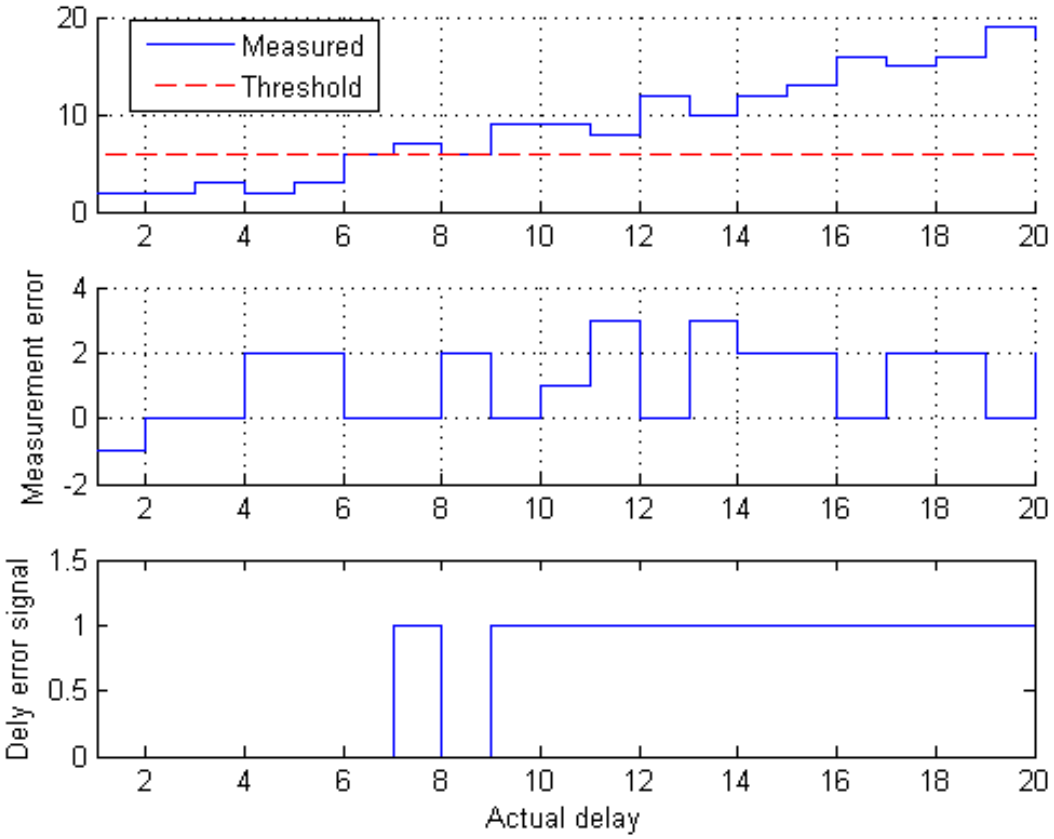


Figure 7.14: Interrupt delay detection with fixed position and rising execution time of the injected uninterruptible code

In order to validate interrupt delay detection implemented by the *IntWDog* subsystem of the test model, Fig. 5.12, a chunk of uninterruptible code was injected into implementation of the *L_RS* variant of the *Bistable* component. Two experiments were performed. Firstly twenty tests were performed with the code situated in the same place inside the implementation but with duration of the uninterruptible code execution ranging from one to twenty CPU cycles. Measured interrupt delay, delay measurement error and delay error signal generated by the *IntWDog* subsystem implementation were recorded and are plotted against injected delay in Fig. 7.14. Here it can be seen that interrupt delay is correctly measured with -1/+3 CPU cycles variation and that all delays greater than eight CPU cycles are reported as error.

In the second experiment, an uninterruptible code with fixed execution time of twenty CPU cycles was injected in every possible position inside the implementation of the *L_RS* variant of the *Bistable* component. Because its' DSC implementation has six lines of assembly code, seven tests were performed. Measured interrupt delay, delay measurement error and delay error signal generated by the *IntWDog* subsystem implementation were recorded and are plotted against position of the delay injection in Fig. 7.15. The delay is correctly measured with -1/+1 variation and reported as error in each of the seven injection positions.

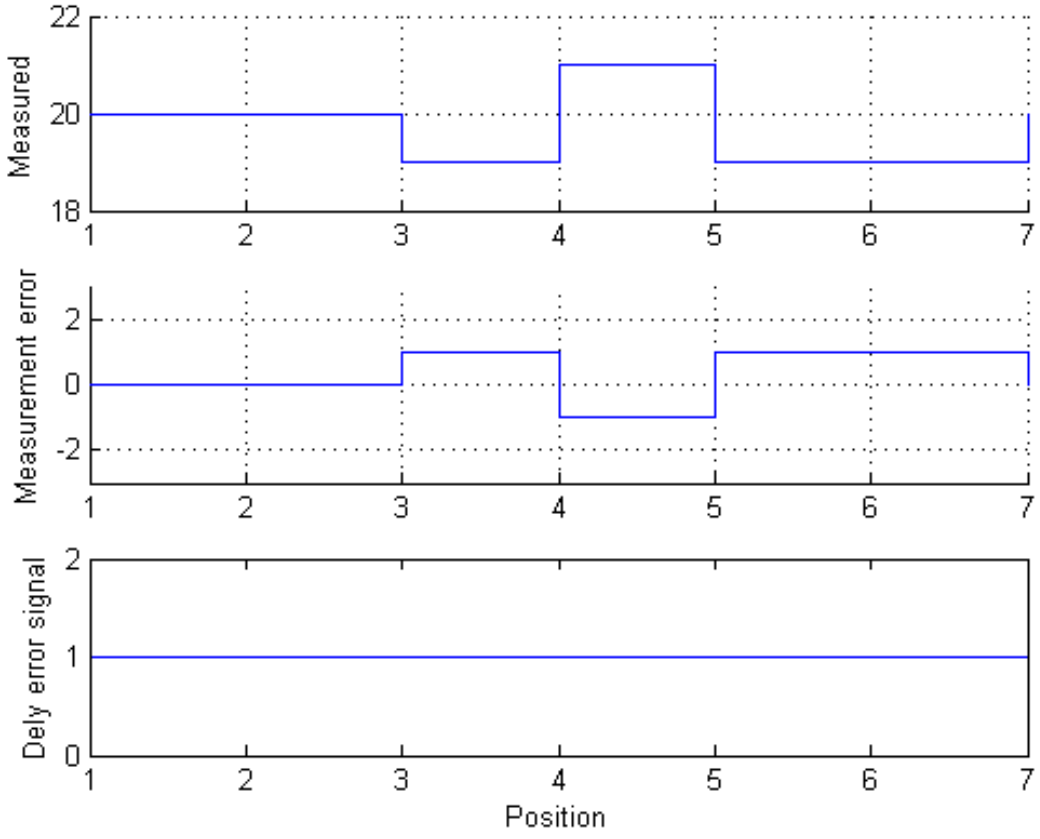


Figure 7.15: Interrupt delay detection with variable position and fixed execution time of the injected uninterruptible code

7.2.5 Interrupt Vulnerability

A component is vulnerable to interrupts if it uses some resource that is not protected by the context switching procedure. Execution of such component can be interrupted by a task with higher priority that contains code which also uses the critical unprotected resource. When execution of the interrupted component is than resumed, the shared resource can be corrupted resulting in an incorrect operation of the component.

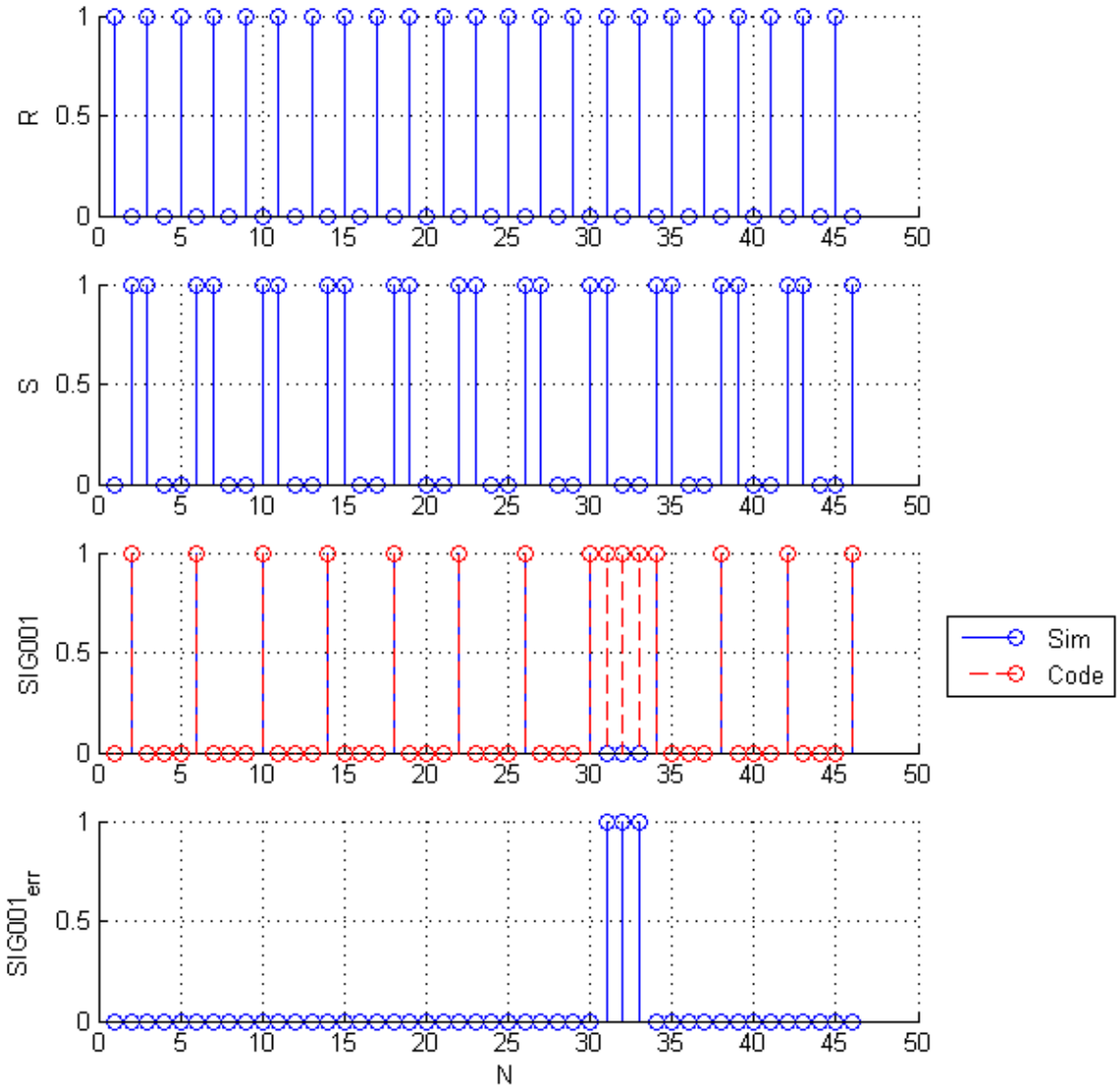


Figure 7.16: Interrupt vulnerability detection in L_RS *variant of the* Bistable *component*

Interrupt vulnerability detection has been validated by injecting a piece of erroneous code into implementation of the *L_RS* variant of the *Bistable* component. The added code stores variant's output into an unprotected memory location, loads it from the same location, and only then outputs it. Because 100% interrupt coverage is ensured by the test pattern, execution sequence should at one point look as:

1. tested variant on CT2 calculates its' output and stores it to unprotected location,
2. CT1 interrupts CT2,
3. variant instance on the CT1 modifies the unprotected location by storing its' output value,
4. CT2 resumes,
5. tested variant on CT2 loads corrupted value from the unprotected location and outputs it.

Test results for the interrupt vulnerable implementation of the *L_RS* variant of the *Bistable* component are shown in Fig. 7.16. Reset and set inputs to the component are shown in sub-figures designated by *R* and *S*, respectively. Simulation and code execution traces are compared in *SIG001* sub-figure and their difference is displayed in $SIG001_{err}$ sub-figure. Discrepancy between simulation and code execution traces indicates an error in the variant implementation.

If the test is repeated with code on CT1 suspended, i.e. if the tested component is not interrupted by its' instances on the higher priority task, than there is no difference between simulation and code execution traces. This way, test engineer can quickly determine whether the problem is in functional logic or in resource manipulation.

7.3 *LogicalOperator* Component

The *LogicalOperator* GRAPlab component masks native Simulink block *Logical Operator* in order to limit its' configuration space to area suitable for GRAP code generation. The mask also provides interface for code generation settings. Fig. 7.17 shows two tabs of the component's dialog window. The *Settings* tab provides configuration of the component while the *Code generation* enables parametrization of the code generation for the individual component.

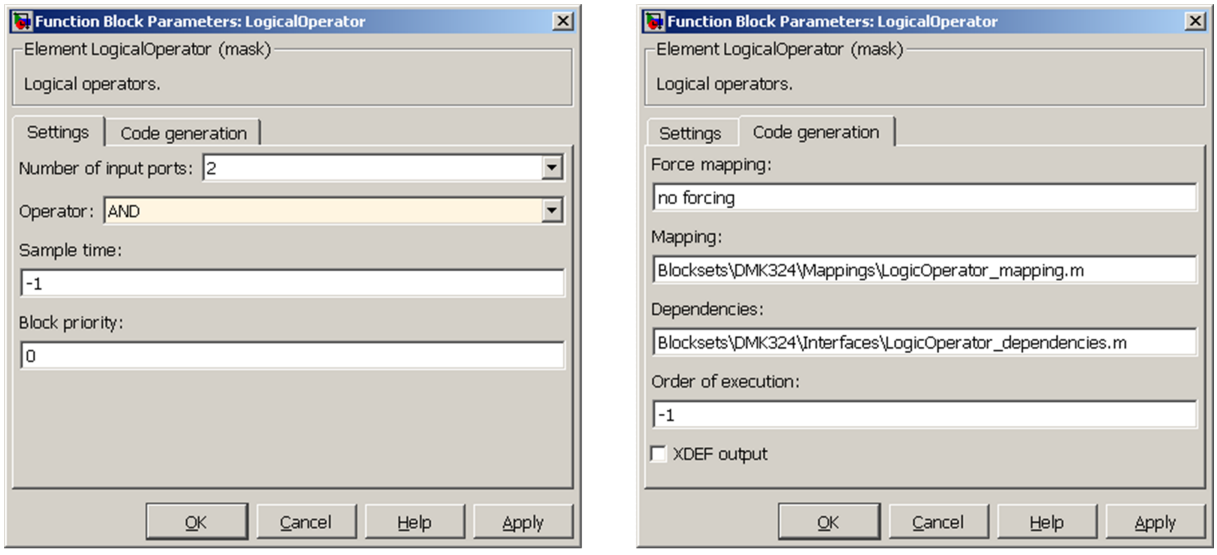


Figure 7.17: LogicalOperator *component's a)* Settings *dialog and b)* Code generation *dialog*

7.3.1 Variants and Configurations

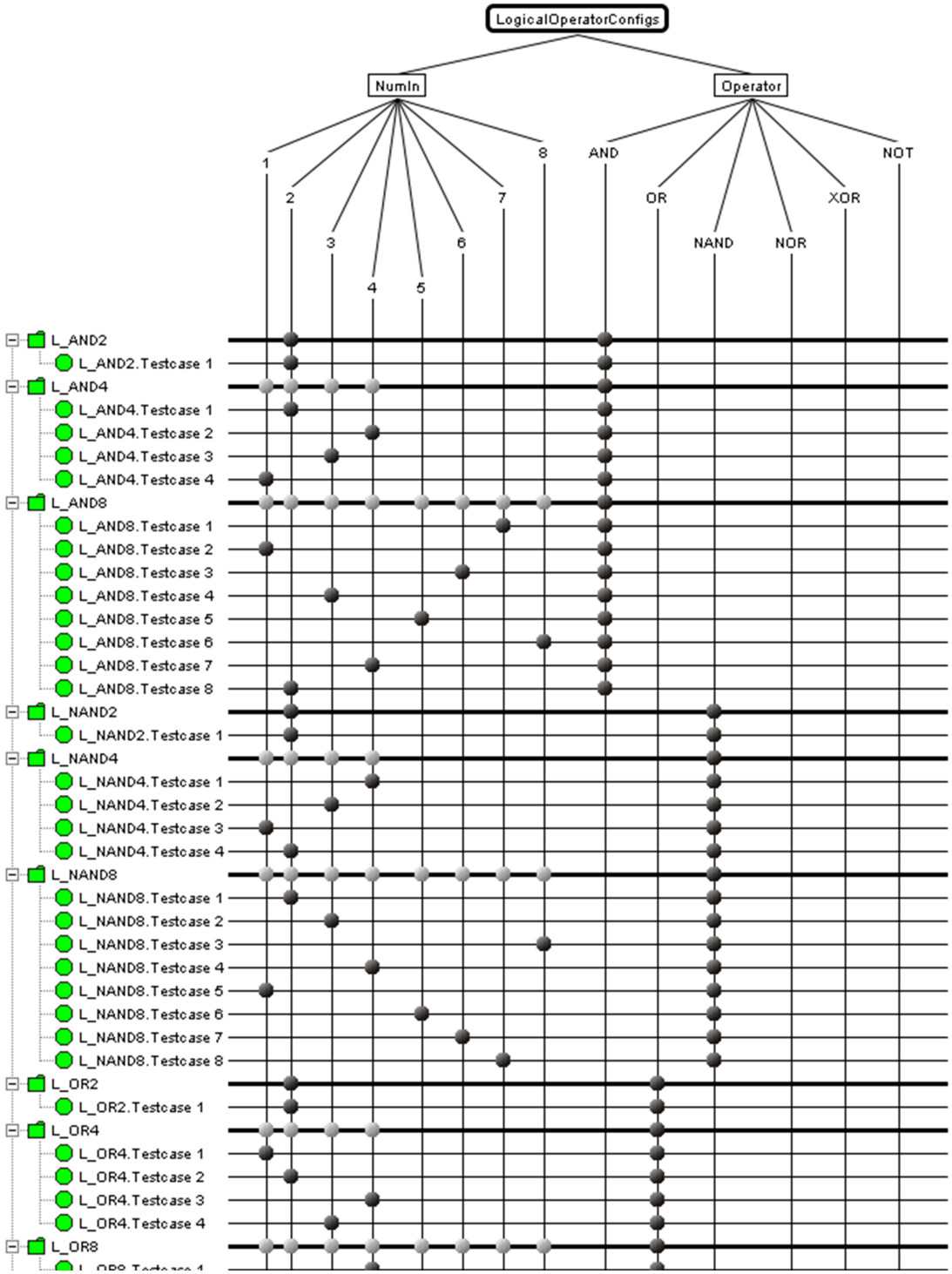


Figure 7.18: LogicalOperator *component variants and configurations, 1st part*

Depending on the *Number of input ports* and *Operator* selections on the dialog window in Fig. 7.17, the *LogicalOperator* component has fourteen variants. Variants have from one to eight configurations, based on number of connected inputs. Configuration classification tree for

the *LogicalOperator* component is shown in figures 7.18 and 7.19.

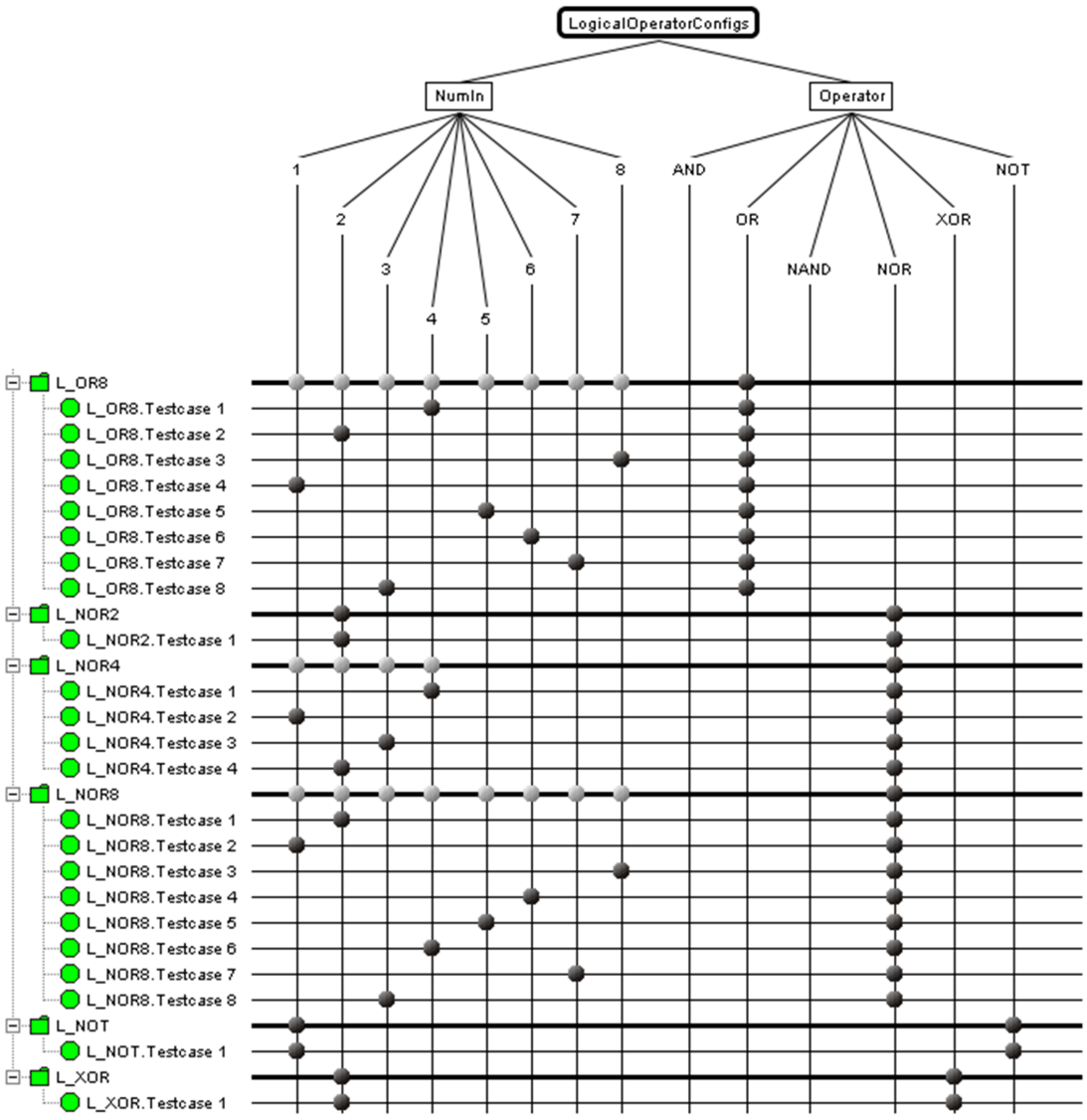


Figure 7.19: LogicalOperator *component variants and configurations, 2nd part*

7.3.2 Vectors

For each of the fifty-four configurations of fourteen variants of the *LogicalOperator* component, a set of test vectors is automatically generated. *LogicalOperator*'s inputs can assume two discrete values: *true* and *false*. Vectors are generated so that every combination of input values is selected in one test step. Vectors classification tree and corresponding signal waveforms for *TC002* configuration of the *L_AND4* variant of the *LogicalOperator* component are presented in figures 7.20 and 7.21, respectively.

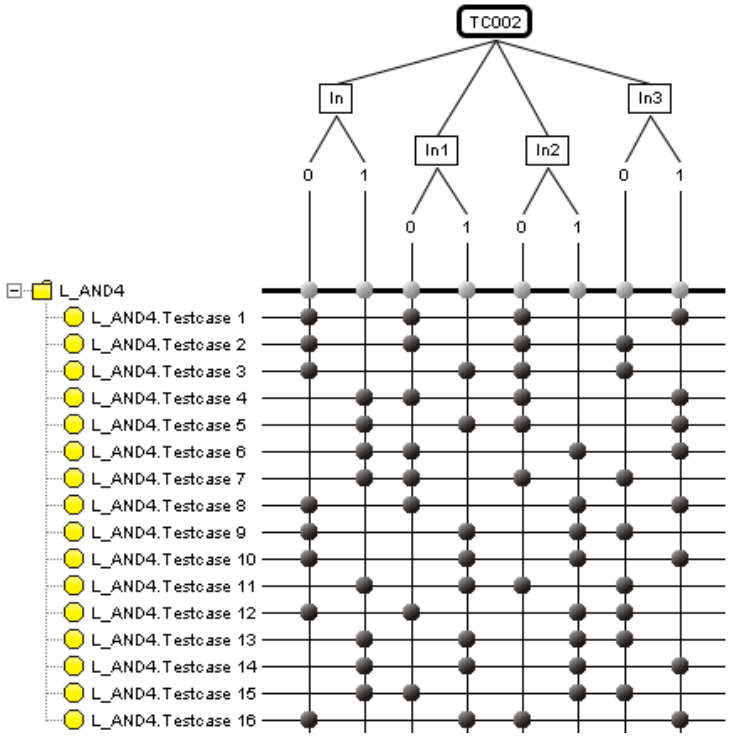


Figure 7.20: Test vector classification tree for the TC002 *configuration of the* L_AND4 *variant of the* LogicalOperator *component*

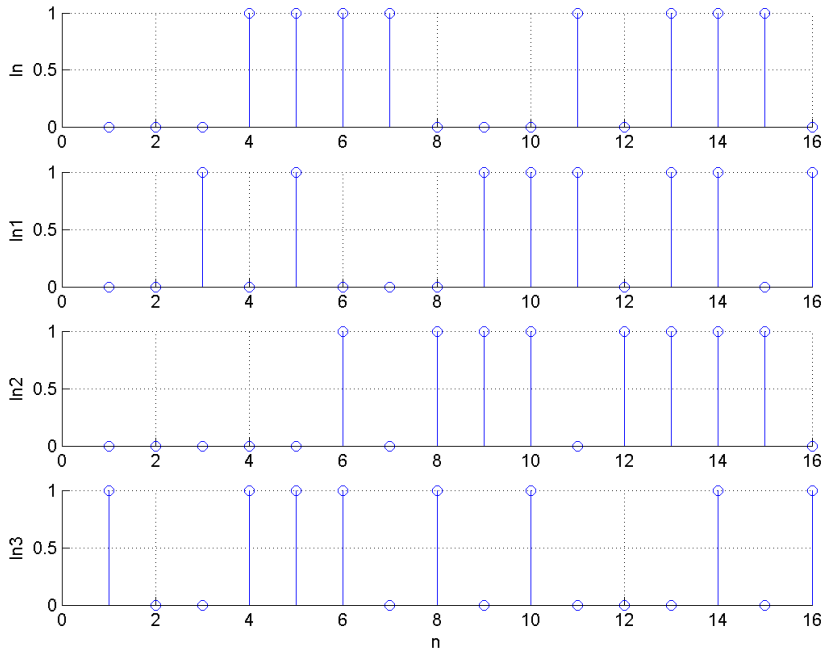


Figure 7.21: Test vector waveforms for the TC002 *configuration of the* L_AND4 *variant of the* LogicalOperator *component*

7.3.3 Models

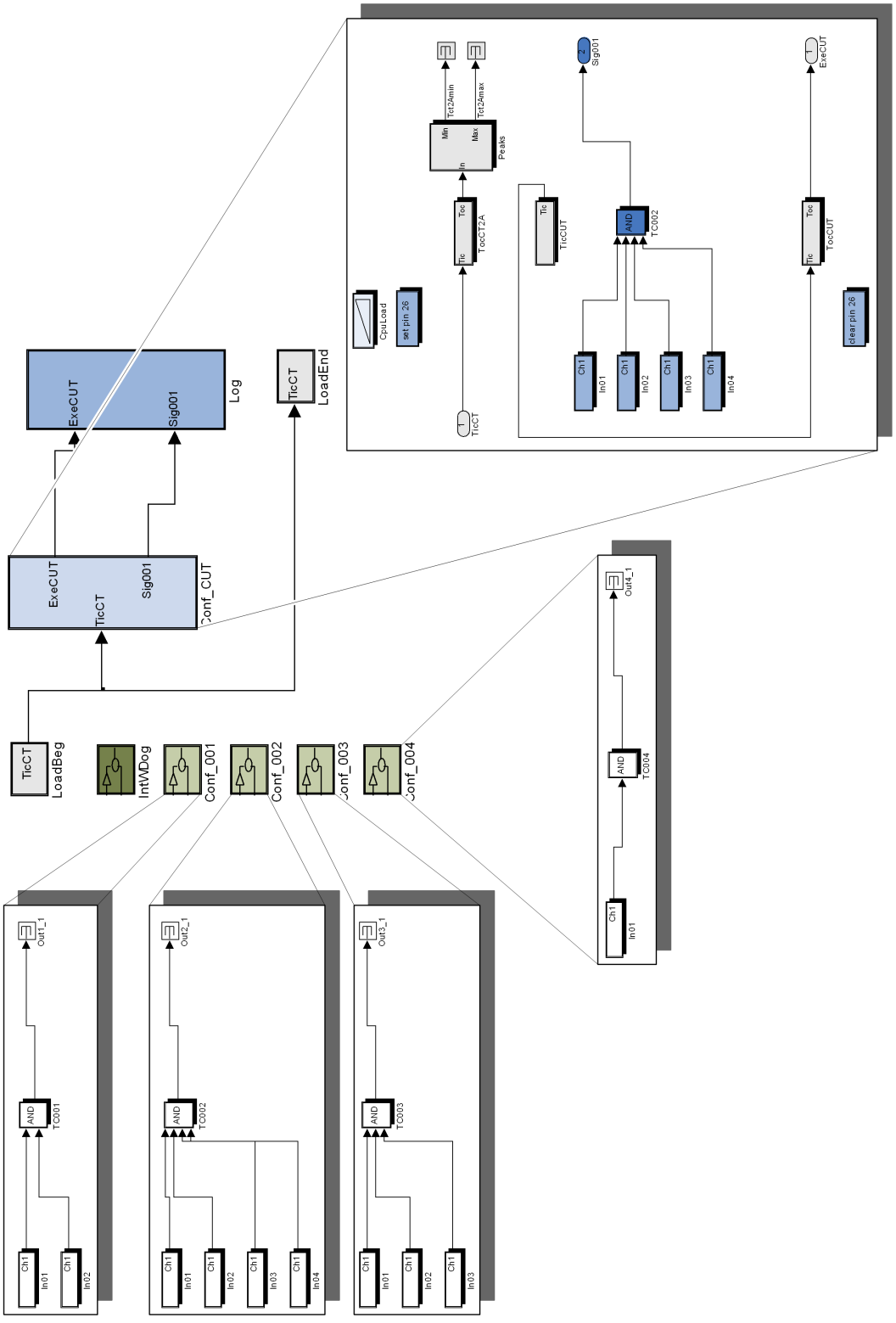


Figure 7.22: Test model of the TC002 *configuration of the* L_AND4 *variant of the* LogicalOperator *component*

The next step in test case creation is generation of test models. A separate test model for each of the fifty-four configurations of fourteen variants of the *LogicalOperator* component is automatically generated based on the RTTP introduced in section 5.4. An example of the test model for the *TC002* configuration of the *L_AND4* variant of the *LogicalOperator* component is shown in Fig. 7.22. All four configurations of the *L_AND4* variant with corresponding input vectors are placed on the high priority task CT1 inside *Conf_001* to *Conf_004* subsystems. The tested configuration *TC002* is situated inside *Conf_CUT* subsystem executed on the low priority task CT2.

7.3.4 Executable code and test results

Two instances of executable code were generated from each of the test models in order to measure execution times of parts of the code. Based on these measurements, parameters for the actual test runs were determined, test models were updated and simulated, and the final executable tests were generated. This process is detailed in section 5.5.

Graphical representation of the functional aspect of the test results for the *TC002* configuration of the *L_AND4* variant of the *LogicalOperator* component is shown in Fig. 7.23. Four input signals from Fig. 7.21 are here all shown in the first sub-figure, second sub-figure shows comparison of the simulation and code execution traces, and in the third sub-figure their difference is shown.

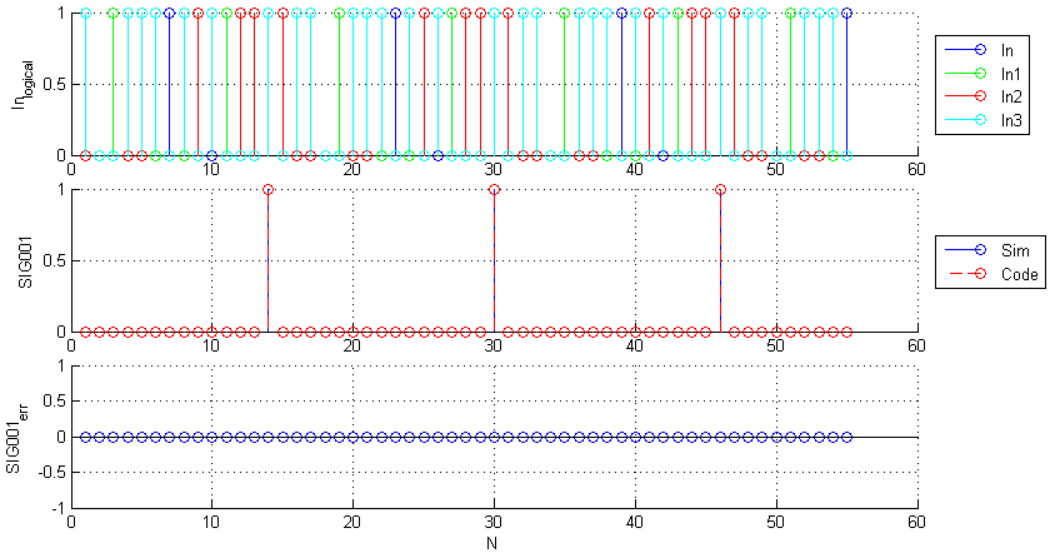


Figure 7.23: Test vectors, output and output error of the TC002 *configuration of the* L_AND4 *variant of the* LogicalOperator *component*

Similar graphical representations are automatically generated for each of the fifty-four configurations of fourteen variants of the *LogicalOperator* component. This allows test engineer to quickly check the test results and provides first input for analysis in case of test failure.

Real time properties of the tested component, namely execution times and introduced interrupt delay, are provided in Tab. 7.1. Three execution times of the configurations are provided in the table:

- BCET – is the minimal execution time,
- ACET – is the average execution time, and
- WCET – is the maximal execution time.

As can be seen in the table, the *LogicalOperator* is a highly deterministic component with always the same execution time, that is BCET, ACET and WCET are the same, and with minimal introduced interrupt response delay.

Table 7.1: Summary of the LogicalOperator *component testing*

Variant	**Config.**	**BCET**	**ACET**	**WCET**	**Delay**
L_AND2	TC001	6	6	6	1
L_AND4	TC001	6	6	6	1
	TC002	10	10	10	1
	TC003	8	8	8	1
	TC004	4	4	4	1
L_AND8	TC001	16	16	16	1
	TC002	4	4	4	1
	TC003	14	14	14	1
	TC004	8	8	8	1
	TC005	12	12	12	1
	TC006	18	18	18	1
	TC007	10	10	10	1
	TC008	6	6	6	1
L_NAND2	TC001	7	7	7	1
L_NAND4	TC001	11	11	11	1
	TC002	9	9	9	1
	TC003	5	5	5	1
	TC004	7	7	7	1
L_NAND8	TC001	7	7	7	1
	TC002	9	9	9	1
	TC003	19	19	19	1
	TC004	11	11	11	1
	TC005	5	5	5	1
	TC006	13	13	13	1
	TC007	15	15	15	1
	TC008	17	17	17	1

Table 7.1: Continued from previous page...

Variant	Config.	BCET	ACET	WCET	Delay
L_NOR2	TC001	7	7	7	1
L_NOR4	TC001	11	11	11	1
	TC002	5	5	5	1
	TC003	9	9	9	1
	TC004	7	7	7	1
L_NOR8	TC001	7	7	7	1
	TC002	5	5	5	1
	TC003	19	19	19	1
	TC004	15	15	15	1
	TC005	13	13	13	1
	TC006	11	11	11	1
	TC007	17	17	17	1
	TC008	9	9	9	1
L_NOT	TC001	5	5	5	1
L_OR2	TC001	6	6	6	1
L_OR4	TC001	4	4	4	1
	TC002	6	6	6	1
	TC003	10	10	10	1
	TC004	8	8	8	1
L_OR8	TC001	10	10	10	1
	TC002	6	6	6	1
	TC003	18	18	18	1
	TC004	4	4	4	1
	TC005	12	12	12	1
	TC006	14	14	14	1
	TC007	16	16	16	1
	TC008	8	8	8	1
L_XOR	TC001	6	6	6	1

Another validated real-time property that is not shown in the Tab. 7.1 is immunity to interrupts, i.e. the ability of the component to correctly fulfils its' function disregarding interrupts by the higher priority task. All code execution traces match simulation traces of the test models which, together with 100% interrupt coverage proven in section 7.2, shows that this component is immune to interrupts.

Chapter 8

Case Study: Safety Platform Component Testing

Safety Platform is a part of the railway crossing (level crossing) control system. Its' purpose is to secure the crossing by entering safe state upon activation. The system can be activated in a number of ways: when train triggers activation contacts, by command from the station or locally. After system activation, light and sound signals are activated and half barriers are lowered. When train leaves protected area, light and sound signals are deactivated and half barriers are raised, Fig. 8.1.

Figure 8.1: An example of level crossing with half barrier and light signalization

The level crossing control system is comprised of:

- *Safety Platform* – is the central part of the control system and it consists of two redundant channels, *Safety Platform A* and *Safety Platform B*. Here relevant data is stored and safety relevant decisions are taken.
- *Data Logging Subsystem* – consists of *Data Display Module* and its function is to log and display all important events in the system.

- *Remote Control Subsystem* – enables remote control over the system.
- *Wayside Equipment Subsystem* – includes all equipment mounted near railway or road, like railway station devices, absolute permissive block devices, half barriers etc.
- *Train Detection Subsystem* – consists of different types of equipment necessary for reliable detection of train presence. This could be axle counters, induction loops or other detections subsystem.

8.1 Safety Platform

Safety Platform consists of hardware, system software and application software. This section provides a short overview of the hardware and systems software. The application software for the safety platform is outside the scope of this thesis.

Besides fulfilling customer requirements, the safety platform should comply to requirements of Safety Integrity Level (SIL) 4 according to standards:

- EN 50126 (*Railway applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety*),
- EN 50128 (*Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*), and
- EN 50129 (*Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling*).

8.1.1 Safety Platform Hardware

Safety platform hardware is organized in a modular structure with electronic modules grouped inside mounting racks. A photo of safety platform mounting rack with installed electronic modules taken during clearances and creepage distance testing is shown in Fig. 8.2.

Figure 8.2: Safety platform mounting rack with installed electronic modules

The safety hardware platform includes four electronic modules based on Silicon Laboratories' C8051F580 microcontroller, [128]:

- *Central Control Module* – is the central processing module of the safety platform. It executes main application program, receives states of digital inputs and determines states of all outputs. It is also responsible for communication with other channel, with data logger, and with remote control.
- *Input Digital Module* – is a 28-channel intelligent digital input module that supports input sequence logging, digital filtering, and other functions.
- *Output Relay Module* – is intended for control of different executive functions by means of four safety relays.
- *Wayside Driver Module* – controls wayside light signalization (bulbs or LED arrays) and sound signalization (bells).

The enumerated electronic modules are powered by the *Power Supply Module* and they communicate with each other via *Backplane*.

8.1.2 Safety Platform System Software

Safety platform system software comprises of firmware, that includes the real-time kernel and the monitor, and libraries of program elements, each of which includes a library of graphical symbols and an assembly macro library.

Real-time kernel is responsible for handling of task execution order and priorities. In the Safety Platform, number of concurrent periodic tasks is reduced to only one. There exist one more task used for serial inter-channel or RS485 communication.

Monitor part of the firmware is responsible for communication with the user. It enables basic operations like application program loading or erasing, viewing and changing system parameters, signals, and memory locations, and similar operations.

In short, safety platform system software comprises of same type of elements organized in same fashion as in all other proprietary embedded control systems, described in section 1.1.

8.1.3 Safety Integrity Level

The most severe requirement placed upon the Safety Platform was that it has to be certified as a SIL 4 system. The European standard EN 50126, [129], defines safety integrity as:

"The likelihood of a system satisfactorily performing the required safety functions under all the stated conditions within a stated period of time."

The same standard than defines SIL as:

"One of a number of defined discrete levels for specifying the safety integrity requirements of the safety functions to be allocated to the safety related system. Safety Integrity Level with the highest figure has the highest level of safety integrity."

In [130], the Safety Integrity Levels are determined with regard to targeted failures per year for high demand systems (e.g. car brakes) and to probability of failure on demand for low demand systems (e.g. car air bag), table 8.1.

Table 8.1: Safety Integrity Levels

SIL	High demand rate (failures/year)	Low demand rate (probability of failure on demand)
4	$\geq 10^{-5}$ to $< 10^{-4}$	$\geq 10^{-5}$ to $< 10^{-4}$
3	$\geq 10^{-4}$ to $< 10^{-3}$	$\geq 10^{-4}$ to $< 10^{-3}$
2	$\geq 10^{-3}$ to $< 10^{-2}$	$\geq 10^{-3}$ to $< 10^{-2}$
1	$\geq 10^{-2}$ to $< 10^{-1}$	$\geq 10^{-2}$ to $< 10^{-1}$

Railway crossing control system, and Safety Platform as its' part, are considered as a high demand rate system so SIL 4 prescribes that targeted number of failures per year should be between 10^{-5} and 10^{-4}. This means that the system is expected to operate without failure between 10,000 and 100,000 years.

As a part of the Safety Platform, its' software must also be certified as SIL 4. To achieve this, all development processes have been conducted in accordance with European standard EN 50128, [131]. The software component testing presented in the remainder of this chapter is a part of this process.

8.2 Component Testing for Safety Platform

This section provides an overview of differences between generic component testing process, chapters 5 and 7, and its' variant customized for Safety Platform. Also some specific examples from each phase of testing are presented.

8.2.1 Real-Time Properties

Software component real-time testing described in chapter 5 can be significantly simplified for the testing of the Safety Platform's atomic software components. This is due to the fact that the Safety Platform's real-time kernel supports only one concurrent cyclic task so there is no need to validate interrupt-related real-time properties.

The software component real-time testing is in absence of interrupts reduced to component execution time measurement. The C8051F580 microcontroller that constitutes the backbone of the Safety Platform is operating with 24 MHz system clock so each CPU cycle takes 41.67 ns to complete. In order to precisely measure software execution time, the application program should have access to a timer running on the same frequency. Unfortunately, only a timer running on 24/12 MHz = 2 MHz is available to the application program. All execution time measurements are thus in $N_{SYSCLK}/12$ so elements that execute in less than 12 clocks can have BCET = 0. For example, this is the case with *GATE* component which has BCET of zero accomplished when the component's operation is suspended via *ENABLE* input, see table 8.3.

8.2.2 Variants and Configurations

Safety Platform and DSC GRAPlab blocksets are composed of the so called "linked blocks" that point to block definitions in a common block repository. The source blocks are general so that they cover functionality for both target systems and could be further expanded in case of introduction of a new target. The general block's interface and code generation mapping are limited according to the target, as described in chapter 4. This approach can lead to oversized configuration classification tree in cases where these limitations are substantial.

For example, in order to simplify component implementation and testing, all ports of atomic components (GRAP programming elements) must be connected in an application program for Safety Platform. This results in Safety Platform's components usually having less variants and configurations than components in DSC development environment. An example of such case is *LogicalOperator* component which in Safety Platform has configuration classification tree with eight variants each of which has a single configuration, Fig. 8.3. The same component in DSC environment has fourteen variant with fifty-four configurations altogether, see section 7.3.

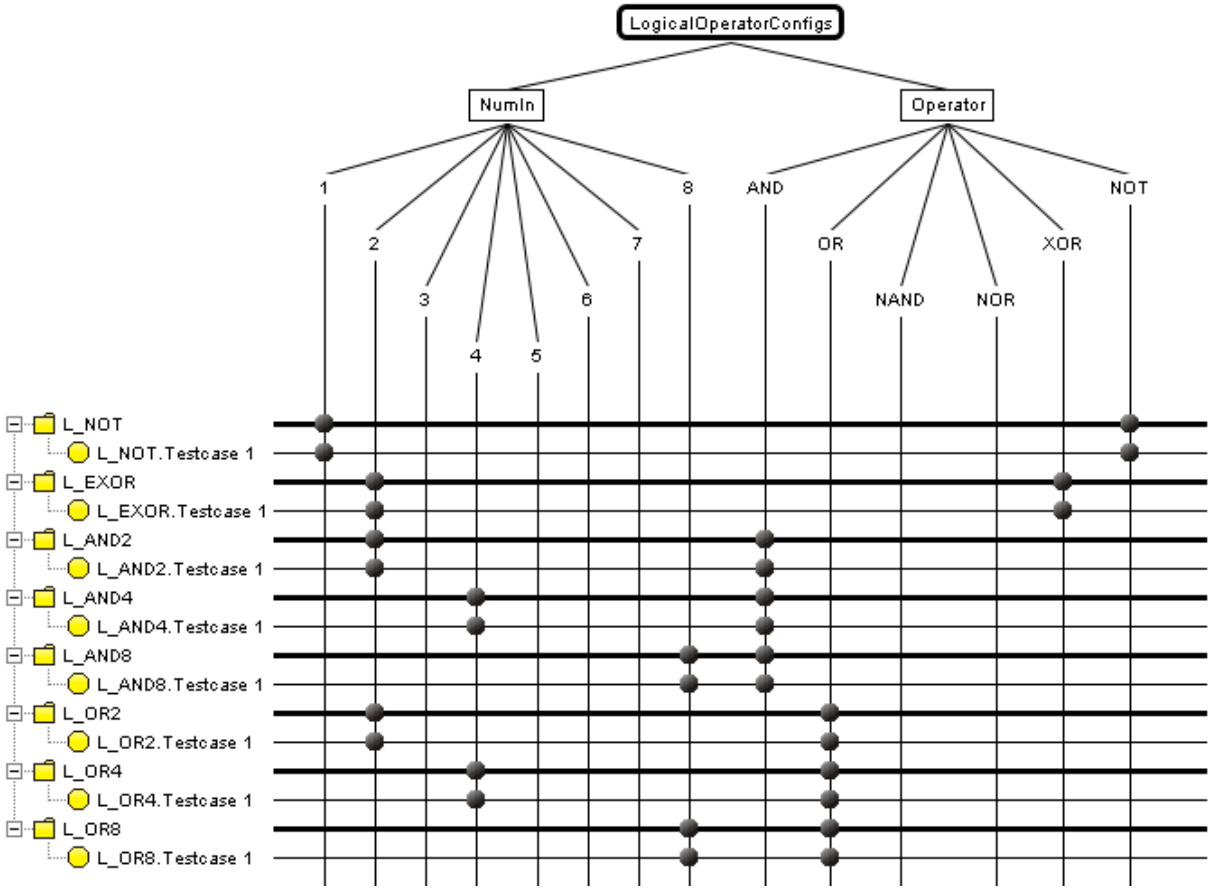


Figure 8.3: LogicalOperator *component variants and configurations classification tree is greatly simplified in comparison to the same component in DSC environment*

It is possible for one configuration of the component variant to be implemented with more than one atomic components. An example for this is the *CRCCALC* component for calculation of Cyclic Redundancy Check (CRC) codes. In Safety Platform, CRC algorithm is implemented

using lookup tables. Two atomic components comprise an implementation: one component defines the lookup table and the other performs the actual calculations. CRC calculation using "Baicheva" and "Modbus" polynomials are supported by two different lookup tables. Besides polynomial selection, the *CRCCALC* component can be configured to have 4 or 8 inputs. These two options result in a configuration classification tree shown in Fig. 8.4.

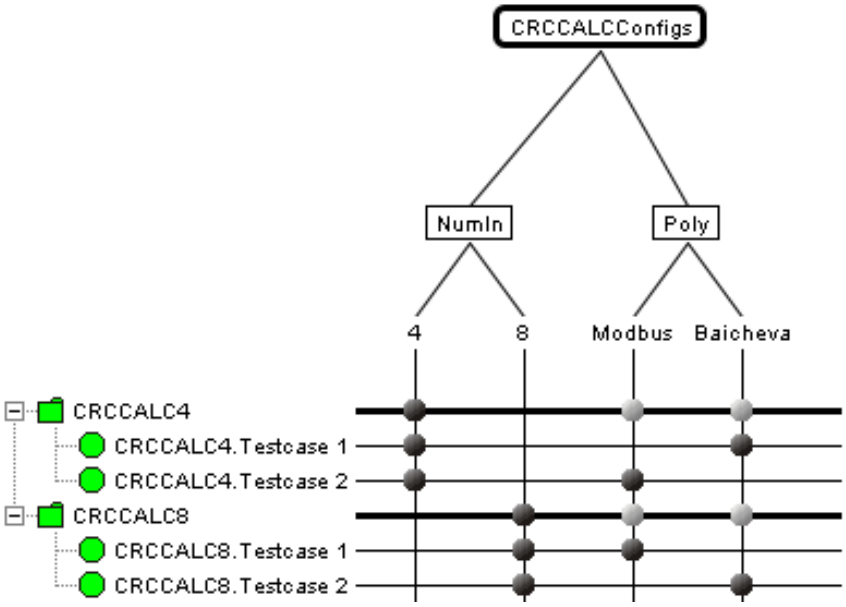


Figure 8.4: CRCCALC *component variants and configurations*

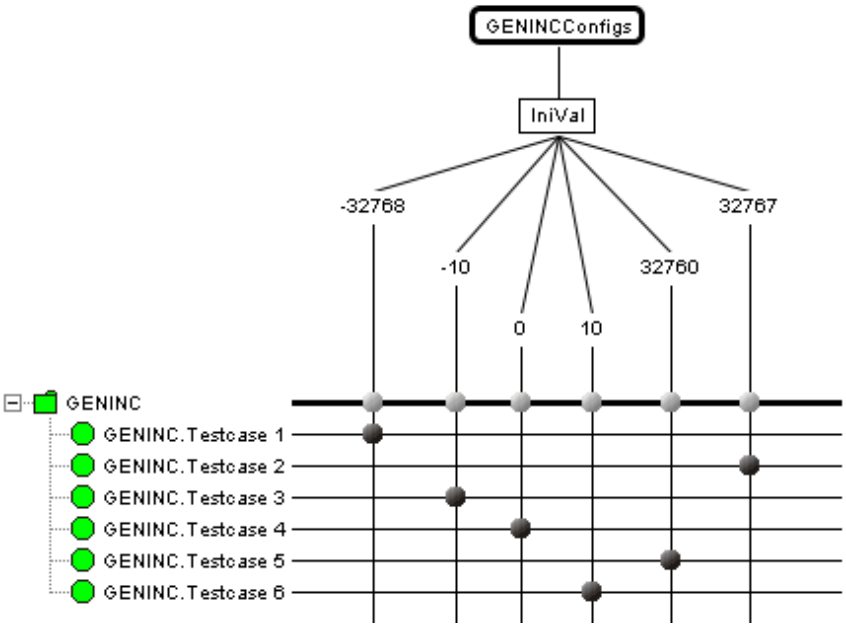


Figure 8.5: GENINC *component variants and configurations*

If atomic component's input is considered only during initialization, than it is represented as a parameter of the GRAPlab component and is partitioned in configuration classification

tree, rather than in vector classification tree as other "regular" inputs. This is the case with *INIVAL* input of the *GENINC* atomic component for generation of incrementing sequence. This component is usually used for detection of lost messages in different communication channels and the mentioned input determines the value from which the incrementing sequence is started. The configuration classification tree for *GENINC* component is shown in 8.5.

In some events, though the input is considered during run-time it makes little sense to change it during execution. This applies to filtering constant of the *LFILTP* variant of the *Debounce* component. *LFILTP* atomic component is actually a low-pass filter of a packed logical signal. Up to sixteen logical, i.e. boolean, signals can be packed into one 16-bit signal of the "packed logical" type saving this way bandwidth, if signals are to be transferred through a communication channel, or memory, if they are to be stored locally. *LFILTP* atomic component performs low pass filtering of the individual bits of such packed signals. In other words, it debounces the packed logical signals. Automatically generated classification tree for the *Debounce* component has been manually modified with sensible filtering constant values, Fig. 8.6.

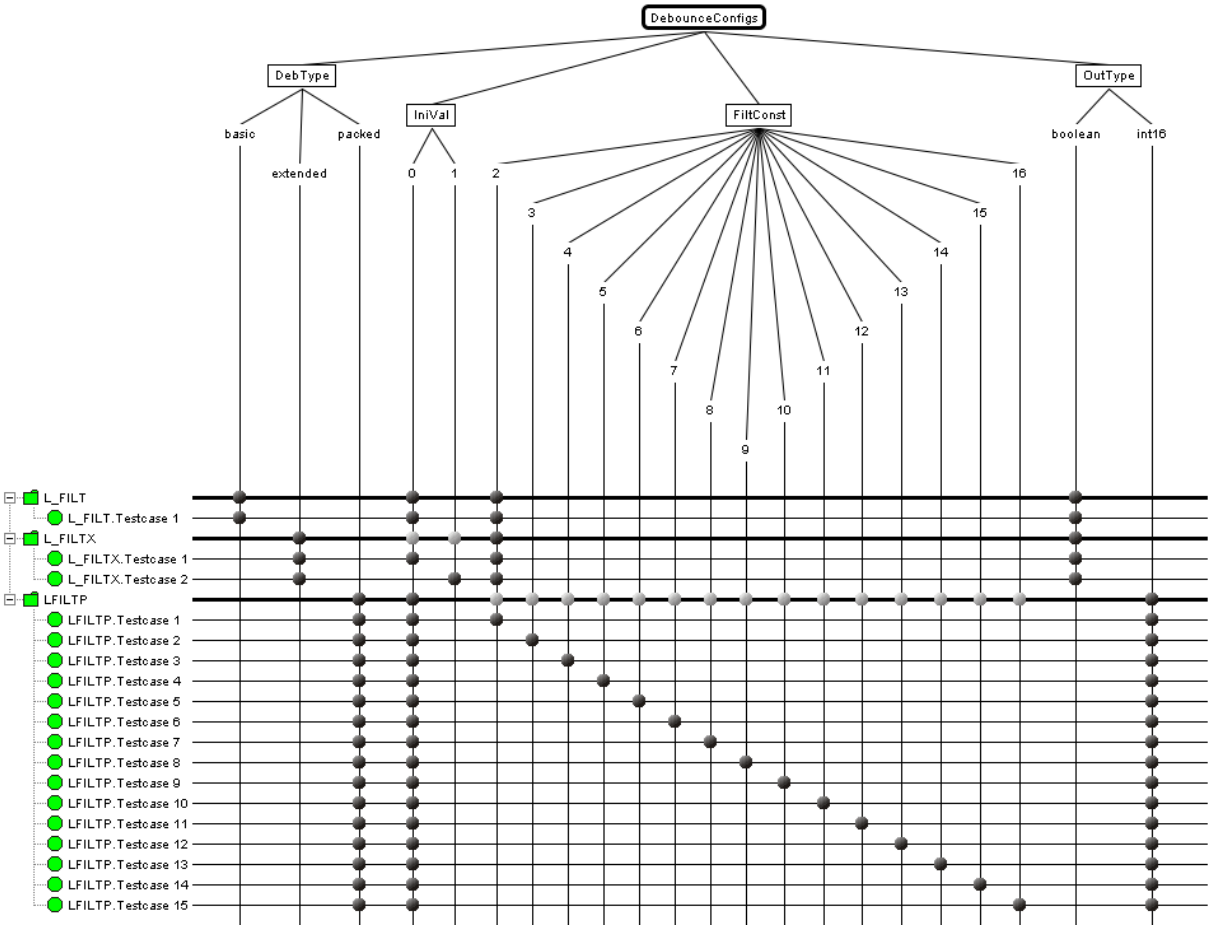


Figure 8.6: Debounce *component variants and configurations*

8.2.3 Vectors

Automatic test vector generation based on data partitioning described in section 5.3 cannot always bring the tested component into all states that need to be validated. In such cases, test

vector classification tree can be manually modified to produce desired test vectors.

For example, the *CHKINC* component represents the second part of the communication channel monitoring, alongside *GENINC* component mentioned in previous subsection. *CHKINC* checks validity of the received incrementing sequence generated on the remote node an thus enables detection of lost or corrupted messages. The input sequence is declared valid only if consecutive samples are incremented by one. Automatic test vector generation scheme would never (i.e. chances are very small) produce such sequence so vector classification tree was manually modified as shown in Fig. 8.7. Equivalent classes that correspond to classification representing *Input* input are organized in pairs that produce valid incrementing sequence. Between these pairs invalid sequence is generated that should result with error detection. *Trigg* input enables component operation and *Reset* input resets the error count output of the component.

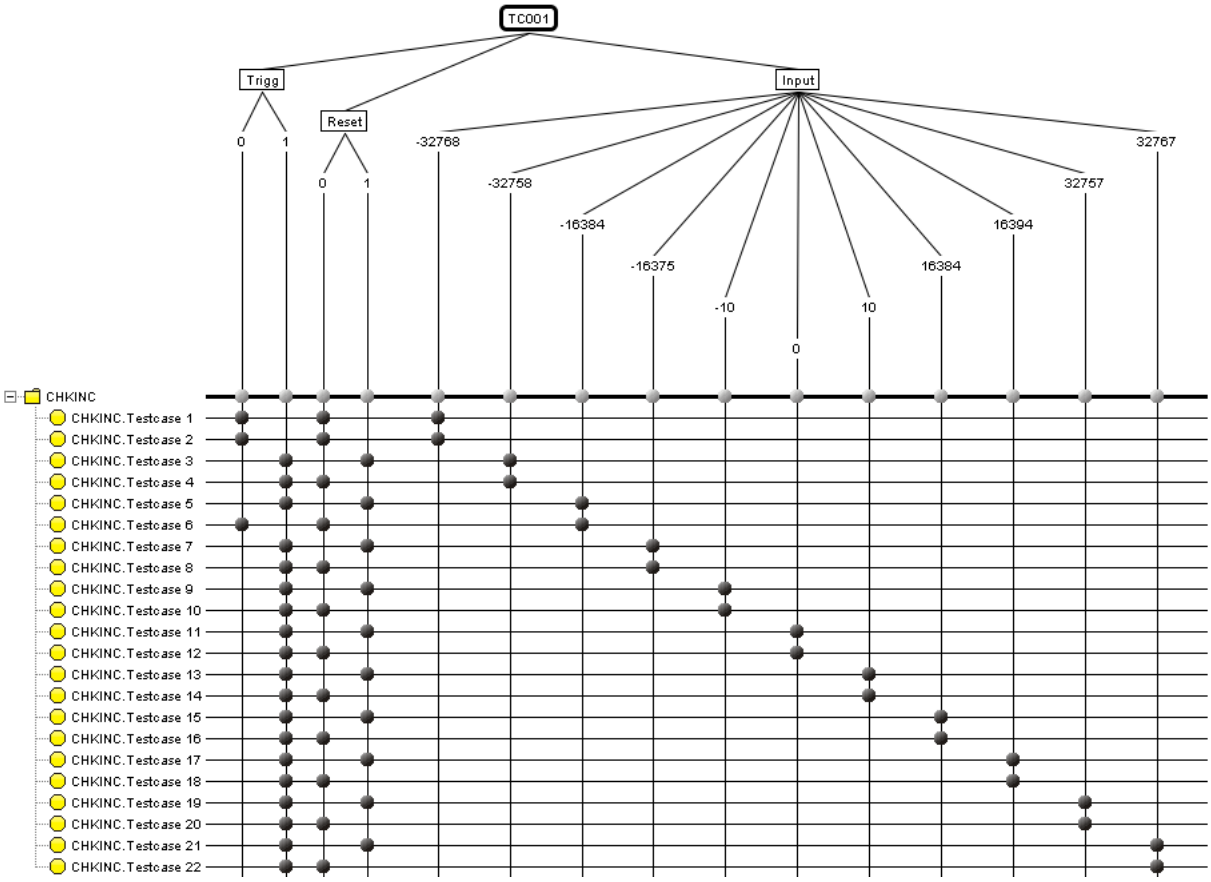


Figure 8.7: CHKINC *vector classification tree*

Waveforms of the test vectors generated based on classification tree in Fig. 8.7 are shown in Fig. 8.8. The scale of the *Input* input is relatively large to cover the whole 16-bit integer range so the valid by-one-incrementing sequences are not visible. One of these ramps has therefore been enlarged in Fig. 8.8.

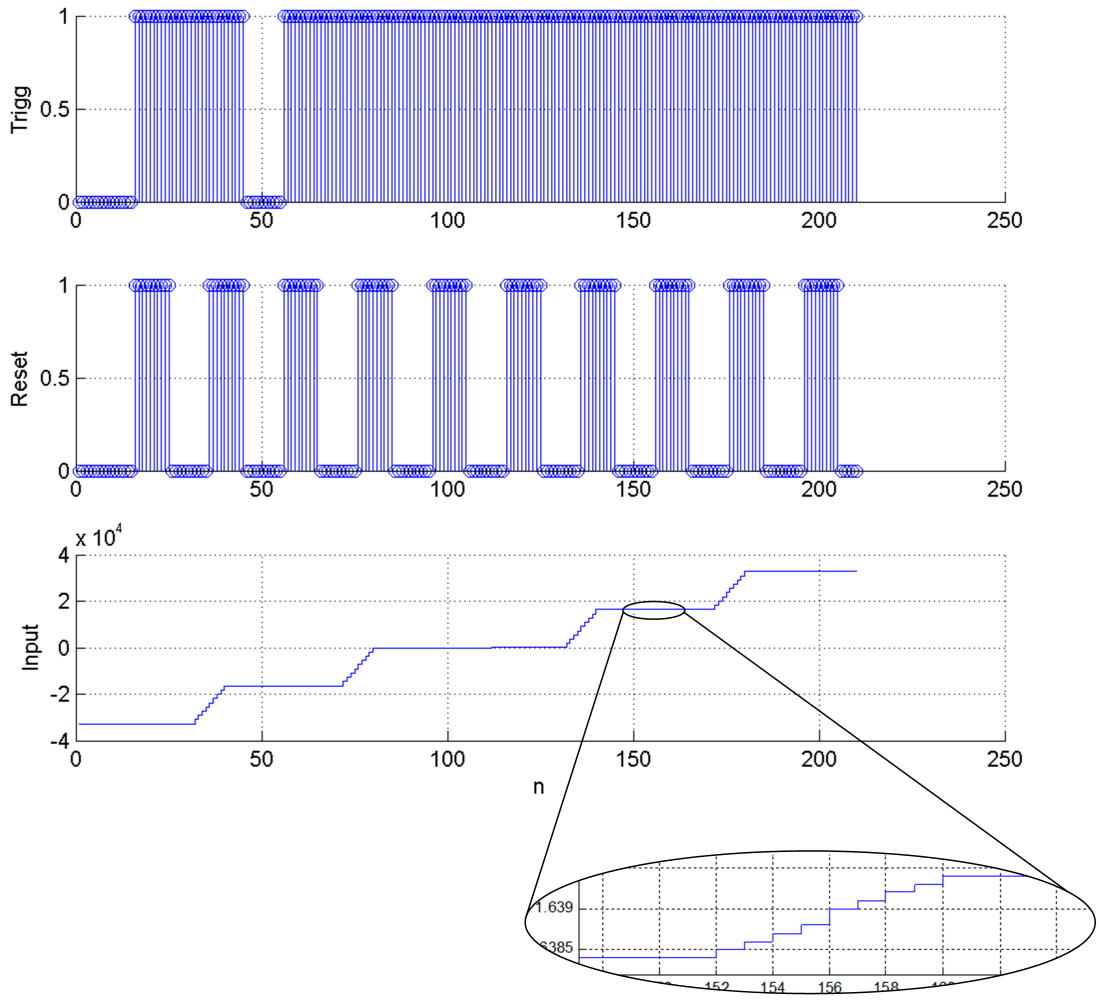


Figure 8.8: CHKINC *test vector waveforms*

Besides varying filter constants through different configurations of the component variant, as is done for *LFILTP* variant of the *Debounce* component in the previous subsection, the filter constant can be manually defined through vector classification tree. This has been done for the *L_FILT* variant of the *Debounce* component and the resulting vector classification tree and test vector waveforms are shown in figures 8.9 and 8.10, respectively.

On the other hand, automatic test vector generation works well for most of the components, e.g. figures 8.11 and 8.12 show fully automatically generated vector classification tree and waveforms for a configuration of the *CRCCALC* component's *CRCCALC4* variant.

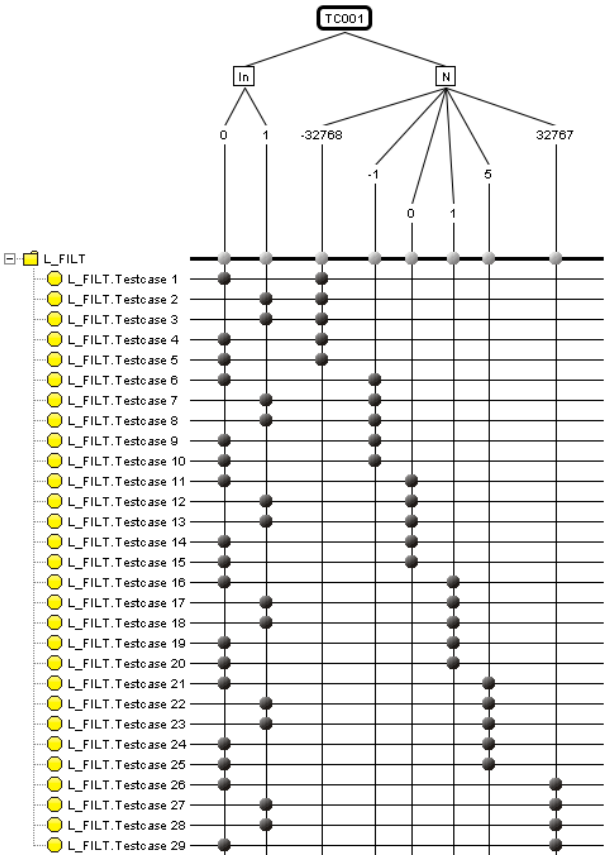


Figure 8.9: L_FILT *vector classification tree*

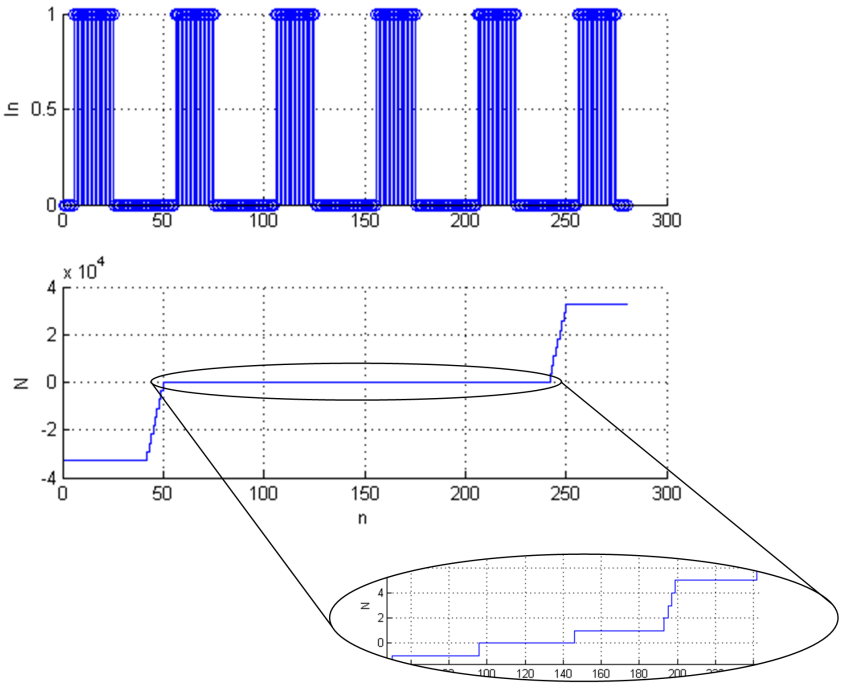


Figure 8.10: L_FILT *test vector waveforms*

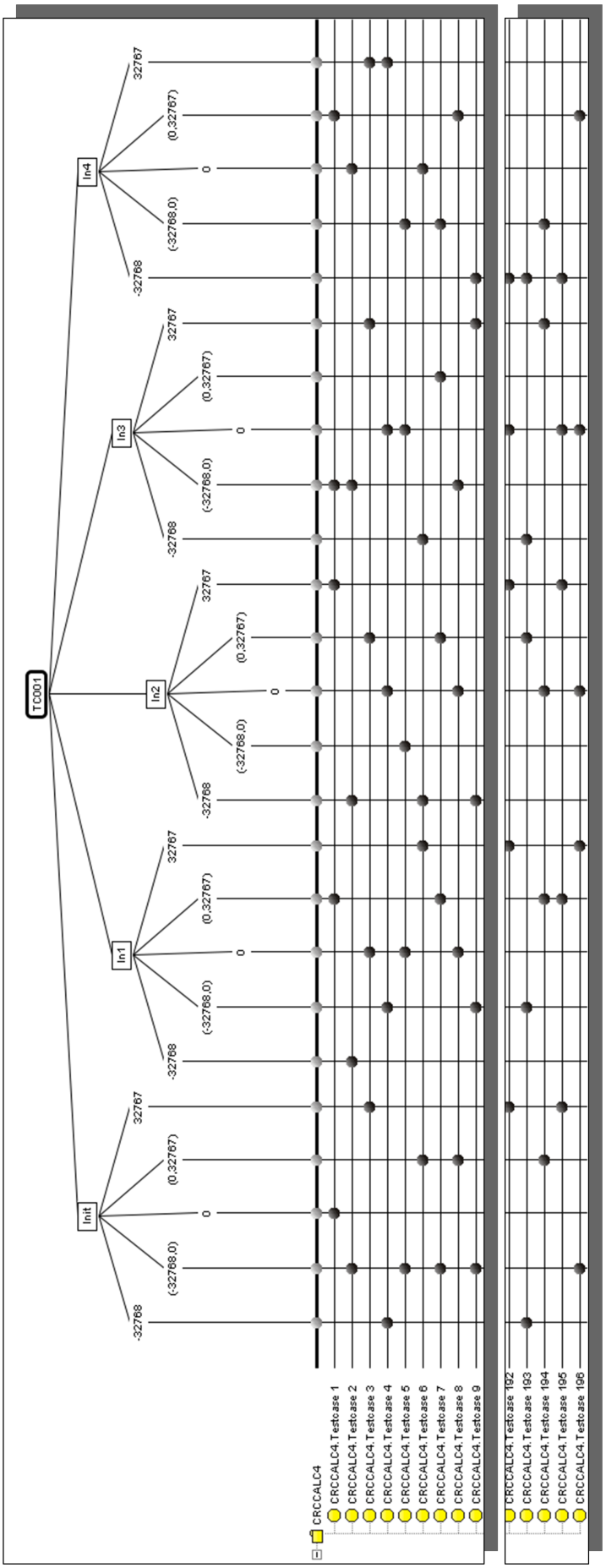


Figure 8.11: CRCCALC4 *vector classification tree*

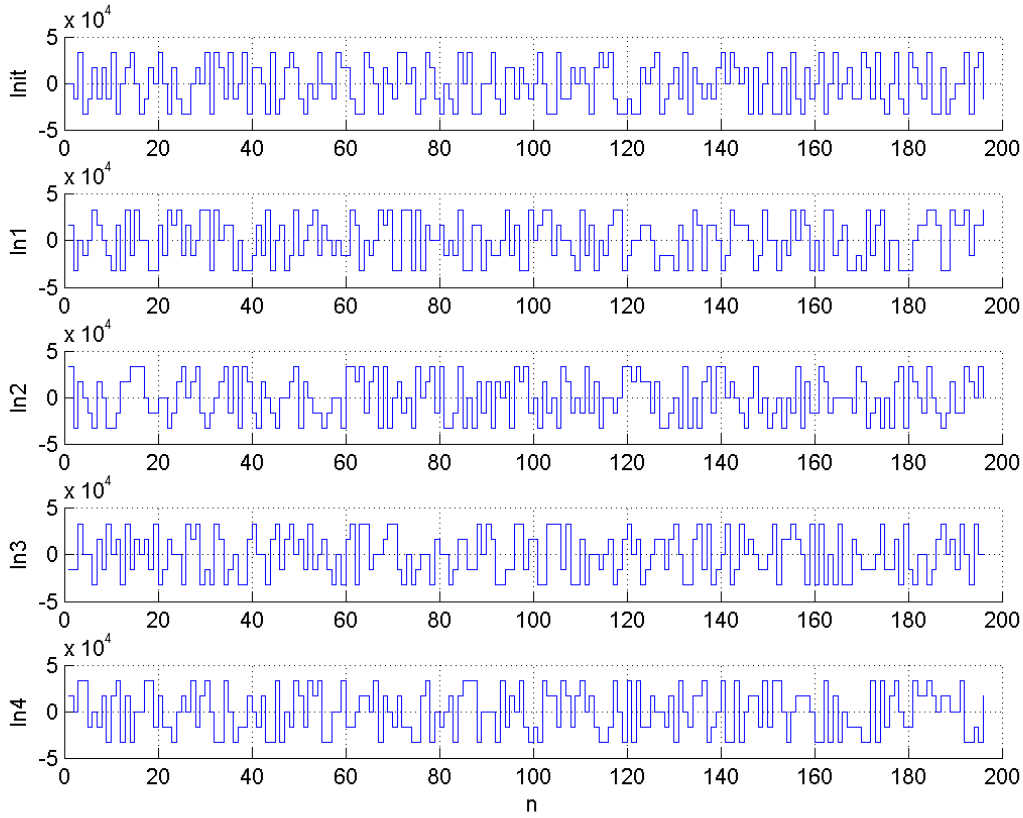


Figure 8.12: CRCCALC4 *test vector waveforms*

Automatic test vector generation procedure can result in substantial vector classification tree if equivalent classes of an input are expanded by specific values beyond the ones determined by data type. For example, *UnPackL* component unpacks a 16-bit arithmetical input into sixteen logical (boolean) signals. For this component's test vector, not only equivalent classes determined by the data type of the input are interesting, but also all the powers of two inside the data range. This kind of partitioning results in vector classification tree in Fig. 8.13.

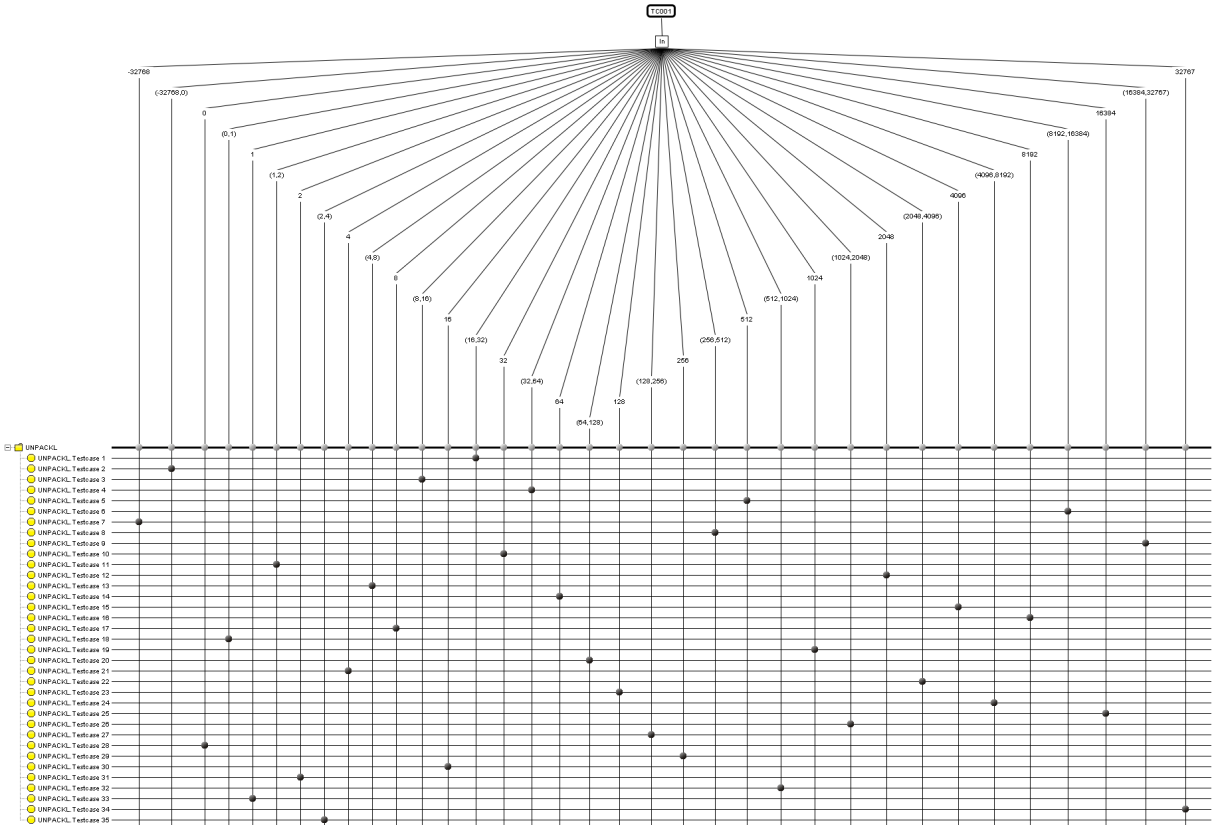


Figure 8.13: UNPACKL *test vector classification tree*

8.2.4 Models

The RTTP introduced in section 5.4, Fig. 5.7, can be significantly reduced for Safety Platform because of the absence of concurrent tasks. The reduced pattern, shown in Fig. 8.14, consists of test vector reproduction, tested configuration and its execution time measurement, and logging.

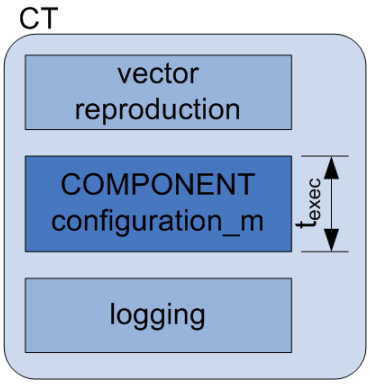


Figure 8.14: RTTP adjusted for Safety Platform

The root view of the test model pattern is shown in Fig. 8.15, the pattern is given on the left-hand side and the right-hand side provides example instance for the *SUM2* variant of the *Summation* component. The pattern is instantiated by populating *Conf_CUT* subsystem with

the tested configuration, as shown in Fig. 8.16, and by connecting this subsystem with the *Log* subsystem for signal logging. The *Log* subsystem pattern and instance are same as in section 5.4, Fig. 5.11.

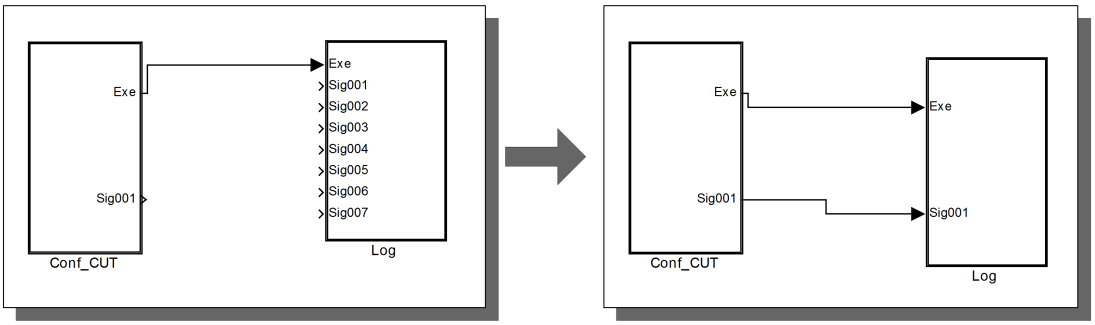


Figure 8.15: Safety Platform test model root; pattern on the left and instantiated on the right

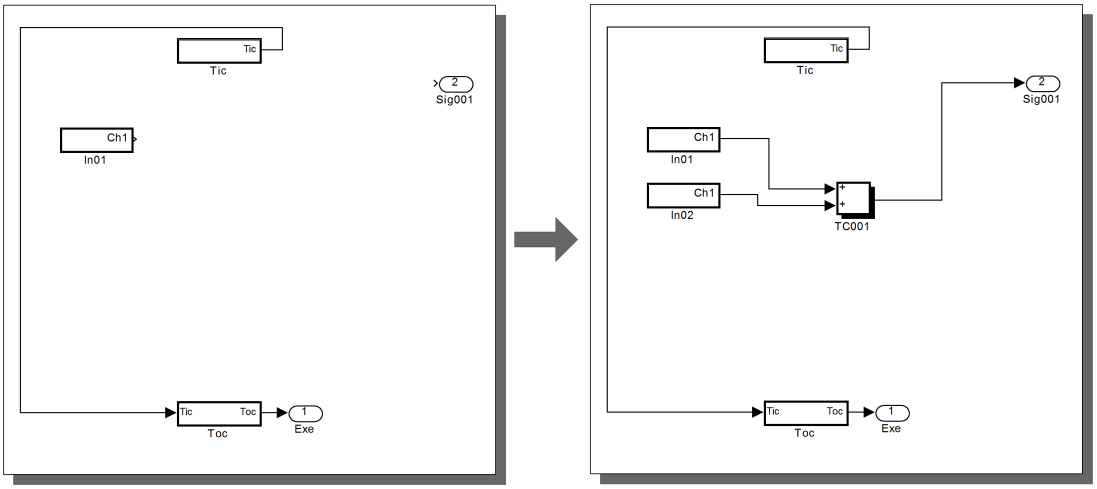


Figure 8.16: Safety Platform test model Conf_CUT *subsystem; pattern on the left and instantiated on the right*

8.2.5 Multiple Runs

Test models deviate slightly from the presented pattern if the configuration under test has a large number of outputs. The *RECORDER* component used for logging of simulation and code execution traces can record up to eight signals. This is not sufficient for testing of, for example, the *UNPACKL* variant of the *UnPackL* component that unpacks a 16-bit integer into sixteen logical signals, all of which must be recorded during test, Fig. 8.17 a). Such situations are detected during automatic test model generation and appropriate number of test model "runs" is generated. In each run, the tested component is supplied with the same test vectors and different outputs are recorded. When all the runs have been simulated, simulation traces from different runs are combined into one trace effectively producing a single simulation trace. In case of *UNPACKL* test model, three test model runs have been generated as depicted in Fig. 8.17 b), c), and d). Models shown in this figure have been manually arranged to avoid clutter produced by the Simulink autorouting during automatic model generation.

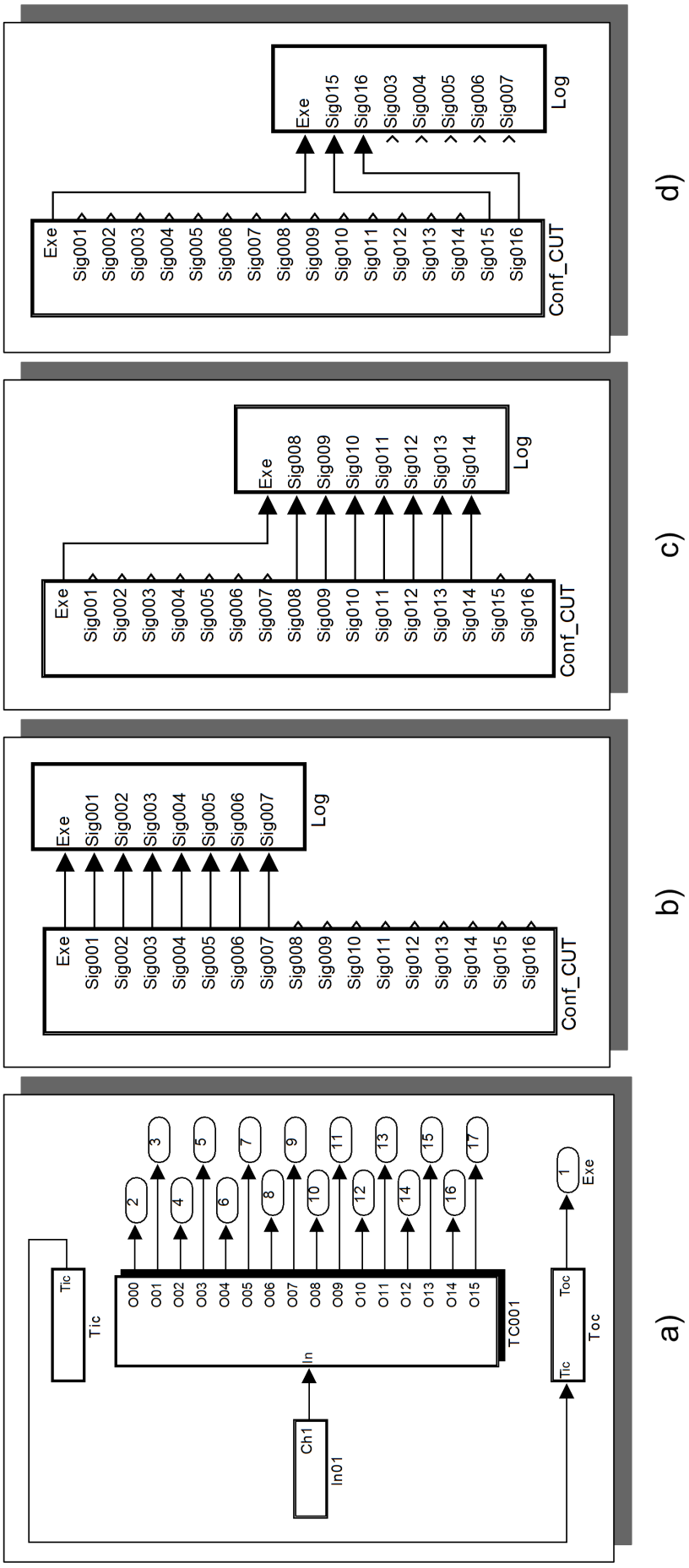


Figure 8.17: UNPACKL *test model: a)* Conf_CUT *subsystem, b) 1st test run, c) 2nd test run, d) 3rd test run*

Automated executable code generation and execution for models with multiple runs follows a similar procedure as during simulation: code is generated, loaded and executed for each run and execution traces are retrieved. Code execution traces from multiple runs are composed into a single trace that can be compared to the composed simulation trace on a one-to-one basis. A comparison of composed simulation and code execution traces for the *UNPACKL* variant of the *UnPackL* component is shown in Fig. 8.18.

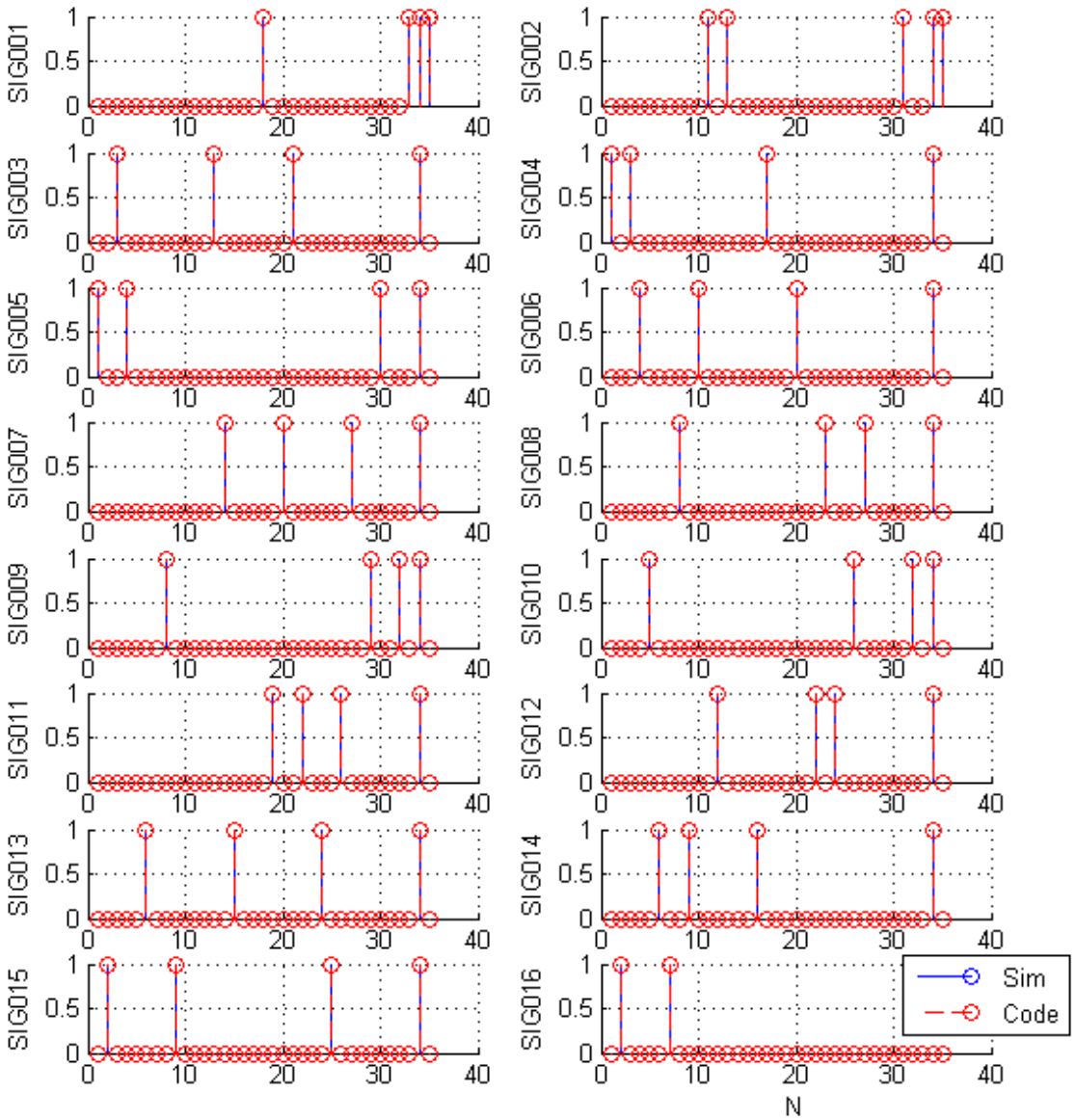


Figure 8.18: Comparison of composed simulation and code execution traces

8.2.6 Testing Results

The described testing procedure has been conducted for all software components of the Safety Platform that have inputs controllable and outputs observable from within the application program. Special and hardware dependent components, e.g. drivers for digital inputs and outputs, have been either manually tested or the testing has been performed indirectly through test cases of the respective hardware unit.

Discovered defects together with their analysis and solutions are listed in Tab. 8.2. Two errors in GRAP atomic component libraries were found and corrected: one error in graphical library and one in the macro library. The three remaining defects impact atomic components' inputs/outputs which are not used in application programs. They were amended by removing the unused ports from the corresponding GRAPlab components and assigning "NC" (not connected) label to the macro calls during code generation.

Table 8.2: Defects found during Safety Platform component testing

Component	**Variant**	**Defect**	**Analysis**	**Solution**
CHKINC	CHKINC	Incorrect LDIFF for constant INPUT (255 instead of -1).	LDIFF is auxilary output, not used in applications.	Simulink component modified.
Copy	COPY	Defines logical output instead of arithmetical.	Error in graphical library.	Corrected.
CRCCALC	all	ENABLE input not functional.	Input not used.	Simulink component modified.
GENINC	GENINC	OV output is never active.	A bug in element, but OV output is not used.	Simulink component modified.
Summation	DIFF	Negative overflow to -32767 instead to -32768.	A bug in element implementation.	Corrected.

After correcting the listed defects, all tested components have passed the functional test. Table 8.3 summarizes real-time properties of the Safety Platform's software components. The average execution time values listed in the table should be considered with a dose of caution. These values depend on input vectors and would in most cases be different for different input sequences.

Table 8.3: Real-time properties of the Safety Platform software components

Component	Variant	Config.	BCET	ACET	WCET
Bistable	L_RS	TC001	2	2.25	3
	L_SR	TC001	2	2.25	3
CHKINC	CHKINC	TC001	2	6.67	10
COMP	COMP	TC001	4	4.52	5
CRCCALC	CRCCALC4	TC001	20	20.55	21
		TC002	20	20.56	21
	CRCCALC8	TC001	38	38.54	39
		TC002	38	38.51	39
Copy	COPY	TC001	2	2.4	3
	L_COPY	TC001	1	1.5	2
Debounce	LFILTP	TC001	10	10.67	11
		TC002	14	14.45	15
		TC003	17	17.34	18
		TC004	20	20.65	21
		TC005	24	24.40	25
		TC006	27	27.33	28
		TC007	30	30.65	31
		TC008	34	34.40	35
		TC009	37	37.39	38
		TC010	40	40.62	41
		TC011	44	44.44	45
		TC012	47	47.33	48
		TC013	50	50.69	51
		TC014	54	54.43	55
		TC015	57	57.37	58
	L_FILT	TC001	2	4.06	10
	L_FILTX	TC001	2	3.98	11
		TC002	2	6.01	10
DelayEdge	L_TOFF	TC001	1	3.19	7
	L_TON	TC001	1	3.09	6
EQTEST	EQTEST	TC001	2	2.65	4
FILTLPS	FILTLPS	TC001	12	13.17	14

Table 8.3: Continued from previous page...

Component	Variant	Config.	BCET	ACET	WCET
GENINC	GENINC	TC001	1	1.58	3
		TC002	1	1.64	3
		TC003	1	1.57	3
		TC004	1	1.7	3
		TC005	1	1.58	3
		TC006	1	1.66	3
Gate	GATE	TC001	0	1.6	3
HiLoJoin	HILOJOIN	TC001	1	1.36	2
HiLoSplit	HILOSPLT	TC001	2	2.4	3
L_OSC	L_OSC	TC001	4	4.86	7
	L_OSCE	TC001	2	3.91	8
	L_OSCR	TC001	4	6.47	9
LogicalOperator	L_AND2	TC001	1	1.75	2
	L_AND4	TC001	2	2.68	4
	L_AND8	TC001	6	6	6
	L_EXOR	TC001	1	1.75	2
	L_NOT	TC001	1	1.5	2
	L_OR2	TC001	1	1.75	2
	L_OR4	TC001	2	2.68	4
	L_OR8	TC001	6	6	6
PackL	PACKL	TC001	13	14.58	16
	PACKL8	TC001	6	7.01	8
Peaks	PEAKS	TC001	3	5.61	9
Summation	DIFF	TC001	3	3.38	5
	SUM2	TC001	3	3.54	4
Switch	L_SWITCH	TC001	2	2.12	3
	SWITCH	TC001	2	2.82	3
Timer	TIMER	TC001	1	2.73	8
Trigg	L_TRIGG	TC001	2	2.30	3
UnPackL	UNPACKL	TC001	2	2.30	3
	UNPACKL8	TC001	2	2.30	3

Table 8.3: Continued from previous page...

Component	Variant	Config.	BCET	ACET	WCET
UnitDelay	DELAY	TC001	4	4.22	5
	L_DELAY1	TC001	2	2.9	3

The testing process was automatically documented using LaTeX report patters, section 5.6. This resulted in a 1183 pages of test report that was part of official documentation set submitted during certification of the Safety Platform.

Chapter 9

Case Study: Photovoltaic Maximum Power Point Tracking Algorithm Testing

Control algorithm model-based real-time testing is presented on case study of MPPT algorithm. This algorithm is used to extract maximum power from a power source with typically non-linear current over voltage characteristic. Common example for MPPT application is a photovoltaic inverter. Solar panels' nonlinear voltage-current characteristic has a distinct maximum power point (MPP) dependent on environmental factors, such as temperature and irradiation. To harvest maximum power from solar panels, they must continuously operate at their MPP, so controllers of all photovoltaic inverters employ some method for maximum power point tracking, [132]. In this case, a MPPT algorithm is used to control interface between the photovoltaic field and the load, i.e. to create an "adaptable load", by means of power inverter.

This chapter presents implemented MPPT algorithm, model of photovoltaic panels used in its' development, and real-time testing of the algorithm according to MoBREST-IT method. Embedded target used in this case study is based on TI's TMS320F28335 floating-point DSC, as introduced in section 7.1.

9.1 Photovoltaic panels and maximum power point tracking model

The root view of the Simulink model with meteorological conditions subsystem *Meteo*, photovoltaic panels' model subsystem *Panels*, and MPPT algorithm implementation subsystem *MPPT* is shown in Fig. 9.1. Period of the cyclic task at which the MPPT algorithm executes is by several orders of magnitude longer than inverter's step response to operating point change, [133]. That is why inverter response from the viewpoint of MPPT algorithm is approximated as instant and is not included in the model.

Photovoltaic fields are constructed out of a number of photovoltaic panels connected in series and parallels. Photovoltaic panels are, again, constructed out of a number of photovoltaic cells. Therefore, to validate an MPPT algorithm, a sufficiently accurate model of photovoltaic cell is needed. As shown in [134], photovoltaic cell can be modeled with equation 9-1, where

I and V are cell output current and voltage, I_0 is saturation current, R_S and R_P are cell shunt and series resistance, I_S is the current generated by sunlight and A, T and k are diode ideality factor, cell temperature and Boltzmann constant, respectively. Typical characteristics of a photovoltaic cell modelled by equation 9-1 are given in Fig. 9.2. In model on Fig. 9.1, the panels are modeled by an M-function. Photovoltaic field model was sized by increasing cell outputs until satisfactory photovoltaic field power at MPP was achieved.

$$I = I_S - I_0(e^{\frac{q(V+I\dot{R}_S)}{kTA}} - 1) - \frac{V + I\dot{R}_S}{R_P} \tag{9-1}$$

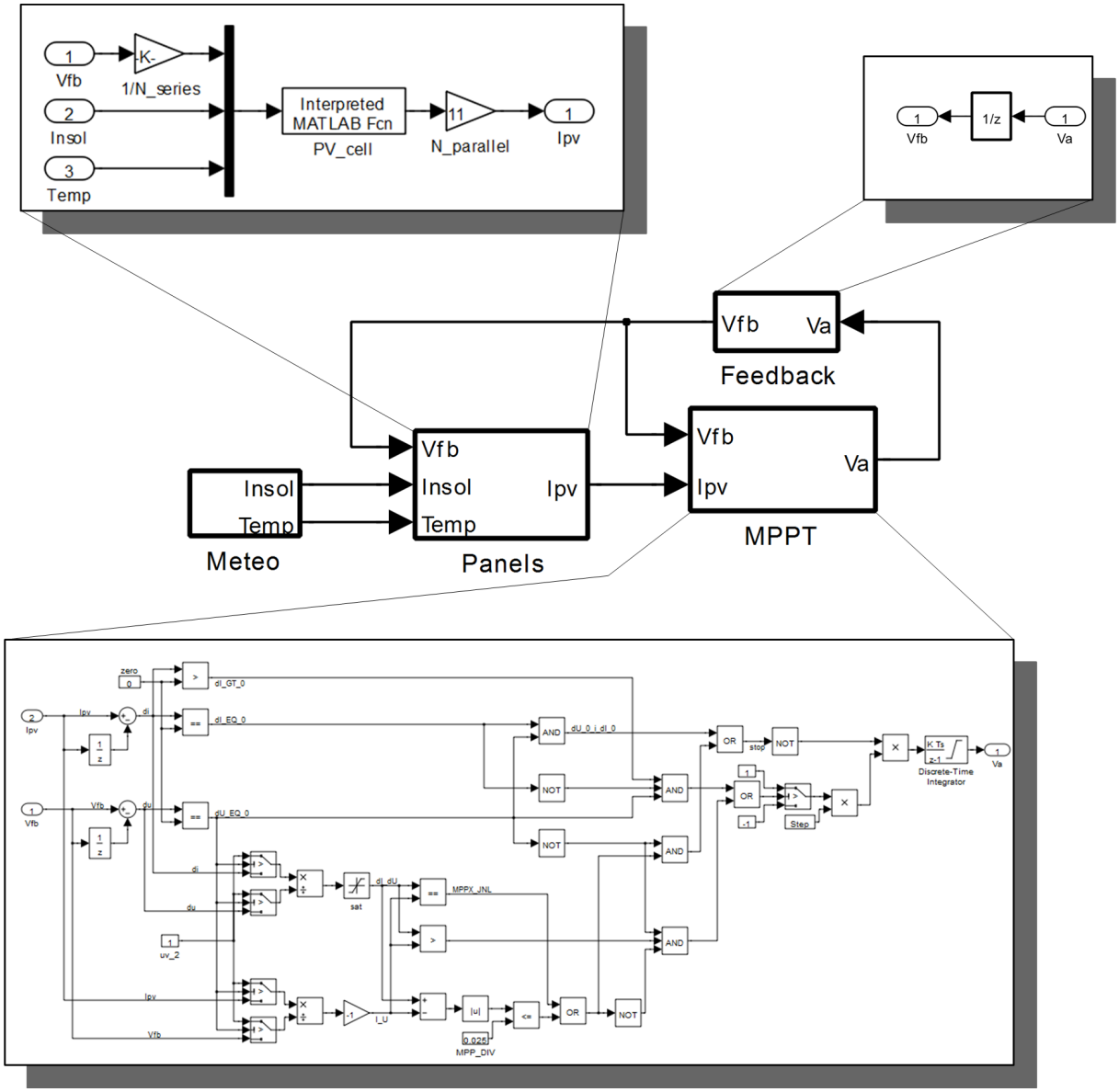


Figure 9.1: Model of the photovoltaic panels and MPPT algorithm

Many different MPPT algorithms have been developed and implemented with various degrees of success, [132, 135]. In general, MPPT algorithms can be divided into “hill climbing” types, advanced MPPTs based e.g. on neural networks or on fuzzy logic, and “approximate”

types, where part of the photovoltaic cell characteristics are known or estimated. In practice, mostly “hill climbing” methods are used, as advanced algorithms can be hard to implement due to computing requirements, while other types do not operate with adequate precision for modern systems, [136]. Tested algorithm is a “hill climbing” variation known as *Incremental Conductance* algorithm, [137]. The algorithm is based on assessment of the slope of power-voltage curve of the photovoltaic panel.

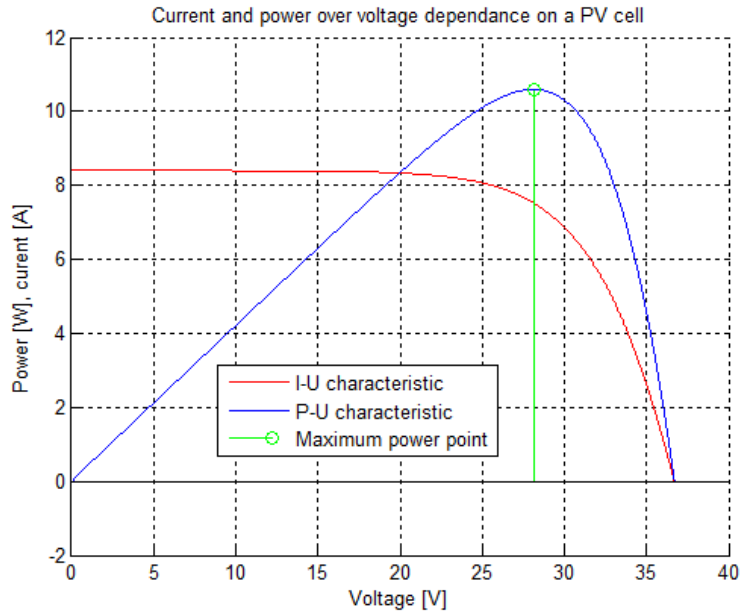


Figure 9.2: Photovoltaic cell characteristics

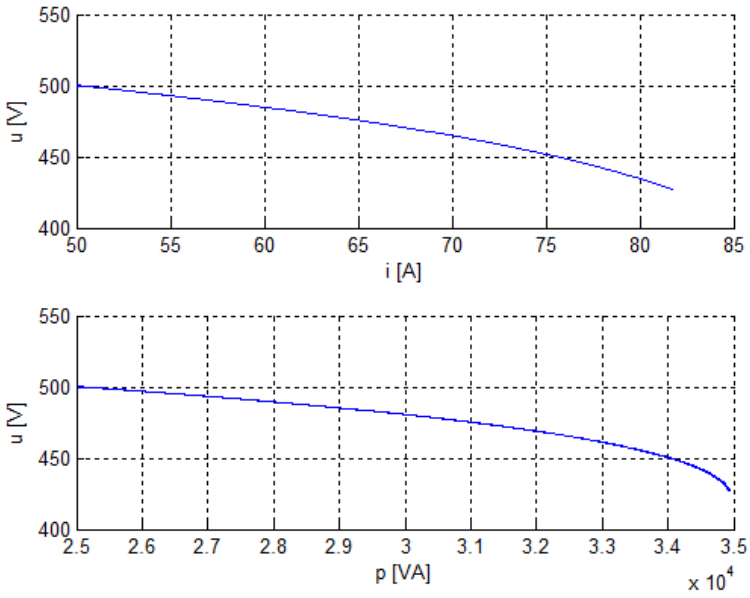


Figure 9.3: Response of the model in Fig. 9.1 to insolation step

Environmental condition changes are relatively slow compared to photovoltaic inverter dy-

namics so testing of the real-time properties of the MPPT algorithm is not computationally demanding. However, this application was chosen for demonstration of integration testing methods because photovoltaic panels model as well as MPPT algorithm model are both simple enough to provide a "school room" example while being at the same time complex enough not to be trivial. More importantly, parallel to this research, the author was part of a team developing a photovoltaic inverter, Fig. 7.3 in section 7.1, so presented results represent real-world application of the proposed method.

Response of the model in Fig. 9.1 to insolation step change is shown in Fig. 9.3. This response is used as on oracle in next stages of testing.

9.2 Open-loop test

Test model for the functional stage of open-loop testing is created by replacing *MPPT* subsystem in model on Fig. 9.1 with subsystem shown in Fig. 9.4. The new subsystem pads tested algorithm with interface blocks, see section 6.1, and replaces native Simulink blocks with functionally equivalent blocks from the GRAPlab blockset, in order to enable automatic code generation as described in chapter 4.

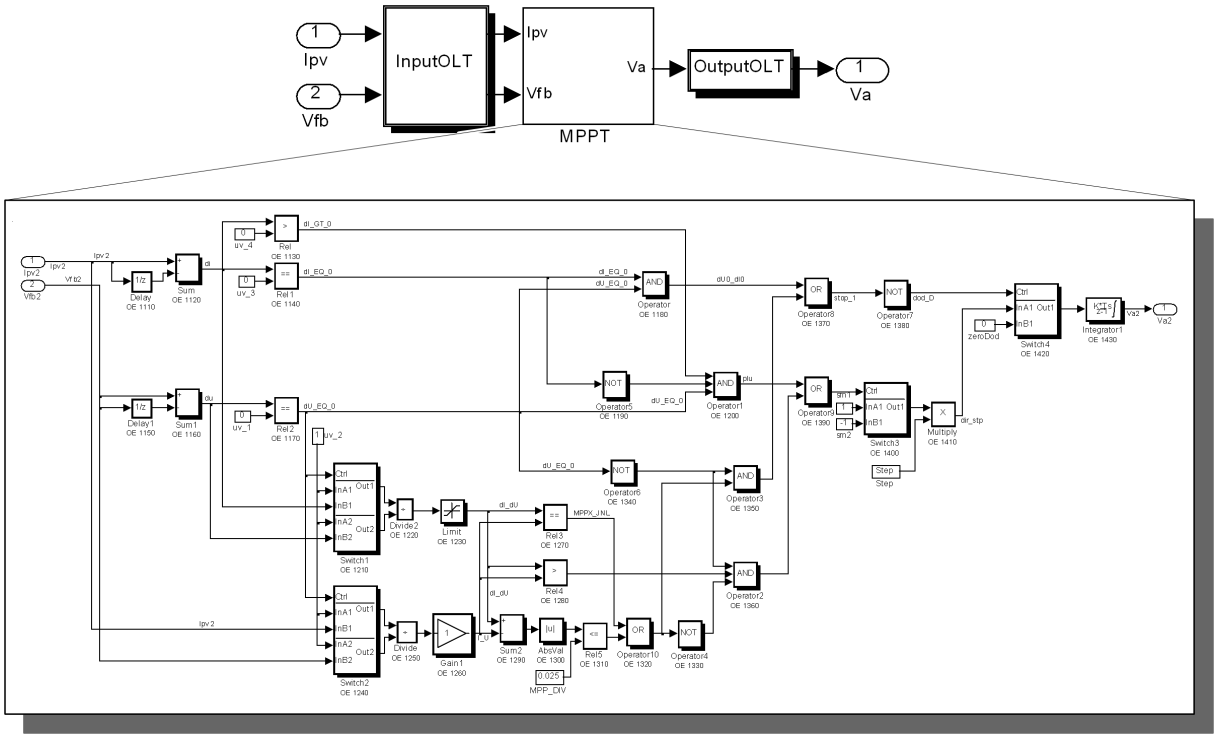


Figure 9.4: MPPT subsystem for first stage of open-loop testing

Fig. 9.5 shows MPPT algorithm integrated into the RTTP for real-time stage of open-loop testing. The same MPPT model prepared for code generation that was used in first stage is also used in real-time testing, but here the *MPPT* subsystem containing it is present on two places: in *SUT_CT1* subsystem executed on CT1 and in *SUT_CT2* subsystem executed on CT2.

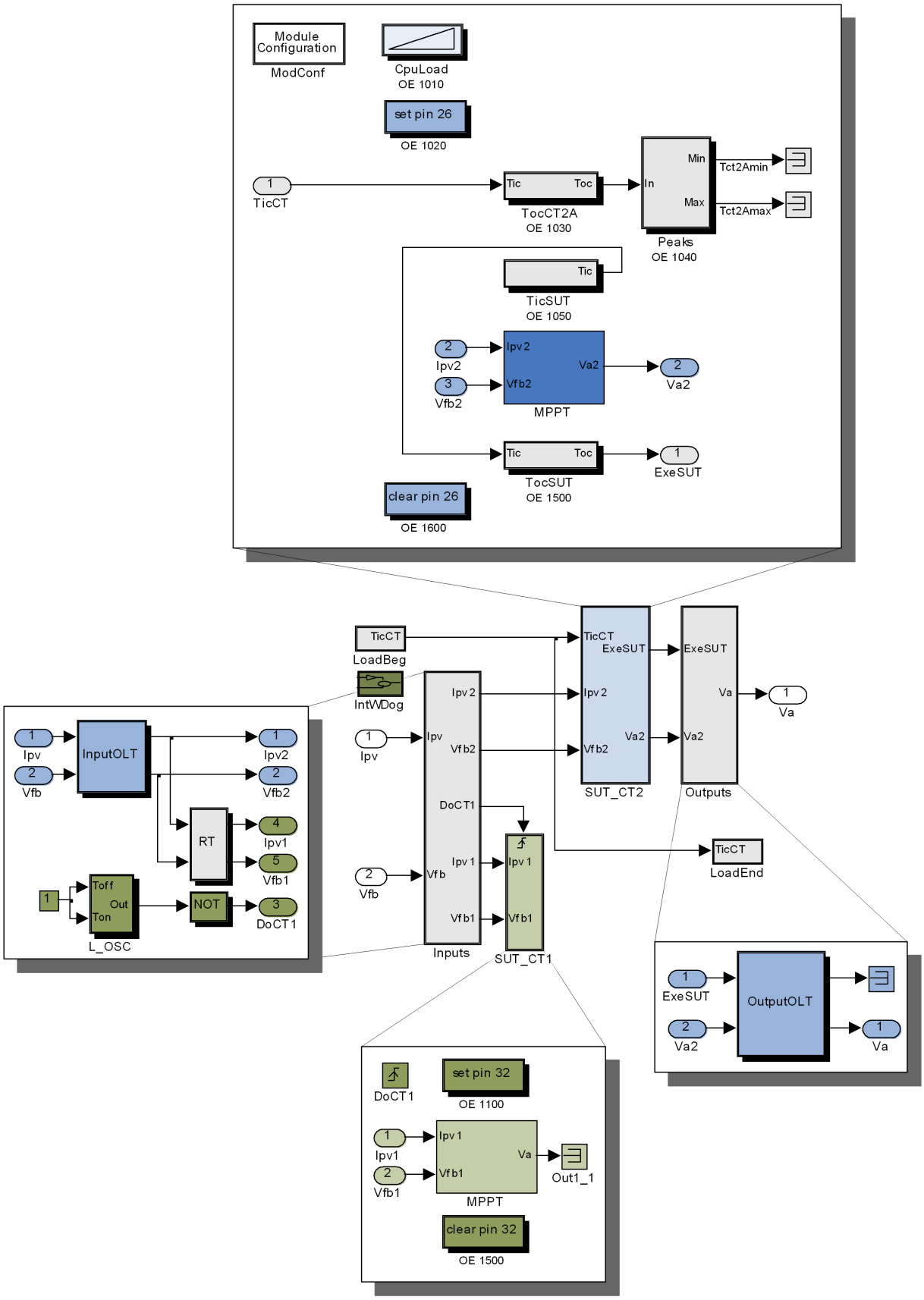


Figure 9.5: MPPT algorithm model integrated into RTTP

Functional behaviour of the MPPT algorithm implementation is the same in both stages of

open-loop testing and execution traces perfectly match simulation traces, Fig. 9.6.

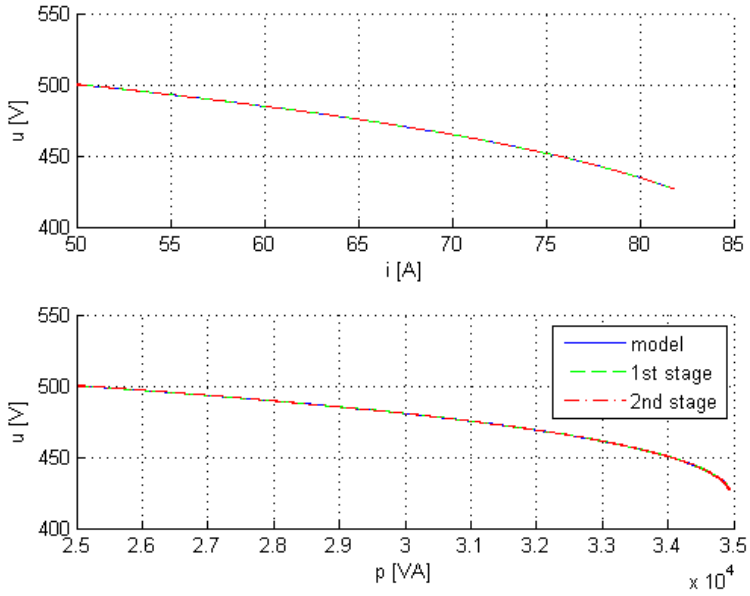


Figure 9.6: Open-loop MPPT testing functional results

RTTP used in control algorithm real-time integration testing is derived from RTTP for component testing. In order to validate these changes, task execution order of test application for MPPT algorithm open-loop real-time testing was recorded by oscilloscope, Fig. 9.7-9.10. Execution of pieces of code is marked by setting a DSC's GPIO pin at start of the code execution and by clearing the same pin at code execution end. This way, execution timing relationships between parts of target application were determined. In figures 9.7-9.10, signals corresponding to execution of individual parts of target code are labeled with:

- *CT1* – higher priority CT1,
- *SUT_CT1* – instance of the tested algorithm on the CT1,
- *SUT* – instance of the tested algorithm on the CT2, i.e. the one that is actually being tested,
- *CT2* – lower priority CT2.

Fig. 9.7 shows that SUT instance placed on CT1 really executes on each second CT1 execution occasion and synchronously with CT2. This enables the SUT instance on CT1 to interrupt CT2 and ensures that SUT instances on CT1 and on CT2 are in the same state during the interrupt.

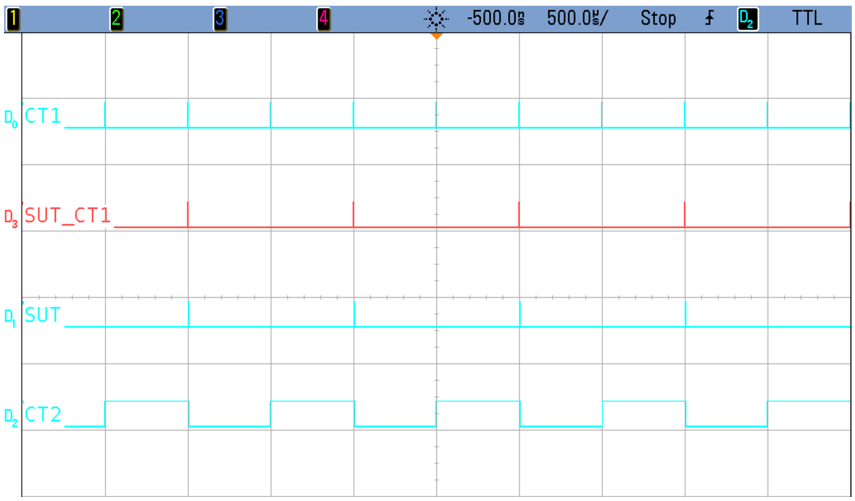


Figure 9.7: SUT instance on CT1 is executed synchronously with CT2 during real-time open-loop testing of the MPPT algorithm

Waveforms on figures 9.8-9.10 represent enlarged parts of the waveforms similar to the one in Fig. 9.7. Fig. 9.8 shows a situation in which variable CPU load generated on the CT2 is below the point at which interrupting of the SUT by the CT1 begins. In Fig. 9.9 variable CPU load positions SUT code so that it is interrupted by the CT1. The *SUT* signal is not cleared when interrupt occurs so it is elongated by the duration of the *CT1* signal pulse. When variable CPU load rises above certain value, SUT code is executed after the point in which it can be interrupted, Fig. 9.10. These tree figures show that variable CPU load ensures interrupting of the SUT code by the CT1 at each possible point.

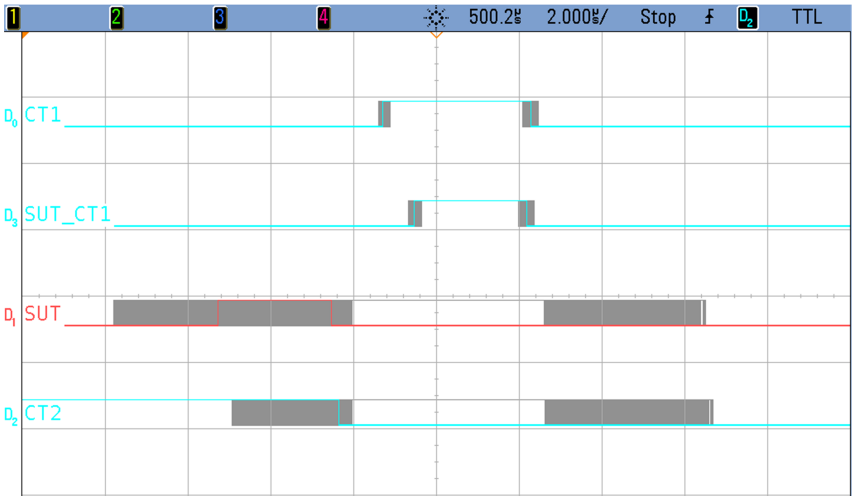


Figure 9.8: Task execution during real-time open-loop testing of the MPPT algorithm before SUT starts being interrupted

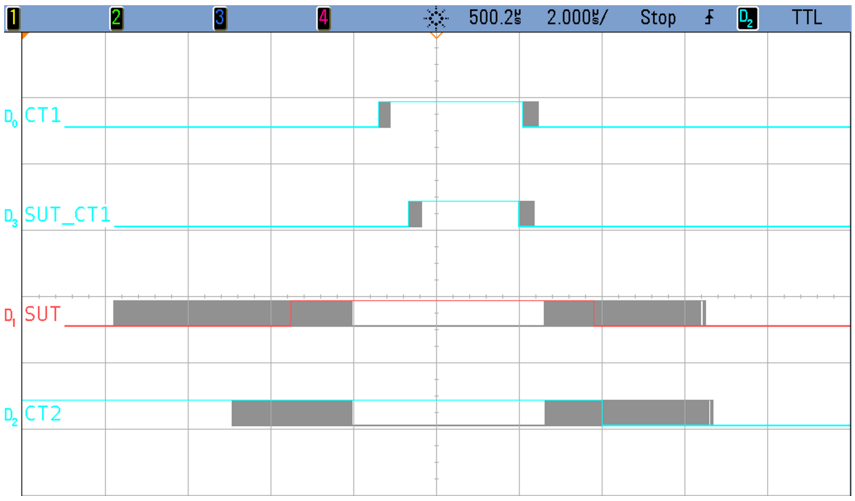


Figure 9.9: Task execution during real-time open-loop testing of the MPPT algorithm when SUT is interrupted

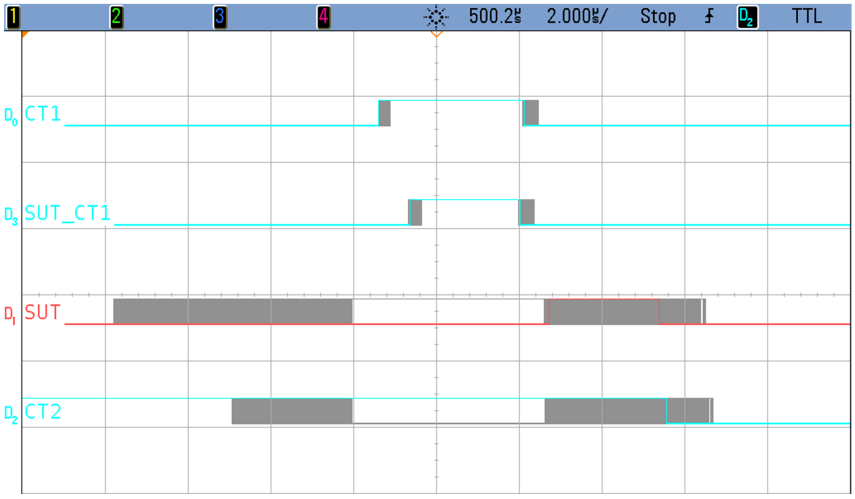


Figure 9.10: Task execution during real-time open-loop testing of the MPPT algorithm after SUT has passed the "interrupt zone"

Execution time of the MPPT algorithm has been measured during open-loop real-time testing and is plotted against algorithm's output in Fig. 9.11. Execution time jumps to 408 CPU cycles at start of execution, oscillates between 395 and 404 cycles during transient, rises to 406 cycles at end of transient and finally stabilizes on 395 cycles in steady state. This kind of measurement can be a starting point for in-depth analysis of SUT real-time behaviour.

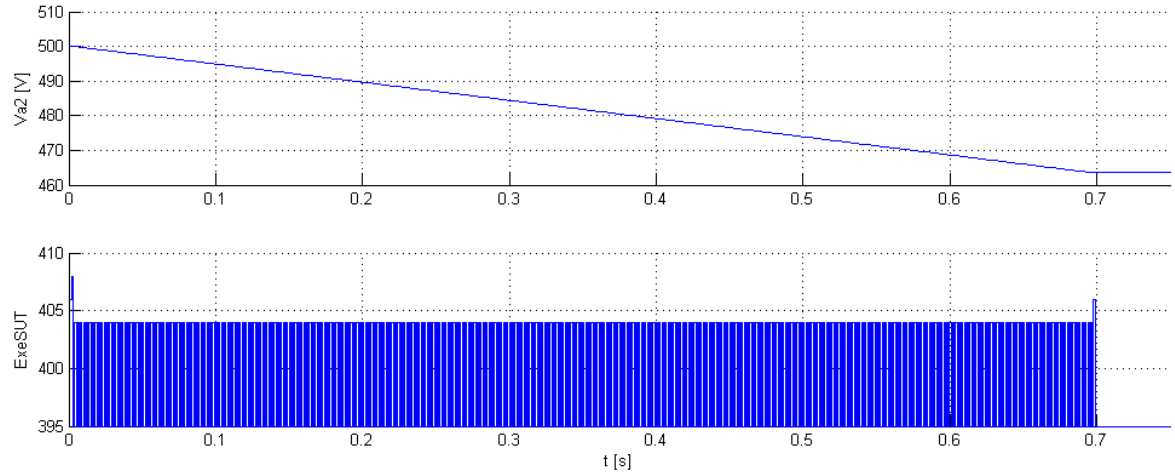


Figure 9.11: SUT output versus its' execution time

Maximal interrupt delay introduced by the MPPT algorithm implementation was determined to be 2 processor cycles.

9.3 Closed-loop test with simulated environment

MPPT algorithm model is prepared for non-real time closed-loop testing by adding *InputPiL* and *OutputPiL* blocks into signal path in same way as in Fig. 6.4, section 6.2. In model on Fig. 9.12, two additional blocks are added. *RunCT1* block sends a command to the target to perform one execution of the code generated from the MPPT subsystem in each simulation step. This way the algorithm executed on the target is fed with input and algorithm output is passed back into the model in a step-by-step manner. The *EndAP* block stops target application when simulation time elapses.

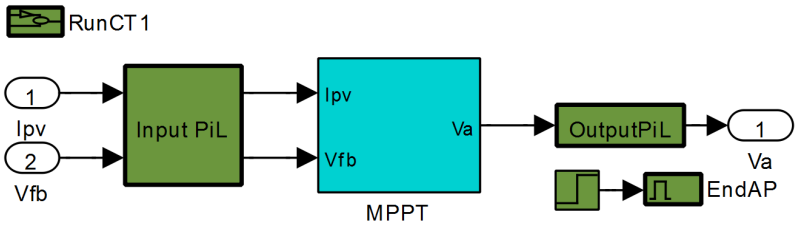


Figure 9.12: MPPT subsystem for non-real-time closed-loop testing

Fig. 9.13 shows B2B comparison of U-I and U-P characteristics for cases when feedback loop is closed with model of the MPPT algorithm and when feedback is closed with code executed on the target. Here a mismatch can be seen caused by a single step delay introduced by the RTW serial communication. Nevertheless, code response follows simulation traces and reaches the same steady state.

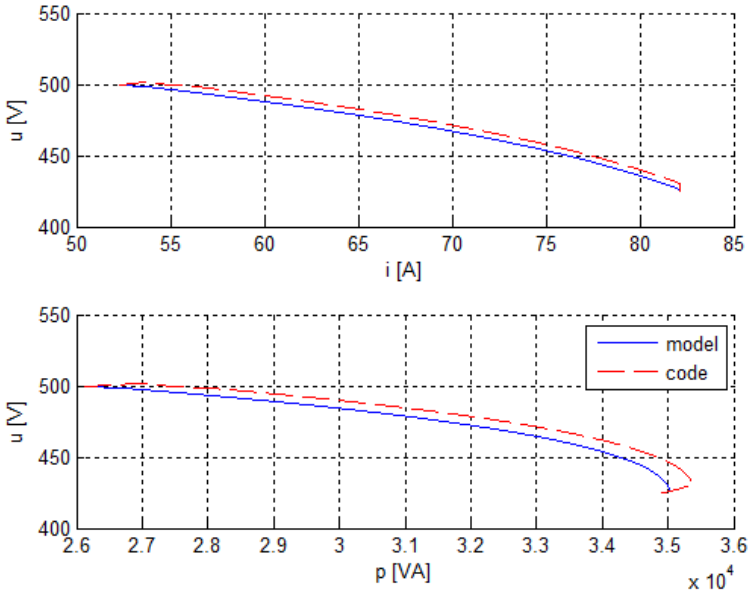


Figure 9.13: Non-real-time closed-loop MPPT testing functional results

A workaround the one step delay problem of the RTW serial communication could probably be devised. However, this was not pursued because the focus of the thesis is on real-time testing. This step is provided here as a proof of concept which shows that presented methods and tools are suitable for non-real time testing with interaction between simulation environment and real-time embedded target.

9.4 Real-time closed-loop test

In order to validate real-time properties of the MPPT algorithm implementation, the model of its' environment, i.e. the photovoltaic panels model, must be executed in real-time. One of goals of this case study was to show that this can be realized without expensive real-time simulators. This was achieved by preparing the panels model for ACG and by executing generated code on the embedded target, alongside the MPPT algorithm implementation.

ACG procedure doesn't support code generation from M-functions, so functional panels model in Fig. 9.1 must be reimplemented with Simulink blocks. In Fig. 9.14, implementational panels model is broken down into two subsystems. *Panels1* subsystem implements part of the M-function that is executed once in each simulation step, while *Panels2* subsystem implements iterative part of the M-function.

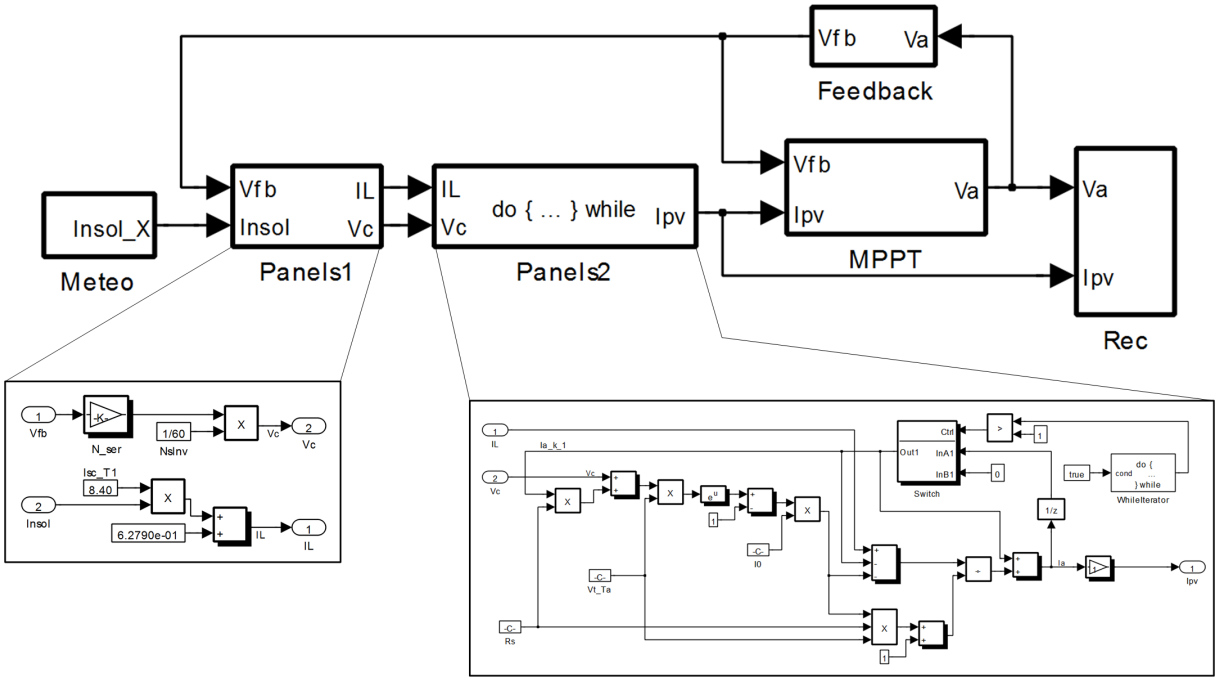


Figure 9.14: Model for real-time closed-loop testing

Before using photovoltaic panels model in MPPT algorithm testing, it has been functionally validated in an open-loop test. Simulation traces, i.e. test vectors and oracle, have been taken from open-loop testing of the MPPT algorithm described section 9.2. This means that a loop including functional model of photovoltaic panels and the MPPT model has been closed inside simulation environment, simulation was performed, and inputs to photovoltaic panels model have been recorded as well as its' output. Next, implementational model was simulated with obtained test vector at its' inputs. Final step in model validation was ACG and execution of the generated code on the embedded target, again with the same test vectors. Outputs of the functional model, implementational model and executable code of the photovoltaic panels model are compared in Fig. 9.15.

Outputs of the implementational model and of the code match perfectly, but differ from output of the functional model. The difference arises because of GRAPlab's approximate implementation of exponent function (block marked with e^u in Fig. 9.14) with Maclaurin series according to equation 9-2. This is exactly a kind of problem that can arise during design of closed-loop real-time algorithm test. Refinement of environment functional model into a form suitable for code generation can require simplifications and approximations that influence its' functional behaviour. At this point in test design, the test and/or application engineer must decide whether such deviations are acceptable. If not, closed-loop real-time testing is unfortunately not applicable to that specific application.

$$e^u = \sum_{n=0}^{\inf} \frac{u^n}{n!} \approx \sum_{n=0}^{16} \frac{u^n}{n!} \tag{9-2}$$

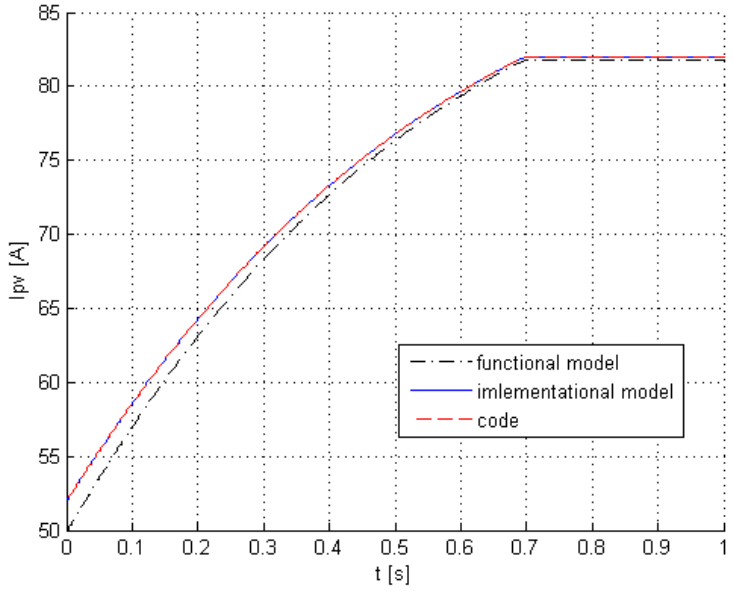


Figure 9.15: Responses of photovoltaic panel models

If quality of photovoltaic panels model is assumed to be satisfactory, testing process can be resumed by simulating model in Fig. 9.14, by generating code from it, and by executing the code on the embedded target. Recorded simulation traces match code execution traces perfectly, but they both differ from responses of the open-loop test. This discrepancy is caused by approximate implementation of photovoltaic panels model.

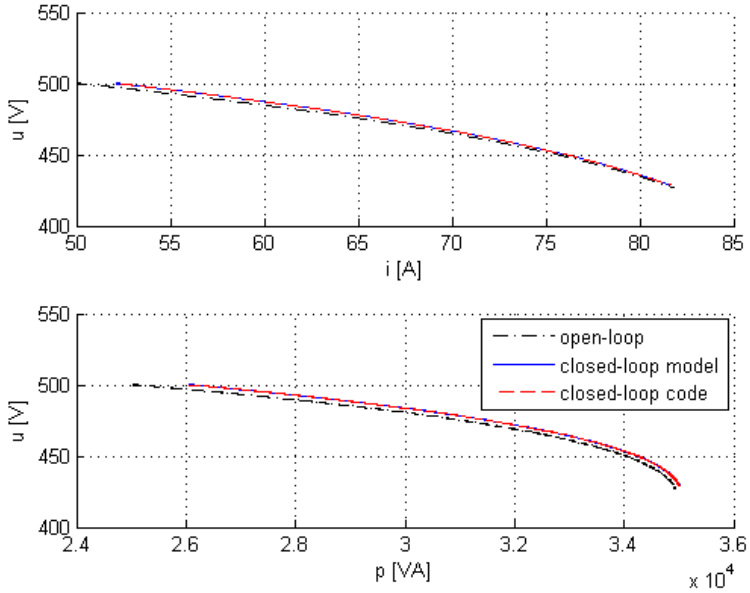


Figure 9.16: Real-time closed-loop MPPT testing functional results

This section presented closed-loop testing of the MPPT algorithm where photovoltaic panel model implementation and MPPT algorithm implementation have been executed in a closed-

loop on the real-time embedded target. Real-time properties of the MPPT algorithm have not been validated here because they have already been determined in open-loop test, section 9.2. If *MPPT* subsystem in model on Fig. 9.14 was embedded inside RTTP, similar as has been done in open-loop test, than real-time properties could be tested alongside functional operation. The main purpose of this section is to show that real-time testing of control algorithms in a closed-loop on embedded target is possible and to point out potential problems that could arise in designing such tests.

Chapter 10

Summary, Conclusion, and Outlook

10.1 Summary

In the first chapter of the thesis, background of the research has been presented. Application domain of the thesis' results has been designated to be low-volume safety critical hard real-time embedded control systems that should have long lifetime. Legacy software development environment, encompassing development tools, code generation scheme, importance of component reuse, and application development process, has been presented. Next, it has been shown how advantages and drawbacks of the legacy embedded software development process motivated the research and research gaps gave been identified in integrating legacy components into MBD of embedded control systems and in MBT of real-time properties of such systems. Three contributions have been proposed to fill these gaps.

In chapter 2, relevant terms regarding embedded systems, such as "real-time" and "hybrid", have been defined. The growing importance of embedded systems in general and of particular problems this thesis dealt with has been stressed through an overview of embedded market trends. A discussion on non-functional requirements has been provided which has shown divergence of their definitions in literature but has also explained their impact on embedded control systems in focus of conducted research. Emphasis has been placed on timing related non-functional requirements, so processing resources management and task execution analysis have been explored in more detail. Since the focus of the thesis has been on component based embedded control software development, definitions of the term CBSE have been discussed and it has been shown how proprietary components used in case studies fit into these definitions. Next, structure and organization of the proprietary software components has been elaborated. An overview of publications on model-based control software development has been presented by exploring how they relate to development aspects such as REQM, modeling, model transformations, ACG, and testing. Advantages of using patterns in software development and testing have been identified at the end of this chapter.

Chapter 3 has provided an analysis of selected MBT tools and methods by placing each of the investigated approaches into MBT taxonomy. The taxonomy available in literature has been adapted for real-time embedded control systems considered in the thesis by expanding it with additional categories and options. This analysis has shown that there is very little support

in investigated tools and methods for real-time unit and integration testing of hybrid embedded control systems. The MoBREST method has been shortly presented to show that it can fill the identified gap.

An approach to MBD of embedded real-time control software that systematically integrates legacy components has been presented in chapter 4. The approach has been showcased on integration of legacy assembler software components into MATLAB/Simulink environment via a toolbox developed in scope of the research. A Simulink blockset, a set of Simulink components, has been constructed where each component comprises graphical symbol with dialog window, the so called *Simulink masked block*, a dependency definition file, and an initialization function. It has been shown how these blocks can be used to create models or parts of models (*subsystems*) which can participate in a flexible ACG procedure that utilizes legacy software components. Mapping of high level software components (Simulink blocks) onto atomic components (legacy assembler macros) has been implemented using mapping definition files.

Model-Based Real-time Embedded System Testing method has been elaborated in chapter 5. The method uses Classification Tree Method extensively so it has been explained on an example from the literature. The remainder of the chapter has dealt with each step of the MoBREST component testing in more detail. Component configuration space partitioning has been presented that decomposes a software component on the MBD level into *variants*, representing software components on the embedded target implementation level, and further into *configurations*, representing parametrized instances of implementational components. The real-time testing pattern used in automated generation of test models for validation of functional and of real-time properties has been elaborated. It has been emphasized that test vectors, i.e. inputs to the test object, can be introduced from previous stages of development or that they can be generated using the developed tool. An iterative executable code generation and execution process has been introduced that ensures the desired test coverage. Finally, adjudication and test documentation procedures have been described.

In chapter 6, the MoBREST method has been adapted for integration testing of complex software control structures. Modifications to the real-time testing pattern have been presented and braking the testing process into three steps has been proposed: open-loop testing, closed-loop step-by-step testing and closed-loop real-time testing. Open-loop testing, which doesn't take environment (model) into account, has been further decomposed into two phases: firstly, functional testing is performed and, secondly, the system under test is incorporated into RTTP for real-time properties validation. Closed-loop step-by-step testing has been presented where model of the environment is executed inside simulation environment on a standard personal computer so real-time properties cannot be validated. Elements of the first two steps of the MoBREST integration testing have been merged into the third step, the closed-loop real-time testing. Here it has been presumed that the model of the environment can be accommodated for ACG and that the real-time embedded target has enough resources to execute it alongside the SUT incorporated in RTTP. If the said assumptions have been fulfilled, than validation of real-time properties of the SUT can be conducted in a closed control loop.

A case study on testing software components for Texas Instruments's digital signal controller has been presented in chapter 7. Firstly, the target controller family has been described

and a short overview of applications of members of this family in proprietary real-time control systems has been given. Next, the specific controller and experimental kit on which the study was performed have been introduced. Validation of the RTTP has been conducted to establish confidence in the testing tool and results it provides. Execution time measurement, variable CPU load generation, interrupt coverage, and interrupt delay detection functions of the RTTP have been checked. The whole component testing process and its outputs have been illustrated on an example of *LogicalOperator* component which performs Boolean operations. Partitioning of its' configuration space has produced fourteen variants which have fifty-four configurations all together so fifty-four sets of test vectors and the same number of test models have been automatically constructed. Executable applications have been automatically generated and executed and test results have been presented.

In chapter 8, real world application of MoBREST method in testing of software components for the *Safety Platform* has been summarized. The presented testing process has been conducted during certification of the Safety Platform as a part of railway crossing control system. The Safety Platform, based on 8051 series microcontroller, has been shortly introduced at the beginning of the chapter. Next, it has been shown that Safety Platform's software architecture deviates from assumptions made during design of MoBREST method. The RTTP and the MoBREST testing procedure have been adapted (simplified) and successfully applied to the Safety Platform. Every step in the testing process has been elaborated through examples of selected software components that have shown characteristic properties in the particular step.

The final case study that deals with real-time testing of photovoltaic MPPT algorithm is presented in chapter 9. The implemented MPPT algorithm, its' model as well as model of photovoltaic panels used during testing have been described. The model of the MPPT algorithm has then been accommodated to the ACG procedure and used in functional open-loop testing. The same model has been incorporated in the RTTP and real-time properties of the algorithm have been validated in the second stage of open-loop testing. The RTTP has been slightly adapted for control algorithm testing so these changes have been validated, similar to RTTP validation in chapter 7. Functional testing of the MPPT algorithm in a closed control loop with photovoltaic panels model simulated on a personal computer has been conducted in a step-by-step manner. This test has shown viability of proposed methods and developed tools for conventional PiL testing but has also revealed issues in commercial tool used to establish communication between real-time embedded target and simulation environment. In order to conduct real-time testing of the algorithm in a closed control loop, photovoltaic panels model has been adapted for ACG and tested in a functional open-loop test. It has been shown that panels model can be executed on the embedded target alongside the MPPT algorithm and that real-time testing can be performed with closed control loop.

10.2 Conclusion

Main goals of this thesis were to investigate possibilities of structured and systematic integration of legacy software components into MBD of embedded control systems and to propose methods and tools for validation of real-time properties of these systems. Suitable answers to

these questions were found neither in literature nor in available commercial tools. However, many partial answers were discovered and their elements were employed in methods and tools developed during the research. The remainder of this section summarizes most important contributions of the thesis.

The proposed ACG scheme, and MATLAB/Simulink toolbox GRAPlab which implements it, have shown that legacy software components can indeed be systematically integrated in development of embedded control software based on a model. The approach makes no assumptions as to programming language of the legacy components so even components written in assembly languages can be integrated in the development process. This is a more general solution to the legacy components issue than those available in investigated existing approaches which presume that the legacy code is in some higher programing language, mostly in C. It has been shown that this approach provides flexibility in managing components and facilitates their reuse, that it provides highly customizable ACG and enables linking of newly generated code with legacy object files. Extensive validation of the GRAPlab approach has been conducted through construction of test models and executable code generation in component and integration testing for two different embedded targets. This approach promises to ease transition from legacy embedded control systems development into modern MBD by transferring knowledge and confidence condensed in legacy software components across the gap.

A novel model-based method for validation of real-time properties of embedded control systems has been presented. The MoBREST-CT method performs partitioning of the tested component on two levels: firstly, component configuration space is partitioned so that each parametrized implementation of the component represents one test case and, secondly, input data ranges of the software component are partitioned during test vector generation. The former represents a novel application of the CTM. Besides generating test inputs, the method can be conducted with test vectors imported from previous stages of development. This way, MoBREST-CT method can supplement other MBT approaches to provide desired test coverage. For instance, thorough functional testing can be conducted using some of the available MBT tools followed by real-time properties validation using MoBREST-CT method and vectors from the functional testing phase. All steps of the MoBREST-CT method, from test generation, through test execution to its' documentation, can be fully automated, but they can also be tweaked or performed completely manually. This enables full control of the tests on the one side and effortless regression testing of large number of components on the other side.

It has been shown that MoBREST method can, with some adjustments, also be applied to validation of real-time properties of complex control software structures. A novel approach of generating executable target code from environment model and executing it alongside system under test in a real-time closed-loop test has been presented. If ACG of the environment code is not applicable, real-time propertied of the SUT can be validated in an open-loop real-time test. The tools developed in course of the research have been shown to be appropriate for conventional functional PiL testing where SUT executed on the target system is tested with simulated model of the environment.

The MoBREST method, with both its' component and integration testing subsets, fulfills most of the requirements placed on the generic MBT process in section 2.3:

- High level of automation aligns MoBREST method with interdisciplinary and iterative nature of MBD.
- Possibilities to reuse vectors from previous stages of testing and to compare functional test results with results of previous or subsequent tests provide link to testing on other integration levels.
- Usage of test patterns ensures systematic test design which achieves desired test coverage without redundancy and eases navigation among possibly hundreds of test cases.
- All test artifacts have graphical representations: classification trees (for configuration space partitioning and test vector generation), signal waveforms (for test vectors and test results), and models (for executable code). These are all systematically organized in automatically generated test documentation so tests are highly readable which enables stakeholders from different domains to participate in testing process.

The only requirement not met is the one for reactive testing where test cases depend on system behaviour. Reactive testing is primarily interesting in testing non-deterministic systems, which are not in scope of this thesis. Besides, real-time reactive test would require on-line test generation and control to be executed on the target system alongside the test application. This would significantly complicate the testing process and possibly overload the target's memory and processing resources.

Capabilities of GRAPlab toolbox and of MoBREST method have been shown through three case studies. Correct operation of the RTTP has been validated on a nanosecond scale by experiments in the first study. The second case study displayed great flexibility of the MoBREST-CT method. The final study has shown three different approaches to integration testing provided by the MoBREST-IT method, two of which are novel techniques for real-time properties validation. Credibility of the proposed methods and implemented tools stems from the fact that they have been applied during development of real-life industrial products based on two different hardware platforms: railway crossing control system and control system for the photovoltaic power inverter. In case of railway crossing control system, the proposed methods and tools have contributed to untypically short certification process; certification of similar systems is usually lengthy procedure with several iterations.

10.3 Outlook

Integration of legacy software components into MBD of embedded control systems presented in this thesis and implemented by the developed GRAPlab toolbox represents a first step toward modernization of development tools and processes in a conservative industrial environment. The shift is alleviated by reusing tried and proven components and its final goal is building confidence in the newly introduced methodologies. The next step in this direction would be using the GRAPlab ACG method alongside commercial code generator, e.g. *Simulink Coder* [138]. This way, automatically generated C/C++ code could be slowly introduced into development process, firstly in non critical tasks and later, as the confidence grows, even for safety critical

functions. The commercial code generator must in such case be configured to generate code that complies to restrictions inflicted by the proprietary tools and real-time kernel. Another research direction, but along the same lines, would be adaptation of GRAPlab code generation to commercial RTOS, e.g. to TI-RTOS [139].

Seamless integration of real-time properties validation, provided by the MoBREST method, with model-based functional testing would simplify the testing process and would facilitate its' usage. The integration scheme should be indiscriminatory with regard to targeted functional testing approach so that as many as possible of the abundant existing functional MBT methods can benefit from it. Such integration would enable the test engineer to use the most appropriate testing tool and, when functional testing is over, to validate real-time properties of the SUT with minimal effort.

Both GRAPlab and MoBREST are just peaces of a greater puzzle called model-based development of embedded control systems. As such they provide limited benefits on their own and their full potential can be realized only by integrating them in a complete MBD process. Modality of integration depends on application domain, on structure of the development team, and on existing toolchain into which they should be integrated. Case studies presented in the thesis provide an example of initial steps toward such integration, but additional research is necessary to achieve all round real-time MBD environment.

Acronyms

ACET	*average-case execution time*
ACG	*automated code generation*
AE	*application engineer*
AP	*application program*
AUTOSAR	*AUTomotive Open System ARchitecture*
B2B	*Back-to-Back*
BCET	*best-case execution time*
CAN	*controller area network*
CBSE	*Component-Based Software Engineering*
COFF	*Common Object File Format*
COTS	*commercial off-the-shelve*
CPU	*central processing unit*
CRC	*Cyclic Redundancy Check*
CT1	*cyclic task 1*
CT2	*cyclic task 2*
CTM	*Classification Tree Method*
CTM/ES	*Classification Tree Method for Embedded Systems*
DSC	*digital signal controller*
DSP	*digital signal processor*
ECU	*electronic control unit*
ESMoL	*Embedded Systems Modeling Language*
FBD	*Function Block Diagram*

FM	*functional module*
GPIO	*general purpose input-output*
GRAP	*GRaphical Application Programming tool*
GRAPlab	*GRAP Laboratory*
GReAT	*Graph Rewriting and Transformation*
HiL	*Hardware-in-the-Loop*
HTG	*Hybrid systems Test Generation*
HWM	*hardware module*
I2C	*inter-integrated circuit*
IDE	*integrated development environment*
KEEI	*KONČAR - Electrical Engineering Institute Inc.*
LIN	*local interconnect network*
MARTE	*Model and Analysis of Real-Time Embedded System*
MBD	*Model Based Development*
MBT	*model based testing*
McBSP	*multi-channel buffered serial port*
MCU	*microcontroller*
MD	*module developer*
MDD	*Model Driven Development*
MiL	*Model-in-the-Loop*
MiLEST	*Model-in-the-Loop for Embedded System Test*
MoBREST	*Model-Based Real-time Embedded System Testing*
MoBREST-CT	*MoBREST Component Testing*
MoBREST-IT	*MoBREST Integration Testing*
MPP	*maximum power point*
MPPT	*maximum power point tracking*
MTest	*Model Test*
N/A	*not applicable*

PDF	*Portable Document Format*
PiL	*Processor-in-the-Loop*
PL	*project leader*
PLC	*programmable logic controller*
PWM	*pulse width modulation*
REQM	*requirements management*
RTOS	*Real-Time Operating System*
RTTP	*real-time testing pattern*
RTW	*Real-Time Windows*
SBS	*sequence-based specification*
SCADE	*Safety Critical Application Development Environment*
SCI	*serial communication interface*
SE	*system engineer*
SEI	*Software Engineering Institute*
SIL	*Safety Integrity Level*
SiL	*Software-in-the-Loop*
SL DV	*Simulink Design Verifier*
SL VV	*Simulink Verification and Validation*
SPI	*serial peripheral interface*
SUT	*system under test*
SysML	*Systems Modeling Language*
TE	*test engineer*
TI	*Texas Instruments*
TPT	*Time Partitioning Test*
TPT-VM	*TPT Virtual Machine*
UML	*Unified Modeling Language*
VETESS	*Verification of Embedded systems for vehicles using automatic TESt generation from Specification*
VTB	*Virtual Test Bed*

VTB-RT	*Virtual Test Bed – Real Time extension*
WCET	*worst-case execution time*

List of Figures

List of Tables

Bibliography

[1] Ebert, C., Jones, C., "Embedded software: Facts, Figures, and Future", *Computer*, Vol. 42, No. 4, 2009, pp. 42–52.

[2] Zander-Nowicka, J., "Model-based Testing of Real-Time Embedded Systems in the Automotive Domain", PhD thesis, Technische Universität Berlin, Berlin, Germany, 2009.

[3] Hjertström, A., Nyström, D., Sjödin, M., "A Data-Entity Approach for Component-Based Real-Time Embedded Systems Development", In *IEEE Conference on Emerging Technologies & Factory Automation (ETFA 2009)*, Mallorca, Spain, September 2009.

[4] Schätz, B., Pretschner, A., Huber, F., Philipps, J., "Model-Based Development of Embedded Systems", Technical report, Technische Universität München 2002.

[5] Rau, A., "Model-Based Development of Embedded Automotive Control Systems", PhD thesis, University of Tübingen, Tübingen, Germany, 2002.

[6] Ueda, K., Baloh, M., Baloh, M., "Converting Legacy Embedded Control Software to Executable Specifications", In *MathWorks' International Automotive Conference*, Stuttgart, Germany, May 2006.

[7] Marijan, S., "Sustainability of embedded control systems for rail vehicles and power generation units", PhD thesis, University of Zagreb, Zagreb, Croatia, 2011.

[8] Babić, J., Marijan, S., Petrović, I., "The comparison of MATLAB/Simulink and proprietary code generator efficiency", In *Proceedings of the International Conference on Electrical Drives and Power Electronics, EDPE 2009*, Dubrovnik, Croatia, October 2009., pp. 12–14.

[9] Stürmer, I., "Systematic Testing of Code Generation Tools: A Test Suite-oriented Approach for Safeguarding Model-based Code Generation", PhD thesis, Technische Universität Berlin, Berlin, Germany, 2006.

[10] Marijan, S., "Control electronics of TMK2200 type tramcar for the City of Zagreb", In *Proc. International Symposium on Industrial Electronics, ISIE 2005*, Dubrovnik, Croatia, June 2005., pp. 1617–1622.

[11] Marijan, S., "Vehicle control unit for the light rail applications", In *Proc. 13th International Conference on Electrical Drives and Power Electronics, EDPE 2005*, Dubrovnik, Croatia, September 2005.

[12] Marijan, S., Petrović, I., "Platform based development of embedded systems for traction and power engineering applications – experiences and challenges", In *Proc. 5th IEEE International Conference on Industrial Informatics, INDIN 2007*, Vienna, Austria, July 2007.

[13] Helmerich, A., Koch, N., Braun, L. M. P., Dornbusch, P., Gruler, A., Keil, P., Leisibach, R., Romberg, J., Schätz, B., Wild, T., Wimmel, G., "Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area", Technical report, FAST GmbH, Munich, Germany and Technische Universität München, Germany 2005.

[14] Chowdhury, P. K., "Symbolic Interpretation of Legacy Assembly Language", Master's thesis, McMaster University, Hamilton, Ontario, 2005.

[15] Baloh, M., Raghav, G., Sivashankar, S., "Key Considerations in the Translation of Legacy Embedded Control Software to Model Based Executable Specifications", In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, Munich, Germany, October 2006.

[16] France, R., Rumpe, B., "Model-driven Development of Complex Software: A Research Roadmap", In *Proceedings of Future of Software Engineering (FOSE 2007)*, Minneapolis, MN, USA, May 2007.

[17] Toeppe, S., Ranville, S., Bostic, D., Wang, Y., "Practical Validation of Model Based Code Generation for Automotive Applications", In *Proceedings of the 18th Digital Avionics Systems Conference, Gateway to the New Millennium*, St Louis, MO, USA, October 1999.

[18] Ward, M. P., Zedan, H., Hardcastle, T., "Legacy Assembler Reengineering and Migration", In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, Chicago, Illinois, USA, September 2004.

[19] Jürjens, J., Reiss, D., Trachtenherz, D., "Model-Based Quality Assurance of Automotive Software", In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, MoDELS '08*, Toulouse, France, September/October 2008.

[20] Dang, T., "Model-Based Testing for Embedded Systems"„ chapter Part IV: Specific Approaches, Model-Based Testing of Hybrid Systems.

[21] Bienmüller, T., Brockmeyer, U., Sandmann, G., "Automatic Validation of Simulink/Stateflow Models - Formal Verification of Safety-Critical Requirements", In *International Automotive Conference*, Stuttgart, Germany, June 2004.

[22] Lehmann, E., "Time Partition Testing: A Method for Testing Dynamic Functional Behaviour", In *Proceedings of TEST2000*, London, Great Britain, 2000, pp. 1–11.

[23] Bringmann, E., Kramer, A., "Model-Based Testing of Automotive Systems", In *1st International Conference on Software Testing, Verification, and Validation*, Lillehammer, Norway, April 2008., pp. 485–493.

[24] PikeTec, GmbH, "TPT - model-based testing of embedded control systems", available at: http://www.piketec.com/products/tpt.php (3 March 2014.).

[25] Lu, B., Wu, X., Figueroa, H., Monti, A., "A Low-Cost Real-Time Hardware-in-the-Loop Testing Approach of Power Electronics Controls", *IEEE Transactions on Industrial Electronics*, Vol. 54, No. 2, 2007, pp. 919–931.

[26] Neukirchen, H. W., "Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests", PhD thesis, Georg-August-Universität zu Göttingen, Göttingen, Germany, 2004.

[27] Pfaller, C., Fleischmann, A., Hartmannd, J., Rappl, M., Rittmann, S., Wild, D., "On the Integration of Design and Test: A Model-Based Approach for Embedded Systems", In *International Conference on Software Engineering, Proceedings of the 2006 international workshop on Automation of software test*, Shanghai, China, May 2006., pp. 15–21.

[28] Bauer, T., Böhr, F., Eschbach, R., "On MiL, HiL, Statistical Testing, Reuse, and Efforts", In *Proceedings of the 1st Workshop on Model-based Testing in Practice, MoTiP 2008*, Berlin, Germany, June 2008.

[29] Philipps, J., Hahn, G., Pretschner, A., Stauner, T., "Prototype-Based Tests for Hybrid Reactive Systems", In *Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping*, San Diego, CA, USA, June 2003.

[30] Henzinger, T. A., "The Theory of Hybrid Automata", In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, New Brunswick, New Jersey, USA, July 1996.

[31] "ARTEMIS Strategic Research Agenda 2011", Technical report, ARTEMIS Joint Undertaking 2011.

[32] "World electronics Industry outlook 2015-2020, Challenges and opportunities ahead, Europe at a crossroad", Technical report, DECISION Etudes Conseil, Paris, France presentation held at Electronica 2012, Munich, Germany, 2012.

[33] "2013 Embedded Market Study", Technical report, UBM Tech Embedded 2013.

[34] Glinz, M., "On Non-Functional Requirements", In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE 2007)*, New Delhi, India, October 2007.

[35] Neukirchner, M., Stein, S., Ernst, R., "SMFF : System Models for Free", In *Proceedings of the 2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2011)*, Porto, Portugal, July 2011.

[36] Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K., "Volume I: Market Assessment of Component-Based Software Engineering", Technical report, Software Engineering Institute Technical Note CMU/SEI-2001-TN-007, 2000.

[37] Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K., "Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition", Technical report, Software Engineering Institute Technical Report CMU/SEI-2000-TR-008 ESC-TR-2000-007, 2000.

[38] Feljan, J., "Design-Time Verification of Component-Based Embedded Systems With Respect to Extra-Functional Properties", In *Proceedings of the 16th International Workshop on Component-Oriented Programming (WCOP)*, Boulder, Colorado, USA, June 2011.

[39] Rastofer, U., Bellosa, F., "An Approach to Component-Based Software Engineering for Distributed Embedded Real-Time Systems", In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2000)*, Orlando, Florida, USA, July 2000.

[40] Sentilles, S., Vulgarakis, A., Bureš, T., Carlson, J., Crnković, I., "A Component Model for Control-Intensive Distributed Embedded Systems", In *11th International Symposium on Component Based Software Engineering*, Karlsruhe, Germany, October 2008.

[41] Sentilles, S., Štěpán, P., Carlson, J., Crnković, I., "Integration of Extra-Functional Properties in Component Models", In *12th International Symposium on Component Based Software Engineering (CBSE)*, East Stroudsburg, Pennsylvania, USA, June 2009.

[42] Hamouche, R., Kocik, R., "Component-Based and Aspect-Oriented Methodology and Tool for Real-Time Embedded Control Systems Design", In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, March 2012., pp. 1421–1424.

[43] Lipka, R., Potuzak, T., Brada, P., Herout, P., "Verification of SimCo - Simulation Tool for Testing of Component-based Applications", In *Proceedings of the EUROCON*, Zagreb, Croatia, July 2013., pp. 467–474.

[44] Autosar Consortium, "AUTomotive Open System Architecture", available at: http://www.autosar.org/ (8 February 2014.).

[45] Schreiner, D., "Dissertation Component Based Communication Middleware for AUTOSAR", PhD thesis, Technische Universität Wien, Fakultät für Informatik, Wien, Austria, 2009.

[46] Quang, T. L., "Component Design Tool for Embedded System Components", Master's thesis, Mälardalen University, Department of Computer Science and Electronics, 2008.

[47] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., Stauner, T., "One evaluation of model-based testing and its automation", In

Proceedings of the 27th international conference on Software engineering, St. Louis, USA, May 2005., pp. 392–401.

[48] Boulanger, J.-L., Đao, V. Q., "Requirements engineering in a model-based methodology for embedded automotive software", In *IEEE International Conference on Research, Innovation and Vision for the Future, RIVF 2008.*, Ho Chi Minh City, China, July 2008., pp. 263–268.

[49] Siegl, S., Hielscher, K.-S., German, R., "Model Based Requirements Analysis and Testing of Automotive Systems with Timed Usage Models", In *18th IEEE International Requirements Engineering Conference*, Sydney, Australia, September 2010., pp. 345–350.

[50] Dubois, H., Peraldi-Frati, M.-A., Lakhal, F., "A model for requirements traceability in a heterogeneous model-based design process: Application to automotive embedded systems", In *15th IEEE International Conference on Engineering of Complex Computer Systems*, Oxford , UK, March 2010., pp. 233–242.

[51] Geisberger, E., Grünbauer, J., Schätz, B., "A Model-Based Approach To Requirements Analysis", In *Methods for Modelling Software Systems, MMOSS*, Dagstuhl, Germany, August 2007., pp. 1862–4405.

[52] Jnior, V. A. D. S., Vijaykumar, N. L., "Generating model-based test cases from natural language requirements for space application software", *Software Quality Journal*, Vol. 20, No. 1, 2011, pp. 77–143.

[53] OMG, "A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2", Technical Report ptc/2008-06-09, Object Management Group consortium 2008.

[54] Wang, Y., Yikun, L. X. W., "Modeling Embedded Software Test Requirement Based on MARTE", In *Proceedings of the 7th International Conference on Software Security and Reliability-Companion (SERE-C)*, Gaithersburg, Maryland, USA, June 2013., pp. 109–115.

[55] Wild, D., "AutoFOCUS 2: The Picture Book", Technical report, Technische Universität München 2006.

[56] Gambuzza, A., Koert, D., "A Concept for Improving the Reusability of Mechatronic System Models", In *In Proc. of the Workshop on Object-oriente Modeling of Embedded Real-Time Systems, OMER3*, Paderborn, Germany, October 2005., pp. 43–48.

[57] Mellor, A. J., Ulber, T., "Executable and Translatable UML", In *3rd Workshop on Object-oriented Modeling of Embedded Real-Time Systems, OMER3*, Paderborn, Germany, October 2005., pp. 69–72.

[58] Raghav, G., Gopalswamy, S., Radhakrishnan, K., Hugues, J., Delange, J., "Model based code generation for distributed embedded systems", In *European Congress on Embedded Real-Time Software,ERTS 2010*, Toulouse, France, May 2010., pp. 1–9.

[59] Prenninger, W., Pretschner, A., "Abstractions for Model-Based Testing", *Electronic Notes in Theoretical Computer Science*, Vol. 116, 2005, pp. 59–71.

[60] Agrawal, A., Karsai, G., Shi, F., "A UML-based graph transformation approach for implementing domain-specific model transformations", *International Journal on Software and Systems Modeling*, 2003, pp. 1–19.

[61] Eyisi, E., Zhang, Z., Koutsoukos, X., Porter, J., Karsai, G., Sztipanovits, J., "Model-Based Control Design and Integration of Cyberphysical Systems: An Adaptive Cruise Control Case Study", *Journal of Control Science and Engineering*, Vol. 2013, 2013.

[62] Sendall, S., Kozaczynski, W., "Model transformation: the heart and soul of model-driven software development", *IEEE Software*, Vol. 20, No. 5, 2003, pp. 42–45.

[63] Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A. L., Freund, U., E. Schlenker, H. J. W., "Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development", In *Proceedings of the conference on Design, Automation and Test in Europe, DATE05*, Munich, Germany, March 2005., pp. 1044–1049.

[64] Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S., Ratiu, D., "Seamless Model-based Development: from Isolated Tools to Integrated Model Engineering Environments", *Proceedings of the IEEE, Special Issue on Aerospace and Automotive Software*, Vol. 98, No. 4, 2010., pp. 526–545.

[65] Mäkinen, M. A., "Model Based Approach to Software Testing", Master's thesis, Helsinki University of Technology, 2007.

[66] Jones, C., "Applied Software Measurement", The McGraw-Hill Companies, 2008.

[67] Utting, M., Pretschner, A., Legeard, B., "A taxonomy of model-based testing", Technical report, University of Waikato 2006.

[68] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S., "A Pattern Language: Towns, Buildings, Construction", Oxford University Press, 1977.

[69] Zander-Nowicka, J., Pérez, A. M., Schieferdecker, I., Dai, Z. R., "Test Design Patterns for Embedded Systems", In *Proceedings of the 10th International Conference on Quality Engineering in Software Technology (CONQUEST 2007)*, Potsdam, Germany, September 2007., pp. 183–200.

[70] Berkenkotter, K., Kirner, R., "Real-Time and Hybrid Systems Testing",In Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A., , editors, *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)* Springer-Verlag New York, 2005.

[71] Reactive Systems, Inc., "Testing and Validation of Simulink Models with Reactis", available at: http://www.reactive-systems.com/ (8 February 2014.).

[72] Stürmer, I., Conrad, M., Dörr, H., Pepper, P., "Systematic Testing of Model-Based Code Generators", *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, 2007, pp. 622–634.

[73] Venkatesh, R., Shrotri, U., Darke, P., Bokil, P., "Test Generation for Large Automotive Models", In *Proceedings of the 2012 IEEE International Conference on Industrial Technology*, Athens, Grece, March 2012., pp. 662–667.

[74] Baresel, A., Conrad, M., Sadeghipour, S., Wegener, J., "The Interplay between Model Coverage and Code Coverage", In *Proceedings of the Conference On Computer Aided Systems Theory - EUROCAST , 2003*, Las Palmas de Gran Canaria, Canary Islands, Spain, February 2008.

[75] Pfaller, C., Pister, M., "Combining Structural and Functional Test Case Generation", In *Proceedings of Software Engineering 2008, SE08*, Munich, Germany, May 2008.

[76] Conrad, M., Fey, I., Sadeghipour, S., "Systematic Model-Based Testing of Embedded Control Software: The MB3T Approach", In *ICSE 2004 Workshop on Software Engineering for Automotive Systems*, Edinburgh, UK, May 2004., pp. 17–25.

[77] Prowell, S. J., Poore, J. H., "Foundations of Sequence-Based Software Specification", *IEEE Transactions on Software Engineering*, Vol. 29, No. 5, 2003, pp. 417–429.

[78] Bauer, T., Bohr, F., Landmann, D., Beletski, T., Eschbach, R., Poore, J., "From Requirements to Statistical Testing of Embedded Systems", In *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, Minneapolis, MN, USA, May 2007.

[79] Carter, J. M., Lin, L., Poore, J. H., "Automated Functional Testing of Simulink Control Models", In *Proceedings of the 1st Workshop on Model-based Testing in Practice, MoTiP 2008*, Berlin, Germany, June 2008.

[80] Iqbal, M. Z., Arcuri, A., Briand, L., "Empirical Investigation of Search Algorithms for Environment Model-Based Testing of Real-Time Embedded Software", In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*, New York, New York, USA, June 2012., pp. 199–209.

[81] Iyenghar, P., JuergenWuebbelmann, ClemensWesterkamp, Pulvermueller, E., "Model-Based Test Case Generation by Reusing Models From Runtime Monitoring of Deeply Embedded Systems", *Embedded Systems Letters*, Vol. 5, No. 3, 2013, pp. 38–41.

[82] Board, I. S. T. Q., "A taxonomy of model-based testing", Technical report, International Software Testing Qualification Board Produced by the 'Glossary Working Party', Editor: van Veenendaal E., 2010.

[83] Muresan, M., Pitica, D., "Software in the Loop Environment Reliability for Testing Embedded Code", In *Proceedings of the 18th International Symposium for Design and*

Technology in Electronic Packaging (SIITME), Alba Iulia, Romania, October 2012., pp. 325–328.

[84] Shokry, H., Hinchey, M., "Model-Based Verification of Embedded Software", *Computer*, Vol. 42, No. 4, 2009, pp. 53–59.

[85] Zander-Nowicka, J., "Reactive Testing and Test Control of Hybrid Embedded Software", In *Proceedings of the 5th Workshop on System Testing and Validation (STV 2007), in conjunction with ICSSEA 2007*, Paris, France, December 2007., pp. 45–62.

[86] The MathWorks, Inc., "Simulink Verification and Validation", available at: http://www.mathworks.com/products/simverification/ (8 February 2014.).

[87] The MathWorks, Inc., "Simulink Design Verifier", available at: http://www.mathworks.com/products/sldesignverifier/ (8 February 2014.).

[88] Sims, S., DuVarney, D. C., "Experience Report: The Reactis Validation Tool", In *Proceedings of the ICFP '07 Conference*, New York, USA, 2007, pp. 137–140.

[89] dSPACE GmbH, "TargetLink", available at: http://www.dspace.com (8 February 2014.).

[90] BTC Embedded Systems AG, "BTC EmbeddedValidator", available at: http://www.btc-es.de/index.php?idcatside=5&lang=2 (19 March 2014.).

[91] Grochtmann, M., "Test Case Design Using Classification Trees", In *Proceedings of STAR'94*, Washington, D.C., USA, May 1994.

[92] Lamberg, K., Beine, M., Eschmann, M., Otterbach, R., Conrad, M., Fey, I., "Model-based testing of embedded automotive software using MTest", In *SAE World Congress 2004*, Detroit, USA, March 2004.

[93] Conrad, M., Fey, I., "Systematic Model-Based Testing of Embedded Automotive Software", *Electronic Notes in Theoretical Computer Science*, Vol. 111, 2005, pp. 13–26.

[94] Conrad, M., Krupp, A., "An Extension of the Classification-Tree Method for Embedded Systems for the Description of Events", *Electronic Notes in Theoretical Computer Science*, Vol. 164, No. 4, 2006, pp. 3–11.

[95] Alexander Krupp, W. M., "A Systematic Approach to the Test of Combined HW/SW Systems", In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE*, Dresden , Germany, March 2010., pp. 323–236.

[96] Conrad, M., "A Systematic Approach to Testing Automotive Control Software", In *Proc. of Convergence 2004*, Detroit, USA, October 2004.

[97] Model Engineering Solutions GmbH, "MTest classic User Guide", available at: http://www.mtest-classic.com (8 February 2014.).

[98] The MathWorks, Inc., "SystemTest, Product Description", available at: http://www.mathworks.com/products/systemtest/ (8 February 2014.).

[99] The MathWorks, Inc., "Parallel Computing Toolbox", available at: http://www.mathworks.com/products/parallel-computing/ (14 February 2014.).

[100] Dai, Z. R., Engel, K.-D., Truscan, D., Streitferdt, D., Vouffo, A., Rennoch, A., "Test Modeling, Test Generation and Test Execution with Model-Based Testing", Technical Report D.2.1.v2.1, D-Mint Consortium 2009.

[101] ETAS GmbH, "Software Products & Systems", available at: http://www.etas.com/en/products/software_products.php (3 March 2014.).

[102] Vector Informatik GmbH, "CANape - Measuring, Calibrating, Diagnosing and Flashing ECUs", available at: http://vector.com/vi_canape_en.html?markierung=CANape (3 March 2014.).

[103] Conrad, M., Sadeghipour, S., Wiesbrock, H. W., "Automatic Evaluation of ECU Software Tests", In *SAE 2005 World Congress*, Detroit, USA, April 2005.

[104] ITPower Solutions GmbH, "MEval", available at: http://www.itpower.de/101-1-MEval-Automatic-signal-comparison-in-MATLABSimulink.html (8 February 2014.).

[105] Zander-Nowicka, J., Mosterman, P. J., Schieferdecker, I., "Quality of Test Specification by Application of Patterns", In *Proceedings of the 15th Conference on Pattern Languages of Programs - PLoP '08*, Nashville, TN, USA, October 2008.

[106] VETESS Consortium, "VETESS - Verification of Embedded systems for vehicles using automatic TESt generation from Specification", available at: http://lifc.univ-fcomte.fr/vetess/index.php (10 March 2014.).

[107] Ambert, F., Bouquet, F., Lasalle, J., Legeard, B., Peureux, F., "Applying an MBT Toolchain to Automotive Embedded Systems: Case Study Reports", In *Proceedings of the 4-th Int. Conf. on Advances in System Testing and Validation Lifecycle, VALID'12*, Lisbon, Portugal, November 2012., pp. 139–144.

[108] Gauthier, J.-M., "Test Generation for RTES from SysML Models: Context, Motivations and Research Proposal", In *Proceedings of the 6-th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Luxembourg, March 2013., pp. 503–504.

[109] Enoiu, E. P., Doganay, K., Bohlin, M., Sundmark, D., Pettersson, P., "MOS: An Integrated Model-Based and Search-Based Testing Tool for Function Block Diagrams", In *Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, San Francisco, CA, USA, May 2013., pp. 55–60.

[110] Enoiu, E. P., "Model-based Test Suite Generation for Function Block Diagrams using the UPPAAL Model Checker", In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Luxembourg, March 2013., pp. 158–167.

[111] ETAS Group, "ETAS RT2: Test for Model-in-the-Loop and Software-in-the-Loop", available at: http://www.etas.com/en/products/rt2.php (16 March 2014.).

[112] Dormoy, F. X., "Scade 6: a model based solution for safety critical software development", In *Proceedings of the 4th European Congress on Embedded Real Time Software, ERTS '08*, Toulouse, France, January 2008.

[113] Technologies, E., "SCADE Suite 6.4 Technical Data Sheet", Technical Report SC-TDS-6.4 - 21/01/13, Esterel Technologies 2013.

[114] Technologies, E., "SCADE Suite Model Test Coverage 6.2 Technical Data Sheet", Technical Report MTC-TDS-6.2 - 19/04/11, Esterel Technologies 2011.

[115] Dajani-Brown, S., Cofer, D., Bouali, A., "Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems"„ chapter Formal Verification of an Avionics Sensor Voter Using SCADE.

[116] McElroy, J., "Reliving Pressure for UAV Software Development", *Electronic Product Design and Test*, 2009.

[117] The MathWorks, Inc., "Block Masks", available at: http://www.mathworks.com/help/simulink/ug/block-masks.html (8 February 2014.).

[118] The MathWorks, Inc., "Use S-Functions in Models", available at: http://www.mathworks.com/help/simulink/sfg/using-s-functions-in-models.html (8 February 2014.).

[119] Babić, J., Marijan, S., Petrović, I., "Introducing Model-Based Techniques into Development of Real-Time Embedded Applications", *AUTOMATIKA: Journal for Control, Measurement, Electronics, Computing and Communications*, Vol. 52, No. 4, 2011, pp. 329–338.

[120] Mattavelli, P., "Digital Control of dc-dc Boost Converters with Inductor Current Estimation", In *Nineteenth Annual IEEE Applied Power Electronics Conference and Exposition, APEC '04.*, Anaheim, California, USA, February 2004., pp. 74–80.

[121] Kaner, C., "Software Negligence and Testing Coverage", In *Proceedings of Fifth International Conference on Software Testing, Analysis, and Review (STAR 96)*, Orlando, Florida, USA, 1996.

[122] Berner & Mattner Systemtechnik GmbH, "Classification Tree Editor CTE XL", available at: http://www.berner-mattner.com/en/berner-mattner-home/products/cte/index.html (8 February 2014.).

[123] Jonathan Bennett & AutoIt Team, "AutoIt v3", available at: http://www.autoitscript.com/site/autoit/ (8 February 2014.).

[124] The MathWorks, Inc., "Real-Time Windows Target", available at: http://www.mathworks.com/products/rtwt/ (8 February 2014.).

[125] Wikipedia, "Digital signal controller", available at: http://en.wikipedia.org/wiki/Digital_-signal_controller (8 February 2014.).

[126] Texas Instruments, Inc., "Overview for C2000 32-bit Real-time Control MCUs", available at: http://www.ti.com/lsds/ti/microcontroller/32-bit_c2000/overview.page (8 February 2014.).

[127] "TMS320F28335, TMS320F28334, TMS320F28332, TMS320F28235, TMS320F28234, TMS320F28232 Digital Signal Controllers (DSCs) Data Manual", Technical Report SPRS439I, Texas Instruments, Inc. 2011.

[128] "C8051F58x/F59x Mixed Signal ISP Flash MCU Family", Technical report, Silicon Labs, Inc. 2011.

[129] "Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)", Technical Report EN 50126:1999, European Committee for Electrotechnical Standardization 1999.

[130] Smith, D. J., "Reliability, Maintainability and Risk", Elsevier Butterworth-Heinemann, 2005.

[131] "Railway applications - Communications, signaling and processing systems - Software for railway control and protection systems", Technical Report EN 50128:2001, European Committee for Electrotechnical Standardization 2001.

[132] Morales, D. S., "Maximum Power Point Tracking Algorithms for Photovoltaic Applications", Master's thesis, Aalto University, School od Science and Technology, Faculty of Electronics, Communications and Automation, 2010.

[133] Teodorescu, R., Liserre, M., Rodriguez, P., "Grid Converters for Photovoltaic and Wind Power Systems Grid Converters for Photovoltaic and Wind Power Systems", John Wiley & Sons, 2011.

[134] González-Longatt, F. M., "Model of Photovoltaic Module in Matlab", In *2DO Congreso Iberoamericano de estudiantes de ingenieria electrica, electronica y computacion, II CIB-ELEC 2006*, Puerto la Cruz, Venezuela, April 2006.

[135] Femia, N., Petrone, G., Spagnuolo, G., Vitelli, M., "Power Electronics and Control Techniques for Maximum Energy Harvesting in Photovoltaic Systems", CRC Press, 2012.

[136] Babic, J., Čihak, T., Marijan, S., "Model-Based Development of MPPT Algorithm with Legacy Components Integration", In *Proceedings of the International Conference on Electrical Drives and Power Electronics, EDPE 2013*, Dubrovnik, Croatia, October 2013.

[137] Sera, D., Kerekes, T., Teodorescu, R., Blaabjerg, F., "Improved MPPT Algorithms for Rapidly Changing Environmental Conditions", In *Proceedings of the 12th International Power Electronics and Motion Control Conference, EPE-PEMC 2006*, Portoroz, Slovenia, August–September 2006., pp. 1614–1619.

[138] The MathWorks, Inc., "Simulink Coder", available at: http://www.mathworks.com/products/simulink-coder/description1.html (8 February 2014.).

[139] Texas Instruments, Inc., "TI-RTOS: Real-Time Operating System (RTOS)", available at: http://www.ti.com/tool/ti-rtos (8 February 2014.).

Curriculum Vitae

Josip Babić was born in 1982 in Slavonski Brod, Croatia where he completed elementary and secondary schools. In 2006 he received B.Sc. degree from the Faculty of Electrical Engineering, University of Zagreb.

Since 2007 he has been with Section for Embedded Systems of Power Electronics and Control Department at Končar - Electrical Engineering Institute in Zagreb, where he is currently employed. He is responsible for research and development of embedded systems based on digital signal microcontrollers. He participated in development of embedded control systems for synchronous machine excitation systems, for traction converters in tramcars and in trains, for multi-system converters in passenger coaches, for auxiliary power supplies for trains and trams, and for wind turbines. Besides that, he was also involved in development of monitoring systems for structural vibrations in wind turbine, for rotating machinery, and for power transformers.

His main research interests are model based development and testing in the field of embedded real-time systems. He is author or co-author of six papers published in a journal and in proceedings of domestic and foreign conferences.

In 2009 he and his colleagues received national award ARCA for main control unit of low floor train. Wind turbine control system for strong and turbulent wind conditions developed by his colleagues and him also won national award ARCA in 2012.

Životopis

Josip Babić rođen je 1982 u Slavonskom Brodu gdje je završio osnovnu i srednju školu. Godine 2006 primio je diplomu Fakulteta elektrotehnike i računarstva na Sveučilištu u Zagrebu.

Od 2007 zaposlen je u Odjelu za ugradbene računalne sustave Zavoda za energetsku elektroniku i upravljanje pri Končar - Institutu za elektrotehniku. Njegove odgovornosti obuhvaćaju istraživanje i razvoj ugradbenih računalnih sustava zasnovanih na signalnim mikrokontrolerima. Sudjelovao je u razvoju ugradbenih sustava upravljanja za sustave uzbude sinkronih strojeva, za pretvarač glavnog pogona tramvaja, za višesistemske vagonske pretvarače, za pomoćne pogone tramvaja i vlaka i za vjetroagregat. Pored toga, bio je uključen u razvoj sustava nadzora strukturnih vibracija vjetroagregata, rotacijskih strojeva i energetskih transformatora.

Provodi istraživanja na polju modelskog razvoja i vrednovanja programske podrške ugradbenih računalnih sustava za rad u stvarnom vremenu. Autor je ili koautor šest znanstvenih radova objavljenih u jednom časopisu i u zbornicima međunarodnih konferencija.

Zajedno s kolegama dobitnik je nacionalne nagrade ARCA 2009. godine za centralno računalo niskopodnog elektromotornog vlaka. Sustav upravljanja vjetroagregatom za područja jakih i turbulentnih vjetrova, u čijem razvoju je sudjelovao, također je 2012. godine osvojio nacionalnu nagradu ARCA.

Publications

- Babić, J., Čihak, T., Marijan, S., "Model-Based Development of MPPT Algorithm with Legacy Components Integration", In Proceedings of the International Conference on Electrical Drives and Power Electronics, EDPE 2013, Dubrovnik, Croatia, October 2013.
- Babić, J., Marijan, S., Petrović, I., "Introducing Model-Based Techniques into Development of Real-Time Embedded Applications", AUTOMATIKA: Journal for Control, Measurement, Electronics, Computing and Communications, Vol. 52, No. 4, 2011, pp. 329-338.
- Babić, J., Marijan, S., Petrović, I., "The comparison of MATLAB/Simulink and proprietary code generator efficiency", In Proceedings of the International Conference on Electrical Drives and Power Electronics, EDPE 2009, Dubrovnik, Croatia, October 2009, pp. 12-14.
- Tečec, Z.; Babić, J.; Petrović, I., "Implementation of Fuzzy-Model Based Autotuning Power System Stabilizer" In Proceedings of the International Conference on Electrical Drives and Power Electronics, EDPE 2009, Dubrovnik, Croatia, October 2009.
- Babić, J.; Budišić, Marko; Petrović, I., "Dynamic Window based Force Reflection for Safe Teleoperation of A Mobile Robot via Internet" In Proceedings of the 2007 IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM2007, Zurich, Switzerland, September 2007.
- Petrović, I.; Babić, J.; Budišić, M., "Teleoperation Of Collaborative Mobile Robots With Force Feedback Over Internet" In Proceedings of the Fourth International Conference on Informatics in Control, Automation and Robotics, Angers, France, May 2007, pp. 430-437
- Babić, J., "Force Feedback in Mobile Robots Teleoperation", pre-Bologna graduate thesis, Faculty of electrical engineering and computing, University of Zagreb, July 2006.