University of West Bohemia

Faculty of Applied Sciences

DOCTORAL THESIS

2012 Dipl.-Ing.(FH) Michael Steindl, M.Eng.

Západočeská univerzita v Plzni
Fakulta aplikovaných věd

VYHODNOCENÍ A URČENÍ POŘADÍ INTEGRACE V KOMPONENTOVĚ ORIENTOVANÝCH VESTAVĚNÝCH SYSTÉMECH

Michael Steindl

disertační práce
k získání akademického titulu doktor
v oboru Informatika a výpočetní technika

Školitel: Doc. Ing. Stanislav Racek, CSc.

Katedra: Katedra informatiky a výpočetní techniky

Plzeň 2012

University of West Bohemia
Faculty of Applied Sciences

EVALUATION AND DETERMINATION OF INTEGRATION ORDERS IN COMPONENT BASED EMBEDDED SYSTEMS

Michael Steindl

Doctoral Thesis
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in specialization
Computer Science and Engineering

Supervisor: Doc. Ing. Stanislav Racek, CSc.

Department: Department of Computer Science and Engineering

Pilsen 2012

Abstract

Embedded software systems are getting more and more complex. The demand for new features and functions led to an increasing complexity in the design and development of these systems. One answer to handle this complexity is component-based development, in which systems are built of individual independent software components. These components shall work together to provide the set or specified subset of the capabilities the final system will provide. One important aspect of the component-based development approach is software integration. Individual components have to be put together and their interactions have to be verified. The crucial point of integration is the order in which components are combined. State-of-the-art approaches (*eg.* top-down or bottom-up integration) are only coarse guidelines and rely strongly on integrators expertise. More elaborate methods in which an algorithm is used to derive an integration order are only available for object-oriented software and cannot be directly used in procedural programming languages, *eg.* the language C. To deal with these challenges, parameters are identified the software integration process is subjected to and metrics are developed in order to evaluate a certain integration order. Furthermore, an optimization approach based on simulated annealing is presented which is used to derive an integration order with respect to the proposed parameters in a powerful and reliable manner.

Zusammenfassung

Die Komplexitaet eingebetteter Systeme steigt kontinuierlich. Mit dem Bedarf an neuen Funktionalitaeten nimmt der Anspruch an den Design- und Entwicklungsprozess dieser Systeme immer mehr zu. Ein vielversprechender Ansatz diesem Anspruch zu begegnen ist die komponentenbasierte Entwicklung. Hierbei werden Systeme aus moeglichst unabhaengigen Einzelkomponenten aufgebaut, die im Zusammenspiel die Anforderungen an das Gesamtsystem erfuellen. Ein wichtiger Aspekt bei diesem Ansatz ist die Software Integration, bei der die einzelnen Komponenten zu einem funktionierenden Gesamtsystem zusammengesetzt und die Interaktionen untereinander getestet werden. Der entscheidende Punkt hierbei ist die Reihenfolge mit der dies geschieht. Uebliche Ansaetze, wie z.B. Top-Down oder Bottom-up Integration bieten lediglich grobe Richtlinien und sind stark von der Expertise des jeweiligen Integrators abhaengig. Algorithmische Ansaetze, welche eine praezise Reihenfolge vorgeben, sind bisher nur fuer Objekt-orientierte Software verfuegbar und koennen nur sehr eingeschraenkt fuer prozedurale Programmiersprachen (z.B. C) verwendet werden. Aufgrund der sehr starken Verbreitung prozeduraler Programmiersprachen werden in dieser Arbeit Parameter und Metriken definiert, mit den Integrationsreihenfolgen fuer prozedurale Programmiersprachen bewertet werden koennen. Zusaetzlich wird ein Optimierungsverfahren auf Basis des Simulated Annealing Ansatzes vorgestellt. Mit diesem Verfahren kann zuverlaessig eine praezise Integrationsreihenfolge auf Basis der definierten Parameter gewonnen werden.

Abstrakt

Vestavěné počítačové systémy jsou stále složitější. Požadavky na nové vlastnosti a funkce vedly ke vzrůstající složitosti procesů návrhu a vývoje těchto systémů. Jeden způsob jak zvládnout tuto složitost je komponentově orientovaný vývoj, při kterém jsou systémy stavěny z jednotlivých nezávislých softwarových komponent. Tyto komponenty musí spolupracovat při zajištťování množiny nebo podmnožiny služeb, které by cílový systém měl poskytovat. Jeden důležitý aspekt komponentově orientovaného přístupu je softwarová integrace. Jednotlivé komponenty musí být sestaveny a jejich interakce musí být ověřeny. Klíčovým bodem integrace je pořadí, ve kterém jsou použité komponenty kombinovány. Současné přístupy (např. integrace top-down nebo bottom-up) poskytují pouze hrubé postupy a silně spoléhají na zkušenost návrháře. Více propracované algoritmické metody jsou dostupné pouze pro objektově orientovaný software a nemohou být přímo použity v procedurálních programovacích jazycích, například v jazyce C. V předložené disertační práci jsou identifikovány parametry integračního procesu a jsou navrženy metriky umožňující vyhodnocení určitého pořadí integrace komponent. Dále je prezentován optimalizační přístup, založený na tzv. simulovaném žíhání, který lze využít k odvození pořadí integrace komponent efektivním a spolehlivým způsobem.

Declaration of Authenticity

I hereby declare that this doctoral thesis is my own original and sole work.
Only sources listed in the bibliography were used.

Čestné prohlášení

Prohlašuji tímto, že tato disertační práce je původní a vypracoval jsem jí samostatně.
Použil jsem jen citované zdroje uvedené v přehledu literatury.

In Pilsen

V Plzni dne 05.11.12

[illegible]

To Paul and Kathrin

Acknowledgements

Many people have either directly and indirectly contributed to this thesis. I would like to take the opportunity to thank all of them for their support. First and foremost, I would like to sincerely thank my supervisors Doc. Ing. Stanislav Racek, CSc. and Prof. Dr. Juergen Mottok for their guidance, invaluable suggestions, continued encouragement and support during these years. I am also very grateful to Prof. Dr. Michael Niemetz, Prof. Dr. Hans Meier and Ing. Premek Brada, MSc, PhD for their time spend on reviews and fruitful discussions. Special thanks go to my former colleagues at the University of Applied Sciences Regensburg, Dr. Michael Deubzer, Dr. Martin Hobelberger and Michael Schorer for their friendship, encouragement and helpful advice over the years. I will never forget the time we spent on joint research and conference travels, as well as the fruitful discussions and the social activities in the evenings. Finally, my very special thanks belong to Monika and Alfons Steindl for their warm support and for always believing in me all these years.

Contents

List of Figures

Abbrevations

N_{ST}	Number of stubs
N_{TD}	Number of test drivers
E_{TE}	Test effort including test drivers
E_{SC}	Overall stub complexity
$E_{\overline{SC}}$	Normalized overall stub complexity
E_{TC}	Test complexity
E_C	Planning effort for a single component
E_S	Planning effort for the overall integration

Chapter 1

Introduction

The integration of software components is an important aspect of embedded system development. This thesis addresses the software integration process with a particular emphasis on the integration strategy which describes the order of component integration. This introduction presents the motivation behind this work, basic concepts, problem formulation and contributions.

1.1 Motivation

Embedded software systems are getting more and more complex. An example for this is given by the automotive industry: up to over 100 microcontrollers in a premium car is the normal case and about 80 % of all the innovations in modern cars are achieved by software [49] which leads to an increasing number of functionalities realized by software (Figure 1.1). The number of line of code has increased from zero to to tens of millions within the last 30 years and about 270 functions a user interacts with are implemented by software in modern cars [14]. Also the cost of software is growing, a study by Mercer Management Consulting and Hypovereinsbank quantifies the value percentage of software in a car in 2010 by 13% (in comparison with 4% in 2000) [41, 32]. Additionally, more and more responsible challenges like safety- and security-critical scenarios are tackled which lead to high demands on software dependability, maintainability and testability [67].

During the last years, approaches have been emerged that aimed to cope with this challenges. One of these approaches is the use of component-based software development which has proven to support the development of complex software solutions [63]. In component-based software development, systems are built of individual independent soft-

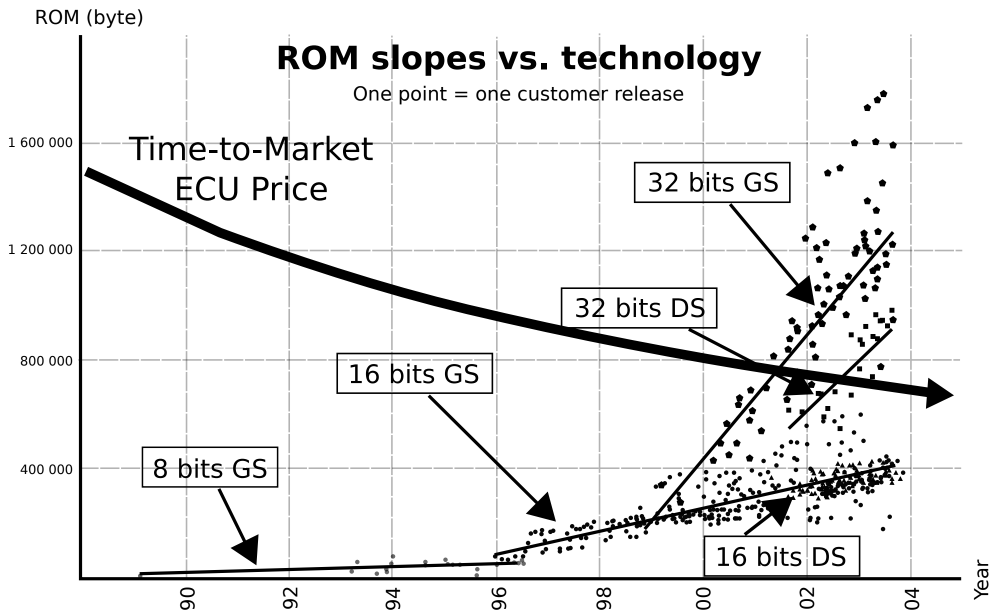


Figure 1.1: Evolution of automotive engine control units (ECUs) [20]. Within 14 years, time-to-market and the ECU price have been roughly cut to a half whereas the need for ROM is increasing drastically.

ware components. These components shall work together to provide the set or specified subset of the capabilities the final system will provide. A major requirement which is attended to component-based software development is the reuse of existing components [54]. To build up a system by use of components which are already tested and proved in other systems leads to shorter development times and a higher reliability [33],[36]. In fact, automotive software functionalities are subjected to only little changes between the vehicle generations, they actually differs mostly not more than 10% [14]. Therfore, a substantial part of software may be reused.

However, typically much more than 10% of software is modified in a new vehicle. Due to high cost pressure on hardware parts, software is strongly optimized to a certain hardware configuration and cannot be ported to new hardware with reasonable effort [14]. Additionally, the reuse objectives often differ between OEMs and suppliers and are not as standardized as needed for the use of eg. software product lines [14, 72]. Given that the total number of software components in an active General Motors car is about 1200 [48], a significant portion of new components must be integrated to a system at each development step. Furthermore, there is a shortage of adequate methods and strategies for the integration of those components, especially for determining an order of integration. It should be mentioned that simply tryout every possible integration order and choose the best is hardly possible since there are $n!$ integration orders for n components. Therefore, for example, there exist about 2.4×10^{18} possibilities when only 20 components should be integrated.

This thesis deals with the evaluation and determination of an integration order in a component-based embedded systems and is divided into four main chapters. Chapter I introduces the motivation behind this work. In Chapter II, the necessary background needed to understand the thesis is presented. Furthermore the state of the art in the world and the goals of the thesis are described. Chapter III introduces the parameters software integration is subjected to and Chapter IV presents a simulated annealing approach to derive an integration order with respect to these parameters. The thesis concludes with a conclusion and further work.

Chapter 2

State of the Art Software Integration Strategies

A software integration strategy is needed to provide software testers a guideline to perform software integration testing activities in a rational way. It usually describes an order in which components are integrated and tested [26]. There are several approaches to devise integration test orders which could be basically divided into two groups: *formalized* and *non-formalized* approaches. Non-formalized approaches (*eg.* [5], [26], [6]) are depending heavily on the expertise of the integrator whereas formalized strategies offer a more distinct description of the integration sequence. This makes formalized approaches more suitable to situations, where integration has to be carried out with a limited understanding of the complete system. This is the case in many of the complex software systems in the automotive industry.

In the following, the basic concepts in component based software development are described. After that the current state of the art for deriving an integration order is presented. The chapter concludes with a discussion of the presented approaches and defines the goals of the thesis.

2.1 Component-based software development

In the terms of component-based software engineering, a software system is a set of components connected to each other in order to provide a set of features. Four main principles are covered by this approach [22]:

1. Reusability: Components may be reused in different systems

2. Substitutability: Different implementations of a component may be used

3. Extensibility: The functionality of individual components may be increased

4. Composability: Components may be composed to provided a desired functionality

There is a lively discussion about what is exactly a component. A very general definition for a component is formulated by Brown in [15]: *A component is an independently deliverable piece of functionality providing access to the services through interfaces.* However, Kopetz claims in [40], that an ideal component is a *self-contained computer with its own hardware (processor, memory, communication interface, interface to the controlled object) and software (application programs, operating system), which performs a set of well-defined functions within the distributed computer system.* Since this work deals with software integration, the expression *component* is used for software components only and the widely accepted definition given by Szyperski [68] in 2002 is used: *A software component is a unit of composition with contractually specified interfaces and explicit context dependency only. A software component can be deployed independently and is subject to composition by third parties.*

Among many other challenges in component-based software development (*eg.* [21]), components must be put together to form the entire software system. Therefore components must be integrated which can be illustrated as a mechanical process of wiring components together. In [34], software integration is defined as *the process of combining software components, hardware components, or both into an overall system.* To interact with each other, the interfaces of components are connected by *dependencies.* If a component `C2` uses one or more service(s) of another component `C1`, the formulation `C2` ”depends” on `C1` is used (Figure 2.1). The testing of these dependencies, called integration testing, *insures the consistency of component interfaces and whether the components pass data and control correctly, which results in successful integration of dependent components* [25]. In other words, integration testing ensures the correct interaction between already tested components. Software integration and integration testing are often used synonymous and are not distinguishable in literature.

If a system should be integrated and several components not available yet, this components can be replaced by *stubs.* In [34] a stub is defined as *a skeletal or special-purpose implementation of a software module, used to develop or test a module that calls or is otherwise dependent on it* and, second, *as a computer program statement substituting for the*

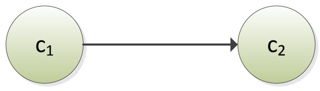


Figure 2.1: Dependency between two components were component C2 depends on C1

body of a software module that is or will be defined elsewhere. In Figure 2.2, the integration of a component (C2) which depends on a component which is not available yet (C1).

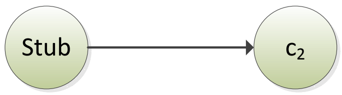


Figure 2.2: Integration of component C2 which depends on a not available component (C1.)

To perform integration testing, components must be stimulated with test data and the test results must be observed (Figure 2.3). To achieve this, another special component implementation is necessary, a *test driver*. A test driver is defined as *a software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results* [34].

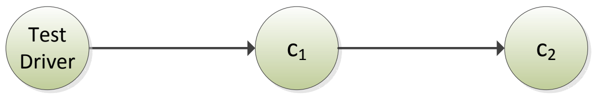


Figure 2.3: Test driver needed to stimulate the interaction between C1 and C2 and to observe the test results.

An integration strategy describes the order in which components are integrated and thus integration testing is performed.

2.2 Integration strategies

In the following state of the art integration strategies are presented. One integration strategy has an exceptional position: *big bang integration*. Big bang integration is not an integration strategy in a classical meaning. All components are built and brought together in the system without regard for inter-component dependencies or risk. This leads to difficulties in fault identification. If a failure is encountered, all components are equally under suspicion.

In [5], big bang integration is characterized by Beizer as follows:

In its purest (and vilest) form, big bang integration is no method at all - 'Let's fire it up and see if it works! It doesn't of course.

There are only a few situations where big bang integration is indicated:

- The system is stabilized, i.e. only few components were added or changed since the last passed test.
- The system is small and testable, i.e. all components have passed the component test.
- The system is monolithic and cannot be exercised separately.

Since in complex embedded systems none of these situation is present in most cases, big bang integration is not a proper solution and it usually creates more problems than it solves. Due to this, big bang integration is not further considered in this document.

2.2.1 Non-formalized approaches

Non-formalized approaches are very generic guidelines to derive an integration order and are based either on an architectural or on a functional view on the system. Well-known strategies such as *bottom up* (Figure 2.4) or *top down* (Figure 2.5) integration are representatives of architectural sight. Bottom up integration achieves stepwise verification of the interfaces between tightly coupled components. Components with the least number of dependencies are integrated first. When these components pass, their test drivers are replaced with their clients and another round of integration begins. This strategy is suited for responsibility-based designs and systems of components with stable and robust interface definitions. Bottom up integration is also widely used if a project is started from scratch. In the top down approach the order of integration is reversed in comparison with the bottom up pattern. The top level component is coded first and the unavailable lower level components are implemented by stubs. After that, the stubs are replaced stage wise by full implementations and the next lower level of components is stubbed. As a combination

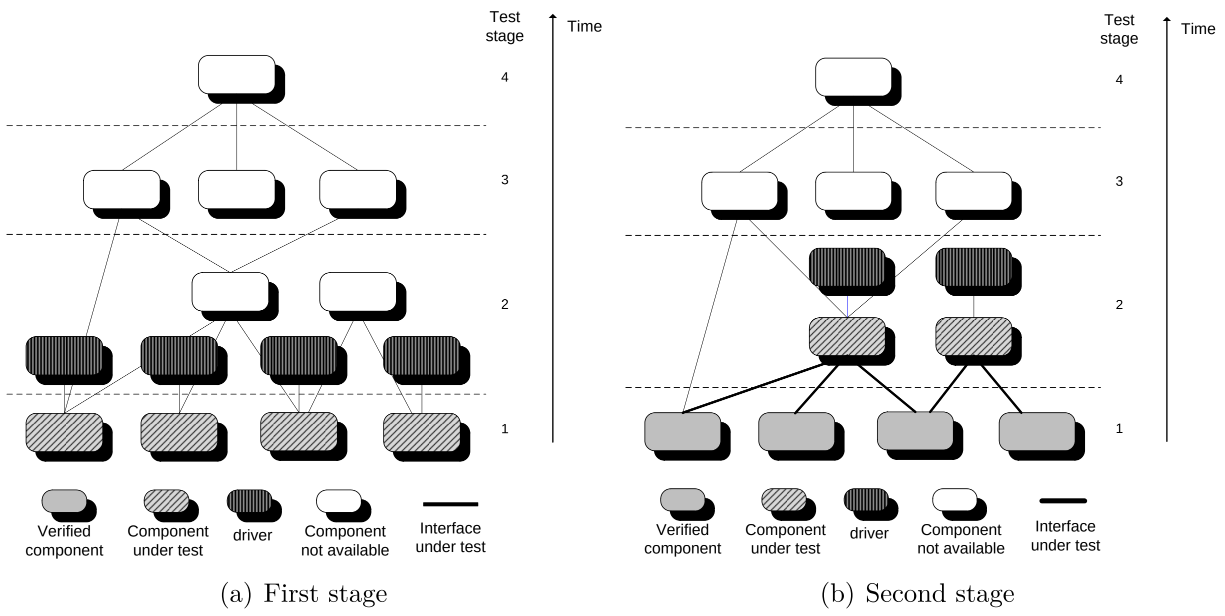
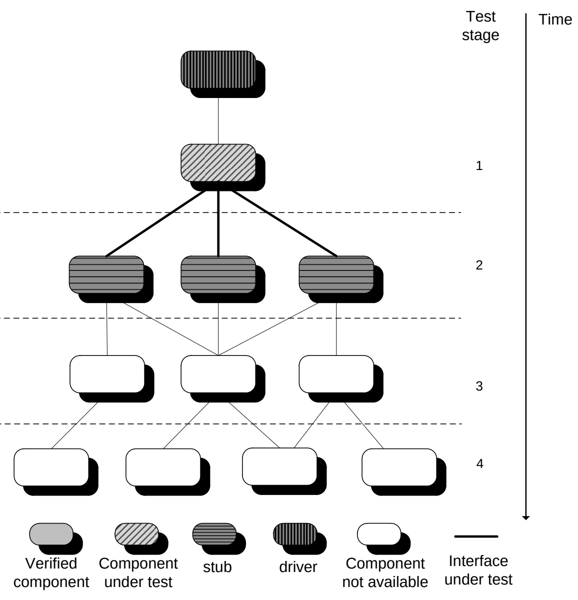


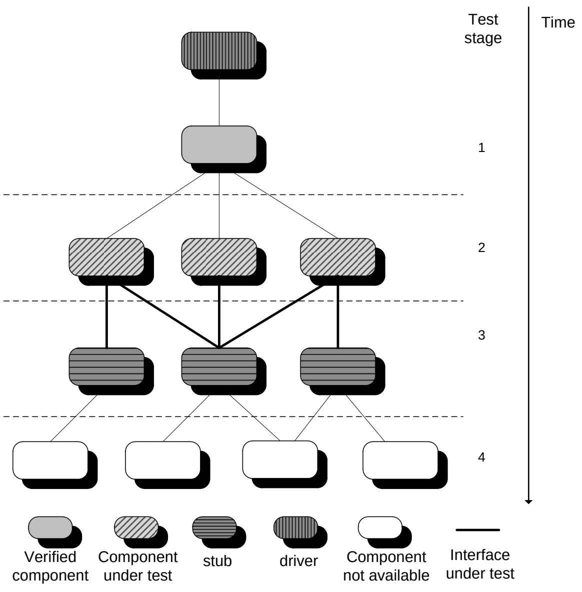
Figure 2.4: Bottom-up integration in the first and second stage

of bottom-up and top-down integration, *outside in integration* can be used. The integration is started both from the hardware and from the user/environment interaction side simultaneously, an increasing number of stubs and test drivers are needed in this strategy.

Representative strategies which are based on a functional view of the system are *use case driven* , *test case driven*, *risk driven* and *schedule driven* integration. In test case driven integration, components associated to a specific test case are identified and integrated. A welcome side effect of this pattern is the generation of statistical data on test coverage. Use case driven integration means that the focus of the integration order is to complete designated features of the software system. In [25], risk driven integration is referred to a prioritization of components on a continuum of most critical to least critical to not critical. This is also known as *hardest first integration* or *critical module integration.* Weak interface specifications between components or runtime requirements of interconnected components imply high risks during integrations. Also modules which were hard to implement can be integrated at the beginning so that the time for fixing problems is as long as possible. In *schedule driven integration*, components are integrated according to a fixed release plan. Since the implementation of the components is also following this schedule the components are integrated as they become available (first come first serve). This strategy is widely used but could lead to a increasing number of testdrivers and stubs and increases the complexity



(a) first stage



(b) second stage

Figure 2.5: Top down integration in the first and second stage

of integration tests.

2.2.2 Formalized approaches

Formalized strategies are able to calculate a certain integration order based on a system model and at least one objective function. Many work has been done on this field in the context of object oriented software, often referred as inter-class integration test order (ICITO) or class integration test order (CITO) [42]. Approaches presented in literature differ in their objective function and if either graph-based algorithms or heuristic algorithm are used to solve it.

Almost all formal approaches presented in literature have the objective to minimize the number of stubs needed for the integartion. Since it is not always possible to construct a stub that is simpler than the component which should be replaced, stubs become error-prone and require not neglectable testing effort [44, 13]. Also automated stub generation is only possible for very simple stubs, and that kind of stubs consume not much effort anyway. Due to this, minimizing stubs could improve the development process.

Comprehensive work in this field has been proposed by Kung *et al.* in [42]. Their solution is based on a dependency graph, which is a directed graph $G(V, E)$ where V is a set of nodes representing the classes and E is a set of edges representing the dependencies between this classes. Kung *et al.* proposed that, if this graph G has no cycles, a topological sorting algorithm could be used to derive an integration order in which no stubs are needed. If cycles appear, they are removed by identifying strongly connected components (SCCs) and temporarily removing edges between related classes. Those edges are identified by an association relation between the related classes, since this relation represents the weakest coupling between the two related classes of three kinds: inheritance, aggregation and association [65, 42]. If more than one association is a possible edge to remove a cycle, a random selection is performed. For each removed edge a stub is needed.

In [70], Tai and Daniels stated that *finding a minimum set of edges in a cyclic digraph for deletion in order to produce an acyclic digraph is an NP-complete problem and, furthermore, deleting a minimum number of association edges for breaking all cycles does not necessarily imply finding a test order that requires a minimum number of stubs.* In their approach, a *major* and a *minor*-level number is assigned to each class. Since the major-level number only takes inheritance and aggregation dependencies into account, there is no cycle and a topological sorting algorithm can be applied [13].The minor-level numbers are assigned in a second step based on association dependencies. In this step, cycles may appear. In order to identify the class that should be stubbed, a weight is assigned to each edge in a SSC. This weight is defined as the sum of incoming dependencies of the origin node and the number of outgoing dependencies of the target node. Tai and Daniels proposed to remove edges with higher values to break cycles, since they assumed that edges with higher value will break more cycles. However, as Briand *et al.* showed in [13], this must be not always true.

In [45], Vu Le Hanh *et al.* present a strategy called Triskell. The classes which participate in as many cycles as possible are stubbed first. If classes belong to the same number of cycles, the one which its incoming association edges belong to more cycles than the other is stubbed (cf.Kung). The next criterion, in case of equality, the class with the lower incoming degree in the cycle is stubbed, followed by an arbitrary order.

The approach of Le Traon *et al.* [46] uses the Tarjan algorithm [71] to identify components which belong to a cycle (SSCs). To identify the dependencies to remove, a weight for every node within a SSC is computed and the incoming edges of the node with the maximum weight are removed. The weight is defined as the sum of the number of incoming and outgoing front edges, which are edges going from a node to an ancestor. As Briand *et al.* [13] stated, this method is not deterministic in two ways. First, the weight depends on the starting edge of the search algorithm, second, there is no specification what class should be stubbed if there are more than one with the same weight.

Briand *et al.* [13] have improved Le Traons work by taking the kind of relationship between classes into account. They calculate the weight of each association dependency using a modified version of Tai and Daniels' definition, and then break the association dependency with the highest weight. In [73] Wang *et al.* pointed out that in contrast to Traon's approach Briand *et al.* 's approach does not break inheritance and aggregation edges and the weight is calculated more precisely as in Tai and Daniels solution.

In [50] Malloy *et al.* present a strategy that is driven by a parametrized cost model. First, SCCs are identified and then a weight for each edge in a SCC is calculated. In the third step, edges with the smallest weights are removed from the SSC. If no cycles are left, a topological sort is applied to derive the integration order. In order to determine the weights, edges are classified into six groups, association, composition, dependency, inheritance, owned element and polymorphic. Each group has a fix weight, based on their estimation of the cost of stub construction for untested classes. Inheritance edges *eg.* are assigned with a high weight, which makes it unlikely that an inheritance edge will be chosen to break cycle whereas association edges are assigned with a small weight.

All solutions presented so far have the objective to minimize the global number of stubs. Briand and Abdurazik ([13],[1]) stated that the effort for testing is hardly appreciable by this number only. They proposed to take the stub complexity into account, since for a specific stub it may be sufficient to return a simple fixed or random value but it may also be possible to return more specific values which require some algorithmic computations.

One of the first researchers that addresses this issue explicitly was Briand *et al.* in [11]. They used a genetic algorithm and coupling metric to try to break cycles by removing edges that will reduce the complexity of stub construction. A genetic algorithm (GA) is an optimization technique that has the ability to search for a global optimum and avoid getting stuck in a local optimum. The complexities of stubs are computed on the level of coupling they involve. Briand *et al.* as well as others stated that breaking compositions and inheritance relationships would likely lead to complex stubs, due to this they are considered as unbreakable. For remaining dependencies, coupling measure is done in two ways: The number of attributes A locally declare in the target class and the number of methods M locally declare in the target class. Based on this coupling measure, a cost function is defined and a GA is used to find a global minimum.

This approach is extended by Abdurazik and Offutt in [1]. Their solution uses a more detailed coupling measurement. By using UML diagrams, nine different types of coupling are identified: Association Coupling, Aggregation Coupling, Composition Coupling, Usage Dependency Call Coupling, Global Coupling, Inheritance Coupling, Interface Realization Coupling, External Coupling and Exception Coupling. For each coupling type, coupling measures are defined to measure the dependencies between two classes in terms of four attributes: (i) the number a of distinct variables used, (ii) the number of distinct methods called (including constructors), (iii) the number of parameters sent, and (iv) the number of return value types. This four measures are aggregated to one which is called test stub complexity. The authors distinguish between two kinds of stub complexity: specific test stub complexity and total test stub complexity. The reason for this is that certain methods or variables of a server class can be used by a number of clients in the same way. In this case, creating one stub for the server class can satisfy the needs of several clients. The node weight represents the total stub complexity of a class, and the edge weight represents the specific stub complexity of a server class to the client class that is connected by the edge. This coupling measure is added to the cost function defined by Briand *et al.* in [11] and a heuristic algorithm is presented to determine the nodes and edges with minimum total weight to remove so that there are no cycles and a topological sort could be applied. In contrast to Briand, Abdurazik and Offutt allow inheritance and composition relationships to be removed, but assign big constant weights as their stub complexity.

An improvement of this approach is done by Wang *et al.* in [73]. They stated that their solution reduces the complexity by 15.5% and increases the speed by 5.8 times in contrast to Abduraziks and Briands approach. To achieve this for each dependency relationship, its inter-class coupling information (ICCI) includes two aspects: (1) the number of distinct accessed attributes; (2) the number of distinct called methods (including constructors) [11]. This leads to a more conservative coupling measurement. A random interative algorithm is presented to minimize the stub complexity.

Additionally to the minimizing stubs objective, Borner *et al.* stated in [9] that the testability of a system could be improved when designated dependencies are selected and integrated a priori. More detailed, Borner proposed that several error-prone dependencies are identified by analyzing former versions of the software. These manually selected dependencies should be integrated in an early stage of the integration process in order to extend the time for testing. Therefore, components are divided into two groups. All components with dependencies marked as error-prone belong to the first group, component without such dependencies to the second one. The groups are disjoint, if components can be assigned to both groups, they should be assigned to the first group. The integration process starts with the first group. In order to derive an optimal integration order, the cost function provided by Briand in [11] is adopted and a simulated annealing algorithm is used to find a global optimum.

Another approach using a heuristic algorithm is presented by Cabral *et al.* in [18]. The authors formalized the class integration problem as a multi-objective optimization problem. The optimization functions are based on the same coupling measures used in the works of Briand *et al.* in [12] and [11], the attribute complexity and the method complexity. Also, following Briand *et al.* , inheritance and composition dependencies cannot be broken. Based on these measures and constraints, the problem is presented as a search for an integration order that minimizes two objectives, the method and the attribute complexities. To archive this, a multi-objective ant colony optimization algorithm is used. The authors stated that their approach presents better results in complex cases compares with the genetic algorithm used by Briand in [11].

Figure 2.6 sums up the presented approaches for deriving an integration strategy.

2.3 Discussion of presented approaches

The non-formalized approaches presented in section 2.2.1 are still state of the art when it comes to integration of embedded software. However, these are very informal guidelines and mainly motivated by testing issues. Furthermore they implicitly imply an acyclic system. If cycles appear it depends solely on integrators choice which component should be stubbed. As advantages of the *bottom-up integration strategy* the early examination of hardware-software interfaces and the simple test of error handling in case of faulty input values are mentioned, because they can be easy injected by test drivers. The advantages of the *top-down integration strategy* are accordingly the early examination of high level components and the simple test of error handling in case of faulty return values because they can be easily created by stubs.

The main drawback of these strategies is testing becomes difficult over multiple stages. In case of bottom-up it may difficult at higher stages getting low level components to return values necessary to get complete coverage respectively in case of top-down it may difficult to exercise low level components sufficiently because with increasing integration level it becomes more difficult to create proper test situations. In both cases a various number of stubs and test drivers are needed to get a proper test coverage. An approach which

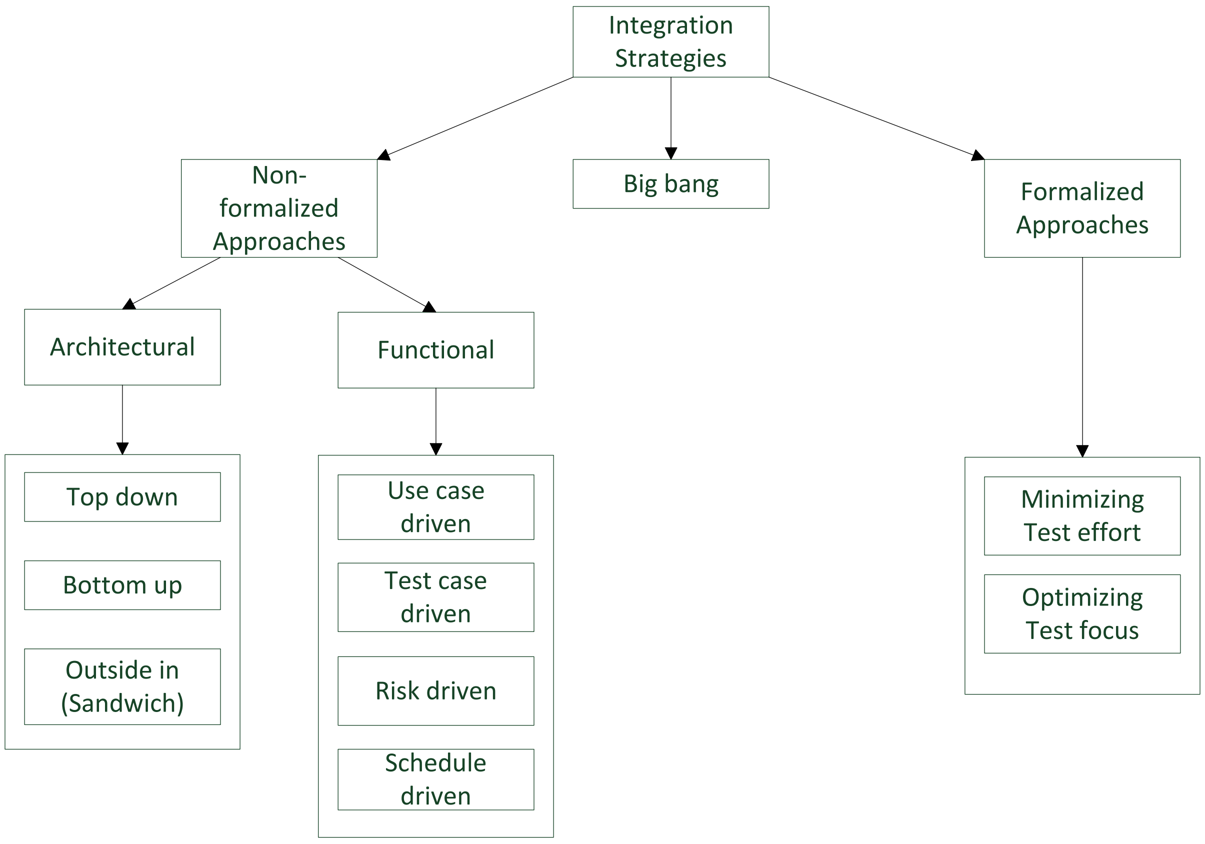


Figure 2.6: Classification of available integration strategies.

addresses this problem is the *Outside-in integration strategy* at the cost of an increasing number of stubs and test drivers needed for integration.

None of this approaches supplies a precise practice to determine an effective integration order, the selection of the next component to integrate depends heavily on the integrators experience.

The formalized approaches presented above supply a precise integration order with two main drawbacks. First they are mainly restricted to object oriented software and, second, they solely focus on minimizing stubs. The only exception here is the approach presented by Borner in [9], in his approach specific components or dependencies could be selected to be integrated a priori. Although the usage of object oriented software is proposed by many researchers (*eg.* [3], [69]), procedural languages , *eg.* the language C, are still used in hard real-time domains like automotive powertrain. Furthermore the need of test drivers is not considered in any available approach.

In all approaches presented in this work only a single component is allowed to be integrated at a single integration step. There may be the possibility to avoid stubs completely and to integrate all components which are part of a cycle in a single integration step. Such components could be identified *eg.* by Tarjan's algorithm and an optimized integration order in that way would integrate as less as possible components in a single step. However, this approach is not further considered due to several reasons. First, the number of components a cycle comprises is unpredictable. Reactive embedded systems usually contain several control loops and there might be a high number of components which have to be integrated in a single step. Furthermore, stubs and test drivers highly facilitate testing which is further discussed in Section 3.3. And, last but not least, this thesis deals with multiple objective for software integration therefore stepwise integration is a more feasible approach.

A survey was conducted by the Laboratory for Safe and Secure Systems (Regensburg University of Applied Sciences) among automotive tier 1 suppliers. The survey was published in [29] and [55]. Software development decision makers of twenty companies have answered questions about their software development process. The questions were especially aimed at their integration process and how they define the integration steps for their software projects. The survey was focused on the order of integration of the components and which problems appear during the integration process.

Figure 2.7 illustrates which integration strategies are most widely used. The survey shows that schedule-driven integration makes up the biggest part of the used integration strategies. This means that the integration order is given by the project schedule and no other constraints are considered. Schedule-driven integration is followed up by bottom up and feature driven integration. Bottom up as well as top down or inside out are mainly coarse guidelines for the integration. The selection of the next component to integrate depends heavily on the integrator's expertise. In this work, such strategies are called non-formalized. The feature driven integration strategy integrates components according to their allocation to product features. Individually considered, this is also only a coarse guideline because there is no prediction on how to integrate components within one feature. The risk driven strategy, which is used in approximately 30 percent of the projects, integrates critical components in a early stage of integration. Due to this the time to test such components is extended. Despite the described disadvantages, Big bang integration is still used commonly.

Figure 2.8 depicts common integration faults which are roughly divided into three groups: static component faults, dynamic component faults and methodical faults. Incompatible data type of components and semantic interface error belong to the static component faults. This is still a problem, despite the growing usage of architecture description languages like UML or AADL, where interfaces can be defined strictly.

Dynamic component faults like violation of timing constraint or overrun in hardware resources are also a problem at the integration stage.

The major part of errors are methodical errors. Methodical errors are errors which are not detected at the integration process but in other stages in the development process. The survey shows that one of the biggest cause for an error is a faulty component. This is by definition not an integration problem but rather a problem of the unit test. Also the faulty/missing documentation should not be handled during the integration process. Problems with stubs or wrong component version also belong to this group, but they play a minor part.

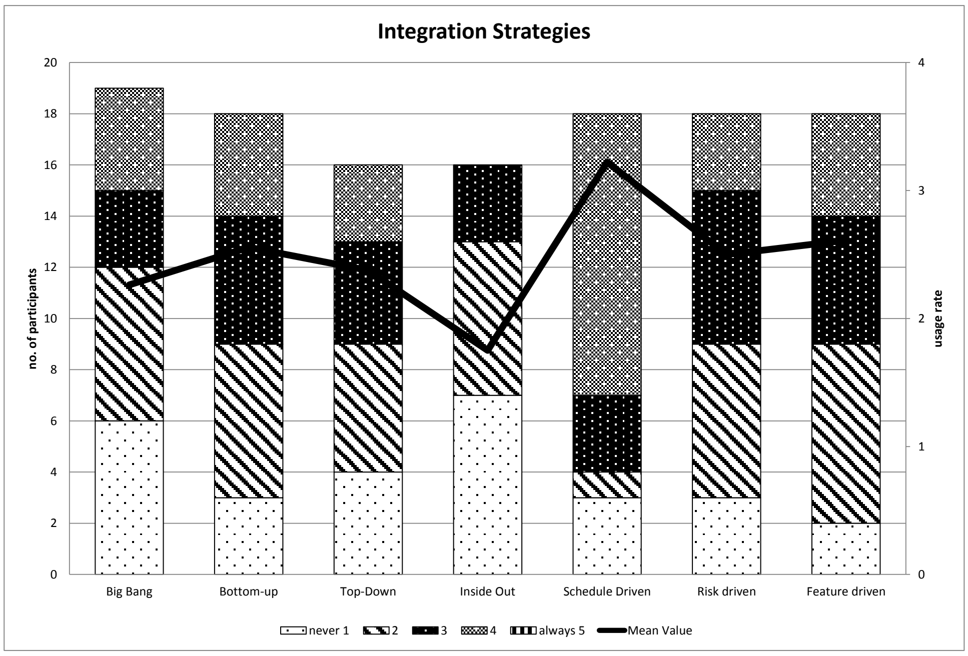


Figure 2.7: Results of the survey, conducted among software developers in the automotive industry([29],[55]). It shows the usage of common integration techniques.

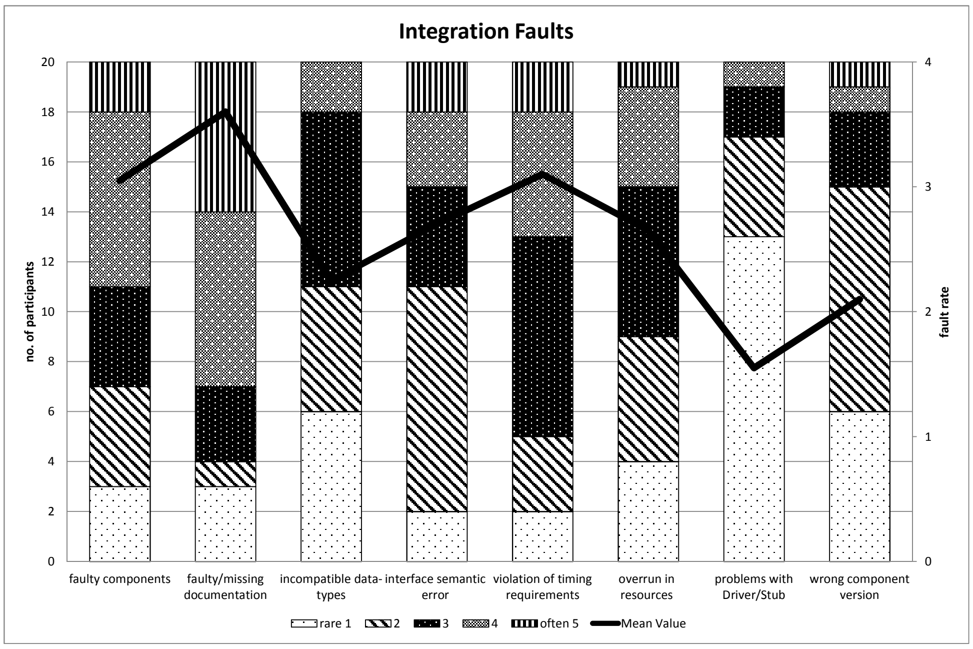


Figure 2.8: Results of the survey, conducted among software developers in the automotive industry ([29],[55]). It shows common integration faults.

2.4 Goals of the thesis

As described in the previous section, the embedded domain has to deal with an continuously growing complexity in their systems and thus systems a build up of individual components. However, no sophisticated methods are available in the embedded domain to determine an integration order which goes with a certain software project. The order of component integration often depends on integrators expertise or is determined by the first come, first serve principle. More elaborate strategies are tailored to object-oriented software and can not be directly used.

To address this challenges, this work pursues the following objectives:

- The first objective is to identify parameters the software integration is subjected to and to define metrics for each parameter in order to evaluate different integration order strategies.
- The second objective is to develop methods and algorithms for optimizing an integration order with respect to the defined parameters.
- The third objective is to validate the developed approaches on real life examples.

Chapter 3

Parameters of Software Integration

This chapter describes the parameters which affect the integration order. In the following, two real life systems are introduced and metrics for test effort, test complexity and planning effort are described and evaluated on these systems. The chapter concludes with a summary of the presented approaches.

3.1 Reference systems

In order to evaluate the proposed parameters, two reference systems are introduced. These real-life examples are taken from the automotive industry. The first one represents an embedded data logger for battery management and consists of 16 components and 23 dependencies. The dependency graph of this system is shown in Figure 3.1.

The second example is an automotive system with 72 components and 172 dependencies including cruise control, window winder and additional control functions. This system was extracted from an AUTOSAR model. These two real systems were used to evaluate the proposed approaches.

3.2 Test Effort

The most common used criteria for evaluating an integration order is called *test effort* and describes the effort for creating stubs needed during integration testing. There are several approaches presented in literature to compute the test effort. The most obvious method was proposed by the authors in [70] and [42], who suggested to simply count the number of classes that need to be stubbed. Briand and Abdurazik ([13],[1]) stated that the effort

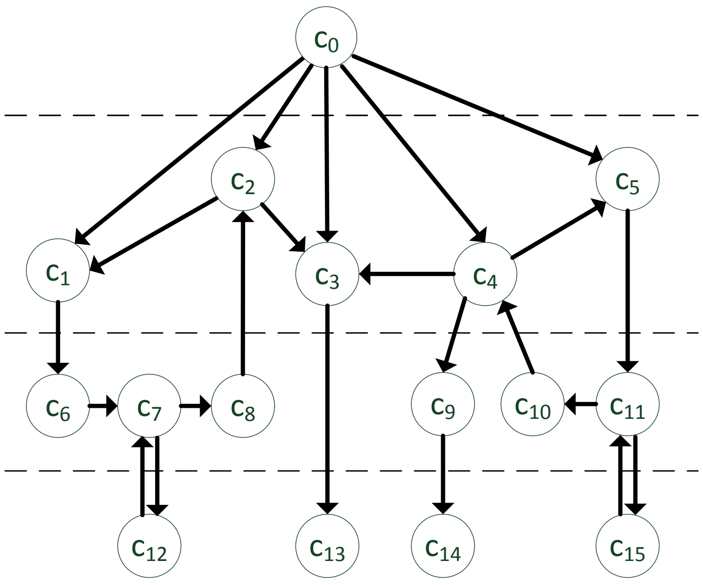


Figure 3.1: Dependency graph of the embedded data logger reference system with 16 components and 26 dependencies.

for testing is hardly appreciable by this number only, since this method assumes that all stubs a equally difficult to create. Therefore they propose to take the stub complexity into account, which is computed by the the number of attributes and the number of methods locally declare in the target class. Another method for estimating the test effort was proposed by Malloy *et al.* in [50]. The use a parameterized cost model for the estimation of the cost of stub construction. This model defines six types of edges in an object-oriented system and adds a constant weight to type of edge which is based on their estimation. This approach was extended by Abdurazik and Offutt in [1], who defines nine typed of edges based on explicit and implicit object-oriented class relationships. For each coupling type, coupling measures are defined to measure the dependencies between classes in terms of four attributes. However, all this metrics are specially tailored to object-oriented software and have a limited expressiveness in the context of non-object-oriented software. In this work, three different approaches for measuring the test effort are proposed. First, simply the number of dependencies and the number of components to be stubbed in a certain integration order are used. Referring to LeTraon *et al.* [46], the terms *specific stub* for a stubbed dependency and *realistic stub* for a stubbed component are used. This most general metric is used to compare available approaches which are ported to non-object oriented software. Next, the number of test drivers are taken into account additionally. As already mentioned, test drivers are also an important part in software integration and need effort for creation. Furthermore, the stub complexity is considered.

3.2.1 Adaptability of existing approaches to C-written component-based embedded software

This section describes the adaptability of existing approaches to C-written component-based embedded software. Note that not every solution is portable to C-written software since they rely strongly on object-oriented concepts.

3.2.1.1 Approach by Kung *et al.*

In Kung *et al.* strategy [42], those edges are identified by an association relation between the related classes in a strongly connected component (SSC), since this relation represents the weakest coupling between the two related classes of three kinds: inheritance, aggregation and association [65]. If more than one association is a possible edge to remove a cycle, a random selection is performed. Since there are more enhanced strategies for edge selection than do it randomly, the approach by Kung *et al.* is not further considered.

3.2.1.2 Approach by Tai & Daniels

In their approach [70], a *major* and a *minor*-level number is assigned to each class. Since the major-level number only takes inheritance and aggregation dependencies into account, there is no cycle and a topological sorting algorithm could be applied [13]. The minor-level numbers are assigned in a second step based on association dependencies. In this step, cycles may appear. In order to identify the class that should be stubbed, a weight is assigned to each edge in a SSC. This weight is defined as the sum of incoming dependencies of the origin node and the number of outgoing dependencies of the target node. In the context of non-object-oriented software, every edge must be seen as an association relation which is possible to break, since there are no different types of dependencies. Thus only the minor-level numbers are relevant since major-level numbers are assigned to inheritance and aggregation dependencies. This leads to only one major-level and Tai and Daniels strategy could be used in the following way:

- identify strongly connected components (SCC)

- assign a $weight(d_i)$ to each edge in a SSC
- incrementally remove edges with highest weight until there are no more cycles

Tai and Daniels stated that the number of classes with the same level number is usually small. However, due to case-studies, this may not be always true in a non-object-oriented embedded system. Since in this work only one component is integrated at a single step, Tai and Daniels suggestion to integrate all components which are connected by dependencies with the same weight in a single step is not useful. Therefore, a random selection between dependencies with the same weight is performed here.

3.2.1.3 Triskell strategy

The strategy proposed by Vu Le Hanh *et al.* in in [45] is a two-part strategy. Since resource allocation is not considered here, only the first part is used. This method does not take into account the type of relationship between components as a priority. To break a cycle, the vertex that participates in as many cycles as possible is stubbed, if classes belong to the same number of cycles the one which its incoming association edges belong to more cycles than the other is stubbed (cf.Kung). In case of non-object-oriented software, this criterion must be generalized to incoming edges. The next criterion, in case of equality, the class with the lower incoming degree in the cycle is stubbed, followed by an arbitrary order.

- progressively stub vertices, which participate in as many cycles as possible
- when two vertices belong to the same number of cycles, stub the vertex which its incoming edges belong to more cycles than the other
- in case of equality the next criterion selects the vertex with the lower incoming degree, followed by an arbitrary selection

3.2.1.4 Approach by Le Traon *et al.*

The approach of Le Traon *et al.* [46] uses the Tarjan algorithm [71] to identify components which belong to a cycle (SSCs). To identify the dependencies to remove, a weight for every node within a SSC is computed and the incoming edges of the node with the maximum weight are removed. The weight is defined as the sum of the number of incoming and outgoing front edges, which are edges going from a node to an ancestor. In the context of non-object-oriented Le Traon *et al.* strategy could be used in the following way:

- identify strongly connected components (SCC) by using Tarjans algorithm
- assign a $weight(d_i)$ to each node in a SSC
- remove incoming edge of the node with the maximum weight
- recursively call the procedure above for every nontrivial SSC (more than one vertex)

Since there is no specification what class should be stubbed if there are more than one with the same weight, a random selection is performed.

3.2.1.5 Approach by Briand *et al.*

Briand *et al.* [13] have improved Le Traons work by taking the kind of relationship between classes into account. They calculate the weight of each association dependency using a modified version of Tai and Daniels' definition, and then break the association dependency with the highest weight. In [73] Wang *et al.* pointed out that in contrast to Traon's approach Briand *et al.* 's approach does not break inheritance and aggregation edges and the weight is calculated more precisely as in Tai and Daniels solution. Since there are no different types of dependencies in non-object-oriented software, Briand *et al.* approach differs from Tai and Daniels in this case of application only in the definition of the weight-function. Therefore, it could be applied in a similar way.

- identify strongly connected components (SCC) by using Tarjan algorithm
- assign a $weight(d_i)$ to each edge in a nontrivial SSC
- remove the edge with the maximum weight
- recursively call the procedure above for every nontrivial SSC (more than one vertex)

3.2.1.6 Approach by Malloy *et al.*

In the strategy presented by Malloy *et al.* in [50], edges are classified into six groups, association, composition, dependency, inheritance, owned element and polymorphic. Since this classification makes only sense in an object-oriented software, this strategy was not ported.

3.2.1.7 Heuristic Approaches

Since minimizing stubs at software integration is a NP-complete problem [70], heuristic approaches have been studied. One of the first researchers using such algorithms was Briand *et al.* in [11]. They used a genetic algorithm (GA) and coupling metric to try to break cycles by removing edges that will reduce the complexity of stub construction. After cycles are broken the top-sort algorithm is applied to derive the test order. The complexities of stubs are measured in two ways: The number of attributes locally declare in the target class and the number of methods locally declare in the target class. Based on this coupling measure, a cost function is defined and a GA is used to find a global minimum. All available heuristic solutions (*eg.* [1],[11],[9]) use this weigh function with some individual extensions. Since this weight-function is specially tailored to object-oriented software, solutions based on this function could not be applied to non-object-oriented software without major modifications.

3.2.1.8 Results

The ported approaches described in the previous section were applied to the two reference system presented in Section 3.1. The needed number of realistic stubs (stubbed components) and specific stubs (stubbed dependencies) in each case are presented in the Tables 3.1 and 3.2. The best result in each case is bold-faced.

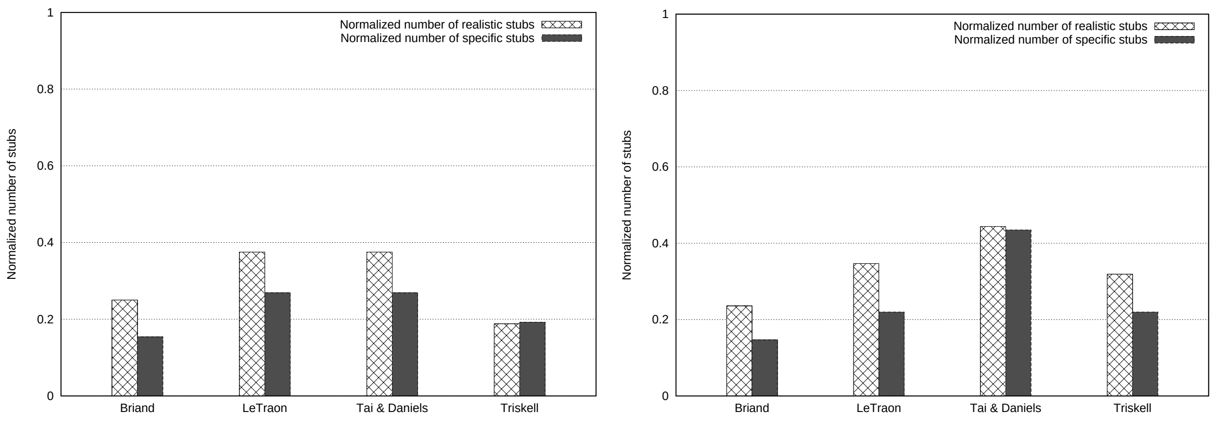
	Briand	LeTraon	Tai & Daniels	Triskell
realistic stubs	4	6	6	**3**
specific stubs	**4**	7	7	5

Table 3.1: Number of realistic and specific stubs needed for the integration of the data logger system with 16 components and 26 dependencies (Figure 3.1).

	Briand	LeTraon	Tai & Daniels	Triskell
realistic stubs	**17**	25	32	23
specific stubs	**26**	39	77	39

Table 3.2: Number of realistic and specific stubs needed for the integration of the AUTOSAR system with 72 components and 177 dependencies.

For a better comparability in the graphical representation, the results are normalized with respect to the size of the system (Figure 3.2(a) and 3.2(b)). In detail, the normalization is done by dividing the actual number of stubs by the maximum possible number of stubs. The maximum possible number of stubs is given by the number of components in the system for realistic stubs and the number of dependencies for specific stubs respectively.



(a) Data logger (b) AUTOSAR system

Figure 3.2: Normalized number of stubs needed for integration

The results indicate that the approach presented from Briand *et al.* and the Triskell strategy presented by Vu Le Hanh *et al.* obtain favorable results in terms of non-object oriented software. Since the Triskell strategy temporarily removes components in order to remove cycles whereas Briands strategy removes dependencies, Triskell may lead to a lower number of realistic stubs whereas Briands approach may result in a lower number of specific stubs. In case of the approaches presented by Tai&Daniels and Le Traon, a significantly number of classes, and dependencies respectively, with the same weight appear. Since a random selection is performed in this case, less good result are obtained by this approaches.

3.2.2 Considering test drivers

The number of test driver which are needed for integration testing is unattended in all available approaches. However, since test drivers are also involved during testing embedded software, they must be also considered. Therefore, the test effort E_{TE} is defined as the number of real stubs N_{ST} plus the number of test drivers N_{TD} which are needed for integration. For a better comparability of systems with different size, this number is normalized. To achieve this, the number of stubs plus the number of test drivers is divided by twice the number of components C since in the worst case each components must be replaced by a stub and a test driver. Equation 3.1 depicts this relationship.

$$E_{TE} = \frac{N_{ST} + N_{TD}}{2C} \tag{3.1}$$

3.2.3 Considering stub complexity

Briand and Abdurazik ([13],[1]) stated that the effort for testing is hardly appreciable by the number of stubs only. They proposed to take the stub complexity into account, since for a stub it may be sufficient to return a simple fixed or random value but it may also be possible to return more specific values which require some algorithmic computations (c.f. Section 2.2.2). However, to take the number of variables locally declared in the target component, as proposed by Briand *et al.* in [11] for object oriented software, is not reasonable for embedded systems since many variables here are globally defined. Also to take the kind of dependency as measurement, as proposed by Abdurazik and Offutt in [1], is only applicable in object-oriented software. An approach for estimating the stub complexity in non-object-oriented software is the usage of requirements based values, as *eg.* function points. Function points were defined by Albrecht in [2] and are a reliable and proven solution to estimate the size of a component. Another approach, which is used in this work, is to use the number of interfaces of a stub in order to determine its complexity. Note that the number of interfaces of a component must not be equal to the number of its dependencies, because multiple interfaces may be served from another individual component which would result in a single dependency. Since a double number of interfaces will produce more than the double complexity, because interactions between these interfaces must be also taken into account, the following polynomial is used to estimate the stub complexity.

The complexity v_i for a stub i is calculated as follows:

$$v_i = \alpha n + \beta n^2 \tag{3.2}$$

where n represents the number of interfaces of a stub i. The polynomial factors α and β are application specific and have to be determined according the intended application.

The overall stub complexity is defined as the sum of the complexities of all stubs.

$$E_{SC} = \sum v_i \tag{3.3}$$

3.3 Integration test complexity (ITC)

Another important criteria for an integration order is testing. Integration testing is one of the time-consuming parts of software development, it requires up to 40% of total system development cost and effort [59]. In [74], Watson stated that a sophisticated integration strategy highly facilitates testing and reduces effort, therefore integration testing is strongly connected to the used integration strategy, which determines the order in which components are integrated. In fact, state-of-art integration strategies are mainly motivated by testing issues. As already mentioned, advantages of the *bottom-up integration strategy* are the early examination of hardware-software interfaces and the simple test of error handling in case of faulty input values, because they can be easily injected by test drivers. The advantages of the *top-down integration strategy* are accordingly the early examination of high level components and the simple test of error handling in case of faulty return values because they can be easily created by stubs. However, these approaches are very informal guidelines relying strongly on integrators expertise. The main drawback of both strategies is that testing becomes difficult over multiple stages. In case of *bottom-up* it may difficult at higher stages getting low level components to return values necessary to get complete coverage. Respectively, in case of top-down it may difficult to exercise low level components sufficiently because with increasing integration level it becomes more difficult to create proper test situations [5]. In both cases a various number of additional stubs and test drivers are needed to obtain a proper test coverage. The formalized approaches, which are mainly aimed to minimize the test effort (described in Section 3.2), do not consider testing over several stages. They do not even take test drivers into account, which are needed to stimulate the system and increase the test effort in general.

In this section, the integration test complexity of a system related to its integration order is analyzed. Therefore general issues of integration testing are presented and a novel metric for measuring the integration test complexity is presented.

3.3.1 General issues of integration testing

In general, the integration test focuses on testing the interaction of components assuming that the details within each module are accurate (verified by module test). Figure 3.3 depicts integration testing in the V development process for component based embedded systems.

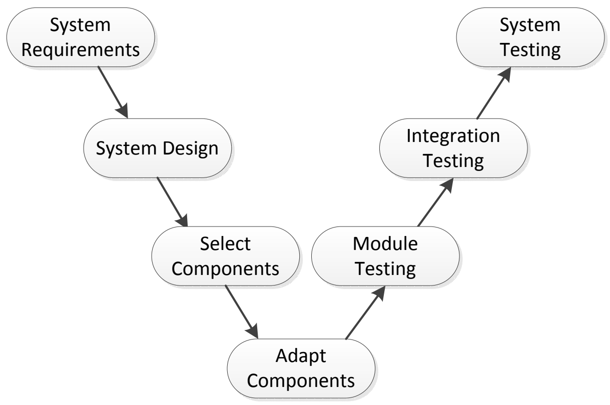


Figure 3.3: V development process for component based embedded systems [23].

In [27], the authors state that an interface can be fully describe by the following characteristic [66]:

- Interface signature (name, data type, return type)
- Interface direction
- Synchronous/asynchronous interface
- Physical range
- Error return values
- Availability of diagnosis mechanism
- Pre- and postconditions
- Invariants
- Required refresh cycle of parameters
- Required response time

- Reentrant functionality
- Synchronization structure
- Dynamic behavior
- Required hardware

Based on the results from Wu *et al.* [75] and Jung *et al.* [35], integration testing may contain the following checks:

- Check if the correct methods/functions are called in the the designated components.
- Check if type inconsistencies (*eg.* 32-bit integer or 16-bit integer) exist.
- Check if numerical inconsistencies (*eg.* prefix ”Mega” means 1,000,000 or 1,048,576) exist.
- Check if physical inconsistencies (*eg.* metric or imperial units) exist.
- Check if global constraints (*eg.* timing constraints, hardware constraints) are violated.

3.3.2 Modeling integration test complexity

Integration test complexity can be interpreted as the *relative ease and expensive of revealing software integration faults* [7] and generally based on the increasing test complexity with increasing integration level. In order to introduce the proposed metric for integration test complexity, a small real life system based on the AUTOSAR Function Inhibition Manager (FIM) is used. In Figure 3.4, the dependency graph of four involved components, the ECU State Manager (EcuM), the Diagnostic Event Manager (Dem), the Function Inhibition Manager (Fim) and the the Diagnostic Communication Manager (Dcm), is shown. The sequence diagram in Figure 3.5 depicts the initialization of the Function Inhibition Manager.

3.3.2.1 Test complexity of a dependency

Integration testing actually means to test the dependencies between the components. Therefore, integration test complexity depends on the complexity of a certain dependency.

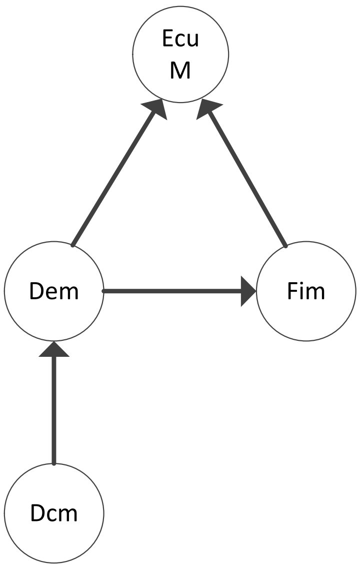


Figure 3.4: Dependency graph of four involved components, the ECU State Manager (EcuM), the Diagnostic Event Manager (Dem), the Function Inhibition Manager (Fim) and the the Diagnostic Communication Manager (Dcm)

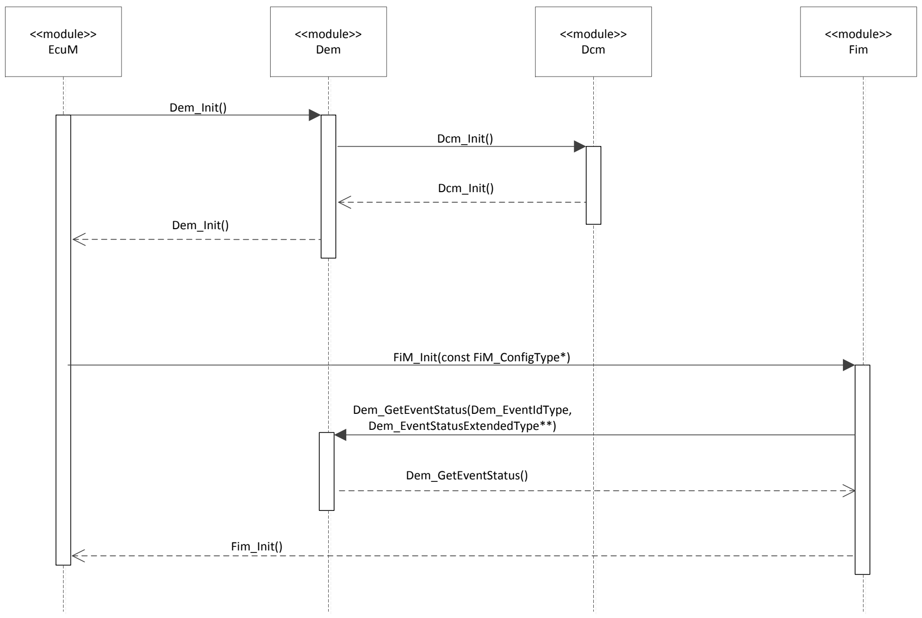


Figure 3.5: Sequence diagram of the initialization of the Function Inhibition Manager (Fim).

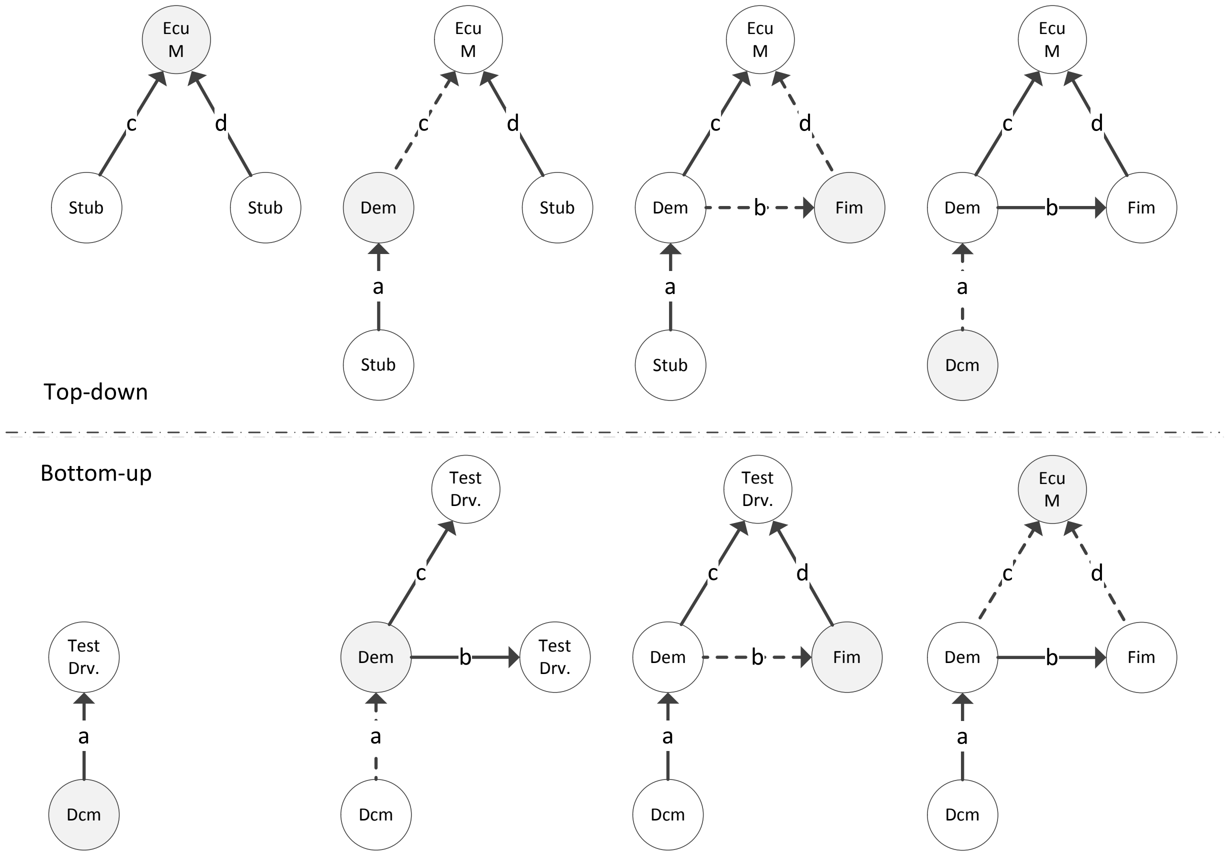


Figure 3.6: Top-down and bottom-up integration of the Function Inhibition Manager. Components are added stepwise, the new one in each step is gray shaded. The interfaces under test (IUT) at each step are shown with broken lines.

For illustration, two possible integration orders of the system presented in Figure 3.4 are discussed.

In Figure 3.6, bottom-up and top-down integration of the Function Inhibition Manager is shown. Note that top-down integration needs no test drivers, whereas bottom-up integration goes without stubs. In the following, the test of the interaction between the Diagnostic Event Manager (Dem) and the Function Inhibition Manager (Fim) will be examined closer. This dependency is marked with (b) in Figure 3.6.

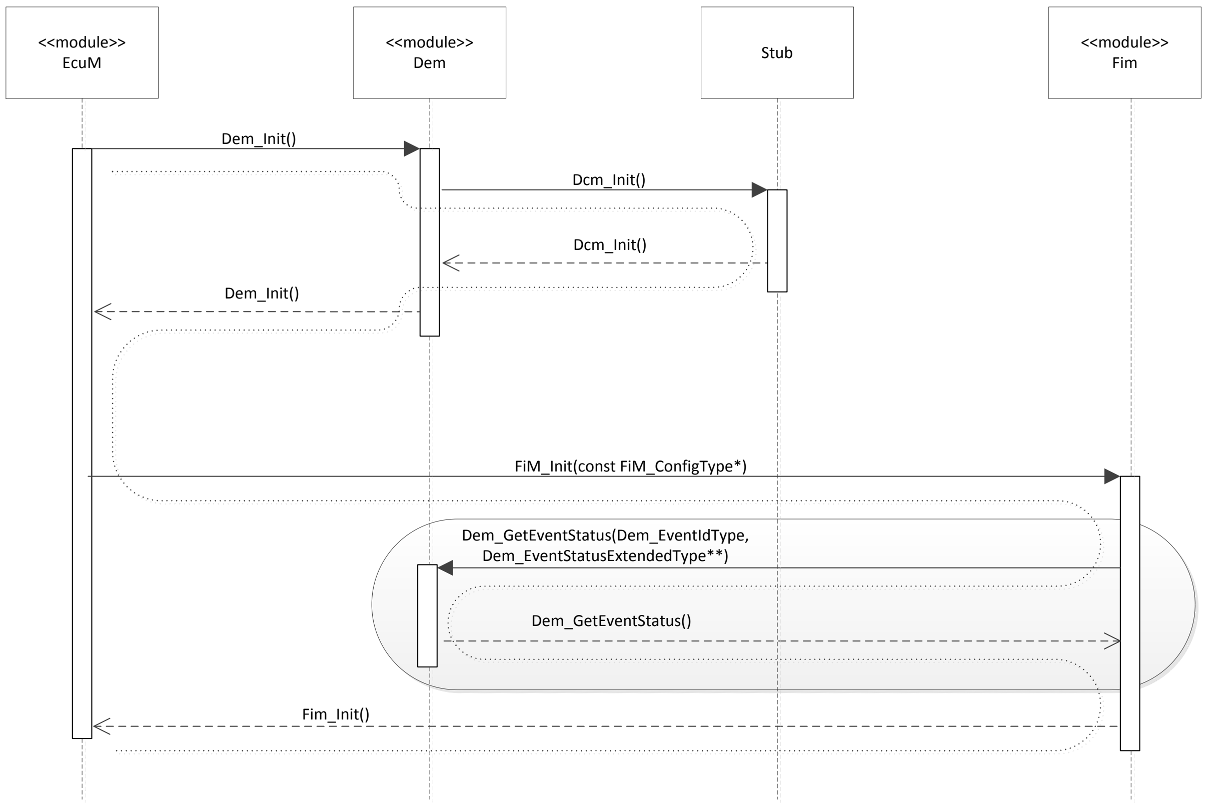


Figure 3.7: Test of the gray shaded dependency in case of top-down integration. Test data (represented by the dotted line) test data must be passed through the EcuM and the Dem.

Figure 3.7 and 3.8 depict the test of this dependency in bottom-up and top-down integration respectively. The dotted line in each figure represents the test data path. In case of top-down integration (Figure 3.7), test data must be passed through the EcuM and the Dem. This makes testing more complex, since *eg.* the stimulation of the Fim with wrong data in order to test its error handler is hardly possible since a possibly existing error handler in the EcuM avoid this. In case of bottom-up integration (Figure 3.8), testing this

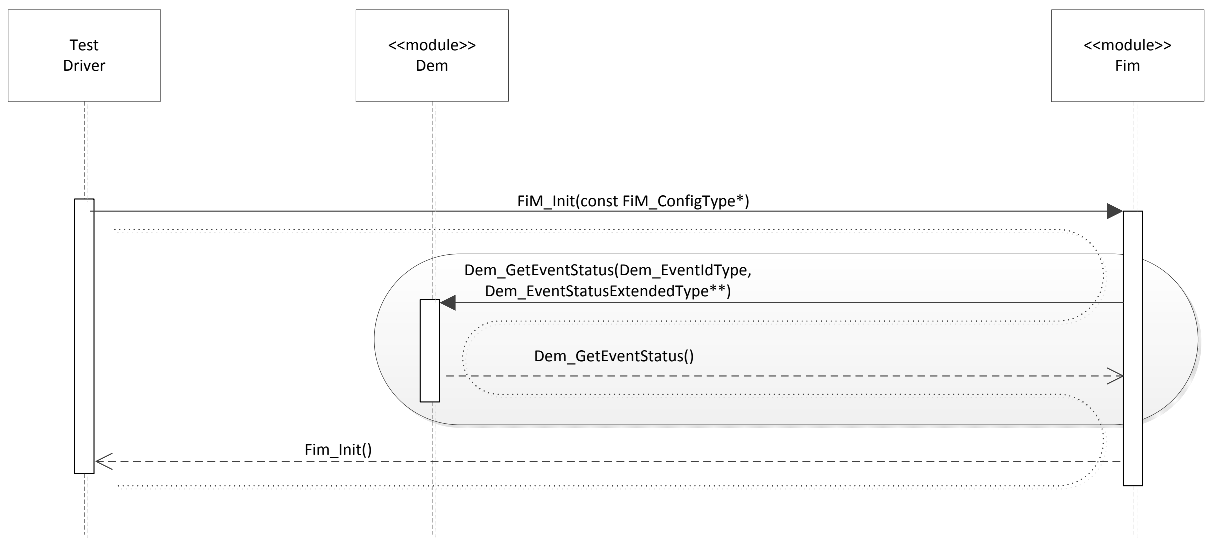

Test Driver
<<module>> Dem
<<module>> Fim
FiM_Init(const FiM_ConfigType*)
Dem_GetEventStatus(Dem_EventIdType, Dem_EventStatusExtendedType**)
Dem_GetEventStatus()
Fim_Init()

Figure 3.8: Test of the gray shaded dependency in case of bottom-up integration.Test data (dotted line) can be directly applied to the interface under test.

dependency becomes more easy since test data can be easily injected by the test driver and the results can be observed. Therefore, roughly speaking, the less test drivers and stubs are involved when testing a certain dependency, the more complex testing becomes. In the following, a formal definition of this relationship is given.

The system is described as a directed graph $G(V, E)$ where V is a set of nodes representing the components and E is a set of edges representing the dependencies between components. The number of components which are potentially involved during the test of an edge $E = \{v_x, v_y\}$ can be calculated by the number of edges incident to v_x, denoted as $d_G(v_x)$) plus the number of edges incident to v_y, denoted as $d_G(v_y)$. As already mentioned, testing can be improved if not all components incident with v_x or v_y are fully implemented (denoted as *real* components) but rather stubs or test drivers. Therefore, the test complexity of $E = \{v_x, v_y\}$ is defined as the quotient of real components incident to v_x and v_y and the number of all components incident to v_x and v_y. Note that at least two components must be real, the start node and the end node of the edge under test. Equation 3.4 depicts this relationship.

$$c_{(x,y)} = \frac{d_G(v_x)(real) + d_G(v_y)(real) - 2}{d_G(v_x) + d_G(v_y) - 2} \tag{3.4}$$

The scalar $c_{x,y}$ describes the test complexity of the dependency between the components C_x and C_y. Note that the test complexity of a dependency without any stubs or test drivers (worst case) is 1, the best case is 0.

Example: In order to exemplify the integration test complexity of a dependency, the complexity of the dependency (b) of the Function Inhibition Manager (Figure 3.6) for top-down and bottom-up integration is calculated.

- **Top-down:** In case of top-down integration, 3 components are incident with the Dem, 2 of them are fully implemented (real), one is a stub. Furthermore, 2 components are incident with Fim, both are real. Therefore, the test complexity can be calculated as follows:

  $$c_b = \frac{2+2-2}{3+2-2} = 0.66$$

- **Bottom-up:** In case of bottom-up integration, also 3 components are incident with the Dem and 2 of them are fully implemented (real) and one is a test driver. However, 2 components are incident with Fim, but only one is real. Therefore, the test complexity can be calculated as follows:

  $$c_b = \frac{2+1-2}{3+2-2} = 0.33$$

As for the result, the complexity of testing this dependency is less complex in case of bottom-up integration compared with top-down. Note that the test effort is equal in both cases.

3.3.2.2 Test complexity of the complete system

Since integration testing in general means testing of more than one dependency, the integration test complexity E_{TC} is introduced. Integration test complexity describes the test complexity of a complete system and is defined as the sum of all individual dependency test complexities divided by the number of dependencies in the system. Equation 3.5 depicts this relationship.

$$E_{TC} = \frac{\sum c_{x,y}}{|E|} \tag{3.5}$$

Example: In order to exemplify the test complexity of a complete system, the integration test complexity of the Function Inhibition Manager (Figure 3.6) for top-down and bottom-up integration is calculated.

- **Top-down:**

$$E_{TC} = \frac{c_a + c_b + c_c + c_d}{4} = \frac{1 + 0.66 + 0 + 1}{4} = 0.665$$

- **Bottom-up:**

$$E_{TC} = \frac{c_a + c_b + c_c + c_d}{4} = \frac{0 + 0.33 + 1 + 1}{4} = 0.583$$

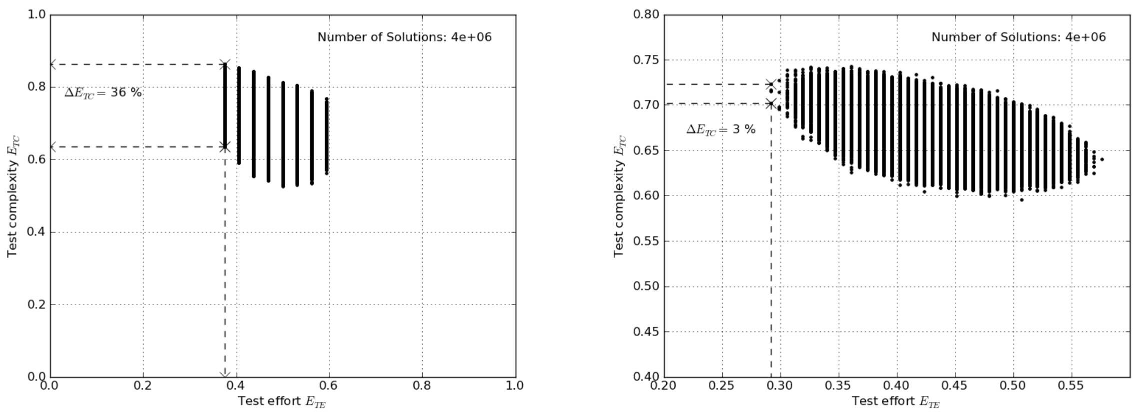
As a result, bottom-up integration provides less test complexity compared with top-down integration with equal test effort.

3.3.3 Test complexity versus Test effort

Based on the considerations above, integration testing has three key questions:

1. How many stubs are needed during integration testing?
2. How many test drivers are needed during integration testing?
3. How complex integration testing becomes?

Question 1 and 2 are questions of test effort. Each stub and each test driver must be constructed and increases the effort for testing. On the other hand, each additional stub or test driver reduces the test complexity of the system. Therefore, test complexity and test effort are contrary goals. In order to illustrate this relation ship, a sample of 4×10^6 possible integration orders are taken from the two reference systems presented in Section 3.1. For each integration order, The test effort E_{TE} (including test drivers, as defined in Section 3.2.2) and the integration test complexity E_{TC} was calculated. Figure 3.9 depicts the test effort E_{TE} against the integration test complexity E_{TC} for both systems.



(a) Data logger (b) AUTOSAR system

Figure 3.9: Test effort against test complexity of 4×10^6 possible integration orders.

The results verify the assumption that improving the test complexity and minimizing the test effort are contrary goals. On the other hand, in case of the data logger system, the test complexity differs over 30% at minimal test effort. This means that the test complexity of this system can be reduced up to 30% without additional effort for stubs and test drivers if a more ellaborate integartion strategy is choosen. In case of the AUTOSAR system only 3% are possible, on the other hand, the test complexity of this system can be reduced significantely with only a small increase of the test effort.

3.4 Integration schedule

Often the project schedule is the crucial factor in the software integration strategy. There are several reasons why a component shall be integrated at a certain point in time. For example, a component is simply not available till then or it should be integrated as early as possible in order to extend time for testing. In addition, components may have deadlines till they shall be integrated because some individual functionality must be available then. Determining an integration order manually is hardly possible in large scale embedded systems since timing constraints are often conflicting and compromises must be found. The following section provides an overview of schedule requirements and presents a metric to evaluate an integration order against these requirements.

3.4.1 Component/Resource release time and deadline

In automotive industries, the software development process is often based on the iterative V-Model process. In this process, a software product is partitioned in several releases, which are delivered at specific points in time during the product development. In every release a specific set of features is implemented. The components, which are associated to each product feature, are integrated in every iteration to form the entire release of the software product. This process, illustrated in Figure 3.10, leads to several individual schedules which must be met throughout the software development process. Since every release is associated to a specific date, an individual deadline for each component can be derived from the date of the release this specific component belongs to. Obviously, if a component belongs to more than a single feature, the deadline is determined by the earlier release. Furthermore, components are not arbitrary available. Since modern complex automotive software systems cover many physical domains, *eg.* combustion process, communication or fluid mechanics, the developers of specific components are not interchangeable as a general rule. This restricts the number of developers which could be assigned to a specific component and the development of components could not be parallelized at will. This leads to the following constraints: First, each components has a date when it became available (release time). If a specific component should be integrated before this date, additional effort must be spend *eg.* to speedup component development or creating a stub. Second, each component has a deadline when it must be integrated. Violating the deadline could lead to serious problems *eg.* reduced functionality or even contract penalties.

Referring to software integration, also the availability of resources is strongly related to the availability of components. Since integration includes testing, human as well as technical resources must be available when a certain component should be integrated. As already mentioned, modern complex software systems cover many physical domains. Therefore, the integration and integration testing of a certain component may only be done by a specialist corresponding to the physical domain on reasonable terms. Those specialists are not arbitrarily available, therefore the availability of certain developers must be considered during integration planning. Additionally, often physical test benches are involved during integration testing, *eg.* engine test benches or hardware-in-the-loop systems. Since this test equipment may be used be more than one development department, the usage of such technical resources must be coordinated.

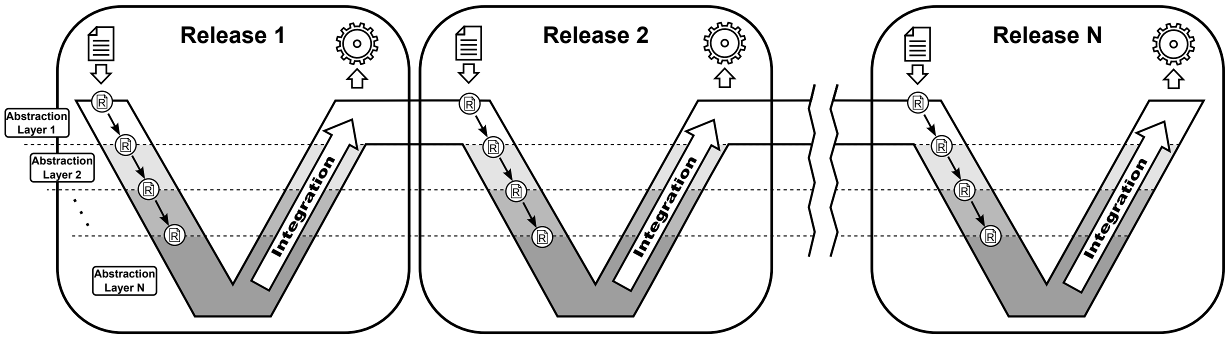


Figure 3.10: Iterative V-Model Process

3.4.2 Risk, Criticality

A special case in integration scheduling occurs if certain components have a high potential to fail. As Borner showed in [9], the integration process could be improved if critical components are integrated in an early state of the process since an early integration extends the time for testing. Critical components are *eg.* components which tend to be error-prone or are located in vital positions in the software architecture. If such a vital component fail, big parts of the entire software system are affected. Developers of software components are facing several risks that the component will not fulfill its desired function, *eg.* timing risks, system functionality risks or performance risks [64]. Various approaches are available how to gather information about how stable a software component is. A very expressive metric is the reuse factor. If a component is already used in a number of projects, there is a relatively modest risk to fail in a new project. Furthermore, there are several statistical methods available to identify error-prone components, *eg.* [4], [62], [57]. Also classic software metrics as *eg.* Cyclomatic Complexity [52] or the Halstead-Metric [31] can be used to determine the risk of a certain component. Moreover, in [56], the authors define several metrics to calculate the criticality of software components, which can be used to determine the risk of a certain component.

In order to extend time for testing and fixing problems an early deadline can be assigned to error-prone components. Thus, such components are scheduled at the beginning of the integration process.

3.4.3 Modeling planning effort for a software integration order

An obvious solution to automatically derive an integration schedule is the usage of a scheduling algorithm. For example, the problem of deriving an integration schedule can be formulated as a resource-constrained project scheduling problem (RCPSP) [10]. At this, software integration consists of $n+1$ activities (integration steps) where each activity has to be processed in order to complete the project. Dependency constraints between components may be handled as precedence constraints (force integration of a component k not to be started before all necessary components k depends on are integrated) and performing the activities requires resources with limited availability (test benches, integrators, *etc*).
However, several problems arise when using such an approach. First, a scheduling problem is represented by an acyclic graph, a software system is acyclic only in the rarest of cases. Although finding a minimum number of edges to delete to get an acyclic graph is a NP-hard problem, there are solutions for this problem at least for object-oriented software ([70], [13]). Also integrating all components of a cycle in a single step may be possible. However, an integration schedule may not be feasible. They may be conflicting parts during the integration, but in contrast to *eg.* assembling a machine, the integration order could be changed in certain cases with additional effort (*eg.* create more stubs, spend overtime, *etc*). This characteristic can be hardly modeled in a scheduling problem. Due to this, a novel approach for modeling the schedule effort for software integration is presented.

3.4.3.1 Scheduling parameters

Based on the concepts of job scheduling we use the following time notation. Time is relative and expressed in equidistant discrete time steps where a single point in a time scale is defined as:

$$t : t \in \mathbb{T}, t > 0 \tag{3.6}$$

The physical representation of this abstract time steps can be fitted to the development process requirements, *eg.* a single time step can represent an hour or a day. In contrast to a typical scheduling problem, release times and deadlines are not hard in general. As already mentioned, the integration of a certain component may be shifted on the time axis with spending additional effort. On the other hand, some points in time are fixed and must not be violated. This behavior can be compared with hard- and soft deadlines in embedded real-time systems. Therefore, the following timing parameters are assigned to each individual component:

Component/resource release time:	r_c
Component/resource deadline:	d_c
Component/resource final release time:	R_c
Component/resource final deadline:	D_c
Estimated duration of component integration:	i_c

The time r_c denotes the release time of a certain component or resource, d_c denotes the deadline for integration. These points in time could be shifted on the time axis with spending additional effort. Additionally, components are labeled with a final release time and a final deadline, R_c and D_c. These points in time can not be shifted and must be strictly adhered to. The time frame i_c denotes the estimated duration for integration and testing. The following equations must be hold:

$$R_c \leq r_c \tag{3.7}$$

$$D_c \geq d_c \tag{3.8}$$

$$D_c - R_c \geq i_c \tag{3.9}$$

In order to deal with multiple resources, individual time frames could be aggregated and the intersections of the time frames can be used:

$$[r_c, d_c] = [r_{c1}, d_{c1}] \cap [r_{c2}, d_{c2}] \cap ... \cap [r_{cn}, d_{cn}] \tag{3.10}$$

$$[R_c, D_c] = [R_{c1}, D_{c1}] \cap [R_{c2}, D_{c2}] \cap ... \cap [R_{cn}, D_{cn}] \tag{3.11}$$

Note that the equations 3.7, 3.8 and 3.9 must be still valid.

3.4.3.2 Calculating the planning effort for integration

As already mentioned, integration a component between r_c and d_c produces the smallest planning effort for integration, defined as 0. The effort rises if integration begins below r_c and/or exceeds d_c. The maximum reasonable planning effort occurs if integration reaches the absolute limits R_c and D_c as is defined by 1. To go beyond the absolute limits R_c and D_c is not possible by definition, therefore the effort then rises to infinity. The progression of the effort between r_c and R_c and d_c and D_c is discussible. For a first approximation a linear progression was chosen in this work. However, several progressions may be possible, *eg.* quadratic progression. A quadratic progression may map the fact that small deviations from schedule produce negligible effort whereas high deadline violations may be not tolerable. Since the effort progression can be easily adapted to the entire development process, in the following only the linear progression is discussed.

In Figure 3.11, the qualitative characteristic of the effort with a linear progression for component integration and testing against time is plotted.

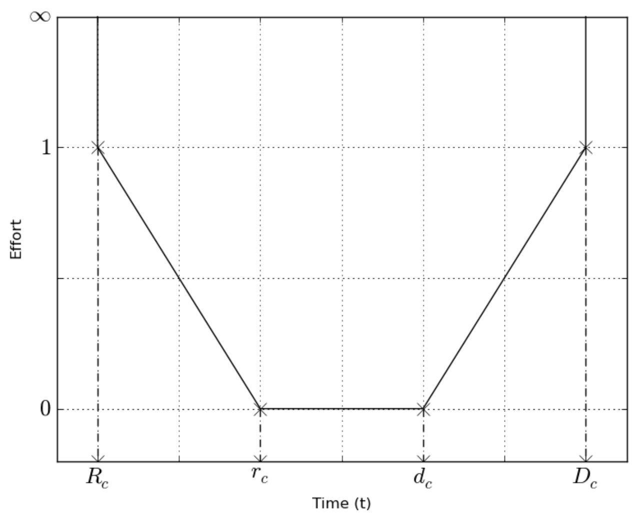


Figure 3.11: Planning effort for the integration and testing of a component against time. Integrating a component between r_c and d_c produces the smallest effort defined as 0. The planning effort for integration rises linearly if integration begins below r_c and/or exceeds d_c. The maximum reasonable effort occurs if integration reaches the absolute limits R_c and D_c as is defined as 1. To go beyond the absolute limits R_c and D_c is not possible by definition, therefore the effort then rises to infinity.

For a quantitative description of the planning effort the following approach is used: Let

t_b denote the begin of the component integration and t_e the end with $[t_b, t_e] = i_c$. The sum $\sum i_c$ denotes the overall duration of the integration. The planning effort for integrating a single component against time can be formulated as follows:

$$E_c(t) = \begin{cases} 0, & \text{if } t_b \geq r_c \wedge t_e \leq d_c \\ \frac{r_c - t_b + t_e - d_c}{\sum i_c} & \text{if } R_c \leq t_b < r_c \;\; \wedge \;\; D_c \geq t_e > d_c \\ \infty & \text{if } t_b < R_c \vee t_e > D_c \end{cases} \tag{3.12}$$

The planning effort for the overall integration is determined by the sum of individual effort divided by the number of components to integrate denoted by C.

$$E_s(t) = \frac{\sum E_c(t)}{C} \tag{3.13}$$

Another important characteristic is the maximum occurring time shift:

$$\hat{E}_c(t) = max(E_c(t)) \tag{3.14}$$

Example: In order to exemplify the proposed metric, an example with 5 components is illustrated. The following timing parameters are assumed for the individual components (Table 3.3):

Component	r_c	d_c	R_c	D_c	i_c
C0	0	4	0	∞	1
C1	5	10	0	∞	3
C2	2	6	0	∞	2
C3	8	10	0	∞	1
C4	4	9	0	∞	3

Table 3.3: Timing parameters of a example system with 5 components. Note that the absolute constraints are deactivated by setting R_c to zero and D_c to ∞.

In Figure 3.13, the results of two different integration orders are presents. In top-down integration (Figure 3.13(a)), the maximum occurring effort is caused by C1 ($\hat{E}_c = 0.4$)

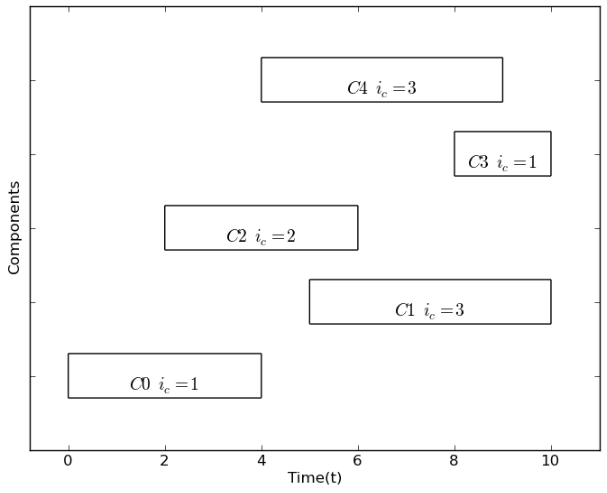
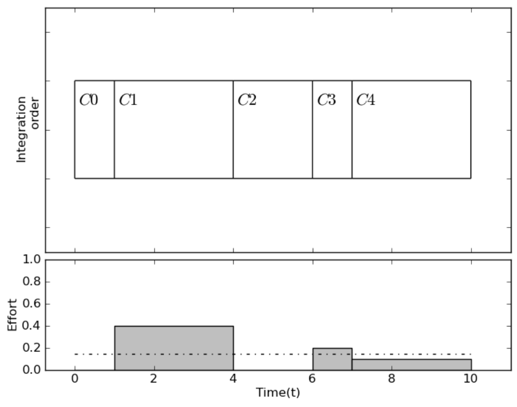
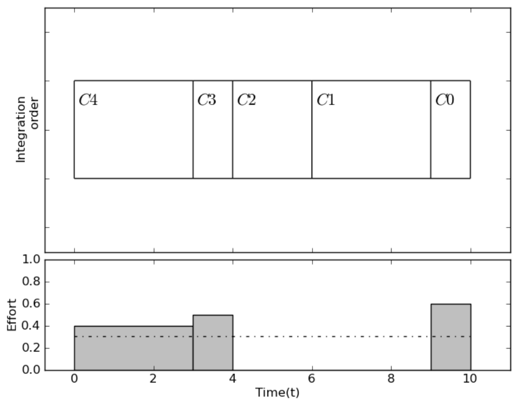


Figure 3.12: Graphical representation of the timing parameters presented in Table 3.3

since it should be integrated at $t = 1$ and becomes available not until $t = 5$. The effort for the overall integration $E_s(t) = 0.14$. In bottom-up integration (Figure 3.13(b)), the maximum occurring effort is caused by C0 ($\hat{E}_c = 0.6$) since it should be finished at $t = 4$ and is actually finished at $t = 10$. The effort for the overall integration $E_s(t) = 0.3$. Based on this metric, top-down integration causes less effort in relation to timing constraints.



(a) Top-down integration of the example system. The maximum occurring effort is caused by C1 ($\hat{E}_c = 0.4$) since it should be integrated at $t = 1$ and becomes available not until $t = 5$. The effort for the overall integration ($E_s(t) = 0.14$) is represented by the dotted line.



(b) Bottom-up integration of the example system. The maximum occurring effort is caused by C0 ($\hat{E}_c = 0.6$) since it should be finished at $t = 4$ and is actually finished at $t = 10$. The effort for the overall integration ($E_s(t) = 0.3$) is represented by the dotted line.

Figure 3.13: Occurring effort for integration caused by schedule in two different integration orders.

Chapter 4

Deriving an integration order in a component-based embedded system using simulated annealing

In Chapter 3, important parameters which affect the integration order were analyzed. While these parameters and the corresponding metrics help system integrators to evaluate a certain integration order, they will not provide an order which meets the corresponding requirements.

To overcome this restriction, a novel approach for deriving an integration order is presented. The approach described in the following section optimizes an integration order with respect to a single parameter as well as combinations of them. Since deriving an integration order is a NP-hard problem, a heuristic optimization approach based on simulated annealing (SA) was used. The method of simulated annealing is a suitable solution for large scale optimization problems. When adapted efficiently to optimization problems, simulated annealing is often characterized by fast convergence and ease of implementation for real-world problems. For example, simulated annealing has effectively ”solved” the famous traveling salesman problem (TSP) of finding the shortest cyclical itinerary for a traveling salesman who must visit each of N cities in turn [61]. Also minimizing interferences among connecting wires in complex integrated circuits has been successfully done by simulated annealing [58].

Simulated annealing is based on the analogy between finding a global minimum of a cost function for a combinatorial optimization problem and the slow cooling down of metal to its minimum energy state. Independently developed by Kirkpatrick *et al.* [37]

and Černý [19], SA is based on an algorithm proposed by Metropolis *et al.* [53] which describes the evolution of a solid to thermal equilibrium. In [43], Laarhoven describes simulated annealing as follow: Initially, the control parameter (the temperature T) is given a high value and a sequence of configurations of the combinatorial optimization problem is generated as follows. As in the iterative improvement algorithm a generation mechanism is defined, so that, given a configuration i, another configuration j can be obtained by choosing an element from the neighborhood of i randomly. Let C denote the cost of a configuration and $\Delta C_{ij} = C(j) - C(i)$, than the probability for configuration j to be the next configuration in the sequence is given by 1, if $\Delta C_{ij} \leq 0$, and by $exp(-\frac{\Delta C_{ij}}{k_B T})$, $\Delta C_{ij} > 0$. Thus, there is a non-zero probability of continuing with a configuration with higher cost than the current configuration. This process is continued until equilibrium is reached. The temperature is lowered in steps, with the system being allowed to approach equilibrium for each step by generating a sequence of configurations in the previously described way. The algorithm is terminated if no deteriorations are accepted anymore.

In the following sections, the concept of simulated annealing is adapted to the integration order problem (IOP).

4.1 Simulated annealing

The problem of optimizing an integration order is formulated as a combinatorial minimization problem with a discrete configuration space which represents the possible integration orders. In general, SA could be adapted to an combinatorial optimization problem with the definition of its four specific parts [61]:

- Configuration
- Rearrangement
- Annealing schedule
- Cost function

4.1.1 Configuration

The configuration represents a solution, including the initial solution, of the problem. The components are numbered $i = 0...C - 1$, where C represents the number of components of the software system. The configuration spaces denotes all possible permutations of C. Therefore a configuration is a permutation of the number $0...C-1$, interpreted as the order in which components are integrated. The initial solution is selected randomly.

4.1.2 Rearrangement

Rearrangement describes the mechanism for neighbor generation. An essential requirement for simulated annealing is that the rearrangement mechanism provides a move from the initial state to the optimal state in a sufficiently small number of steps. Based on the configuration definition, a rearrangement function that swaps two arbitrary components can get from any state (integration order) to any other state in $(C-1)$ steps. More efficient sets of moves are strongly connected to specific problems and the corresponding cost function. In the TSP, for example, it is expected that swapping two consecutive cities in a tour have a moderate effect on its energy (length); whereas swapping two arbitrary cities is far more likely to increase its length than to decrease it, even though an consecutive swap may need more steps to the optimal solution (a consecutive swap needs $n(n-1)/2$) steps in contrast to $(n-1)$ steps when the cities are selected randomly) [76]. Other approaches are mentioned *eg.* in [47].

The cost functions which are used in this work are strongly non-linear and depend not necessarily on close neighborhood. For example, the schedule of a certain component is completely independent of its direct neighbor. In order to provide a general solution for different cost functions, the neighbor generation is done randomly.

4.1.3 Annealing schedule

The cooling schedule describes the temperature distribution. In [28], German and German introduced a logarithmic cooling scheme. It has been proven that this schedule will lead the system to the global minimum state with an infinite time to run [30]. Since finite computing times are much more preferred in practical implementations, simpler schedules are used. In this work, the most common *exponential cooling schedule*, which was originally proposed by Kirkpatrick in [37], was used. This schedule is rather based on empirical rules than on theoretical studies [43].

In general, four different parameter must be discussed in a cooling schedule.

Start temperature of the algorithm T_{max} :
Kirkpatrick proposes the following empirical rule for determining the initial value of the temperature: Select a large value for T_{max} and perform a number of transitions. If the acceptance ratio x, defined by the number of accepted transitions divided by the number of proposed transitions (number of steps at each temperature), is less than a given value x_0 (in [37] and [24] $x_0 = 0.8$), double T_{max}. In this work, T_{max} was determined in this way.

End temperature of the algorithm T_{min} :
The optimization process could be either stopped by limiting the number of of temperature steps [8] or if no improvement (no new best solution) is found at one temperature [17]. The latter one was used in this work.

Number of steps at each temperature: I
On each step, the temperature must be held constant for an appropriate number of steps in order to keep the system close to equilibrium. If this number is to small, the algorithm is likely to converge to a local minimum. Selecting a value which depends polynomially on the size of the problem has been found to work well in many real-world problems [8].

Attenuation factor: α
The attenuation factor indicates how much the temperature is decreased at each step. Exponential cooling decreases the temperature in steps according to $T_{n+1} = \alpha T_n$ where $0 < \alpha < 1$. A good value for α is 0.95 which was first proposed by Kirkpatrick *et al.* in [37], but also wildly used by others ([60, 8, 16]) and also in this work.

4.1.4 Cost function

The cost function describes the goal of the minimization. In case of the TSP, the cost function may be simply the resulting length of the journey. In the following sections, several reasonable cost functions with respect to the integration parameters described in Section 3 are presented and evaluated on real life examples.

4.2 Minimizing the test effort

As already mentioned, almost all formal approaches presented in literature have the objective to minimize the the test effort, which actually means to minimize the number of stubs needed for the integration (c.f. Section 3.2). Additionally to minimize the number of stubs, this work deals also with the objective to minimize the stub complexity.

4.2.1 Minimizing the number of stubs

4.2.1.1 Cost function

In order to minimize the absolute number of stubs, the following cost functions are defined. Equation 4.1 is used to minimize the number of specific stubs, Equation 4.2 for minimizing the number of real stubs.

$$E_{spec}(o) = \sum_{i=1}^{C} d_i, \tag{4.1}$$

where d_i represents the number of dependencies stubbed in a certain integration step. Since a certain specific stub may be needed in different integration steps, each individual specific stub must be counted only once.

$$E_{real}(o) = \sum_{i=1}^{C} c_i, \tag{4.2}$$

where c_i represents the number of components stubbed in a certain integration step. Similar to specific stubs, each individual realistic stub must be counted only once.

4.2.1.2 Case study

In order to demonstrate the powerfulness of the presented approach, it was applied to the two reference systems introduced in Section 3.1 and compared with the two best graph-based results presented in Section 3.2.1. In Table 4.2 and 4.4 and Figure 4.1(a) and 4.1(b) respectively, the obtained results are presented. The tables give the absolute number of stubs which are needed for integration. The simulated annealing approach was applied with two optimization goals, minimizing realistic stubs marked with SA(r) (Equation 4.2) and minimizing specific stubs denoted as SA(s) (Equation 4.1). Since simulated annealing is a heuristic approach, it may yield different results on different runs. Each SA optimization was run 10 times with nearly identical results and the best result is listed. In Figure 4.2, the distribution of the resulting number of specific stubs in case of the AUTOSAR system is presented exemplarily. For a better comparability the results presented in the Figures are normalized to 1. The normalization is represented by the number of stubs divided by the maximum possible number of stubs. The maximum number of stubs is given by the number of components in the system or the number of dependencies respectively. Additionally, the corresponding annealing schedules used for the optimization are listed (Table 4.1 and 4.3).

Parameter	SA(s)	SA(r)
T_{max}	26.0	11.0
T_{min}	0.13	0.09
α	0.08	0.8
Steps at each temperature	6100	6100

Table 4.1: Annealing schedules for the embedded data logger

	Briand	Triskell	SA(s)	SA(r)
realistic stubs	4	3	4	**2**
specific stubs	**4**	5	**4**	4

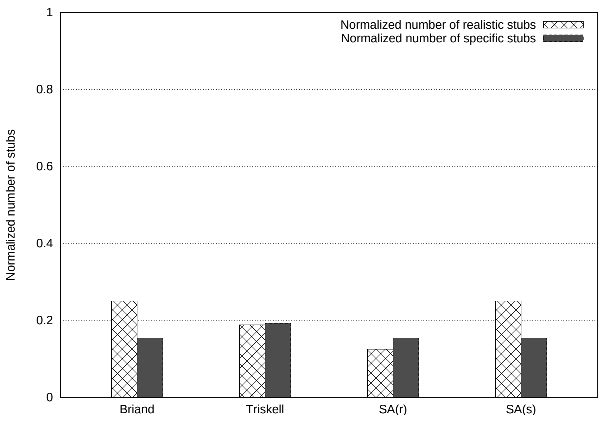
Table 4.2: Number of realistic and specific stubs needed for the integration of the data logger system with 16 components and 26 dependencies (Figure 3.1).

Parameter	SA(s)	SA(r)
T_{max}	59.0	11.0
T_{min}	0.13	0.06
α	0.08	0.8
Steps at each temperature	240	240

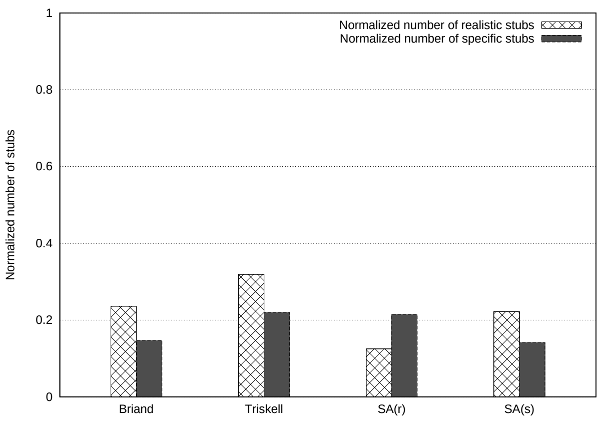
Table 4.3: Annealing schedules for the AUTOSAR system

	Briand	Triskell	SA(s)	SA(r)
realistic stubs	17	23	16	**9**
specific stubs	26	39	**25**	38

Table 4.4: Number of realistic and specific stubs needed for the integration of the AUTOSAR system with 72 components and 177 dependencies.



(a) Data logger



(b) AUTOSAR system

Figure 4.1: Normalized number of stubs needed for integration

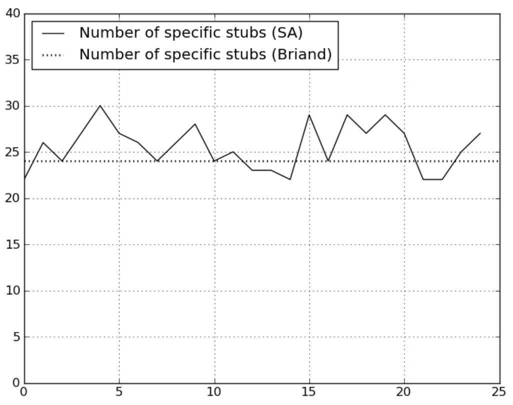


Figure 4.2: Distribution of the resulting number of specific stubs of the AUTOSAR system using simulated annealing.

4.2.1.3 Summary

The results indicate that the proposed approach provides at least comparable results in comparison to the graph-based solutions in case of specific stubs. In case of realistic stubs, which denotes the number of components to be stubbed, the simulated annealing approach obtains significantly better results on both reference systems. In Figure 4.2, the distribution of the resulting number of specific stubs of the AUTOSAR system when applying the proposed cost function is presented. The results show that the proposed approach is a reasonable solution for minimizing the number of stubs in component-based embedded system and can at least keep up with available solutions.

4.2.2 Minimizing overall stub complexity

4.2.2.1 Cost function

In order to minimize the overall stub complexity, the sum of all individual stub complexities as defined in Section 3.2.3, Equation 3.3 is used. Therefore, the corresponding cost function is defied as follows:

$$E_{compl}(o) = E_{SC} \tag{4.3}$$

4.2.2.2 Case study

In order to evaluate the proposed cost function, random generated complexities were assigned to the components in the embedded data logger (Section 3.1). A uniform distribution of random values between 1 and 20 were calculated and assigned to the components as shown in Table 4.5.

Component	0	1	2	3	4	5	6	7	8
Stub complexity	3	4	7	16	11	4	1	9	10

Component	9	10	11	12	13	14	15
Stub complexity	11	17	15	4	3	19	19

Table 4.5: Randomly generated stub complexities assigned to the individual components. The mean value of all complexities $m = 10$.

Table 4.6 shows the resulting stub complexity which occurs if components are integrated with the objective to minimize the number of real stubs (c.f. Equation 4.2). Since the stub complexity is not taken into account, component 7, which provides a high complexity, has to be stubbed.

Int. order	10, 0, 15, 8, 2, 4, 1, 12, 5, 11, 3, 13, 9, 6, 14, 7
real Stubs	11, 7
Stub complexity	24
T_{max}	11.0
T_{min}	0.09
α	0.80
Steps	6100

Table 4.6: Resulting stub complexity which occurs if components are integrated with the objective to minimize the number of real stubs (c.f. Equation 4.2). Since the stub complexity is not taken into account, component 7 which provides a high complexity has to be stubbed.

For the results presented in Table 4.7 the cost function for minimizing the stub complexity was used (Equation 4.3) In this case, the complex component 7 has not to be stubbed. Although 3 real stubs are needed in this case, the overall complexity is less than in Table 4.6, since the more simple components 6 and 12 have to be stubbed instead of the high complex component 7.

Int. order	10, 15, 7, 12, 0, 4, 8, 5, 2, 9, 1, 14, 3, 13, 11, 6
real Stubs	11, 6, 12
Stub complexity	20
T_{max}	180.0
T_{min}	0.58
α	0.80
Steps	5600

Table 4.7: Resulting stub complexity which occurs if components are integrated with the objective to minimize the overall stub complexity (c.f. Equation 3.3). The complex component 7 has not to be stubbed. Although 3 real stubs are needed in this case, the overall complexity is less than in Table 4.6, since the more simple components 6 and 12 have to be stubbed.

In some cases an individual component shall not be stubbed for different reasons. This objective can be achieved by assigning an infinite complexity to the corresponding components. In Table 4.8, a very high complexity ($v = 100000$, infinity is hardly possible for practical reasons) is assigned to the components 7 and 11. In the obtained integration order, the corresponding components have not be stubbed. However, the resulting stubs differ widely in their individual complexity. The maximum difference $\Delta v = v_{15} - v_6 = 0.19 - 0.01 = 0.18$.

Int. order	11, 10, 7, 15, 12, 8, 0, 2, 1, 4, 3, 13, 5, 9, 14, 6
real Stubs	5, 15, 6, 12
Stub complexity	28
T_{max}	8000.0
T_{min}	0.67
α	0.80
Steps	4400

Table 4.8: In this case, very high complexities ($v = 1000$) are assigned to the components 7 and 11. The results indicate that component 7 and 11 have not to be stubbed. However, the resulting stubs differ widely in their individual complexity. The maximum difference $\Delta v = v_{15} - v_6 = 0.19 - 0.01 = 0.18$

In order to obtain more uniform individual stub complexities, square addition was evalued. The corresponding cost function is shown in Equation 4.4.

$$E_{compl}(o) = \sum v_i^2 \tag{4.4}$$

However, as shown in Table 4.9, there is no difference in the results when using square addition of individual stub complexities instead of a linear addition in this system.

4.2.2.3 Summary

In this section an approach for taking the stub complexity into account was described. Additionally to the proposed way to derive the individual stub complexities, many other sources could be used, including information obtained from similar projects, developers experience or requirement based values. The presented SA approach provides a powerful and robust solution to derive an integration order with a minimum overall complexity. This

Int. order	7, 12, 0, 11, 10, 8, 2, 4, 3, 5, 13, 9, 15, 14, 1, 6
real Stubs	12, 6, 5, 15
Stub complexity	394
T_{max}	5300000.0
T_{min}	2.2
α	0.80
Steps	4900

Table 4.9: In this case, also very high complexities (1000) are assigned to the components 7 and 11. In contrast to Table 4.8, the cost function defined in Equation 4.4 was used for minimization. The results indicate that there is no difference, the components which have to be stubbed and therefore the resulting stub complexity are equal to Table 4.8. The difference in the stub complexity results only by square addition.

solution includes also the possibility to exclude components from stubbing by the use of very high complexities. This may be a strong advantage against all graph based solutions.

4.3 Optimizing test effort and integration test complexity

As stated in Section 3.3, reducing the integration test complexity and minimizing the test effort are contrary goals. In order to optimize both, a multi objective approach is described in this section. Multi objective optimization deals with more than one objective function and is defined in [38] as follows. Given an n-dimensional decision variable vector $x = x_1, ..., x_n$ in the solution space X, find a vector x^* that minimizes a given set of k objective functions $z(x^*) = z_1(x^*), ..., z_k(x^*)$. As Konak *et al.* stated in [38], there are two general approaches for solving a multi-objective optimization problem. The first is to create a single scalar objective function by usage of *eg.* utility theory or weighted sum method or to determine an entire Pareto optimal solution set or a representative subset. In this work, the weighted sum method is used. Since no objective should be preferred, equal weights are assigned. In order to make the results more comparable, equal stub complexities are assumed, therefore only the number of realistic stubs and test drivers is taken into account. However, the approach could be easily extended and the complexity of stubs and even the complexity of test driver can be taken into account.

4.3.1 Cost function

In order to compare the obtained results, the following three objective functions were applied to the embedded data logger system introduced in Section 3.1.

- Optimizing test effort:	$E_{(o)} = E_{TE}$
- Optimizing test complexity:	$E_{(o)} = E_{TC}$
- Optimizing both:	$E_{(o)} = \sqrt{E_{TE}^2 + E_{TC}^2}$

The first one, optimizing test effort, optimizes the extended test effort as described in Section 3.2.2. Additionally to the number of stubs, necessary test driver are also taken into consideration. As already mentioned, equal stub complexity is assumed and only the number of stubs is taken into account.

The second objective function optimizes the system test complexity as introduced in Section 3.3.2.2. The third one optimizes both, the test effort and the system testability.

In order to achieve a preferable uniform optimization of both objectives, the individual weight are set to one and square addition is used.

4.3.2 Case study

Figure 4.3 illustrates a sample of 4×10^6 integration orders of the embedded data logger. Note that this are only 0.00002% of all possible solutions. The minimum test effort occurring in this sample is $E_{TE}(min) = 0.375$, the best occurring test complexity is $E_{TC}(min) = 0.525$. In Figure 4.4 the optmization results of minimizing the test effort and optimizing the test complexity are shown. For each objective function, 1000 optimization runs were performed and plotted. In case of optimizing the test effort (Figure 4.4(a)), the optimization yields the same minimum value as the minimum value occuring in Figure 4.3. In case of optimizing the test complexity, the optimization yields a slightly lower value as the minimum value occuring in Figure 4.3. In Figure 4.5, the results of the multiobjective optimization are presented. Following the multi objective approach, the test complexity can be reduced for 16% with a increase in test effort of 17%, thus the presented multi objective approach represents a good compromise between additional test effort and reducing the test complexity. Since simulated annealing is a heuristic approach, each run may yield different results. Figure 4.5 depicts also that the optimization results differ only in a negligible way.

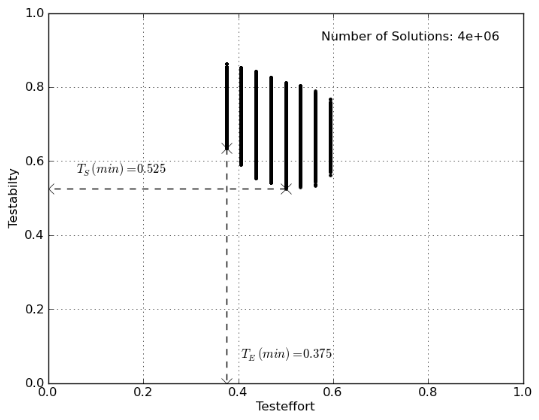
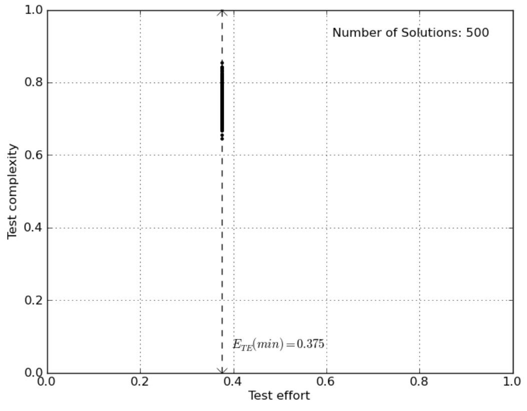
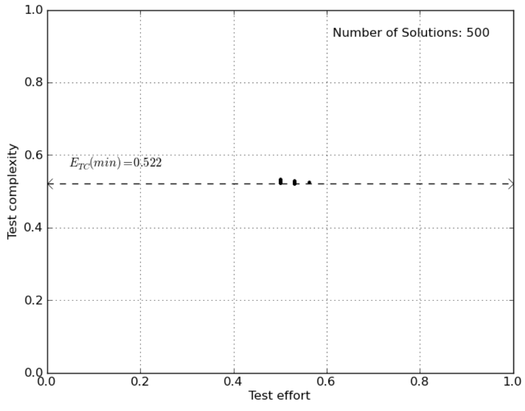


Figure 4.3: Sample of 4×10^6 integration orders of the embedded data logger. The minimum test effort occurring in this sample is $E_{TE}(min) = 0.375$, the lowest occurring test complexity is $E_{TC}(min) = 0.525$.

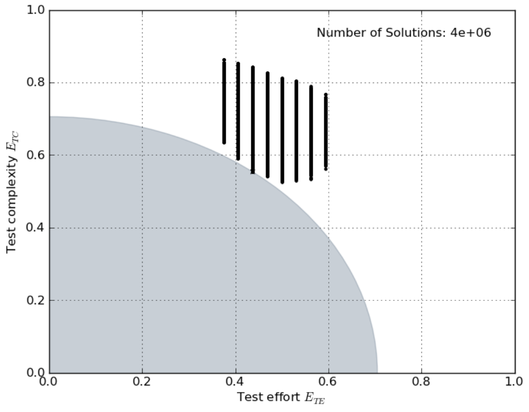


(a) Optimizing the extended test effort of the embedded data logger.

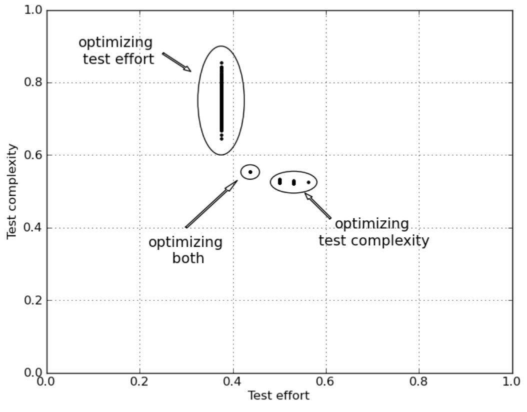


(b) Optimizing the system test complexity of the embedded data logger

Figure 4.4: Optimization results of the embedded data logger. 1000 values were calculated for each configuration.



(a) Results of the multi objective optimization results in contrast to a stochastic sample



(b) Results of the different optimization goals. 1000 values were calculated for each configuration.

Figure 4.5: Optimization results of the embedded data logger including the multi objective results.

4.3.3 Summary

The results indicate that minimizing the integration test complexity not necessarily increases the test effort. The test complexity vary significantly with a constant test effort, therefore the test complexity could be reduced without additional stubs or test drivers.

The proposed multi objective approach provides a reasonable and stable way to derive a compromise between these two contrary goals.

4.4 Optimizing integration schedule

Determining an integration order which meets the schedule is hardly possible in large scale embedded systems. Often schedule parameters are conflicting and compromises must be found. In this section, an cost function for deriving an integration order with a minimum planning effort is presented. The approach is based on the timing model presented in Section 3.4.3.

4.4.1 Cost function

Referring to Section 3.4.3 there are two possible cost functions which denote the optimization goal. On the one hand the overall effort for integration $E_s(t)$ and on the other the maximum occurring effort $\hat{E}_c(t)$. In practice however, several small differences to the optimal schedule cause usually less problems then a single big one. In order to verify this assumption, both cost function were evaluated.

$$E_{(o)} = \hat{E}_c(t) \tag{4.5}$$

$$E_{(o)} = E_s(t) \tag{4.6}$$

4.4.2 Case study

In this section both optimization goals, minimizing the maximum occurring value (Equation 4.5) and minimizing the mean value Equation (4.6), are applied to three example systems and the results are presented.

Experiment 1: The system defined in Table 3.3 was optimized. In Figure 4.6, this system was optimized with the objective to minimize the maximum occurring value $\hat{E}_c(t)$, In Figure 4.7 the same system was integrated with the objective to minimize the mean value $E_s(t)$.

The results presented in Table 4.12 indicate that there is no difference between both objective functions. Both lead to equal integration orders.

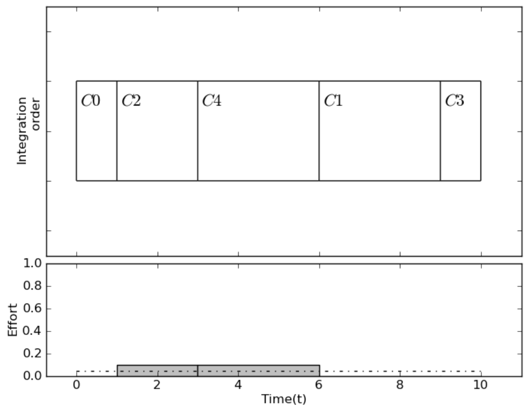


Figure 4.6: Integration sequence of five components optimized by using the proposed simulated annealing approach with the objective to minimize the **maximum occurring value** $\hat{E}_c(t)$ (Equation 4.5) The corresponding parameters are shown in Table 4.10.

T_{max}	3500.0
T_{min}	11.0
α	0.80
Steps	36000.0

Table 4.10: Annealing parameter used to obtain the results presented in Figure 4.6

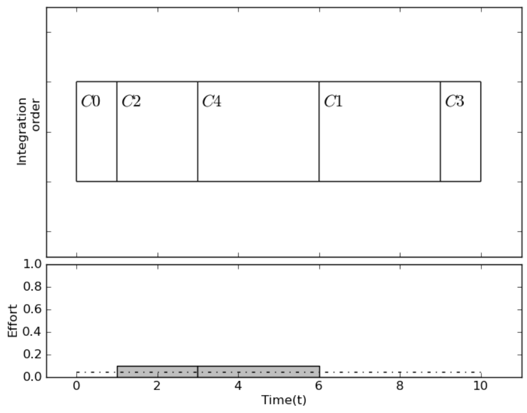


Figure 4.7: Integration sequence of five components optimized by using the proposed simulated annealing approach with the objective to minimize the **mean value** $E_s(t)$ (Equation 4.6). The corresponding parameters are shown in Table 4.11.

T_{max}	80000.0
T_{min}	110.0
α	0.80
$Steps$	36000.0

Table 4.11: Annealing parameter used to obtain the results presented in Figure 4.7

	top-down	min $\hat{E}_c(t)$	min $E_s(t)$
$\hat{E}_c(t)$	0.40	0.1	0.1
$E_s(t)$	0.14	0.04	0.04

Table 4.12: Optimization results of the example system with 5 components defined in Table 3.3.

Experiment 2: The embedded data logger introduced in Section 3.1 was equipped with timing parameters according to Table 4.13. In Figure 4.8 the corresponding planning effort for top-down integration is shown. In Figure 4.9, this system was optimized with the objective to minimize the maximum occurring value $\hat{E}_c(t)$, In Figure 4.10 the same system was integrated with the objective to minimize the mean value $E_s(t)$.

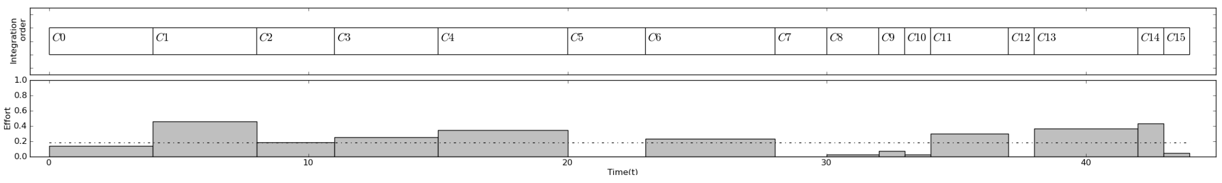


Figure 4.8: Top-down integration sequence of the embedded data logger introduced in Section 3.1 with respect to schedule constraints according to Table 4.13 **without optimization**.

The results presented in Table 4.16 indicate that the maximum occurring value $\hat{E}_c(t)$ is equal in both cases. However, the mean value $E_s(t)$ is significantly lower when using Equation 4.6. In both cases the assumed high risk components (C2, C3, C4) are scheduled to the beginning of the integration process.

Component	r_c	d_c	R_c	D_c	i_c
C0	6	42	0	∞	4
C1	24	34	0	∞	4
C2	0	3	0	∞	3
C3	0	4	0	∞	4
C4	0	5	0	∞	5
C5	2	23	0	∞	3
C6	2	18	0	∞	5
C7	8	41	0	∞	2
C8	31	35	0	∞	2
C9	35	38	0	∞	1
C10	1	33	0	∞	1
C11	0	24	0	∞	3
C12	34	40	0	∞	1
C13	11	26	0	∞	4
C14	3	24	0	∞	1
C15	39	42	0	∞	1

Table 4.13: Timing parameters for the embedded data logger introduced in Section 3.1. Note that the absolute constraints are deactivated by setting R_c to zero and D_c to ∞. The bold faced components (C2, C3, C4) are considered as high risk components, therefore they are label with $r_c = 0$ and the earliest possible deadline $d_c = i_c$.

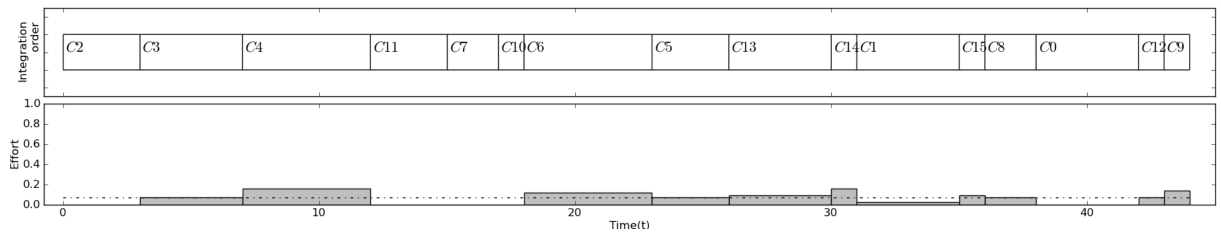


Figure 4.9: Integration sequence of the embedded data logger equipped with the timing parameters of Table 4.13 and optimized by the simulated annealing approach with the objective to minimize the **maximum occurring value** $\hat{E}_c(t)$ (Equation 4.5) The corresponding parameters are shown in Table 4.14.

T_{max}	710.0
T_{min}	3.5
α	0.80
$Steps$	6300.0

Table 4.14: Annealing parameter used to obtain the results presented in Figure 4.9

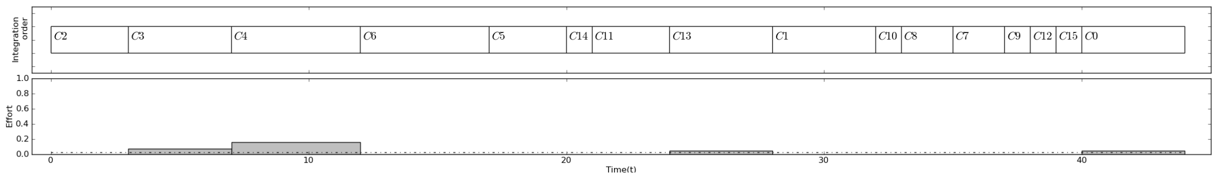


Figure 4.10: Integration sequence of an embedded data logger equipped with the timing parameters of Table 4.13 and optimized by the simulated annealing approach with the objective to minimize the **mean value** $E_s(t)$ (Equation 4.6). The corresponding parameters are shown in Table 4.15.

T_{max}	410000.0
T_{min}	170.0
α	0.80
$Steps$	6100.0

Table 4.15: Annealing parameter used to obtain the results presented in Figure 4.10

	top-down	min $\hat{E}_c(t)$	min $E_s(t)$
$\hat{E}_c(t)$	0.455	0.159	0.159
$E_s(t)$	0.178	0.065	0.020

Table 4.16: Schedule optimization results of the embedded data logger equipped with the timing parameters defined in Table 4.13.

Experiment 3: The embedded data logger introduced in Section 3.1 was equipped with alternative timing parameters define in Table 4.17. In Figure 4.11 the corresponding schedule effort for top-down integration is shown. In Figure 4.12, this system was optimized with the objective to minimize the maximum occurring value $\hat{E}_c(t)$, In Figure 4.13 the same system was integrated with the objective to minimize the mean value $E_s(t)$.

Component	r_c	d_c	R_c	D_c	i_c
C0	29	48	0	∞	5
C1	16	26	0	∞	3
C2	26	32	0	∞	5
C3	15	47	0	∞	4
C4	25	43	0	∞	3
C5	19	30	0	∞	1
C6	44	50	0	∞	6
C7	0	5	0	∞	5
C8	0	3	0	∞	3
C9	4	18	0	∞	3
C10	13	30	0	∞	3
C11	1	5	0	∞	1
C12	2	37	0	∞	1
C13	8	41	0	∞	4
C14	5	55	0	∞	5
C15	3	25	0	∞	3

Table 4.17: Alternative timing parameters for the embedded data logger introduced in Section 3.1. Note that the absolute constraints are deactivated by setting R_c to zero and D_c to ∞. The bold faced components (C7, C8) are considered as high risk components, therefore they are label with $r_c = 0$ and the earliest possible deadline $d_c = i_c$.

The results presented in Table 4.20 indicate that also in this case the maximum occurring value $\hat{E}_c(t)$ is equal in both cases. Again, the mean value $E_s(t)$ is significantly lower when using Equation 4.6. In both cases the assumed high risk components (C8, C7) are almost scheduled to the beginning of the integration process.

Discussion: The experiments 1-3 have shown that minimizing the mean value $E_s(t)$ leads to comparable results in the maximum occurring value $\hat{E}_c(t)$. The mean value itself,

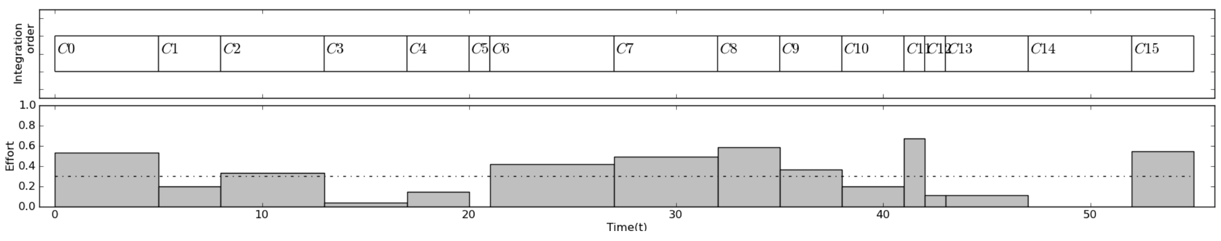


Figure 4.11: Top-down integration sequence of the embedded data logger introduced in Section 3.1 with respect to schedule constraints according to Table 4.17 **without optimization**.

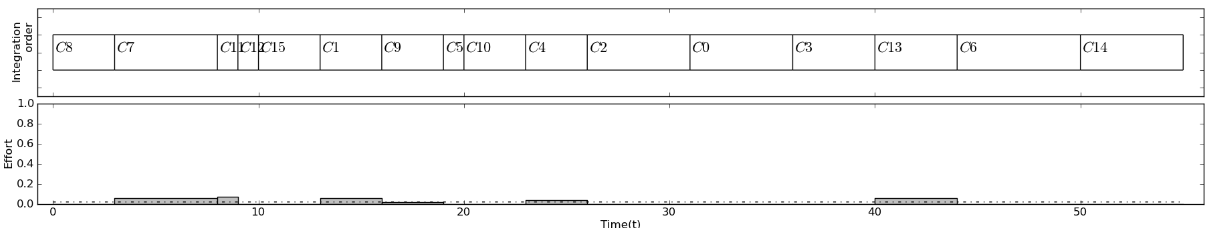


Figure 4.12: Integration sequence of the embedded data logger equipped with the timing parameters of Table 4.17 and optimized by the simulated annealing approach with the objective to minimize the **maximum occurring value** $\hat{E}_c(t)$ (Equation 4.5) The corresponding parameters are shown in Table 4.18.

T_{max}	1100.0
T_{min}	3.9
α	0.80
$Steps$	6400.0

Table 4.18: Annealing parameter used to obtain the results presented in Figure 4.12

T_{max}	350000.0
T_{min}	220.0
α	0.80
$Steps$	6500.0

Table 4.19: Annealing parameter used to obtain the results presented in Figure 4.13

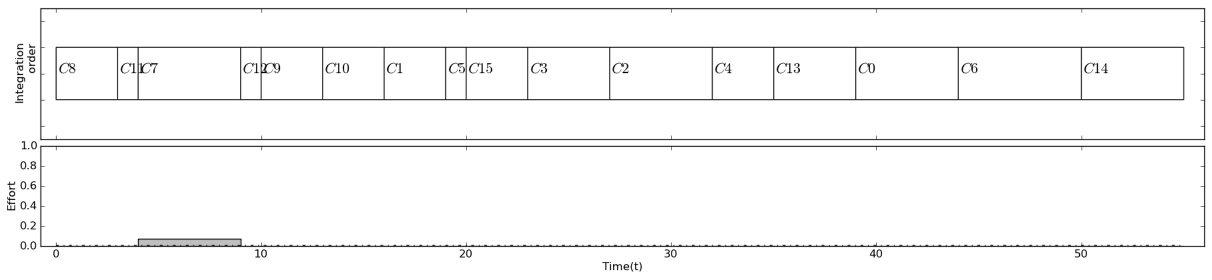


Figure 4.13: Integration sequence of an embedded data logger equipped with the timing parameters of Table 4.17 and optimized by the simulated annealing approach with the objective to minimize the **mean value** $E_s(t)$ (Equation 4.6). The corresponding parameters are shown in Table 4.19.

	top-down	min $\hat{E}_c(t)$	min $E_s(t)$
$\hat{E}_c(t)$	0.465	0.073	0.073
$E_s(t)$	0.180	0.018	0.005

Table 4.20: Schedule optimization results of the embedded data logger equipped with the timing parameters defined in Table 4.17.

however, is significantly lower as shown in experiment 2 and 3. Therefore minimizing the mean value $E_s(t)$ will be the better choice in most of the cases.

Experiment 4: This experiment deals with the proposed absolute deadlines and release times. Therefore, the timing parameters defined in Table 4.13 are extended with absolute constraints and presented in Table 4.21 . Since assigning infinity costs if R_c or D_c are violated is hardly possible, the value 100000 is used as infinity.

Component	r_c	d_c	R_c	D_c	i_c
C0	6	42	0	**42**	4
C1	24	34	0	∞	4
C2	0	3	0	∞	3
C3	0	4	0	∞	4
C4	0	5	0	∞	5
C5	2	23	0	∞	3
C6	2	18	0	∞	5
C7	8	41	0	∞	2
C8	31	35	0	∞	2
C9	35	38	**35**	∞	1
C10	1	33	0	∞	1
C11	0	24	0	∞	3
C12	34	40	0	∞	1
C13	11	26	0	∞	4
C14	3	24	0	∞	1
C15	39	42	0	∞	1

Table 4.21: Timing parameters for the embedded data logger introduced in Section 3.1. The parameters are equal to Table 4.13, except the absolute deadline added to C1 and the absolute release time added to C9

In Figure 4.14, the integration sequence of the embedded data logger equipped with absolute timing constraints (Table 4.21) and optimized by the simulated annealing approach with the objective to minimize the maximum occurring value $\hat{E}_c(t)$ (Equation 4.5). Figure 4.15 presents the integration sequence with the objective to minimize the mean value $E_s(t)$ (Equation 4.6). In Table 4.24, a comparison of three different integration strategies is given. The results indicate that all absolute deadlines are met with both objective functions. Also

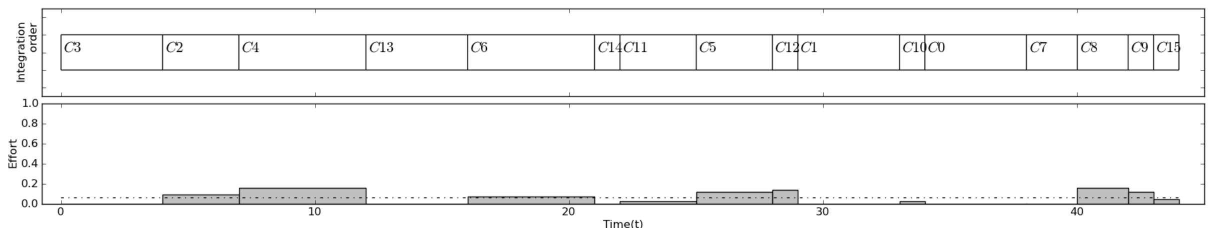


Figure 4.14: Integration sequence of the embedded data logger equipped with absolute timing constraints (Table 4.21) and optimized by the simulated annealing approach with the objective to minimize the **maximum occurring value** $\hat{E}_c(t)$ (Equation 4.5). The corresponding parameters are shown in Table 4.22.

T_{max}	5100000.0
T_{min}	3.3
α	0.80
Steps	6400.0

Table 4.22: Annealing parameter used to obtain the results presented in Figure 4.14

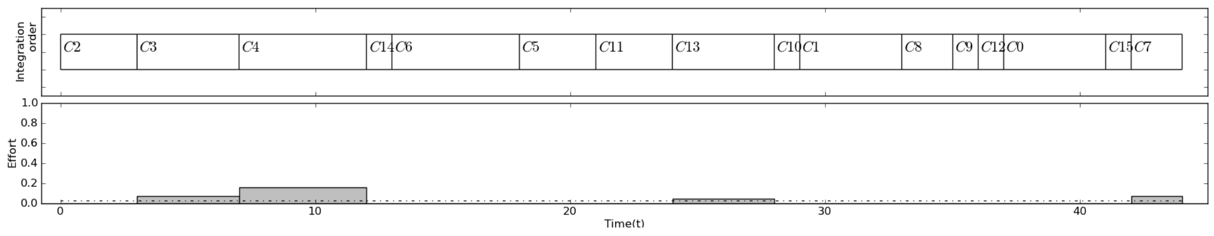


Figure 4.15: integration sequence of the embedded data logger equipped with absolute timing constraints (Table 4.21) and optimized by the simulated annealing approach with the objective to minimize the **mean value** $E_s(t)$ (Equation 4.6). The corresponding parameters are shown in Table 4.23.

T_{max}	200000000.0
T_{min}	200.0
α	0.8
Steps	6300

Table 4.23: Annealing parameter used to obtain the results presented in Figure 4.15

	top-down	min $\hat{E}_c(t)$	min $E_s(t)$
$\hat{E}_c(t)$	0.455	0.159	0.159
$E_s(t)$	0.178	0.058	0.021

Table 4.24: Schedule optimization results of the embedded data logger equipped with the timing parameters defined in Table 4.21.

the assumed high risk components are again scheduled to the beginning of the integration process. Of course, the effort is increased compared to the integration without absolute timing constraints (4.20) since the solution space is restricted. This means, however, that an increasing number of absolute timing constraints may increase the potential effort to the point of infinity.

4.4.3 Summary

The results presented in this section show that the proposed cost function provides a method to minimize the planning effort in software integration. It has been shown that minimizing the mean value yields better results compared with minimizing the maximum occurring value. The possibility to assign absolute deadlines and release times provides a method to apply hard constraints which must be met. However, this constraints should be sparsely used, since unfortunately choices may lead to an unfeasible schedule.

4.5 Optimizing integration schedule and stub complexity

Another reasonable optimization goal is to minimize the stub complexity (c.f. Section 3.2.3) and to minimize the schedule effort at the same time (Section 4.4). The main difference to the multi objective optimization of test effort and test complexity presented in Section 4.3 is that the optimization goals are independent and not contrary. Whereas test effort and test complexity are contrary goals, meaning reducing one increases the other, stub complexity and schedule effort are completely independent. In this section, a multi objective approach with a linear combination and a square addition is evaluated.

4.5.1 Cost function

In order to obtain equal weights for both objective functions, the stub complexity presented in Section 3.2.3 has to be normalized to 1. In order to achieve this, the sum of the complexities of the stubbed components is divided by the sum of the complexities of all components. Equation 4.7 depicts this relationship.

$$E_{\overline{SC}} = \frac{\sum\limits_{i=0}^{S} v_i}{\sum\limits_{i=0}^{C} v_i}, \tag{4.7}$$

where v_i represents the complexity which will occur of component i shall be stubbed and S the number of stubs.

In order to optimize both objectives, the schedule cost function (Equation 4.6) and the normalized stub complexity cost function (Equation 4.7) are added. In the following, linear addition (Equation 4.8) and square addition (Equation 4.9) are evaluated.

$$E_{(o)} = E_s + E_{\overline{SC}} \tag{4.8}$$

$$E_{(o)} = \sqrt{E_s^2 + E_{\overline{SC}}^2} \tag{4.9}$$

4.5.2 Case study

In the following, the cost functions were applied to the embedded data logger system. As for the schedule, the timing parameters defined in Table 4.13 and the parameters defined in Table 4.17 were used. In both cases, the values presented in Table 4.5 were used as stub complexities. Both cost functions were applied in each case.

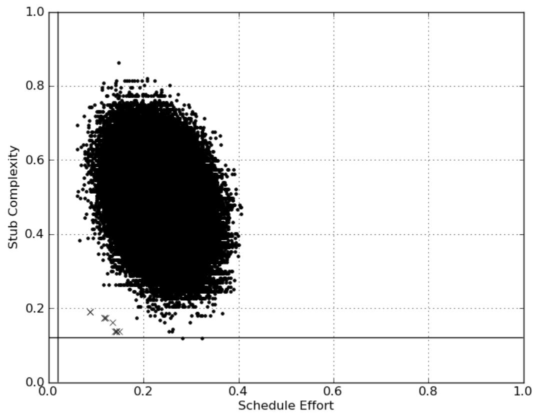


Figure 4.16: A sample with 10^5 possible integration orders was generated and the corresponding schedule mean value and the stub complexity were calculated. These solutions are plotted as points. Furthermore, 50 optimization runs with **square addition** were performed and the results marked with 'x'. The optimization results with the corresponding single objective function are marked with lines. The horizontal line represents the minimum stub complexity, the vertical line the minimum planning effort.

Figure 4.16 and Figure 4.17 depict the result when using the timing parameters defined in Table 4.13. A sample with 10^5 possible integration orders was generated and the corresponding schedule mean value and the stub complexity were calculated. These solutions are plotted as points. Furthermore, 50 optimization runs with square addition were performed and the results marked with 'x' (Figure 4.16). In Figure 4.17, the same sample was used and 50 optimization runs with linear addition were performed. The optimization results with the corresponding single objective function are marked with lines. The horizontal line represents the minimum stub complexity, the vertical line the minimum planning effort. The results indicate that both cost functions obtain favorable results compared with the 10^5 random solutions. However, linear addition produces significantly less variation compared with square addition. The results obtained with linear addition are nearly identical in each run.

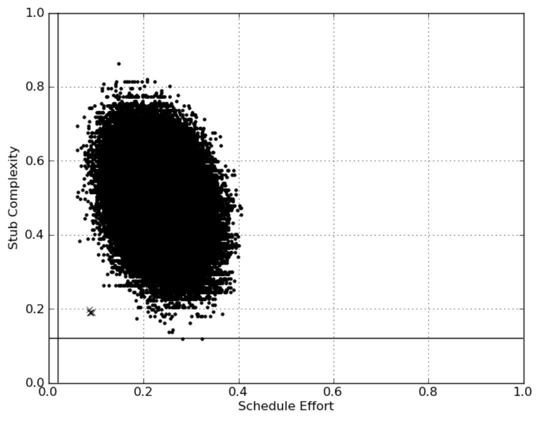


Figure 4.17: A sample with 10^5 possible integration orders was generated and the corresponding schedule mean value and the stub complexity were calculated. These solutions are plotted as points. Furthermore, 50 optimization runs with **linear addition** were performed and the results marked with 'x'. The optimization results with the corresponding single objective function are marked with lines. The horizontal line represents the minimum stub complexity, the vertical line the minimum planning effort.

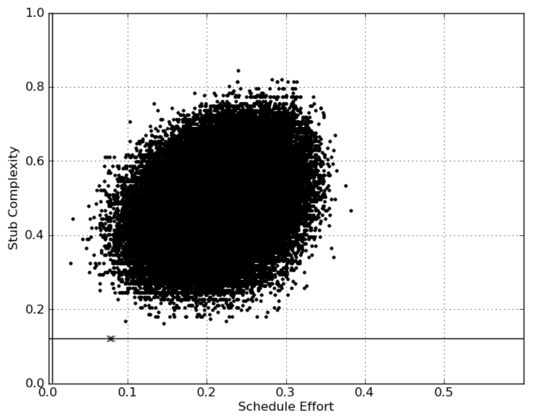


Figure 4.18: A sample with 10^5 possible integration orders was generated and the corresponding schedule mean value and the stub complexity were calculated. These solutions are plotted as points. Furthermore, 100 optimization runs with **square addition** were performed and the results marked with 'x'. The optimization results with the corresponding single objective function are marked with lines. The horizontal line represents the minimum stub complexity, the vertical line the minimum planning effort.

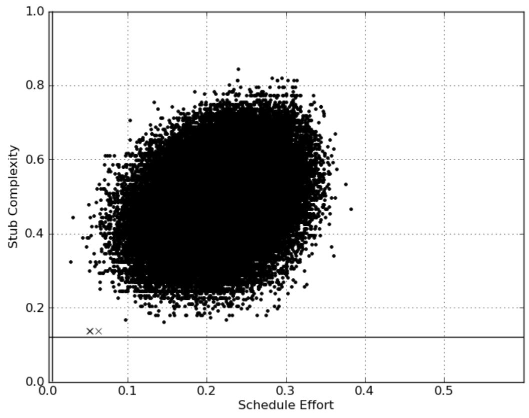


Figure 4.19: A sample with 10^5 possible integration orders was generated and the corresponding schedule mean value and the stub complexity were calculated. These solutions are plotted as points. Furthermore, 100 optimization runs with **linear addition** were performed and the results marked with 'x'. The optimization results with the corresponding single objective function are marked with lines. The horizontal line represents the minimum stub complexity, the vertical line the minimum planning effort.

Figure 4.18 and Figure 4.19 depict the corresponding results when using the timing parameters defined in Table 4.17. The results indicate that using square addition yields very unilateral results. As for the stub complexity, the results are equal to the minimum stub complexity obtained by single objective optimization. Linear addition produces more even results.

4.5.3 Summary

The results presented in this section show that the proposed cost functions provide a method to minimize the stub complexity and the planning effort for integration. However, especially the square addition yields a high scatter in the results. Additionally, no exclusion criteria (the absolute timing constraints R_c and D_c or a infinite stub complexity) were used in this evaluation. Such criteria will reduce the number of possible solutions. In practical problems, a decision maker which has insight to a certain problem will choose among several optimization results the one which satisfies best [51].

4.6 Optimizing test effort, integration testability and schedule

In this section a multi objective optimization approach which optimizes three different integration parameters is proposed. These optimization goals are the extended test effort (Section 3.2.2), integration test complexity (Section 3.3.2.2) and schedule effort (Section 4.4).

4.6.1 Cost function

In order to obtain a suitable energy function, the multi objective approach presented in Section 4.3 is extended by the maximum timing difference $\hat{E}_c(t)$.

$$E_{(o)} = \sqrt{E_{TC}^2 + E_{TE}^2 + \hat{E}_c(t)^2} \tag{4.10}$$

4.6.2 Case study

In Figure 4.20, a sample with 10^6 possible integration orders was generated and the corresponding parameters were calculated. These solutions are plotted as points. Furthermore, 100 optimization runs with the proposed objective function and the results marked with 'x'. For a better visualization, a sphere segment is added to the results (Figure 4.21). The area underneath the sphere houses better solutions. It can be seen that all 10^6 stochastic solution are above the sphere. That indicates that the proposed multi objective approach obtains favorable results compared with the 10^6 random generated integration orders. Furthermore there is no variation in the 100 optimization runs. Therefore, the proposed optimization approach is a useful way to derive an integration order with the proposed objectives.

4.6.3 Summary

In this section, a multi objective optimization approach with the three key parameters, test effort, test complexity and planning effort, is presented. This approach could be easily modified, *eg.* by using the schedule mean value instead of the maximum value. Also stub complexity could be used instead of the number stubs and test drivers. This experiment shows the promising scalability of the proposed approach. All optimization goals can be

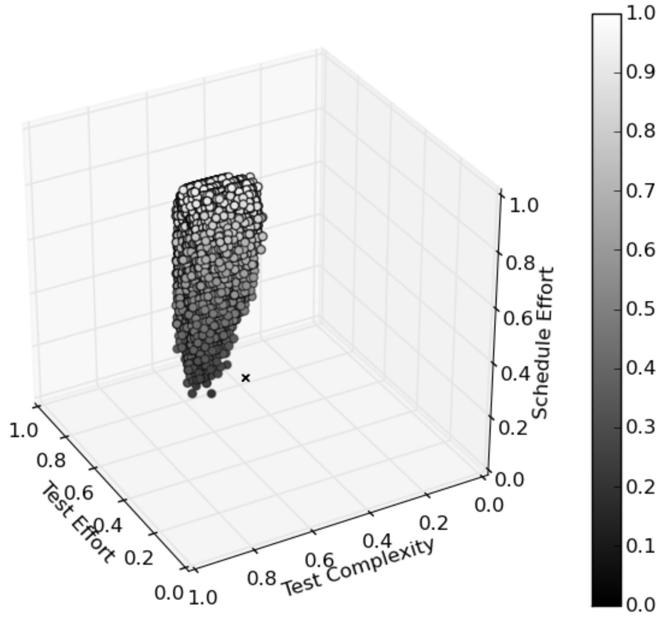


Figure 4.20: A sample with 10^6 possible integration orders was generated and the corresponding parameters were calculated. These solutions are plotted as points. Furthermore, 100 optimization runs with the proposed objective function and the results marked with 'x'

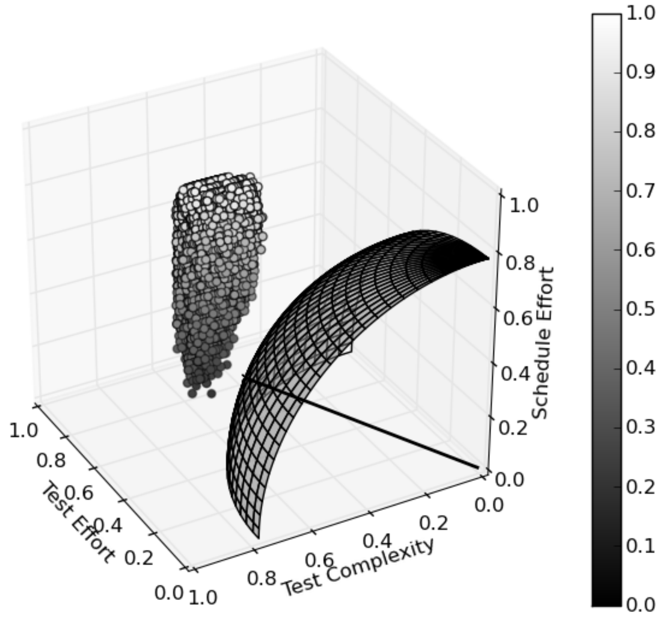


Figure 4.21: Results presented in Figure 4.20 with additional sphere segment for better visualization.

combined with favorable results. This implies that also future extensions can be added without changing the optimization strategy.

Chapter 5

Conclusion

In this work, parameters for software integration and an approach for deriving an integration order based on these parameters using a simulated annealing algorithm are presented. This chapter summarizes theses techniques and points out interesting issues for future work on this topic.

5.1 Conclusion

The most important driver of innovation in modern cars are embedded systems. New-generation cars contain a huge amount of features which would not be possible without the support of electronic devices and their respective software. This demand for new features and functions led to an increasing complexity in the design and development of embedded systems. Due to this high complexity, the task of building such systems becomes increasingly challenging and raises major concerns in critical application domains like the automotive industry. In order to meet this challenge, component-based architectures where introduced to automotive embedded systems. Despite the usage of *eg.* software product lines, a significant portion of new components must be integrated in each development step. This work contributes to the challenge of software integration and pursues the following objectives:

- The first objective is to identify parameters the software integration is subjected to and to define metrics for each parameter in order to evaluate different integration order strategies.
- The second objective is to develop methods and algorithms for optimizing an integration order with respect to the defined parameters.

- The third objective is to validate the developed approaches on real life examples.

As for the parameters of software integration, three key parameters were identified: test effort,test complexity and schedule effort. The integration test effort describes the effort which must be spend on constructing stubs and test drivers. Recent state-of-the-art approaches only consider the number of stubs needed for integration. In this work, three different metrics are presented for determining the test effort. Simply use the number of stubs, take the number of test drivers also into account or additionally assign complexities to these special implementations. The second key parameter is called test complexity and describes the integration testability. This topic addresses the often mentioned disadvantage of hierarchical integration orders (*eg.* bottom-up or top-down) whereas testing becomes difficult over several stages since test data must be passed through more and more components. Simply speaking, the more stubs and test drivers are involved when testing a certain interface, the more easy testing becomes. The third, and maybe the most relevant, parameter is the integration schedule. Due to the presented survey, schedule driven integration is wildly used in practice and is often done manually. In this thesis a formal approach is presented how these timing constraints can be modeled.

In order to derive an integration order with respects to the proposed parameters an optimization approach based on simulated annealing was developed. In addition to minimize the singe objectives test effort and schedule effort, reasonable combinations were evaluated. It has been shown that minimizing the test effort and minimizing test complexity, which are contrary goals, can be performed by the proposed approach in an sophisticated and reliable manner. Also adding the schedule effort as objective yields favorable results. Optimizing the stub complexity and the schedule effort, which are independent goals, is also possible with good results.

5.2 Pros and Cons

The presented approach offers the user a variety of advantages. First, the solution is very scalable. The described optimization goals can be individually used or combined in an almost arbitrary way. The results of the case studies have shown that favorable results are obtained in each case. Therefore, the user is allowed to choose the optimization goals which fits his requirements as its best. The proposed parameters are based on a basic system model which is available in almost all component based development processes, therefore no additional effort is necessary to apply this solution. Furthermore, the presented approach is open-ended, the proposed simulated annealing approach provides a robust solution.

However, the solution yields also a few drawbacks. The proposed metric for test complexity is based on a simplified system model and it must be proven in practice if this model is adequate. Also the stub complexity have to be evaluated in different software projects and need some experience in order to obtain preferable results.

In general, finding an integration order may be an iterative process. Several optimization runs with adjusted parameters may be performed during the development process until a preferable result is obtained.

5.3 Further work

This section presents topics and implications which should be investigated further in order to improve, refine and extend the techniques and methodologies presented in this thesis.

- Performance analysis and optimization: In this thesis no statement about the performance of the proposed optimization approach is given. Roughly speaking, the duration of an optimization run the embedded data logger with 16 components and 23 dependencies including the calculation of an reasonable annealing schedule is about 1 hour on an Intel Core 2 duo. The current implementation uses Python 2.7 and the NetworkX graph library. While the implementation with Python is sufficient for early experiments of the approach and easy to integrate in the development process, a moderate performance gain could be expected from an implementation in C/C++. Also some parallelization of the algorithm may be possible.

- User front end: At this time, only a experimental software version is available. Corresponding parameters have to be manually added to text files, also the system model has be created manually. Since this is very time-consuming and error-prone, a comfortable user front and is necessary. Additionally model transformers which which can transform model from other domains, *eg.* AUTOSAR, to the required format. Also a reasonable strategy for presenting results has to be developed. For this case, the polar comparison introduced by Koenig *et al.* in [39] provides a promising approach.

- System model: The integration parameters presented in this work are based on the dependency graph of the system. This model provides only generic information.

Since the used system models widely differs (Matlab Simulink, UML, etc.), this was a reasonable approach in this work. However, it may be useful to adopt the proposed parameters to a specific and more detailed system model in order to improve the results.

In a nutshell, the proposed approach provides a modular design were individual objectives of software integration could be combined and customized. The user is allowed to select and optimize integration parameters in order to satisfy his demands at its best.

References

[1] A. Abdurazik and J. Offutt. Using coupling-based weights for the class integration and test order problem. *The Computer Journal*, 52(5):557, 2009. 11, 19, 20, 24, 26

[2] A.J. Albrecht and J.E. Gaffney Jr. Software function, source lines of code, and development effort prediction: a software science validation. *Software Engineering, IEEE Transactions on*, (6):639–648, 1983. 26

[3] J. Axelsson. Holistic object-oriented modelling of distributed automotive real-time control applications. In *Object-Oriented Real-Time Distributed Computing, 1999.(ISORC'99) Proceedings. 2nd IEEE International Symposium on*, pages 85–92. IEEE, 1999. 15

[4] V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, 1996. 38

[5] B. Beizer. *Software system testing and quality assurance*. 1984. 4, 7, 27

[6] R. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000. 4

[7] R.V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, 1994. 29

[8] E. Bonomi and J.L. Lutton. The n-city travelling salesman problem: Statistical mechanics and the metropolis algorithm. *SIAM review*, pages 551–568, 1984. 48

[9] L. Borner and B. Paech. Integration Test Order Strategies to Consider Test Focus and Simulation Effort. In *Advances in System Testing and Validation Lifecycle, 2009.*

VALID'09. First International Conference on, pages 80–85. IEEE, 2009. 12, 15, 24, 38

[10] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. *European Journal of Operational Research*, 149(2):268–281, 2003. 39

[11] L.C. Briand, J. Feng, and Y. Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 43–50. ACM, 2002. 11, 12, 13, 24, 26

[12] L.C. Briand, J. Feng, and Y. Labiche. Experimenting with genetic algorithms to devise optimal integration test orders. *Software engineering with computational intelligence*, page 204, 2003. 13

[13] L.C. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, pages 594–607, 2003. 9, 10, 11, 19, 21, 23, 26, 39

[14] M. Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 33–42. ACM, 2006. 1, 2

[15] A.W. Brwan. Background information on cBD, 1997. 5

[16] R.E. Burkard and F. Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European Journal of Operational Research*, 17(2):169–174, 1984. 48

[17] F. Busetti. Simulated annealing overview. *World Wide Web URL www. geocities. com/francorbusetti/saweb. pdf*, 2003. 48

[18] R.D.V. Cabral, A. Pozo, and S.R. Vergilio. A Pareto ant colony algorithm applied to the class integration and test order problem. In *Proceedings of the 22nd IFIP WG*, volume 6, pages 16–29. 13

[19] V. Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985. 46

[20] Eppinger K. Claraz, D. and L. Berentroth. Reuse strategy at siemens VDO automotive: The EMS 2 powertrain platform architecture. *Ingenieurs de l ' Automobile*, page 767, 2004. xv, 2

[21] I. Crnkovic. Component-based software engineering: new challenges in software development. *Software Focus*, 2(4):127–133, 2001. 5

[22] I. Crnkovic. Component-based software engineering for embedded systems. In *Proceedings of the 27th international conference on Software engineering*, pages 712–713. ACM, 2005. 4

[23] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. In *Software Engineering Advances, International Conference on*, pages 44–44. IEEE, 2006. xv, 28

[24] A. Dekkers and E. Aarts. Global optimization and simulated annealing. *Mathematical programming*, 50(1):367–393, 1991. 48

[25] N.S. Eickelmann and D.J. Richardson. What makes one software architecture more testable than another? In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*, pages 65–67. ACM, 1996. 5, 8

[26] J. Gao, H.S.J. Tsao, and Y. Wu. *Testing and quality assurance for component-based software*. Artech House on Demand, 2003. 4

[27] Vera Gebhardt, Gerhard M. Rieger, JÃijrgen Mottok, and Christian GieÃ§elbach. *Funktionale Sicherheit nach ISO 26262: Ein Praxisleitfaden zur Umsetzung*. dpunkt.verlag GmbH, 1., auflage edition, 2 2013. 28

[28] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):721–741, 1984. 47

[29] M. Hafner. Umfrage "software-integration". *Embedded Software Engineering Report*, 2010. xv, 15, 17

[30] B. Hajek. Cooling schedules for optimal annealing. *Mathematics of operations research*, pages 311–329, 1988. 47

[31] M.H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977. 38

[32] B. Hardung, T. Kölzow, and A. Krüger. Reuse of software in distributed embedded automotive systems. In *Proceedings of the 4th ACM international conference on Embedded software*, pages 203–210. ACM, 2004. 1

[33] M. Jaffar-ur Rehman, F. Jabeen, A. Bertolino, and A. Polini. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*, 17(2):95–133, 2007. 2

[34] F. Jay and R. Mayer. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610:1990, 1990. 5, 6

[35] M. Jung and F. Saglietti. Supporting Component and Architectural Re-usage by Detection and Tolerance of Integration Faults. 2005. 29

[36] D. Karlsson. Verification of Component-based Embedded System Designs. 2006. 2

[37] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671, 1983. 45, 47, 48

[38] A. Konak, D.W. Coit, and A.E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, 2006. 56

[39] F. König, D. Boers, F. Slomka, U. Margull, M. Niemetz, and G. Wirrer. Application specific performance indicators for quantitative evaluation of the timing behavior for embedded real-time systems. In *Proceedings of the conference on Design, automation and test in Europe*, pages 519–523. European Design and Automation Association, 2009. 81

[40] H. Kopetz. Component-based design of large distributed real-time systems. *Control Engineering Practice*, 6(1):53–60, 1998. 5

[41] A. KRÜGER, G. WAGNER, N. EHMKE, and S. PROKOP. Wirtschaftliche betrachtungen und mögliche geschäftsmodelle für standard-software. *VDI-Berichte*, pages 1057–1071, 2003. 1

[42] D.C. Kung, J. Gao, P. HsiaYasufumi, and C. Chen. On regression testing of object-oriented programs. *Journal of Systems and Software*, 32(1):21–40, 1996. 9, 10, 19, 21

[43] P.J.M. Laarhoven and E.H.L. Aarts. *Simulated annealing: theory and applications*, volume 37. Springer, 1987. 46, 47

[44] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck, and M.H. Durand. Testing levels for object-oriented software. 2000. 9

[45] V. Le Hanh, K. Akif, Y. Le Traon, and J.M. Jézéque. Selecting an efficient OO integration testing strategy: an experimental comparison of actual strategies. *ECOOP 2001-Object-Oriented Programming*, pages 381–401, 2001. 10, 22

[46] Y. Le Traon, T. Jéron, J.M. Jézéquel, and P. Morel. Efficient object-oriented integration and regression testing. *Reliability, IEEE Transactions on*, 49(1):12–25, 2002. 10, 20, 22

[47] Shen Lin. Computer Solutions of the Traveling Salesman Problem. *Bell System Technical Journal*, 44(10):2245–2269, 1965. 47

[48] K. Lind and R. Heldal. Categorization of real-time software components for code size estimation. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 26. ACM, 2010. 2

[49] D. Lohmann, W. Schroder-Preikschat, and O. Spinczyk. Functional and non-functional properties in a family of embedded operating systems. In *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, pages 413–420. IEEE, 2005. 1

[50] B.A. Malloy, P.J. Clarke, and E.L. Lloyd. A parameterized cost model to order classes for class-based testing of C++ applications. 2003. 11, 20, 23

[51] R.T. Marler and J.S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004. 74

[52] T.J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pages 308–320, 1976. 38

[53] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21:1087, 1953. 46

[54] M. Morisio, M. Ezran, and C. Tully. Success and failure factors in software reuse. *Software Engineering, IEEE Transactions on*, 28(4):340–357, 2002. 2

[55] J. Mottok. Bayerisches it-sicherheitscluster: Automotive forum. http://www.it-speicher.de/itsecurity/, 2009. xv, 15, 17

[56] V.L. Narasimhan and B. Hendradjaya. A new suite of metrics for the integration of software components. In *Proceedings of the The First International Workshop on Object Systems and Software Architectures (WOSSA'2004)*. Citeseer, 2004. 38

[57] T.J. Ostrand and E.J. Weyuker. How to measure success of fault prediction models. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, pages 25–30. ACM, 2007. 38

[58] R. Otten and L. Van Ginneken. *The annealing algorithm*, volume 37. Kluwer Academic Publishers Boston MA., 1989. 45

[59] R. Paul. End-to-end integration testing. In *Quality Software, 2001. Proceedings. Second Asia-Pacific Conference on*, pages 211–220. IEEE, 2001. 27

[60] B.R. Preiss. *Data structures and algorithms with object-oriented design patterns in C++*. A1bazaar, 2008. 48

[61] W. Press, S. Teukolsky, and W. Vetterling. B. flannery," numerical recipes in c, 1992. 45, 46

[62] J. Ratzinger, T. Sigmund, and H.C. Gall. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 35–38. ACM, 2008. 38

[63] D. Rombach. Software nach dem baukastenprinzip. *Fraunhofer Magazin*, (1):30–31, 2003. 1

[64] J. Ropponen and K. Lyytinen. Components of software development risk: How to address them? A project manager survey. *Software Engineering, IEEE Transactions on*, 26(2):98–112, 2000. 38

[65] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, et al. *Object-oriented modeling and design*, volume 38. Prentice hall Englewood Cliffs (NJ), 1991. 10, 21

[66] Ian Sommerville. *Software Engineering (Pearson Studium - IT)*. Pearson Studium, 9. aktual. edition, 3 2012. 28

[67] M. Steindl, J. Mottok, and H. Meier. SES-based framework for fault-tolerant systems. In *Intelligent Solutions in Embedded Systems (WISES), 2010 8th Workshop on*, pages 12–16. IEEE. 1

[68] C. Szyperski, D. Gruntz, and S. Murer. Component software: beyond object-oriented programming. 2002. 5

[69] A. Taghipour and E. Taheripour. Object-oriented programming application in automotive door control performance. In *Proceedings of the 7th WSEAS International Conference on Applied Computer and Applied Computational Science*, pages 353–357. World Scientific and Engineering Academy and Society (WSEAS), 2008. 15

[70] K.C. Tai and FJ Daniels. Test order for inter-class integration testing of object-oriented software. In *Computer Software and Applications Conference, 1997. COMPSAC'97. Proceedings., The Twenty-First Annual International*, pages 602–607. IEEE, 2002. 10, 19, 21, 24, 39

[71] R. Tarjan. Depth-first search and linear grajh algorithms. In *Conference Record 1971 Twelfth Annual Symposium on Switching and Automata Theory*, pages 114–121. IEEE, 1971. 10, 22

[72] S. Voget and M. Becker. Establishing a software product line in an immature domain. *Software Product Lines*, pages 121–168, 2002. 2

[73] Z. Wang, B. Li, L. Wang, M. Wang, and X. Gong. Using Coupling Measure Technique and Random Iterative Algorithm for Inter-Class Integration Test Order Problem. In *2010 34th Annual IEEE Computer Software and Applications Conference Workshops*, pages 329–334. IEEE, 2010. 11, 12, 23

[74] A.H. Watson, T.J. McCabe, and D.R. Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST special Publication*, 500:235, 1996. 27

[75] Y. Wu and M.H.C. Dai Pan. Techniques for testing component-based software. In *iceccs*, page 0222. Published by the IEEE Computer Society, 2001. 29

[76] Z. Zhang and X. Ke. Solving terminal allocation problem using simulated annealing arithmetic. *WSEAS Transactions on Systems*, 7(12):1412–1422, 2008. 47

Author’s Publications

[1] M. Steindl, J. Mottok, H. Meier, F. Schiller, and M. Fruechtl. Migration of SES to FPGA Based Architectural Concepts. In *Softwaretechnik Trends, Gesellschaft fuer Informatik*, ISSN 0720-8928, 2009.

[2] M. Steindl, J. Mottok, H. Meier, F. Schiller, and M. Fruechtl. Diskussion des Einsatzes von Safely Embedded Software in FPGA-Architekturen. In *Proceedings of the 2nd Embedded Software Engineering Congress*, ISBN 978-3-8343-2402-3, pages 655-661, December 2009.

[3] M. Steindl, J. Mottok, H. Meier, F. Schiller, and M. Fruechtl. Safeguarded Processing of Sensor Data. In *Proceedings of the 5th Embedded Real Time Software and Systems Conference (ERTS2 2010), Toulouse, France, May 2010.*

[4] M. Steindl, J. Mottok, and H. Meier. SES-based framework for fault-tolerant systems. In *Proceedings of the 8th IEEE Workshop on Intelligent Solutions in Embedded Systems (WISES)*, ISBN 978-1-4244-5716-8, pages 12–16, Heraklion, Greece, July 2010.

[5] M. Steindl, J. Mottok. Deriving an integration order in a component-based embedded system using simulated annealing. In *Proceedings of the NWK13*, ISBN 978-3-86870-436-5, Goerlitz, Germany, April 2012.

[6] M. Steindl, J. Mottok. Optimizing Software Integration Testing by Considering Integration Testability and Test Effort. In *Proceedings of the 10th IEEE Workshop on Intelligent Solutions in Embedded Systems (WISES)*, ISBN: 978-1-4673-2464-9, Klagenfurt, Austria, July 2012

[7] M. Steindl, J. Mottok. Considering Schedule Requirements of Software Integration in Component based Embedded System. In *Proceedings of the 17th International Conference on Applied Electronics*, ISSN 1803-7232, Pilsen, Czech Republic, Sept. 2012

[8] M. Steindl, J. Mottok. Minimizing the Number of Stubs in Component-based Software Integration by using Simulated Annealing. **Submitted** *to the International Conference on Software Engineering (ICSE),*San Francisco, USA, May 2013.

[9] M. Steindl, J. Mottok. Optimizing Software Integration in Component-based Embedded Systems by Using Simulated Annealing. **Submitted** *to the IEEE Region 8 Conference EuroCon 2013*, Zagreb, Croatia, July 2013.