School of Humanities and Informatics
Final Year Project, Advanced level, 30 credits
Spring term 2009

Methods for Testing Concurrent Software

Ramon Radnoci

Methods for Testing Concurrent Software

Submitted by Ramon Radnoci to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the School of Humanities and Informatics.

June 24, 2009

I hereby certify that all material in this dissertation which is not my own work has been identified and that no work is included for which a degree has already been conferred on me.

Signature:_____

Methods for Testing Concurrent Software

Abstract

Most software today is concurrent and are used in everything from cell-phones, washing machines, cars to aircraft control systems. The reliability of the concurrent software may be more or less critical, depending on which a.o. domain it is functioning in. Irrespective of domain, the concurrent software must be sufficiently reliable.

It is therefore interesting to study how adaptable test methods for sequential software are to test concurrent software. Novel test methods for concurrent software can be developed by adapting test methods for sequential software. In this dissertation, adaptability factors have been identified by conducting a literature survey over state-of-the-art test methods. Directions taken in the research of concurrent software testing is described by the survey. The survey also demonstrates differences and similarities between test methods.

Three research contributions has been achieved by this dissertation. First, this dissertation presents a survey over state-of-the-art-test methods. The second contribution is the identified adaptability factors that should be added to a test method for sequential software, that will be adapted to test concurrent software. Finally, the third contribution to the field of concurrent software testing is the identified future work in areas where test methods for concurrent software has not been researched much or at all.

Keywords: concurrent software testing, software testing, adaptability, test methods

Acknowledgements

First of all I would like to thank my supervisor Birgitta Lindström at University of Skövde for her excellent supervision and the amount of time she spent on this dissertation. Our supervision sessions was both instructive and enjoyable.

Second, I also want to thank my supervisor Farzad Fooladvandi at Saab Microwave Systems Skövde for his exceptional supervision. He has given me valuable advice and feedback. A thank also goes to colleagues at Saab Microwave Systems for their interesting questions during my presentations.

Third, I want to thank my examiner Gunnar Mathiason for the feedback given on my dissertation and presentations.

Fourth, I would like to thank Prof. Jeff Offutt from George Mason University for his invaluable advice and feedback and time spent on my dissertation.

A thank also goes to Prof. Sang H. Son from University of Virginia for his invaluable feedback on my presentation. He has also given me additional angle of approaches to consider.

I also want to thank Prof. Sten F. Andler, Marcus Brohede and Johan Grahn for their feedback on my presentation for the Distributed Real-Time Systems group.

And finally, I want to thank my opponent Jonas Mellin for asking relevant and interesting questions during my last presentation and defend of this dissertation, and also for the feedback given on my dissertation.

All of you have helped to increase the quality of this dissertation.

Ramon Radnoci
June 24, 2009

Software testing can be used to show the presence of faults, but never to show their absence!

Edsger W. Dijkstra

Contents

List of Figures

1 Introduction

Methods for Testing Concurrent Software has been written because there is a need for test methods for concurrent software. Software testing is mature in general, because research has finally met practise. But there is a lack for test methods applicable on concurrent software. That is a problem because software today are mostly concurrent. It is what is being developed and hence used. Also, the hardware industry pushes forward into this direction by releasing hardware that can utilize the power of concurrency more efficiently.

Software must be sufficiently reliable. How reliable depends on the domain it is operating in. It is especially important to make it possible to test concurrent software, because the probability of occurring faults may increase. That is because concurrent software consists of competing and cooperating processes. Thus, additional fault types exist in concurrent software compared to sequential software.

The execution order between processes in concurrent software cannot be foretold, which is another reason making concurrent software intricate to test. These differences increases the probability of faults. Test methods for sequential software cannot reveal fault types specific for concurrent software, because they are not developed to focus on them. Thus, there is a need for test methods for testing concurrent software.

One hypothesis was that although test methods for sequential software cannot reveal specific faults for concurrent software, there may exist common properties between test methods for sequential software and those developed for concurrent software. Thus, it may be possible to adapt a test method developed to test sequential software, to test concurrent software.

Another hypothesis was that every test method for concurrent software may contain common properties. That is, properties that every test method for concurrent software must have in order to reveal faults in concurrent software. Hence, the intention of this dissertation was to investigate what must be added to a test method developed for sequential software, to test concurrent software.

Target readers are computer scientists, software testers and researchers.

1.1 Dissertation Outline

Section 2 gives a background to software testing and concurrency in general, as well as their relations. Section 3 provides information about the problem domain for the dissertation. That includes both aim and objectives, and also methods for how to achieve them. Section 4 presents the literature survey, that is, an overview of test methods. Section 5 presents factors that is important to consider when adapting a test method for testing concurrent software. Finally, section 7 presents conclusions, related work as well as future work.

2 Background

This section introduces fundamental concepts and presents the background for the dissertation. Section 2.1 introduce software testing in general. Section 2.2 provides information about concurrency and how it becomes a problem for software testing. Finally, section 2.3 presents fault types that are particularly interesting when the software is concurrent.

2.1 Software Testing

We are surrounded by software products. Everything from electric razors and washing machines to cellphones, cars and control systems for nuclear plants, contain software. We are dependent of software today and we expect them to work, that is, that they are sufficiently reliable. Hence, it is important that they keep on working without malfunctioning. One method for increasing the reliability of software is testing. *Software testing* is the evaluation of software by observing its execution (Ammann & Offutt 2008).

Software testing should reveal faults. A *fault* is a static implementation mistake made by the programmer. If the software reaches a fault, it may enter an incorrect internal state, that is an *error*. If the software enters an error, it may propagate to an externally incorrect behavior. That is called a *failure*. Test cases should provide clear feedback so that revealed faults can be easily corrected (Ammann & Offutt 2008, Beizer 1990).

It is necessary to observe when, why and what the software does in order to distinguish between its correct and erroneous behavior. *Observability* is the capability to observe both internal and external behavior during test execution. When a software is observed, its behavior can be estimated. However, the software must facilitate such an observation, that is, it must be *observable* (Schütz 1994).

Three conditions that are necessary for observing a failure:

Figure 1: The RIP-model after Ammann & Offutt (2008)

Reach means that the location(-s) of the fault in the software must be reached by execution. The state of the location must be incorrect after the location has been executed (*Infect*). If the infected state propagate to cause an output of the software to be incorrect, it is referred to as *Propagation*. This fault/failure model was originally developed by Morell (1988) under the name PIE. *PIE* abbreviates Propagation, Infection and Execution.

Three important properties are tightly connected to the conditions in the RIP-model. That is, *controllability* together with *reproducibility* which con-

cerns the reach and infect condition, and *observability* that concerns the propagate condition.

The first property, controllability, is the amount of influence the tester has over the software when executing test cases on it.

The second property, requires that the software is *reproducible* during testing. Tests are reproducible if the behavior of the software is the same each time the same test cases are executed. Otherwise it is not guaranteed that repeated executions activates the same error. Hence, reproducibility is a part of controllability. Reproducibility is the part that concerns reproducing, for example, the same execution orders and input values. The process of repeatedly execute test cases after faults have been corrected or after the software has been changed (Schütz 1994), is called *regression testing* (Adrion et al. 1982). The main purpose of regression testing is to ensure that revealed faults have been accurately corrected and/or, that the modifications did not introduce new, undesired effects or faults (Schütz 1994).

The third property, observability, not only concerns the software, but also its environment. That is because the correctness of the software's behavior can often only be evaluated with respect to what is happening in its environment (Schütz 1994). For instance, is the answer *no* to the question if there exist a *"Kim Lundberg"* in the database correct? That depends on the situation, that is, what is in the database. If Kim Lundberg do not exist in the database, then the answer is correct, but otherwise it is not.

Observability is the ability to observe what is happening in the software when it executes. For instance, databases are considered to have low observability (Gait 1986). Consider the case when an end-user deletes a customer from a database. It is not difficult to check if the customer was deleted, but also other customers may have been deleted. In order to check that, the whole database needs to be checked (Lindström 2009).

If it is not possible to decide if the software did the right action or not given its outputs, the internal state should be traced. For instance, printout statements can be added in the source code to increase observability. However, by adding printout statements, the schedule and the outcome of a race condition may be affected so that different output and behavior occur. This observability problem is called *the probe effect*. Gait (1986) further describes that probe effects can be observed in the altered behavior of concurrent software. Either a non-functioning concurrent software works with inserted delays, or a functioning concurrent software stops working when previously embedded implicit delays are removed, relocated or perhaps changed in value (Gait 1986).

Several test process models exist. One of them is the V-model, that defines different testing activities and maps these to development activities during the software development process (Rook 1990). Notwithstanding, software testing is challenging and several limitations exist. Software testing cannot show the absence of faults. Software testing can only show the presence of faults (Dahl et al. 1972). That is because the number of paths in the software

grows quickly with the number of nested loops and branch statements (Beizer 1990).

Software testing is difficult to perform partly because the complexity of software increases (Schütz 1994, Taipale & Smolander 2006). This problem is increased when the software is concurrent (Nilsson 2006).

2.2 Test issues for concurrent software

Concurrent software is a software containing at least two processes, that cooperate with each other to perform a given task (Andrews 2000, Iyengar & Debnath 1993). Concurrent software has become increasingly important in the past few years (Lu et al. 2007).

The inherent complexity of concurrent software makes them prone to faults. It is generally impossible to test concurrent software exhaustively because of the huge interleaving space (Gait 1986). The *interleaving space* is the total number of execution orders between processes (Lu et al. 2007). Concurrent software can behave anomalously due to, for example, synchronization faults. Synchronization faults are independent of the number of processors.

The presence of concurrent processes leads to race conditions. A *race condition* is a situation when at least two processes tries to access the same memory location or other shared resource, and at least one of them writes to that memory location (Balasundaram & Kennedy 1989). The outcome of race conditions is determined by several factors such as processor load, network traffic, non-determinism in the communication protocol and timing of events caring the race (Schütz 1994, McDowell & Helmbold 1989). Which process that wins the race is non-deterministic.

It is in general impossible to foretell the outcome of race conditions. Certain interleavings may, for instance, reveal synchronization faults, whereas most others cannot. Hence, concurrent software is not predictable with respect to its interleaving space. Changing one of the factors just mentioned, for example, the processor load, is enough in order to get a different outcome of such a race. Moreoever, a different outcome of a race condition may give a different system behavior (Schütz 1994). The non-deterministic behavior of concurrent software tends to make them more difficult to understand, write and debug, compared to sequential software (McDowell & Helmbold 1989).

Observability, reproducibility and controllability, as discussed in section 2.1, is harder to achieve for concurrent software in contrast to sequential software. Concurrent software introduce additional faults types that do not exist in sequential software.

Reproducibility is a common problem when testing concurrent software (Schütz 1990). The behavior of concurrent software is not reproducible by only repeatedly execute it with the same input values (Stone 1988).

Unfortunately, neither do test methods for sequential software focus on synchronization faults, nor do they address the problem of controllability.

Hence, test methods and guidelines for testing concurrent software must be provided.

The sequence of inputs, the schedule (scheme), and the interleaving space are necessary to control when testing concurrent software in order to guarantee reproducibility (McDowell & Helmbold 1989).

If the concurrent software has a race condition, it may execute with different interleavings each time. The fact that the input values are not enough to control the outcome of a race condition, means that the software is non-deterministic.

Deterministic execution can be achieved by using a *capture-and-replay* mechanism. Such a mechanism record the sequence of inputs and the interleaving space. Controllability over the scheduler is required in order to execute the software deterministically. Concurrent software must therefore satisfy *controllability*. A high degree of controllability is necessary to effectively test concurrent software (Nilsson 2006).

2.3 Synchronization faults

As mentioned, concurrency introduces new fault types that do not exist in sequential software. Hence, test methods must search and target additional fault types. Fault types that test methods for sequential software do not target, because they are not developed for it. A short description with common synchronization faults follows.

A critical section is a code section that requires *mutual exclusion*. That is, only one process may be executing that part of the code at a time. This is a classical concurrent software problem called *the critical section problem* (Katseff 1978). A process that wants to execute a critical section, must first obtain a locking object, for example, a semaphore. If another process is holding the locking object, then the first process will be blocked until the lock is available. These locks are like guards that manages the synchronization to the critical section and ensures exclusive access to it (Dijkstra 1965).

Two properties of a concurrent software must hold in order to satisfy correctness and consequently solve the critical section problem. These properties are safety and liveness (Lamport 1977). *Safety* states that nothing bad will happen during execution. Whereas *liveness* states that something good will eventually happen during execution of the software (Lamport 1977).

To satisfy safety and liveness, the properties depicted in figure 2 are necessary.

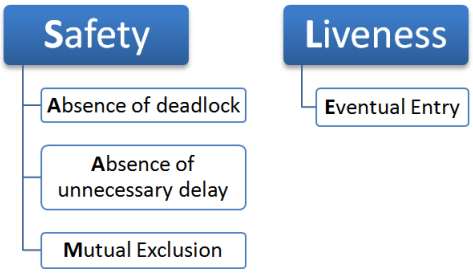

Figure 2: Necessary properties for a solution to the critical section problem

These properties are *global invariants*, that is, conditions that are true in all states the execution of the concurrent software may enter.

Absence of deadlock should be satisfied. *Deadlock* is a condition where two processes are waiting for each other to make progress, but neither of them does. In such a case, a circular wait occur and this condition is called a deadlock (Obermarck 1982).

Absence of *unnecessary delay* is when a process can enter its critical section without being prevented unnecessarily.

The mutual exclusion property has already been described above. These three properties must hold to satisfy safety. In order to satisfy liveness, only the *eventual entry* property must hold. That is, if process X want to use a shared resource, process X will eventually be able to do that sooner or later.

It should be noted that a concurrent software may not be correct by only satisfying safety and liveness. The concurrent software must satisfy these properties to be correct, but it may not be sufficient. Additional properties may be required or not depend on the concurrent software in question. Such properties can for instance be timeliness, maintainability, portability and usability. These properties are examples of quality properties. Which property that is most important to satisfy, depends on the type of concurrent software.

Consider a concurrent software that controls an *Automatic Teller Machine* (ATM). The most important property to satisfy in such a case apart from safety and liveness, may be security. Whereas the most important property for a control system for an aircraft may be safety.

Verifying that a concurrent software satisfyes safety and liveness is intricate, and is usually done by a formal method called model checking (Alba & Chicano 2008). *Model checking* is an automatic technique that can be used to verify the correctness of concurrent software (Clarke 1997). But verifying correctness is still intricate. Model checking cannot replace software testing, it must still be performed since they are complements to each other. That is because model checking utilizes a model that is based on assumptions of the software. Testing is required to test these assumptions in order to guarantee that the model is correct.

3 Problem definition

This section introduces the problem that this dissertation intend to investigate. Aim and objectives are presented together with methods for how to achieve them. A motivation of why the chosen problem is important to investigate is also presented. As well as delimitations and expected outcomes of this dissertation.

3.1 Problem description

Most software today are concurrent and the hardware industry moves towards concurrency by developing hardware that supports the utilization of concurrency more effectively (Olukotun & Hammond 2005). Most test methods do not target synchronization faults, because they are developed for sequential software. Thus, there is a need for test methods that address concurrent software. Unfortunately, few such test methods exists and their efficiency and effectiveness is uncertain. A reason for that is, that empirical studies and benchmarks are rare. Some industries therefore hesitates to use test methods for concurrent software (Bron et al. 2005).

A problem is that most test methods are not applicable on concurrent software. Hence, it is interesting to investigate the prerequisites to adapt sequential test methods to test concurrent software. What is required by a sequential test method to make it suitable to test concurrent software?

The problem statement is as follows:

Effective methods for testing concurrent software are missing. How can methods for testing concurrent software be developed by adapt methods used for testing sequential software?

3.2 Aim and objectives

The aim is as follows:

Investigate the adaptability of test methods, to test concurrent software.

Three objectives must be met in order to achieve the aim:

1. Survey test methods
2. Identify adaptability factors
3. Classify available test methods with respect to approaches

To be able to achieve each objective and hence the aim, several methods have been utilized.

Objective 1: Survey test methods

A literature survey is conducted over test methods to achieve the first objective. A benefit of conducting a literature survey is that an overview over state-of-the-art test methods is given. A drawback is that conducting a literature survey requires a lot of time. An alternative survey method to achieve the first objective is to interview companies.

Although the latter method is beneficial since information from current practice can be achieved, it suffers of an important problem. The problem is that there is a gap between state-of-the-art software testing, that is research, and what is used in practice (Bertolino 2004). Conducting a literature survey is hence the most beneficial method to achieve the first objective.

Objective 2: Identify adaptability factors

An adaptability analysis of test methods in the literature survey has been made to achieve the second objective.

Objective 3: Classify test methods with respect to adaptability factors

The third objective is achieved by classifying test methods. That is important to further refine the survey. The classification is a categorized collection of included test methods.

The classification is used as a basis for the analysis of what kind of sequential test methods that are candidates for an adaptation to test concurrent software. The survey shows that a group of test methods are good candidates to adaptation for testing concurrent software, since they all have common properties, that is criteria, which can be mapped for testing concurrent software.

3.3 Delimitations and Expected outcome

This dissertation do not intend to adapt a test method for sequential software, to test concurrent software. The intention of this dissertation is to provide knowledge that can be seen as guidelines, for how to adapt a test method for sequential software, to test concurrent software. Thus, the expected outcome of this disseratation are two artifacts:

1. A survey over state-of-the-art test methods for concurrent software
2. A set of adaptability factors that can be used to identify which test methods for sequential software that are most suitable to adapt for testing concurrent software

4 Survey: Test methods

A survey over test methods is provided in this section. It should be noted that this survey do not intend to cover all test methods for concurrent software, but to provide an overview of common ones.

4.1 Test methods for sequential software

According to Ammann & Offutt (2008), software can be represented by four type of models: Graph Coverage, Logic Coverage, Input Domain Model (IDM) and Syntactic structures. Model-based test methods are based on such models. Thus, test methods can be categorized with consideration to which model they are based on, if any. A C++ function, as depicted in figure 3, is used to demonstrate the usage of the four type of models.

```
private int getGreatest(int x, int y){
   if(x > y){
      std::cout<<"x is greatest"<<std::endl;
      return x;
   }
   else{
      std::cout<<"y is greatest"<<std::endl;
      return y;
   }
}
```

Figure 3: Function in C++

Graphs has been used for software testing since around 1970. They can be derived from, for example, control flow graphs, Finite State Machines (FSMs) and use cases. Node- and edge-coverage, respectively, are the two most fundamental criteria for graph coverage. Node coverage may be more known as statement coverage, whereas edge coverage correspond to branch coverage. Hence, node coverage aims to execute each statement at least once in sequential software. Similarly, satisfaction of edge coverage means that each branch in the sequential software has been executed at least once. Most other test methods based on graph coverage is an extension of node- or edge-coverage. Figure 4 depicts a graph over the function depicted in figure 3.

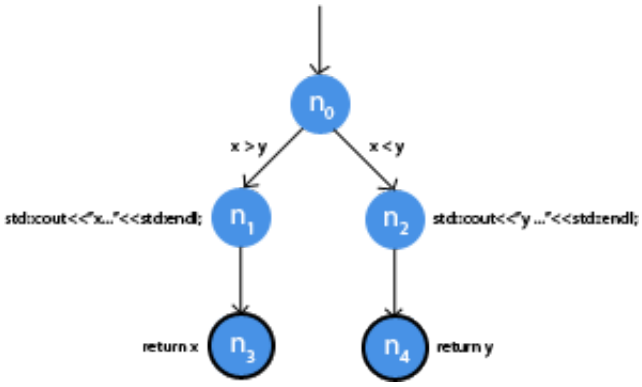

Figure 4: A graph

Around the middle of the 1970s, the research begun to focus on the correlation between a variables definition and its usage. Such a test method is *All-du-paths coverage*. Its intention is to cover the paths from all places in the implementation where a variable is defined, that is where the variable is on the left side of the assignment operator, to the place where it is used, that is, the variable is on the right side of the assignment operator or situations like x++.

Another type of model is *logical coverage*. Logical expressions can be derived from, for example, decision points in the software, statecharts and requirements. A logical model that describes the software's logic is constructed. Whereupon test methods can utilize the logical model to define test requirements.

If a condition can only be true when the software is in a specific state that will never happen, then the condition may never be true. That is called *dead-code*. Such faults can be revealed by logical coverage.

The logical expression of the function in figure 3 is depicted in figure 5.

$$(x > y)$$

Figure 5: Logical Expression

The next category of test models, *IDM*, do not require knowledge about the software's implementation. All possible values that the input parameters can have, is defined for the input domain when using input space partitioning. Input parameters can for instance be variables and parameters to functions in the software. After the input parameters are defined, the input domain is partitioned into regions. These regions are assumed to contain equally useful values from, which are selected from each region (Ammann & Offutt 2008). Each region is usually based on a characteristic of the software, its input or its environment. A *characteristic* is a property that an input parameter can have. For instance if the input parameter is a variable, it may be null, or if it is an array, it may or may not be sorted.

Two approaches of input domain modeling exists, namely a *interface-based* and a *functionality-based* approach. In the former approach, characteristics are identified from the input parameter from the software being tested. Whereas characteristics are identified from a functional or behavioral view of the software being tested when using the latter approach. When the tester has chosen one of these approaches and developed the IDM, then it is time to decide what combinations to use with help of several coverage criteria. A combination of values is then used to test the software.

It is for instance common in various software that a certain function must not take a negative value as argument to the function in question. In such a case, it is useful to feed the function with values that are near the boundary value, that is 0 (zero). Example of boundary values to feed the function in figure 3 is depicted in 6. These values are valid ones. Which values to use is determined by a boundary value analysis. It is desired to test both valid and invalid values.

$$x\{-1,\ 0,\ 999\}$$
$$y\{-1,\ 0,\ 999\}$$

Figure 6: An IDM

Another example is if the sequential software has a function that requires an integer as an argument. Then it is interesting to test to feed the function with for instance a string, a char or a boolean variable.

The workflow when using sequential test methods based on IDM usually consists of two steps. Step one is about partitioning the space of input values for example a function. That is constructing a model over the domain to test, not only the correct one. Whereas the second step is about *combination strategies*. That is considering several partitions at the same time, that is, decide from which combination of partitions [1] to choose values from (Ammann & Offutt 2008).

IDM test methods are important to utilize, because it is often the case that faults may be revealed by giving either invalid type of arguments to a function, or by testing boundary values of the input space.

[1] A set of values

The last category of test models aims to cover the *syntactic structure* of the software's implementation. Figure 7 depicts the syntactic structure of the function in figure 3.

```
private int getGreatest(int x, int y){
   if(x > y){
      std::cout<<"x is greatest"<<std::endl;
      return x;
      Δ return y;   // Embedded mutant
   }
   else{
      std::cout<<"y is greatest"<<std::endl;
      return y;
      Δ return y;   // Embedded mutant
   }
}
```

Figure 7: A syntactic structure

Mutation-based testing, that is regarded to be one of the high quality techniques, belong to this category. It is performed by creating an alternative implementation, that is, a *mutant*, of the software with help of a mutant operator. A *mutant operator* is for instance a modified operator or exchanged variable or a function call.

Static faults such as typing wrong type of operator in the implementation is common and may not always be easy to detect. That is, for instance as depicted in figure 7, where the mutants has exchange *"return x"* to *"return y"*, and vice versa.

4.2 Test methods for concurrent software

Test methods and papers are classified below, in order to grasp methods for software testing and to be able to lift the abstraction level up from detailed descriptions. It should be noted that the following classification gives an overview over test methods for concurrent software. But, it may be the case that papers can belong to several of these classes (e.g., require both reproducibility and reachability).

The classification together with descriptions of test methods under subsection 4.1 and 4.2, shows how selected papers are connected to each other. But also how they relates to the identified adaptability factors as described in section 5.

Each paper is labeled using a code system like [A,B,C]. Where A represents the model that the test method utilize (as depicted by models 1-4 in figure 8), B if the test method target synchronization faults or not (yes/no) and C, if the test method requires deterministic execution of the concurrent software or not (yes/no). It should be noted that B and C are not preselected adaptability factors, but identified ones from selected papers.

For example, a test method that is based on graphs, target synchronization faults and requires deterministic execution will get the label [2,Y,Y]. The label can be mapped to a specific area in figure 8, which depicts a Venn-diagram over all possible combinations.

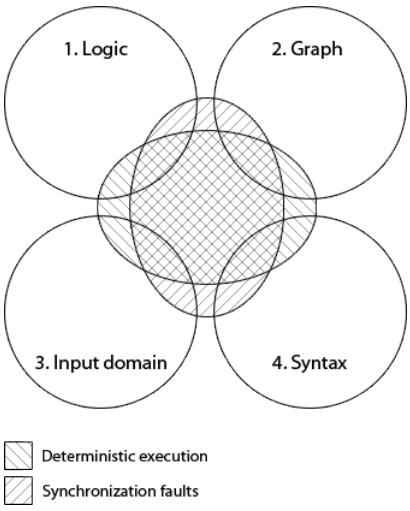

Figure 8: Venn-diagram over the classification after Birgitta Lindström (personal contact via email, 22nd of April, 2009)

It should be noted that this Venn-diagram is based on the classification of the four type of models that can represent a software as discussed in section 4.1. This classification made by Ammann & Offutt (2008) has been extended with the adaptability factors identified in this dissertation. Namely, that a test method for concurrent software must target synchronization faults and that the software must be reproducible by deterministic execution. These adaptability factors are discussed further in section 5.

The research community has taken different directions when it comes to testing of concurrent software. One property that most test methods for concurrent software have in common, is that they intend to target *synchronization faults*. That is issues in the communication between cooperating or competing processes or threads. Examples of such issues that can occur are deadlock and race condition.

As mentioned earlier, test methods tries to tackle these issues differently. Some test methods for concurrent software utilize the softwares implementation as an artifact and starting point (Carver & Tai 1989, Stoller 2002, Sen 2007). Whereas others are utilizing specifications (Chung et al. 1999, Yavuz-Kahveci & Bultan 2002, Tai & Carver 1994).

4.2.1 Reproducibility

All papers focus on revealing synchronization faults in the concurrent software. Most of them also requires deterministic execution of the concurrent software. That is, to make it reproducible. The concurrent software can be reproducible in several ways, which is demonstrated by the following papers. A majority of the papers that requires deterministic execution of the concurrent software under test, are not representing the concurrent software by a model as depicted in figure 8. Hence, these papers are not discussing test methods in that sense, but presents tools for how to achieve deterministic execution.

Richard H. Carver is a researcher in the field of software testing, has proposed a method adapted for concurrent software, based on mutation testing (Carver 1993) [-,Y,Y]. The method is called *Deterministic Execution Mutation Testing* (DEMT) and is a combination of *deterministic testing* and *mutation-based testing*. Deterministic testing is about controlling the interleavings of the concurrent software, so they can be foretold. In contrast, mutation-based testing is when mutants are inserted in the softwares implementation. A *mutant* is either an invalid string or a valid one that follows a different derivation from a preexisting string. DEMT is a general method that can be used for both testing and debugging of concurrent software. Both test- and debug tools has been developed within a constructed framework.

Deterministic execution of concurrent software has been researched already back in the 80's. Tai was early by addressing this issue on concurrent Ada software (Tai 1986) [-,Y,Y]. Tai utilized the *R_PERMIT* method to control the execution orders of concurrent Ada software.

A software P is said to own a single permit called R_PERMIT for having a rendezvous. A request for R_PERMIT must be made by each task in P that requests a rendezvous. The task must wait to receive the R_PERMIT before it can start the rendezvous. When R_PERMIT has been received and the rendezvous is started, R_PERMIT is released immediately. That is to make it possible for the caller for the next rendezvous to receive the R_PERMIT.

Tai continued the research together with Carver on deterministic execution of concurrent Ada software and released another paper on the topic (Carver & Tai 1989) [-,Y,Y]. The execution order of concurrent software can be reproduced with help of language-based *Synchronization Sequence* (SYN-sequence) replay tools. A SYN-sequence is for instance a P- and V-operation on a semaphore, a send and receive event on communication channels, for example, rendezvous. The synchronization sequence in the case of rendezvous is also called an *R-sequence*. Tai addressed the importance of SYN-sequences already in (Tai 1986), although it is called an *R-sequence* there since concurent Ada software uses rendezvous for communicating between tasks.

A language-based tool transforms the concurrent software into a slightly different version. Thereafter it is possible to control the execution of synchronization events in the original version. Also implementation-based replay tools exist, that may be more efficient, but they are not so popular according to Carver & Tai (1989). That is because it is difficult to port such tools to different concurrent software implementations (requires that the concurrent software is written in the same programming language) and is it also more difficult to develop implementation-based replay tools compared to language-based. An implementation-based replay tool transforms the implementation of the programming language that the concurrent software is built upon instead of the implementation of the concurrent software itself. This means that either the compiler, runtime environment or the operating system or all of them is modified.

In 1991, Carver and Tai proposed a test method called *Deterministic Execution Debugging and Testing* (Carver & Tai 1991) [-,Y,Y]. This method can be used to deterministically execute a concurrent software and hence reproduce a certain test case both during debugging and regression testing. Deterministic execution is done towards a given SYN-sequence. Reproducibility is performed by a language-based SYN-sequence replay tool similar to the one described in Carver & Tai (1989).

(Stone 1988) [2,Y,Y] presented a method called *speculative replay* that can reproduce the behavior of a concurrent software from the histories of its individual processes. Histories are divided into dependence blocks by using time dependencies between events in different processes. A *concurrency map* is used to visualize feasible concurrencies among processes.

Thus, Stone's method analyzes the software's execution in retrospect and reproduces the execution from observed events during the original software execution. The concurrency map is used both for visualization of concurrently executing processes, as well as a data structure used for the reproducibility process.

4.2.2 Reachability of graphs

When it comes to test methods, most papers are utilizing a representation of the concurrent software as graphs. Hence, it is nodes and edges that is to be covered (i.e., reached). The majority of these test methods still require that it is possible to execute the concurrent software deterministically.

Another direction taken in the research of testing concurrent software concerns reachability. Many test methods for sequential software is based on reachability, but not so many exist for concurrent software. However, different test methods focusing on reachability for concurrent software exist (Pu & Xu 2008, Lei & Carver 2006), they have one thing in common, that is, reachability. Reachability testing utilizes both non-deterministic and deterministic testing. Lei & Carver (2006) [-,Y,Y] among others, are focusing on reaching specified SYN-sequences in concurrent software.

Also Pu & Xu (2008) [2,Y,Y] focuses on SYN-sequences, but their approach is different since it aims to increase the feasibility of reachability testing. That is done by ensure that a *race variant* is always feasible. A race variant is a variant of a race condition.The *happens-before* relation between race receive events of synchronization pairs is determined by using vector timestamps. A happens-before relation defines a partial order over all instructions executed during a given execution. The happens-before relation is used to change the send partner of race receive events in proper order. When the send partner of race receive event r is changed, all events that occur after r needs to be removed in the original execution.

Reachability test methods often use graphs as a model. The artifact can be both implementation and specification.

Lei & Carver (2005) [-,Y,Y] has proposed an algorithm that does not save the history of synchronization sequences in order to guarantee that every partially-ordered sequence will be exercised exactly once. The algorithm creates a *race table* for a specific *Send-Recieve sequence* (SR-sequence) Q. A unique, partially-ordered race variant of Q is represented on each row of the race table. No analysis of the implementation is required to create a race table. The proposed reachability testing algorithm has been implemented in a reachability test tool called RichTest.

Lei et al. (2007) [2,Y,Y] has proposed a combinatorial testing strategy for concurrent software that is based on reachability testing. But their test method is more efficient since it is not as exhaustive as similar preceeding test methods. Previous methods are exhaustive because they intend to exercise all possible SYN-sequences of a concurrent software given input X. This means that exhaustive reachability testing derives race variants to cover all possible race outcome changes that can be made in a SYN-sequence. In contrast, race variants to cover all possible t-way combinations of the race outcome changes are derived using t-way testing. T-way reachability testing is therefore proposed. The fundamental framework of reachability testing is adopted, but the method only intend to exercise a subset of all SYN-sequences.

Not every race outcome contributes to every fault and many faults can be revealed by interactions between a small number of race outcomes. This hypothesis is the main idea behind the proposed testing strategy. Using reachability testing, it is not guaranteed that different SYN-sequences will be exercised during the execution of the concurrent software. But t-way reachability testing using the t-way testing stategy to guarantee that a different SYN-sequence is exercised each time during execution. T-way testing is a combinatorial testing strategy that selects input values for individual parameters and combines them for creating test cases. Reachability testing derives test sequences on-the-fly without constructing a static model.

Chung et al. (1999) [2,Y,Y] have proposed an approach based on graphs called Message Sequence Charts (MSCs). A *MSC* is a collection of sequencing constraints that restricts the execution behavior of the concurrent software. The interaction between processes may be represented by these graphs. Both non-deterministic and deterministic testing is utilized with the proposed approach.

Reza & Grant (2007) [2,Y,Y] has proposed an approach to integration testing of complex systems. The approach is based on both graphs and logic. That combination is not common. Their approach can be used with a model-oriented software architecture. Several models are used in a model-oriented software architecture approach, where each model is described by different notions. The benefit of such an approach is that each model can be used at the same of different level of abstractions. Architecture significant elements such as interfaces, is for instance described for each individual system by *Diagrammatic Syntactic Theory* (DST). Then, *Hierarchical Predicate Transition Net* (HPrTN [2]) is used to describe the structure and behavior of the concurrent software. Finally, reachability graphs are used by the approach. Several fault types can be revealed by this approach, such as deadlock. But also dead code can be found and invariant, can be checked.

Seo et al. (2006) [2,Y,Y] has proposed a similar approach based on graphs by using both reachability and statecharts. Sequencing constraints between events are analyzed in order to generate the test sequences.

Takahashi et al. (2008) [2,Y,?] proposed an approach for revealing race conditions. It uses the concurrent software's implementation as a starting point. The approach uses several kind of graphs: *Concurrent Module Flow Graph* (CMFG), *All Concurrent Path* (ACP) and *All Concurrent Binominal Path* (ACBP). These graphs which are extended for testing concurrent software, is used to test the concurrent software together with given coverage criteria.

Tai & Carver (1994) [2,Y,Y] have presented a method for how to accomplish coverage and detect violations of constraints written in *Constraints on Succeeding and Preceding Events* (CSPE). Both deterministic- and non-deterministic testing can be used.

Non-deterministic execution of a concurrent software P exercises a sequence of synchronization events (SYN-sequence). Restrictions on the allowed SYN-

[2]Logic

sequences of P are specified by sequencing constraints. The constraints do not have to be complete and they can be derived from formal- or informal specifications of P. Violations of P's constraints can be detected and coverage can be measured by collecting SYN-sequences during non-deterministic testing of P. SYN-sequences can be generated according to P's constraints and then used for deterministic testing of P.

Another approach that has been constructed from test methods for sequential software is the one proposed by (Taylor et al. 1992) [2,Y,Y]. This approach also uses the concurrent software's implementation as artifact and different kind of graphs as a model. The following coverage criteria have been defined: Concurrency State Coverage, State Transition Coverage and finally, Synchronization coverage.

An uncommon approach for utilizing a graphical representation of the concurrent software for testing has been proposed by Shousha et al. (2008) [2,Y,?]. Their approach is based on existing UML models of the concurrent software, but with concurrency properties added with help of a UML profile called SPT. The models are then used with a special genetic algorithm (GA) to detect deadlocks in concurrent software.

The GA is then fed with concurrency information from the SPT profile. Once that is done, the tailored GA automatically retrieve required information from the UML/SPT models and it is then used to detect deadlocks in the concurrent software. Sequence diagrams are particularly beneficial to use, since the stereotypes and tags that is used by the GA occurs in these models.

Also Wong et al. (2005) [2,Y,Y] uses a coverage-based approach by utilizing graph models of the concurrent software. The proposed approach is using white-box testing (structural testing) which often includes the construction of a reachability graph. The method generates SYN-sequences from the reachability graphs. Criteria such as all-node and all-edge coverage should then be satisfied. After the graph is constructed, and SYN-sequences generated, these SYN-sequences together with data input, is used to deterministic testing of the concurrent software.

Another test method for concurrent software that is somewhat similar to Takahashi et al. (2008) and Taylor et al. (1992) is the one proposed by Yang & Chung (1990) [2,Y,Y]. The test method focuses on revealing synchronization faults by utilizing graphs of the concurrent software. The method focus on states, transitions and flows among processes. Path analysis is used in the proposed method by Yang & Chung (1990). The execution behavior of concurrent software is modeled by using a flowgraph and a rendezvous graph.

4.2.3 Automate debugging

Researchers have also tried to automate debugging of concurrent software. Such a paper is the one written by Tzoref et al. (2007) [-,Y,Y].

They focus on a debugging technique that is able to pinpoint locations in concurrent software that are in the vicinity of faults. The approach utilizes two search algorithms to make this possible. Noise can also be inserted in the implementation in order to discover where faults exist. *Noise* are scheduling-modifying statements such as the sleep() method in Java. A noise may hence cause a context switch. When a fault has been revealed, automatic debugging aims to find a minimal subset of the changes required to produce the fault. This approach uses both the syntactic structures and graphs, namely, control flow graphs of the concurrent software.

4.2.4 Analysis techniques

Chen & MacDonald (2008) [2,Y,Y] has proposed a test method that allows tighter collaboration between static- and dynamic analysis of concurrent software. Course-grained analysis is used in static analysis to guide the dynamic analysis to concentrate on a relevant search space. Whereas runtime information is collected during the guided exploration with dynamic analysis.

Another distinction is that faults can be revealed without executing the software with static analysis. Hence, the non-determinism issue is avoided. But when using dynamic analysis, the software is executed directly and faults may be revealed at runtime. Dynamic analysis tries to trigger different interleavings in order to reveal faults. That may be done by inserting so called sleep statements randomly in the implementation. Different interleavings can also be achieved by using an explicit-state model checker that can explore all interleavings systematically.

Thus, static analysis performs the initial part in the search space, whereas dynamic analysis then refines the remaining search space with information gained from the static analysis. Testing a distinct partial order more than once is avoided by using static analysis to guide dynamic analysis.

Another common analysis technique for sequential software is software slicing. *Software slicing* can be used to understand, test and debug software. A static software slice focus on values that are stored in a given variable at a given time. Those parts in the concurrent software that are irrelevant to the chosen values X at given time Y are simply deleted.

Nanda & Ramesh (2000) [2,Y,?] extends the technique for concurrent software by adding support for shared memory, interleavings and mutual exclusion. Their proposed software slicing algorithm utilizes an abstract representation of the concurrent software in form of two type of graphs. A *Threaded Control Flow Graph* (TCFG), which is an extension of a *Control Flow Graph* (CFG) including one node a cobegin and one at coend, and a *Threaded Program Dependence Graph* (TPDG). The TPDG is an extension

to a *PDG*, which captures data and control dependencies between nodes in a single thread. Given these graphs over the concurrent software, slicing criteria are identified as nodes in these graphs. The slicing algorithm then aim to cover these, in order to define and compute more accurate software slices.

Chen & Xu (2001) [2,Y,Y] has also proposed an approach for software slicing. But they focus on concurrent Java software. An efficient static slicing algorithm creates the software slices by using two type of graphs, namely a *Concurrent Control Flow Graph* (CCFG) as well as a *Concurrent Program Dependence Graph* (CPDG). A CCFG represents the synchronization among communicating threads, whereas a CPDG represents dependencies of the concurrent software. Their proposed approach is different from previous ones, because they e.g., do not have sufficiently strong definitions for the graphs to be able to represent all possible software constructs.

4.2.5 Specific synchronization faults

Most papers focus on synchronization faults in general, but papers focusing on specific synchronization faults also exist. These papers mostly focus on deadlock, livelock and race conditions.

Pugh & Ayewah (2007) [-,Y,Y] has taken another direction and focused on revealing faults causing deadlock and livelock, respectively, in concurrent software. A Java-based framework called MultithreadedTC is proposed that allow construction of deterministic unit tests that exercise specific interleavings in concurrent Java software. Common concurrent programming constructs can be tested to see if they work as intended. It is for instance possible to test if locking/unlocking mechanisms used as guards around critical sections.

MultithreadedTC is developed to work well with JUnit [3] Olan (2003). It has partly been inspired by ConAn (Long et al. 2003), a script-based concurrent test framework that also uses a clock to synchronize activities among several threads. MutlithreadedTC uses an external clock on several communicating threads. The clock executes in a separate thread as a daemon that periodically check the status of all threads. If all threads are blocked and at least one thread is waiting for a tick, the clock advances to the next desired tick.

Coffman et al. (1971) mention three approaches for how to deal with deadlocks:

1. Detection and recovery
2. Avoidance
3. Prevention

[3] http://www.junit.org

Kameda (1980) [2,Y,?] early focused on how it is possible to detect deadlock in concurrent software. More specifically, he considered whether or not a concurrent software needs any deadlock avoidance or prevention overhead. Kameda's algorithm can help to reduce the deadlock avoidance overhead if a concurrent software not satisfies absence of deadlock. The algorithm can find a deadlock state and thereby indicating which resources could be involved in a deadlock. However, such a test do not give much direction as how to design a concurrent software that satisfies absence of deadlock.

Another paper focusing on graphs and revealing synchronization faults such as deadlock and race conditions, is (Chen & MacDonald 2007) [2,Y,Y]. The proposed method generates and tests so called *value schedules* of concurrent Java software. Value schedules are read-write assignment statement sequences, also known as *def-use pairs*.

4.2.6 Random testing

Random testing in general is not one of the high quality test methods for software testing. Although, random testing for concurrent software has not been studied so much. One researcher that has focused on this topic recently is Sen (2007, 2008).

In Sen (2007) [2,Y,Y], he proposed an approach for concurrent software based on random testing, that target synchronization faults such as deadlock and race conditions. The proposed approach utilizes ideas from traditional model checking, where partial order reduction is used to minimize the state space explosion problem. Some interleavings in a concurrent software are equivalent to each other since they correspond to different execution orders of various non-interacting or independent instructions from concurrent threads. This fact is exploited by partial order reduction methods. Thus, if a given execution reveals a fault such as deadlock or race condition, then all equivalent execution orders will also reveal that fault. These equivalent interleavings are described in terms of *happens-before* relations. The approach proposed by Sen (2007) performs random testing by choosing thread schedules at random.

In Sen (2008) [-,Y,Y], he continued the research on random testing for concurrent software, but in a slightly different direction. Deterministic execution and the happens-before relation is still important, but the proposed method is now solely concentrated on race conditions. Although, it can be modified for other synchronization faults.

Information about potential race conditions is obtained from an analysis tool to separate real race conditions from false ones. The method has been implemented as an algorithm in a testing and debugging tool called RACEFUZZER. Thus, the tool uses this obtained information about potential race conditions to control a random scheduler of threads, whereupon the real race condition is created with a very high probability. These race conditions are randomly resolved during runtime.

The proposed method is called *race-directed random testing*, and it combines detection of race conditions by utilizing a randomized thread scheduler in order to find real race conditions in a concurrent software. Race conditions are detected by using a an imprecise race detection technique such as *hybrid dynamic race detection*. It computes a set of pairs of software statements that could potentially race in a concurrent execution. RACEFUZZER then executes the concurrent software with a random schedule for each pair in the set, also called *a racing pair of statements*.

Stoller touched the area about random testing of concurrent software already in Stoller (2002) [-,Y,Y]. He proposed a method that uses ideas from model checking, but this method is less systematic and more scalable in constrast to a model checker. Invocations to a *scheduling function* is inserted at selected points in the software's implementation. This scheduling function either causes a context switch or does nothing.

The transformed software is then executed repeatedly in order to test it. If a fault is found given a certain *seed*, then the software is re-executed with the same seed. The seed is usually based on the current time. Reproducibility is important, because it is desired to reproduce the same case. However, this method does not guarantee reproducibility, although it is likely. A *capture and replay mechanism* is required in order to guarantee reproducibility. That is to record the execution order of the concurrent software and then be able to replay exact the same order. Randomness and heuristics that weight the choises are combined in more sophisticated scheduling functions. The proposed method has been implemented in a tool called *Random Scheduling Test* (rstest). Thus, as opposed from general random testing approaches, the proposed one randomizes over the execution orders, rather than input values.

4.2.7 Formal methods

Yavuz-Kahveci & Bultan (2002) [2,Y,?] has proposed a novel method to validate concurrent software. The method work as follows:

1. The concurrency control component of the concurrent software is formally specified
2. The formal specification is automatically verified by an infinite model checker
3. The implementation for the concurrency control component is automatically generated

Traditional validation techniques such as software testing is not efficient for concurrent software due to the state space exploration issue. The state space of a concurrent software increases exponentially with the number of variables and concurrent processes.

Another formal method is proposed by Carver & Durham (1995) [2,Y,Y]. Their formal method is based on process algebra and modal logic that are combined with a practical method for testing concurrent software. Formal methods are integrated, since the verification is often not carried down to the concrete implementation level. In this method, a coverage technique is used that applies whitebox testing techniques with deterministic testing.

The abstract software models the valid behavior of a concurrent software without describing any implementation mechanisms that achieve this behavior. A software's valid behavior can be modeled as the possible *sequence of events* that may be observed of a conforming concrete implementation of the abstract software. This method tests if identified constraints of the implementation is *covered* or *violated* during deterministic execution.

4.2.8 Tool support

Finally, some papers provide tool support for testing concurrent software. Depending on the tool, it may or may not utilize a model as the representation of the concurrent software.

Tool support is provided for some approaches in order to make them more usable and efficient. That is the case with Long et al. (2003) [-,Y,Y] tool that uses an extended approach for testing Java monitors. The tool called *Concurrency Analyzer* (ConAn) supports unit testing of concurrent Java components. ConAn generates drivers that are used for testing monitors in Java. First, the driver automatically executes the statements in the test case in a prescribed order. Second, ConAn automatically analyzes the test output by compare it against the expected output specified in the test case.

Hessel & Pettersson (2007) [2,Y,Y] has written a paper on a test-case generation tool called *Cover* [4] for real-time systems. Test suites are automatically generated by using timed automata models created with the model checker *Uppaal* [5].

A model consisting of a controller- and an environmental part, is constructed of the concurrent software. The behavior of the concurrent software is specified by the controller part. Whereas the components surrounding the controller are specified by the environmental part. Coverage criteria is then described by an *observer language*. An observer is a monitoring automaton that formally describes a coverage criteria.

Generated test suites [6] are then automatically executed by a test driver. Cover generates test cases by performing state space exploration on-the-fly with reachability analysis of the timed automata. Coverage information is combined with the state space. The concurrent software is hence tested with aspect to specified coverage criteria.

[4] http://www.uppaal.org/cover

[5] http://www.uppaal.org

[6] A set of test cases

5 Adaptability factors

This section presents the analysis of the result from section 4, namely the identified adaptability factors. It should be noted that it may exist additional adaptability factors important to consider when adapting a sequential test method to concurrent software.

Adaptability factors has been identified with the intention to aid the development of novel test methods for concurrent software. These adaptability factors should guide to identify which sequential test methods are best candidates to adapt for testing concurrent software, considering the adaptability factors. Because, the adaptability factors are something that a test method for concurrent software should satisfy, it is also the case that these adaptability factors can be used when develop a novel test method for concurrent software from scratch. It should be noted that these adaptability factors can be seen as features that a test method for concurrent software should have, but a test method for sequential software do not have. Thus, when adapting a sequential test method for concurrent software, these adaptability factors are not present in the selected test method for sequential software.

By analyzing the result from the survey in section 4, constantly returning patterns between papers written in the field of concurrent software testing have been identified. Most test methods for concurrent software are based on orders of events with consideration to the synchronization.

Most papers aims to target *synchronization faults* in concurrent software. Hence, it is an indication towards an adaptability factor. Focus on synchronization faults is also reasonable to be an adaptability factor, because it is the kind of fault type introduced by concurrent software. Thus, it is required that a test method for concurrent software target synchronization faults.

Synchronization faults are likely to occur near SYN-sequences, as discussed in section 4.2. Consider the case when the synchronization between a P- and V operation on a semaphore is incorrectly performed. Then, an error is likely to occur. Test methods for concurrent software should focus on such areas in order to reveal synchronization faults.

There may be other kind of faults in concurrent software that do not belong to synchronization faults, but these faults are essential here.

Targeting synchronization faults, as an adaptability factor has been depicted as a vertical ellipse in figure 9.

Another adaptability factor that has been identified is *deterministic execution*. In order to provide reproducibility as described in section 2.2, deterministic execution of the concurrent software under test is required. If the concurrent software is not reproducible, then it is difficult to perform regression testing (described in section 2.2). That is because different interleavings between processes will occur and it will be most unlikely to get exactly the same interleaving if the concurrent software is repeatedly executed. That is a problem because if a fault is revealed and then removed, it will be impos-

sible to check if the fault has been removed, and that additional faults has not been introduced.

Deterministic execution has been depicted as a horizontal ellipse in figure 9.

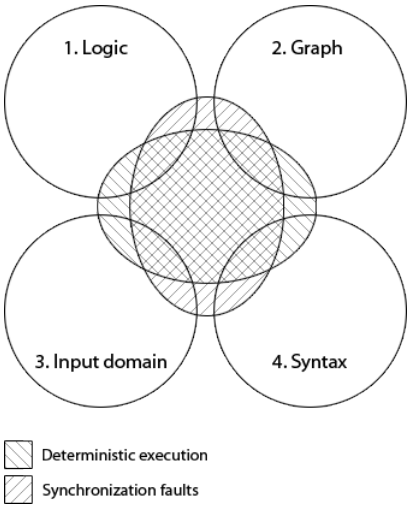

Figure 9: Venn-diagram with adaptability factors after Birgitta Lindström (personal contact via email, 22nd of April, 2009)

6 Related work

This section intend to position this dissertation in the context of others work in the field of software testing. Similarities and differences between this and other studies has been made with consideration to the details of the problem, the approach and the results.

Omar & Mohammed (1991) has conducted a survey over sequential test methods based on functional testing. They point out that a black-box technique can not reveal faults contained in a function that is not explicitly described in the software's specification.

Their survey consider both black- and white-box techniques for sequential software. Their description of test methods are not as detailed as given by this thesis, because they have not described specific test methods. Adaptability of test methods are not considered at all. The conclusion of their study is that black- and white-box techniques should be utilized complementary to be able to reveal all faults in a software.

Another survey conducted by Yang et al. (2006) focus on comparing 17 coverage-based test tools that are not restricted to coverage measurement. Additional features such as software prioritization for testing, assistance in debugging, automatic generation of test cases and customization of test reports are also considered. The intention of their study was first, to understand coverage-based test tools and second, to compare them with an in-house developed test tool called *eXVantage*.

The results from their study shows that each test tool seem to be tailored to a specific software domain given their unique features. Their survey can hence be used to guide the selection of which coverage-based test tool is most suited for a specific software domain. But the adaptability of these test tools has not been considered. Their focus has not been to investigate if and how these test tools can be tailored to suit additional domains or fault types.

Examples of researchers that have adapted a test method for sequential software to test concurrent software also exist. Such an example is conducted by Yang et al. (1998). The goal of their study was to demonstrate that a test method developed for sequential software can be used to test concurrent software if an adaptation has been made. They have developed an algorithm that can find all-du-paths for shared memory concurrent software. A test tool called Delaware Parallel Software Testing Aid (Della Pasta) for concurrent software has been developed to demonstrate the effectiveness and usefulness of their techniques.

They have focused on adapting a specific test method to test concurrent software in their study, but not investigating adaptability in general. Hence, there is a major difference between their study and this dissertation.

7 Conclusions

This section starts by concluding the dissertation, followed by arguments for the reached aim. This dissertation is then related to others work in the field. Arguments for the contributions of this dissertation follows as well as proposed future work.

Papers and books written in the field of sequential- as well as concurrent software testing has been read in order to obtain knowledge about test methods. This has been used as a basis for the literature survey. The intention of conducting a survey about test methods was to discover constantly returning patterns, similarities and differences between them. Directions taken in the research have also been identified. An initial hypothesis was that there should be common factors that test methods must have in order to test concurrent software. Adaptability factors have been identified in section 5 from the survey where test methods have been classified in section 4.

The aim of this dissertation, as defined in section 3.1, has therefore been accomplished.

7.1 Research contribution

The first two outcomes from this dissertation as described in section 3.3 was expected, but an additional one has also been achieved. That is identified open research areas. These *open research areas*, as depicted in figure 10 (the grey areas), are areas in the research where there is a lack (either completely or only a limited amount exist) of test methods for concurrent software. Thus, the result from this dissertation has not only identified what has been done in the research and where, but also what has *not* been done and within which areas. That is also an important contribution, because a lot of future work can be identified there.

A survey over state-of-the-art test methods have been conducted in order to achieve the first outcome. Whereas the second outcome has been achieved by identifying a set of adaptability factors from the survey as basis. Although a survey over test methods has been conducted previously, as discussed under section 6, such a survey like in this dissertation that focus on test methods for concurrent software, has not yet been conducted. Especially not where the focus is on how adaptable a test method is to test concurrent software.

An additional unexpected contribution has also been achieved as just discussed, that is the identification of open research areas. An important outcome from this dissertation is hence not only information of what has been done in the field of testing concurrent software, but also what has not been done.

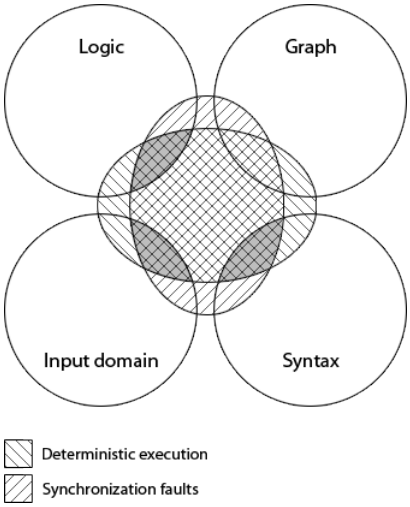

Figure 10: Venn-diagram with open research areas after Birgitta Lindström (personal contact via email, 22nd of April, 2009)

7.2 Future work

Problem domains worth to investigate further have been identified in this dissertation. Proposed future work is hence presented.

The first proposed future work is about investigating what is required to take a test method for sequential software outside the fields of the identified adaptability factors as depicted in figure 10, and move it inside either only one- or both of the identified adaptability factors (ellipses). It may not be sufficient to only add the identified adaptability factors, because sequential test methods may have other properties that make one test method harder to adapt than another.

The second proposed future work is about selecting a test method adaptable to test concurrent software with consideration of the adaptability factors, as discussed in section 5, and adapt it. An analysis should then be performed that discuss the final result. The expected outcome is an adapted test method, that is, a novel test method for testing concurrent software.

The second proposed future work is built upon the first one above. This one is about using the adapted test method to test a concurrent software. Thus, to empirically show whether or not the adapted test method works in practice. It is actually desired to test it on several kind of concurrent software in order to show that the test method can reveal faults in for example, concurrent software in various domains and sizes.

The expected outcome for this proposal is an empirical study. The intention of conducting such a study is to empirically prove that the adapted test method works in practice. Different factors can be considered here. Interesting ones can for instance be to investigate if the test method is easy to use? If it is general? That is, can it be applied on software from various domains? Does it have high requirements on controllability or observability?

References

Adrion, W. R., Branstad, M. A. & Cherniavsky, J. C. (1982), 'Validation, verification, and testing of computer software', *ACM Comput. Surv.* **14**(2), 159–192.

Alba, E. & Chicano, F. (2008), Searching for liveness property violations in concurrent systems with aco, *in* 'GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation', ACM, New York, NY, USA, pp. 1727–1734.

Ammann, P. & Offutt, J. (2008), *Introduction to Software Testing*, Cambridge University Press. ISBN: 978-0-521-88038-1.

Andrews, G. (2000), 'Foundations of multithreaded, parallel, and distributed programming', *University of Arizona, Addison-Wesley* .

Balasundaram, V. & Kennedy, K. (1989), Compile-time detection of race conditions in a parallel program, *in* 'ICS '89: Proceedings of the 3rd international conference on Supercomputing', ACM, New York, NY, USA, pp. 175–185.

Beizer, B. (1990), *Software Testing Techniques*, Van Nostrand Reinhold. ISBN: 0-442-20672-0.

Bertolino, A. (2004), 'The (im)maturity level of software testing', *SIGSOFT Softw. Eng. Notes* **29**(5), 1–4.

Bron, A., Farchi, E., Magid, Y., Nir, Y. & Ur, S. (2005), Applications of synchronization coverage, *in* 'Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming', Chicago, IL, USA.

Carver, R. H. (1993), Mutation-based testing of concurrent programs, *in* 'Proceedings of the International Test Conference', IEEE Computer Society, Baltimore, MD, USA, pp. 845–853.

Carver, R. H. & Durham, R. (1995), Integrating formal methods and testing for concurrent programs, *in* 'ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering', IEEE Computer Society, Washington, DC, USA, pp. 421–425.

Carver, R. H. & Tai, K.-C. (1991), 'Replay and testing for concurrent programs', *IEEE Software* **8**(2), 66–74.

Carver, R. & Tai, K. C. (1989), Deterministic execution testing of concurrent ada programs, *in* 'TRI-Ada '89: Proceedings of the conference on Tri-Ada '89', ACM, New York, NY, USA, pp. 528–544.

Chen, J. & MacDonald, S. (2007), Testing concurrent programs using value schedules, *in* 'ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering', ACM, New York, NY, USA, pp. 313–322.

Chen, J. & MacDonald, S. (2008), Towards a better collaboration of static and dynamic analyses for testing concurrent programs, *in* 'PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems', ACM, New York, NY, USA, pp. 1–9.

Chen, Z. & Xu, B. (2001), 'Slicing concurrent java programs', *SIGPLAN Not.* **36**(4), 41–47.

Chung, I. S., Kim, H. S., Bae, H. S., Kwon, Y. R. & Lee, B. S. (1999), Testing of concurrent programs based on message sequence charts, *in* 'PDSE '99: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems', IEEE Computer Society, Washington, DC, USA, p. 72.

Clarke, E. M. (1997), 'Model checking', *Lecture Notes in Computer Science* **1346**, 54–??
URL: *citeseer.ist.psu.edu/clarke00model.html*

Coffman, E. G., Elphick, M. & Shoshani, A. (1971), 'System deadlocks', *ACM Comput. Surv.* **3**(2), 67–78.

Dahl, O.-J., Dijkstra, E. W. & C.A.R., H. (1972), *Structured Programming*, London: Academic Press. ISBN: 0-12-200550-3.

Dijkstra, E. W. (1965), 'Solution of a problem in concurrent programming control', *Commun. ACM* **8**(9), 569.

Gait, J. (1986), 'A probe effect in concurrent programs', *Software Practice and Experience* **16(3)**, 225–233.

Hessel, A. & Pettersson, P. (2007), Cover - a test-case generation tool for timed systems, *in* A. Petrenko, M. Veanes, J. Tretmans & W. Grieskamp, eds, 'Testing of Software and Communicating Systems: Work-in-Progress and Position Papers, Tool Demonstrations, and Tutorial Abstracts of TestCom/FATES 2007', pp. 31–34.

Iyengar, S. R. & Debnath, N. C. (1993), A general abstract representation for the study of concurrent programs, *in* 'Proceedings of the 1993 ACM conference on Computer science', Indianapolis, Indiana, United States.

Kameda, T. (1980), 'Testing deadlock-freedom of computer systems', *J. ACM* **27**(2), 270–280.

Katseff, H. P. (1978), A new solution to the critical section problem, *in* 'STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing', ACM, New York, NY, USA, pp. 86–88.

Lamport, L. (1977), 'Proving the correctness of multiprocess programs', *IEEE Trans. Softw. Eng.* **3**(2), 125–143.

Lei, Y. & Carver, R. (2005), A new algorithm for reachability testing of concurrent programs, *in* 'ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering', IEEE Computer Society, Washington, DC, USA, pp. 346–355.

Lei, Y. & Carver, R. (2006), 'Reachability testing of concurrent programs', *IEEE Transactions on Software Engineering* **32**(6), 382–403.

Lei, Y., Carver, R. H., Kacker, R. & Kung, D. (2007), 'A combinatorial testing strategy for concurrent programs', *Softw. Test. Verif. Reliab.* **17**(4), 207–225.

Lindström, B. (2009), Testability of Dynamic Real-Time Systems, PhD thesis, University of Skövde. Dissertation No. 1241.

Long, B., Hoffman, D. & Strooper, P. (2001), A concurrency test tool for java monitors, *in* 'COMPASS '95: Proceedings of the 10th Annual Conference on Systems Integrity, Software safety and Process Security', IEEE Computer Society, Gaithersburg, MD, USA, pp. 25–33.

Long, B., Hoffman, D. & Strooper, P. (2003), 'Tool support for testing concurrent java components', *IEEE Transactions on Software Engineering* **29**(6), 555–566.

Lu, S., Jiang, W. & Zhou, Y. (2007), A study of interleaving coverage criteria, *in* 'Proceedings of the 6th joint meeting of the European Software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering', Dubrovnik, Croatia.

Lu, S., Park, S., Seo, E. & Zhou, Y. (2008), 'Learning from mistakes: a comprehensive study on real world concurrency bug characteristics', *SIGARCH Comput. Archit. News* **36**(1), 329–339.

McDowell, C. E. & Helmbold, D. P. (1989), 'Debugging concurrent programs', *ACM Computing Surveys* **21**(4), 593–622.

Morell, L. J. (1988), Theoretical insights into fault-based testing, IEEE Computer Society Press, Banff, Alberta, pp. 45–62.

Nanda, M. G. & Ramesh, S. (2000), Slicing concurrent programs, *in* 'ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis', ACM, New York, NY, USA, pp. 180–190.

Nilsson, R. (2006), A Mutation-based Framework for Automated Testing of Timeliness, PhD thesis, University of Skövde. Dissertation No. 1030.

Obermarck, R. (1982), 'Distributed deadlock detection algorithm', *ACM Trans. Database Syst.* **7**(2), 187–208.

Olan, M. (2003), 'Unit testing: test early, test often', *J. Comput. Small Coll.* **19**(2), 319–328.

Olukotun, K. & Hammond, L. (2005), 'The future of microprocessors', *Queue* **3**(7), 26–29.

Omar, A. A. & Mohammed, F. A. (1991), 'A survey of software functional testing methods', *SIGSOFT Softw. Eng. Notes* **16**(2), 75–82.

Pu, F. & Xu, H.-Y. (2008), A feasible strategy for reachability testing of internet-based concurrent programs, *in* 'ICNSC '08: Proceedings of the International Conference on Networking, Sensing and Control', IEEE, pp. 1559–1564.

Pugh, W. & Ayewah, N. (2007), Unit testing concurrent software, *in* 'ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering', ACM, New York, NY, USA, pp. 513–516.

Reza, H. & Grant, E. S. (2007), 'A method to test concurrent systems using architectural specification', *J. Supercomput.* **39**(3), 347–357.

Rook, P. (1990), *Software Reliability Handbook*, Elsevier Science Inc., New York, NY, USA.

Schütz, W. (1990), A test strategy for the distributed real-time system mars, *in* 'Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering', Tel-Aviv, Israel.

Schütz, W. (1994), 'Fundamental issues in testing distributed real-time systems', *Real-Time Systems* **7(2)**, 129–157.

Sen, K. (2007), Effective random testing of concurrent programs, *in* 'ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering', ACM, New York, NY, USA, pp. 323–332.

Sen, K. (2008), 'Race directed random testing of concurrent programs', *SIGPLAN Not.* **43**(6), 11–21.

Seo, H.-S., Chung, I. S. & Kwon, Y. R. (2006), 'Generating test sequences from statecharts for concurrent program testing', *IEICE - Trans. Inf. Syst.* **E89-D**(4), 1459–1469.

Shousha, M., Briand, L. & Labiche, Y. (2008), A uml/spt model analysis methodology for concurrent systems based on genetic algorithms, *in* 'MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems', Springer-Verlag, Berlin, Heidelberg, pp. 475–489.

Stoller, S. D. (2002), Testing concurrent Java programs using randomized scheduling, *in* 'Proc. Second Workshop on Runtime Verification (RV)', Vol. 70(4) of *Electronic Notes in Theoretical Computer Science*, Elsevier.

Stone, J. M. (1988), Debugging concurrent processes: a case study, *in* 'Proceedings of the SIGPLAN'88 Conference on Programming Language and Implementation', Atlanta, Georgia, United States.

Tai, K.-C. (1986), An approach to testing concurrent ada programs, *in* 'WADAS '86: Proceedings of the third annual Washington Ada symposium on Ada', ACM, New York, NY, USA, pp. 253–264.

Tai, K.-C. (1989), Testing of concurrent software, *in* 'COMPSAC '89: Proceedings of the 13th Annual International Conference on Computer Software and Applications Conference', IEEE Computer Society, Orlando, FL, USA, pp. 62–64.

Tai, K.-C. & Carver, R. H. (1994), Use of sequencing constraints for specifying, testing, and debugging concurrent programs, *in* 'Proceedings of the 1994 International Conference on Parallel and Distributed Systems', IEEE Computer Society, Washington, DC, USA, pp. 280–287.

Taipale, O. & Smolander, K. (2006), Improving software testing by observing practice, *in* 'ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering', ACM, New York, NY, USA, pp. 262–271.

Takahashi, J., Kojima, H. & Furukawa, Z. (2008), Coverage based testing for concurrent software, *in* 'ICDCS '08: The 28th International Conference on Distributed Computing Systems Workshops', IEEE Computer Society, Beijing, China, pp. 533–538.

Taylor, R. N., Levine, D. L. & Kelly, C. D. (1992), 'Structural testing of concurrent programs', *IEEE Trans. Softw. Eng.* **18**(3), 206–215.

Tzoref, R., Ur, S. & Yom-Tov, E. (2007), Instrumenting where it hurts: an automatic concurrent debugging technique, *in* 'ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis', ACM, New York, NY, USA, pp. 27–38.

Wong, E. W., Lei, Y. & Ma, X. (2005), Effective generation of test sequences for structural testing of concurrent programs, *in* 'ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems', IEEE Computer Society, Washington, DC, USA, pp. 539–548.

Yang, C.-S. D., Souter, A. L. & Pollock, L. L. (1998), All-du-path coverage for parallel programs, *in* 'ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis', ACM, New York, NY, USA, pp. 153–162.

Yang, Q., Li, J. J. & Weiss, D. (2006), A survey of coverage based testing tools, *in* 'AST '06: Proceedings of the 2006 international workshop on Automation of software test', ACM, New York, NY, USA, pp. 99–103.

Yang, R.-D. & Chung, C.-G. (1990), A path analysis approach to concurrent program testing, *in* 'Proceedings of the 9th Annual International Phoenix Conference on Computers and Communications', IEEE Computer Society, Scottsdale, AZ, USA, pp. 425–432.

Yavuz-Kahveci, T. & Bultan, T. (2002), Specification, verification, and synthesis of concurrency control components, *in* 'ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis', ACM, New York, NY, USA, pp. 169–179.

A Short summary of selected papers

This appendix gives a short summary of each paper mentioned in the survey in section 4.

1. A Concurrency Test Tool for Java Monitors
Long et al. (2001)

A tool called *Concurrency Analyser* (ConAn) is presented in the paper. ConAn is a tool that generates drivers which are used for testing Java monitors. ConAn is a tool that utilizes an extended method for testing monitors.

First, the tester identifies a set of preconditions that will satisfy loop coverage (branch coverage in some cases) for each monitor operation. Loop coverage is desired to achieve since Java monitors usually contain while-conditions. *Second*, a sequence of monitor invocations is constructed by the tester. Each of these invocations will exercise each operation under all its preconditions. *Third*, the tester specifies the sequence of monitor invocations and the threads that will invoke them. ConAn then automatically generates the test driver (source code) that controls the synchronization between the threads through a clock. The test driver then compares the outputs from the execution against the expected outputs specified in the test case. Processes are synchronized with a clock (for testing only) to increase controllability. *Fourth*, the test software is executed and its output is compared against the specified expected output.

The driver automatically executes the statements in the test case in a prescribed order, and then compares the output against the expected output specified in the test case.

2. Integrating Formal Methods and Testing for Concurrent Programs
Carver & Durham (1995)

This paper presents a method to select test sequences from an formal specification to test its concrete implementation. Formal methods based on process algebra and modal logic are combined with a practical method for testing concurrent software. Formal methods are integrated, since the verification is often not carried down to the concrete implementation level. In this method, a coverage technique is used that applies whitebox testing techniques with deterministic testing.

The abstract software models the valid behavior of a concurrent software without describing any implementation mechanisms that achieve this behavior. A software's valid behavior can be modeled as the possible *sequence of events* that may be observed of a conforming concrete implementation of the abstract software. This method tests if identified constraints of the implementation is *covered* or *violated* during deterministic execution.

Deterministic testing of an implementation P against its abstract software A is performed in the following three steps:

1. A set of tests is selected with the form (X,S), where X is an input and S is an event sequence. Tests may be selected from both A and P.

2. A deterministic execution of P according to X and S is performed for each test (X,S). Whether S is allowed by P with input X is determined by the deterministic execution.

3. Finally, analyze the results from the deterministic execution for conformance to A.

Test selection should be guided by information derived from the abstract software.

Conducted experiments demonstrated that derived constraints from an abstract software provides efficient guidance for test sequence generation.

3. Mutation-based Testing of Concurrent Programs
Carver (1993)

A general method for how to test and debug concurrent software is presented. The method is called *Deterministic Execution Mutation Testing* (DEMT). DEMT is a combination of deterministic testing and mutation-based testing.

A framework within which the issues of non-determinism can be addressed has been developed. That is because the issues with concurrent software must be addressed before current test methods can be applied.

Test and debug tools for concurrent software has been built with help of the framework. The objective is to use current test methods and develop new validation techniques within this framework.

The framework is based on the notion of *Synchronization sequences* (SYN-sequences). A SYN-sequence is *feasible* if the sequence of synchronization events can be exercised during the execution of the concurrent software. The SYN-sequence exercised by a concurrent software is in general non-deterministic. Hence, executing the concurrent software multiple times may exercise *different SYN-sequences* and may also produce *different results*.

Two different methods for testing concurrent software has been proposed. Both of them are useful when using mutation-based testing.

1. Multiple Execution Testing (MET)

   (a) Select a set of inputs of the concurrent software

   (b) For each selected input X, execute the concurrent software with X multiple times and examine the result from each execution.

   Using MET to randomly exercise SYN-sequences is similar to using random testing for selecting input values. Hence, it has been shown that MET should be supplemented with carefully selected test input values and SYN-sequences in order to achieve highly reliable concurrent software.

2. Deterministic Execution Testing (DET)

   (a) Select a set of IN_SYN test cases. Such a test case has the form of (X,R), where X is an input value and R is an SYN-sequence of the concurrent software.

   (b) For each selected IN_SYN test case (X,R):

      i. Determine whether or not R is feasible for the concurrent software given input value X. Do this by attempting to force R to be exercised during the execution of the concurrent software with input value X.

      ii. Then, compare the actual result of the deterministic execution with the expected result of the deterministic execution. An error has been detected if the actual- respective expected result of the deterministic execution differ.

The DEMT framework utilizes both these methods in order to utilize the best parts from both. Shortly, DEMT works as follows:

1. Randomly generate *Rendezvous sequences* (R-sequences) using the MET method until the mutation score has reached a steady value.

2. Select IN_SYN test cases and apply the DET method for achieving an adequate mutation score.

4. Replay and Testing for Concurrent Programs
Carver & Tai (1991)

Deterministic Execution Debugging and Testing is a method for solving the non-deterministic issues caused by concurrent software. The method can solve these issues by force an deterministic execution of the concurrent software during both debugging and regression testing, according to given SYN-sequences.

Example of a SYN-sequence is the start- respective stop the execution of a P-operation by a process on a semaphore, and to start the execution of a V-operation by a process on the semaphore. Note, that there is no need to stop the execution of the V-operation, since it stops itself immediately after the start. Such a sequence is called a PV-sequence.

Deterministic execution debugging is forced since its purpose is to reproduce the executions of the concurrent software, so that debugging information can be collected. Debugging information can be collected by using an interactive debugger and reproducibility tools.

SYN-sequence replay tools for concurrent software can be developed using two different approaches. Either language-based or implementation-based. The scope of the article is the language-based approach, hence the implementation-based approach is not covered. General distinctions between

them is that the language-based reproducibility tools are both easy to develop and highly portable. This is not the case with implementation-based reproducibility tools. Although, they are in general more efficient.

A language-based implementation reproducibility tool for a concurrent language, transforms a concurrent software written in that language into another version of the concurrent software written in the same language, but a version that can control the execution of synchronization events in the original version.

Force deterministic execution according to the input values and SYN-sequences of previous executions of the concurrent software can be done by using such reproducibility tools.

5. Testing of Concurrrent Programs based on Message Sequence Charts
Chung et al. (1999)

The proposed test method is based on conformance relations. Two types of these exist: *behavioral conformance* and *non-determinacy conformance*. Since this method is specification-based, it investigates whether the implementation of a concurrent software conforms with its specification. Synchronization constraints among synchronization events are included in the specification of a concurrent software. These have to be satisfied during the execution of the concurrent software.

The presented method is specification-based and tests a concurrent software against its Message Sequence Chart (MSC). A MSC can be seen as a collection of sequencing constraints that restricts the execution behavior of the concurrent software. MSCs mainly represents the interaction of messages among the softwares different components, e.g., message passing between processes. Module-level testing of concurrent software can be performed using MSCs within a given test framework.

MSC-based testing consists of the following four steps:

1. Elicitate constraints
2. Construct the test driver
3. Non-deterministic testing
4. Deterministic testing

The events modeled in the MSC are ordered by local timestamps. The constraints between the events can be determined by comparing their timestamps.

As a starting point from the elicited sequence constraints from the previous step, a test driver is then constructed for the *Module Under Test* (MUT). A MUT may consist of several processes. The test driver is simple a kind of

state machine that interacts with the MUT during execution of the concurrent software. It does so by mimicking the behaviors of the processes that are not included in the MUT.

Behavioral conformance testing is performed by non-deterministically execute the MUT together with the test driver.

Finally, the MUT is exercised by deterministic execution in order to check the conformance of non-determinism.

A drawback of specification-based testing of concurrent software is that if changes are made to the software, new sequence constraints must be created from scratch. That can be expensive no matter if these are created manually or automated by a computer.

6. Deterministic Execution Testing of Concurrent Ada Programs
Carver & Tai (1989)

More or less the same method presented under method 4 above.

7. A Feasible Strategy for Reachability Testing of Internet-based Concurrent Programs
Pu & Xu (2008)

A feasible strategy for reachability testing of Internet-based concurrent software is proposed in this paper. Three main approaches as been proposed previously regarding testing concurrent software. These are:

1. Non-deterministic testing
2. Deterministic testing
3. Reachability testing

Non-deterministic testing is about executing the concurrent software with input X several times and hope that faults are revealed at least under one execution. Because this method execute the software randomly, the method is not efficient. But it is easy to use.

Specific SYN-sequences are selected when using deterministic testing, so this method is more efficient. But it is not trivial to select what SYN-sequences to use.

Finally, reachability testing is a combination of non-deterministic testing and deterministic testing. A test case is executed deterministically up to a certain point, and the execution then continues non-deterministically from there.

A case study using this proposed method has been conducted, which demonstrated that reachability testing with this feasible strategy greatly decreased the testing of SYN-sequences and race variants. It is also worth to emphasize that no false SYN-sequences were exercised.

Reachability is important using this method since it is built upon exercising SYN-sequences. Exercising given SYN-sequences can be difficult, since

controllability is required. The concurrent software must be forced using deterministic testing, at least to some point, to be able to exercise specific SYN-sequences and not just randomly exercise them. Also, it may not be trivial to control up to which point the concurrent software should be executing deterministically and when it should continue non-deterministically.

8. A combinatorial testing strategy for concurrent programs
Lei et al. (2007)

Reachability testing derives test sequences on-the-fly without constructing a static model. Several reachability testing algorithms exist, but they are exhaustive. That is because they intend to exercise all possible SYN-sequences of a concurrent software given input X.

This means that exhaustive reachability testing derives race variants to cover all possible race outcome changes that can be made in a SYN-sequence. In contrast, race variants to cover all possible t-way combinations of the race outcome changes are derived using t-way testing.

T-way reachability testing is therefore proposed. The fundamental framework of reachability testing is adopted, but the method only intend to exercise a subset of all SYN-sequences.

Not every race outcome contributes to every fault and many faults can be revealed by interactions between a small number of race outcomes. This hypothesis is the main idea behind the proposed testing strategy.

Using reachability testing, it is not guaranteed that different SYN-sequences will be exercised during the execution of the concurrent software. But t-way reachability testing using the t-way testing stategy do guarantee that a different SYN-sequence is exercised each time during execution. T-way testing is a combinatorial testing strategy that selects input values for individual parameters and combines them for creating test cases.

An empirical study has been conducted to investigate the methods effectiveness. The results demonstrated that the number of test sequences that need to be exercised can substantially be reduced, at the same time as faults still can be revealed effectively.

This method have similar issues regarding controllability and reachability as method 7 described above. Apart from that, this methods effectiveness lies much in which SYN-sequences are selected for testing, since not all will be exercised. It means that the methods effectiveness depends much on the algorithm that selects these. If the algorithm is badly implemented, then the effectiveness may be decreased.

9. A New Algorithm for Reachability Testing of Concurrent Programs
Lei & Carver (2005)

The proposed algorithm does not save the history of synchronization sequences in order to guarantee that every partially-ordered sequence will be exercised exactly once. The algorithm creates a *race table* for a specific

Send-Recieve sequence (SR-sequence) Q. A unique, partially-ordered race variant of Q is represented on each row of the race table. No analysis of the implementation is required to create a race table.

10. Reachability Testing of Concurrent Programs
Lei & Carver (2006)

The method above is described in a conference proceeding, whereas this paper is the same but presented more detailed for a journal.

11. Tool Support for Testing Concurrent Java Components
Long et al. (2003)

Two significant test automation issues arises regarding concurrent software. *First*, forcing the execution of a given statement- or branch is difficult. *Second*, so is the automation of the analysis of test outputs. These issues are because of the inherent non-determinism in concurrent software.

The popularity of unit testing has increased and agile software development processes even advocate to integrate unit testing in the software development process. That is for increasing the software quality. However, there is a lack of unit testing tools with support for concurrent software. Therefore, this paper first extends a method for testing monitors and second, tool support for unit testing of concurrent Java components is introduced.

This is the journal version of the conference proceeding: A Concurrency Test Tool for Java Monitors (method 1 described above).

12. A method to test concurrent systems using architectural specification
Reza & Grant (2007)

A method for integration testing of complex concurrent systems is proposed. Integration testing is considered the most problematic level of testing related to development of concurrent software. Integration testing is considered the least well understood level of testing because it is neither unit testing nor completely system testing.

Integration testing is heavily dependent on the software architecture. Testing the software architecture is essential since it has been shown that the architecture can be the difference between a successful or a failure of a software. The software architecture is simply a description of the softwares components, connectors and tis configuration. Components are different computational elements, connectors are communicational elements, whereas the configuration is the softwares overall organization. A test method utilizing the softwares architecture must at least reach the components, connectors and configuration of the software.

The softwares architecture is usually prone to errors such as inconsistency, incompleteness and incorrectness. Hence, it is important to utilize integration testing since these errors may otherwise lead to financial and/or human loss. An *Architectural Description Language* (ADL) can be used to increase the understanding of the softwares architecture. ADLs help to provide clear

semantics together with tools used for description or analysis. A couple of models are usually also used in order to increase the understandability of the softwares architecture.

The proposed integration testing method relies on a model-oriented software architecture. This model-oriented software architecture provides different views of the software, i.e., different level of abstractions. Different behaviors of the software is provided from these views and can hence be tested.

13. Generating Test Sequences from Statecharts for Concurrent Program Testing
Seo et al. (2006)

Using specification-based testing, the softwares implementation is usually analyzed against its specification by executing it with test sequences derived from the specification. A test sequence is usually an interleaving of concurrent events. Test sequences are derived by analyzing the sequencing constraints between events.

The proposed method is utilizing automata-based execution instead of deterministic execution. Automata-based execution allows the software to be executed according to sequences accepted by the automata.

This proposed method is concerned about a concurrent softwares operational behavior. Overall, the method works as follows:

1. Identify information about the occurrence of events in a statechart specification and the information about dependency relation among events.

2. Generate representative interleavings, that represent the behavior of the concurrent software. This is done using partial-order methods.

3. Automata is generated separately from each representative interleaving. The automata will accept all equivalent sequences of a representative interleaving.

Empirical experiments has shown that the proposed method is effective.

An equivalence class in the form of an automaton is generated from each representative interleaving. The softwares execution can be controlled by the automaton to follow a given path that includes the implemented sequence among equivalence sequences. Controllability is hence important.

14. A UML/SPT Model Analysis Methodology for Concurrent Systems Based on Genetic Algorithms
Shousha et al. (2008)

A new method is proposed that is based on the analysis of the design representations of the concurrent software expressed in UML. The models are then used with a special *genetic algorithm* (GA) used to detect deadlocks in concurrent software.

It is required to feed the GA with relevant concurrency information. That is done by using the SPT profile for UML, which provides desired functionality. Once that is done, the tailored GA automatically retrieve required information from the UML/SPT models and it is then used to detect deadlocks in the concurrent software. Sequence diagrams are particularly beneficial to use, since the stereotypes and tags that is used by the GA occurs in these models.

The detection of deadlocks is performed by optimizing the access time of threads to locks. The GA consists of four components:

1. Chromosomes (i.e., representation of the solution)
2. A fitness function (i.e., a fitness of each chromosome)
3. Genetic operations of crossover and mutation that is used to generate new chromosomes
4. Selection operations that choose chromosomes fit for survival

It must be ensured that a deadlock is detected where there is one. Hence, a *Resource Allocation Graph* (RAG) is used when two or more threads are waiting. The GA terminates and the chromosome that yields the deadlock is returned, once a deadlock is detected from the RAG.

A prototype tool called *Concurrency Fault Detector* (CFD) has been built using the proposed method. CFG automatically identifies potential concurrency faults in concurrent software. CFG can currently detect deadlocks, but work is in progress for additional concurrency faults. The goal with CFG is to provide an automated tool that can be applied in *Model Driven Development* (MDD), since it is becoming increasingly popular.

The CFG tool works in the following way:

1. The user selects two categories of information:
   (a) UML/SPT sequence diagrams for the concurrent software
   (b) The execution time interval during which the system is to be analyzed
2. CFG then extracts the required information from the sequence diagrams

The tool consists of three parts:

1. A scheduler
2. A genetic algorithm
3. RAG evaluator

Three case studies has been conducted with promising results. The execution time for CFG has chown to be reasonable even for concurrent software with large search spaces. CFG can be executed several times for a couple of minutes and then nearly be certain to detect deadlocks if there exist some, even in a worst-case scenario.

15. Testing Concurrent Java Programs using Randomized Scheduling

Stoller (2002)

It is well known that errors caused by the inherent non-determinism in concurrent software is hard to reveal. These errors can be pinpointed and verified by a model checker, but these are not easily scalable to large systems. A model checker aim to control all non-determinism (including in the scheduling) and exhaustively explore the softwares possible behaviors. If the softwares state space is not to large, a model checker can even verify correctness.

This paper proposes a method that is less systematic and more scalable in contrast to a model checker. Invocations to a *scheduling function* is inserted at selected points in the softwares implementation. This scheduling function either causes a context switch or does nothing.

The transformed software is executed repeatedly in order to test it. If an error is found given a certain seed, then the software is re-executed with the same seed. That is because it is desired to reproduce the error. Hence, reproducibility is likely but not guaranteed. Since, in order to guarantee reproducibility, a capture-and-replay mechanism is required.

A tool called *Random Scheduling Test* (rstest) has been developed utilizing the proposed method. rstest's scalability has been demonstrated by apply it to ArgoUML [7]. rstest found what is apparently a concurrency-related error within quite a short time. Known errors in smaller systems were easily revealed by rstest. However, more experiments is needed to evaluate the tools effectiveness on large software systems.

16. Use of Sequencing Constraints for Specifying, Testing, and Debugging Concurrent Programs

Tai & Carver (1994)

Non-deterministic execution of a concurrent software P exercises a sequence of synchronization events (i.e., SYN-sequence). Restrictions on the allowed SYN-sequences of P are specified by sequencing constraints. The constraints do not have to be complete and they can be derived from formal- or informal specifications of P. Violations of P's constraints can be detected and coverage can be measured by collecting SYN-sequences during non-deterministic testing of P. SYN-sequences can be generated according to P's constraints and then used for deterministic testing of P.

This paper addresses a method for how to accomplish coverage and detect violations of constraints written in CSPE. Both deterministic- and non-deterministic testing can be used.

[7] Open-source UML modeling environment

17. Testing of Concurrent Software
Tai (1989)

This paper address issues with testing concurrent software and some methods for how it can be done. The inherent non-determinism of concurrent software is a major source to issues when testing concurrent software.

Reproducibility is important for instance when regression testing software. One approach to reproduce a software's behavior is to vary the execution of it. Several type of executions are mentioned:

1. Single execution testing
2. Multiple execution testing
3. Deterministic execution testing

Deterministic execution testing has several advantages over single- and multiple execution testing:

1. Carefully selected SYN-sequences are used to test the concurrent software P. It can detect the existence of invalid, feasible SYN-sequences of P, as well as the existence of valid, infeasible SYN-sequences of P. Single- and multiple execution testing can only exercise feasible SYN-sequences and hence detect the existence of invalid, feasible SYN-sequences of P. Thus, the existence of valid, infeasible SYN-sequences of P can not be detected.
2. If P is executed deterministically and an error is detected, it may be corrected and P can then be tested with the same previous input and SYN-sequence of the erroneous execution. That is desired for checking if the error has been corrected.
3. P can be re-executed deterministically with the same input values and SYN-sequences as previously in order to see that no new errors has been introduced.

18. Coverage Based Testing for Concurrent Software
Takahashi et al. (2008)

A new test method that focus on concurrency faults such as race conditions is proposed. The method works in the following stepwise manner:

1. Model the concurrent software with an enhanced ordinary modeling method. Blocks that are a subset of the concurrent software is defined. These blocks in the concurrent software are combined as a model.
2. Specific coverage criteria is used to test the model.

The proposed coverage method has been developed since current coverage criteria developed for sequential software can not target concurrency faults. *Concurrent Module Flow Graph* (CMFG), *All Concurrent Path* (ACP) and *All Concurrent Binominal Path* (ACBP) is defined in the paper.

Even though the proposed test method is regarded as efficient by the writers of the paper, their case study has shown that the method has not absence of issues. One major issue is that even with the lower level criteria ACBP, there is a large number of test cases. These tend to increase with the number of threads and blocks in the concurrent software.

Hence, it is desired to reduce the number of test cases at first, and second, to decrease the explosion issue with more complex concurrent software. This is important, since there is some indications on that testers hardly test concurrent software. That is because it requires a large effort to test a concurrent software and searching for various concurrency faults. Thus, this required effort should be minimized.

19. Structural Testing of Concurrent Programs
Taylor et al. (1992)

Structural testing techniques developed for sequential software are extended to test concurrent software. Several coverage criterions are described:

1. Concurrency State Coverage
2. State Transition Coverage
3. Synchronization Coverage

Concurrency states defines the structural testing metrics for concurrent software. For instance, the units (e.g., tasks or packages) in an Ada software will be a flowgraph. Each statements in the given unit will then be a node. Finally, each transfer between units and/or nodes will be represented by a directed edge. A sequence of statements represents a path through the graph.

Important to consider is the fact that practical issues exist with static concurrency analysis techniques based on reachability. Issues such that these techniques are limited in practice by the state space explosion problem. Also, it is impractical to analyze a complex concurrent software by building a concurrency graph for it. Another issue is that some of these test methods based on structural testing is ineffective for revealing all type of faults. Thus, structural testing is not itself a complete test method. It should be used as a complement to more comprehensive test methods.

20. Effective Generation of Test Sequences for Structural Testing of Concurrent Programs
Wong et al. (2005)

Constructing a *Reachability Graph* (RG) is common when performing structural testing of a concurrent software. A set of paths is then selected form the graph to satisfy some coverage criterion.

There is a need for methods to generate efficient test sequences to increase the coverage in an effective way. Four methods for test sequence generation is proposed, two based on hot spot prioritization and two based on topological sort. These methods can be used to generate a small set of test sequences that cover all nodes in the graph.

Two methods for testing concurrent software is proposed, namely non-deterministic and deterministic testing, respectively. The former is easy to use, but results in a very limited test coverage so it is inefficient. The latter is more efficient, but the main challenge is how to generate an effective set of SYN-sequences.

Constructing a RG of the concurrent software is a common approach to generate SYN-sequences. A state in the software is represented by a node in the graph, and a transition between reachable states in the software is represented by an edge in the graph.

At least one software execution can exercise each path in the graph, thus they are feasible SYN-sequences. Structural testing of a concurrent software P, is usually performed by the following steps:

1. Generate a set of paths from the RG, that should satisfy some structural testing criteria such as the all-node or all-edge criterion.
2. These SYN-sequences inclusive its data input is input to a deterministic execution environment for test execution.

Different strategies exist to decide which nodes in the graph that should be covered first.

A case study has been conducted that has shown that the proposed method can effectively generate a small set of test sequences to cover all nodes and edges in a RG. Some advantages of using a small set of test sequences is the management of them and that it is easier to manage the output verification. But examine the methods fault detection effectiveness of these test sequences is also important. At the time this paper was conducted, no real defect data has been used for examining the methods fault detection effectiveness in practice.

21. A Path Analysis Approach to Concurrent Program Testing
Yang & Chung (1990)

A method to test concurrent software using path analysis is presented. There has been a lack of these methods, since most of them is developed for sequential software and hence can not be applied directly on concurrent software.

The execution behavior of the concurrent software is modeled with a *concurrent path model*. An execution of the concurrent software includes a concurrent path comprised of the paths of all concurrent tasks. But, also the task synchronizations are modeled as a concurrent route to traverse the concurrent path included in the execution.

It is known that some new issues arise when testing concurrent software compared to sequential one. For instance, how the execution behavior should

be modeled and how to control the execution. The execution behavior of a concurrent software is modeled with a flowgraph and a rendezvous graph. Another difference aport from sequential software is that it is not enough to model the execution behavior of a concurrent software by the input and sequence of statements that is involved in the execution. That is because a concurrent software has synchronized tasks.

The static and dynamic structure of the concurrent software is separately modeled. The static structure is a syntactic view of the execution behavior. That is, the possible execution flow of statements. Whereas the dynamic structure is a run-time view, which is the possible rendezvous relationship among concurrent tasks.

The static structure is represented by a flowgraph, whereas the dynamic structure is represented by the rendezvous relationship of concurrent tasks of this execution.

An integrated testing environment called ConTest has been developed to realize the idea presented in this paper.

22. An approach to testing concurrent Ada programs
Tai (1986)

A previously common approach for testing concurrent software is with repeated execution. That is to execute the concurrent software repeatedly with the same input values. However, several issues exist with this method, since it is random.

Deterministic execution is hence proposed as an approach to use instead. The reproducibility issue can be solved by using it with the R_PERMIT method.

The following steps are performed in order to detect errors in a concurrent software P with input X using deterministic execution:

1. A set U of R-sequences are selected
2. A concurrent software P is transformed to P' using a reproducibility method. P only needs to be transformed once per all input values.
3. Finally, P' is executed with (X,S) [8] for each R-sequence S in U.

The R_PERMIT method can not reproduce a concurrent Ada software that contains:

1. Select statements with else- or delay alternatives
2. Conditional or timed entry call statements, or
3. Statements using shared variables or the COUNT attributes of entries

[8] X is an input value, whereas S is a rendezvous sequence

Both theoretical and empirical studies has shown that the deterministic execution approach is very effective for detecting rendezvous errors in concurrent Ada software.

23. Debugging concurrent processes: a case study
Stone (1988)

The behavior of a concurrent software can be reproduced from the histories of its individual processes by utilizing an approach called *speculative replay*. Histories are divided into dependence blocks by using known time dependences between events in different processes. Possible concurrencies among processes are visualized by a so called *concurrency map*.

When using this replay approach, known dependences are preserved and process histories generated during replay are compared with those that were logged during the original execution of the software.

Thus, the proposed approach studies the software's execution in retrospect and reproduces the execution from observed events during the original software execution.

The concurrency map is used both for visualization of concurrently executing processes, but it is also a data structure used for the reproducibility process. When reproducing the concurrent software's execution, a supervisor process control the re-execution of the concurrent software and also enforces a duplication of the process histories created during the original execution. A concurrent software can be reproduced by inserting a breakpoint at each point where a successor event occurs. The processes are thereafter re-executed and histories corresponding to their original execution histories are created.

24. Effective random testing of concurrent programs
Sen (2007)

The proposed approach utilizes ideas from traditional model checking. Model checking tries to systematically explore all thread schedules, hence it can prove correctness for a software with very high confidence. But, model checking suffer from the *state explosion* issue. That is, model checking does not scale well with the size of the software. The reason of this state explosion issue, is that the number of possible interleavings among processes in a concurrent software often grows exponentially with the length of the execution.

Perform random testing by choosing thread schedules at random is a simple and inexpensive alternative approach to model checking. The proposed approach uses partial order reduction methods to minimize the state explosion issue.

Some interleavings in a concurrent software are equivalent to each other since they correspond to different execution orders of various non-interacting or independent instructions from concurrent threads. This fact is exploited by partial order reduction methods. Thus, different execution orders from non-interacting instructions from concurrent threads will result in the same overall final state. So, if a given execution reveals a fault such as a deadlock

or race condition, then all equivalent execution orders will also reveal the fault.

These equivalent interleavings are described in terms of *happens-before* relations. A happens-before relation defines a partial order over all instructions executed during a given execution. Thus, concurrent executions that have the same happens-before relation are said to be equivalent. Partial order reduction methods aims to explore at least one execution from each partial order. But also to avoid exploring more than one execution from the same partial order.

Empirical studies have been conducted, but their validity is unsure. That is because it is not clear if the selected benchmarks are representative of parallell Java software used in practice. However, concurrent software from several different domains have been selected for benchmarking in order to reduce the just mentioned uncertainty.

The benchmarks have shown that the proposed approach that uses a systematic algorithm for random testing concurrent software, is indeed an effective approach for revealing faults in concurrent software. This random testing method is simple and inexpensive.

25. Instrumenting Where it Hurts - An Automatic Concurrent Debugging Technique
Tzoref et al. (2007)

Two algorithms are created and evaluated to automatically pinpoint locations in the software that are in the vicinity of faults. This paper is about search algorithms that can find the root causes of faults in concurrent software.

Noise can be inserted in the source code to discover whether faults exist or not. The noise are schedule-modifying statements such as sleep() and yield(). A context switch will be attempted by a noise. Once a fault has been founded, automatic debugging aims to find a minimal subset of the changes required to produce the fault. The understanding of the core requirement of the fault can be increased by finding such a minimal subset.

Experiments that has been conducted has demonstrated the methods effectivity.

26. Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics
Lu et al. (2008)

This journal article not really addresses a test method, although it provides important information about real world bug characteristics in concurrent software.

Concurrent software are becoming prevalent by the fact that multi-core processors is widespread not only in servers, but also desktop machines. Hence, the inherent issues with concurrent programming not only concerns elite programmers, but also novice programmers.

Writing concurrent software is hard, without doubt. A reason for that is because programmers tend to think sequentially when writing concurrent software. Hence, they easily make mistakes since concurrent programming requires a different way to think.

Because programmers tend to think sequentially when programming concurrent software, they tend to assume that small code regions will be executed atomically. Another issue is that programmers often assume a given order between threads, but they forget to enforce this order.

Software testing is a critical step for *revealing* software faults before the release of the software. The interleaving space is a major challenge for testing concurrent software. A *fault-revealing input* as well as a *fault-triggering* execution interleaving is required to reveal concurrency faults. Hence, achieving a complete test coverage of concurrent software requires that every possible interleaving for each input test case is covered. Unfortunately, this is infeasible in practice due to the state-explosion problem.

Designing effective test cases is important for concurrent software testing and it requires knowledge about *the manifestation conditions of real world concurrency faults*. Thus, what conditions are needed besides the input values to the software in order to reliably trigger a concurrency fault? For instance, how many threads, variables or accesses are usually involved in the faults manifestation?

If the manifestation of real world concurrency faults are better understood, it may help model checkers to alleviate its state-explosion problem by prioritize the software states.

Concurrency is mostly used in server software to handle concurrent requests from clients. Whereas concurrency in client software is mostly used to synchronize multiple *Graphical User Interface* (GUI) sessions and threads working in background.

27. Race Directed Random Testing of Concurrent Programs
Sen (2008)

A novel randomized dynamic analysis method is proposed. The method utilizes information about potential race conditions obtained from a current analysis tool to separate real race conditions from false ones. The method is implemented as an algorithm in a testing and debugging tool called RACEFUZZER. RACEFUZZER is an automatic tool, thus e.g., no manual inspection is needed in order to separate real race conditions from false warnings.

So, the tool uses this obtained information about potential race conditions to control a random scheduler of threads, whereupon the real race condition is created with a very high probability. These race conditions are randomly resolved during runtime.

The proposed method which is called *race-directed random testing*, combines detection of race conditions by utilizing a randomized thread scheduler in order to find real race conditions in a concurrent software. The tool works with high probability and it can discover if the detected race conditions may

cause an exception or an error in the software. Race conditions are detected by using a so called *imprecise race detection technique*.

RACEFUZZER is able to replay the execution of a test case on a concurrent software by picking the same seed for random number generation. That is feasible since the tool ensures that only one thread is executing during runtime. Also, all non-determinism is resolved by picking the next thread to execute by using random numbers. RACEFUZZER has hence support for deterministic replay of a given test case, which is important when debugging a race condition.

Empirical studies has shown that the RACEFUZZER tool was able to detect all known real race conditions in known benchmarks which was used. That is an indication on that no real race conditions that was predicted and manually confirmed by other dynamic analysis tools was missed. Conducted empirical studies also demonstrated that the tool effectively can find subtle faults in complex software.

28. Slicing Concurrent Java Programs
Chen & Xu (2001)

Two graphs called *Concurrent Control Flow Graph* (CCFG) and *Concurrent Program Dependence Graph* (CPDG) are presented in the paper which is used by a software slicing algorithm.

Software slicing can be used during the debugging process to effectively narrow the focus of attention to relevant parts of the concurrent software. A slice is a couple of statements in a software that may either directly or indirectly affect computed values at a given point during runtime. Software slicing can not only be used for debugging purpose, but also to increase the understanding of a software, testing, maintenance and complexity measurement.

The above mentioned CCFG is used to represent the synchronization among communicating threads. Control- and data flow of threads is usually not independent since inter-thread synchronization and communications may exist in the software. Hence, another graphs is presented for slicing concurrent Java software, namely CPDG. That is a dependence-based representation of the concurrent software.

A CCFG consists of several thread control graphs where each represents a single thread in the software and a special kind of edges represents interactions among threads.

The authors of this paper states that this method is efficient, but wheres the proof? Case study? Empricial study? Results?

29. Slicing Concurrent Programs
Nanda & Ramesh (2000)

This method is similar to the above one and it should hence haves similar issues (method 28).

30. Unit Testing Concurrent Software
Pugh & Ayewah (2007)

The proposed MultithreadedTC is not really a test method, hence it is a Java-based test framwork for writing test cases that exercise given interleavings in concurrent software.

No empirical studies are presented, so not much can be said about its effectivity. It is possible to construct deterministic and reproducible unit tests with the proposed test framework.

31. Testing Deadlock-Freedom of Computer Systems
Kameda (1980)

This paper aim to investigate if it is feasible to get deadlock in a concurrent software where no efforts has been made to avoid it. An efficient algorithm for the purpose has been developed.

32. Testing Concurrent Programs using Value Schedules
Chen & MacDonald (2007)

A novel technique for revealing concurrency faults is proposed. The method generates and tests *value schedules* of concurrent Java software. Value schedules are read-write assignment sequences, also known as *def-use pairs*.

The proposed method works like follows:

1. Feasible sequences of critical concurrent events are produced
2. Thread interleavings that fulfill those event sequences are produced and tested to decide software correctness.

Information collected from static analysis and from runtime state space exploration are combined in order to derive the feasible sequences of critical events and their fulfilling interleavings. Those thread interleavings are executed deterministically in an explicit state model checker to test the software's correctness.

Two approaches has foremost been used for dealing with the inherent non-determinism in concurrent software to reveal concurrency faults. The first approach is about completely bypass the non-determinism and reveal concurrency faults by utilizing static analysis techniques. The second approach is to verify the concurrent software by dynamic analysis or model checking.

This method works in the reverse direction compared to most fault detection methods. The method starts from a feasible value schedule. Then, the software's correctness is tried to be tested by producing and testing a thread interleaving that could fulfill this value schedule. The value schedule is feasible if such a thread interleaving is found.

There are three main approaches for revealing concurrency faults:

1. Static analysis

2. Dynamic analysis

3. Model checking

Different analysis methods are performed iteratively to reveal potential faults with static analysis. Whereas dynamic analysis reveal tries to reveal concurrency faults by executing the software with different thread interleavings.

Experiments has been conducted which demonstrated the methods efficiency.

33. Specification, Verification, and Synthesis of Concurrency Control Components
Yavuz-Kahveci & Bultan (2002)

Traditional validation techniques such as software testing is not efficient for concurrent software due to the state space exploration issue. The state space of a concurrent software increases exponentially with the number of variables and concurrent processes.

A new method for validation of concurrent software is therefore proposed in this paper. The method works as follows:

1. Formally specify the concurrency control component of the concurrent software

2. An infinite model checker automatically verifies the formal specification

3. The implementation for the concurrency control component is automatically generated

The concurrent software is formally specified in a monitor model called Action Language. It is argued that monitor specifications can be specified in a higher level of abstraction by using the Action Language.

An automated abstraction technique called *counting abstraction* is utilized by the proposed method. It can be used to verify monitor specifications defined in the Action Language. Counting abstraction can automatically verify the properties of any given monitor model for an arbitrary number of processes.

Several benefits can be gained by combining specification, verification and synthesis, as the proposed method does. The major are:

1. The need for condition variables, wait- and signal operations are eliminated. Hence, a monitor specification is a higher level abstraction of a solution compared to a monitor implementation.

2. Action Language Verifier can be used to verify Action Language specifications.

3. Java monitor implementations can be automatically translated from verified monitor specifications in the Action Language. The correctness of the translated implementation is guaranteed by construction.

34. Towards a Better Collaboration of Static and Dynamic Analyses for Testing Concurrent Programs
Chen & MacDonald (2008)

The proposed test method allows a tighter collaboration between static- and dynamic analysis of concurrent software. Course-grained analysis is used in static analysis to guide the dynamic analysis to concentrate on a relevant search space. Whereas runtime information is collected during the guided exploration with dynamic analysis.

Another distinction is that faults can be revealed without executing the software with static analysis. Hence, the non-determinism issue is avoided. But when using dynamic analysis, the software is executed directly and faults may be revealed at runtime. Dynamic analysis tries to trigger different interleavings in order to reveal faults. That may be done by inserting so called sleep statements randomly in the source code. Different interleavings can also be achieved by using an explicit-state model checker that can explore all interleavings systematically.

Thus, static analysis performs the initial part in the search space, whereas dynamic analysis then refines the remaining search space with information gained from the static analysis. Testing a distinct partial order more than once is avoided by using static analysis to guide dynamic analysis.

Some early experiments has been conducted that shows that this method has some improvements over *Java Path Finder* (JPF).

35. A test-case generation tool for times systems
Hessel & Pettersson (2007)
A model of the concurrent software under test is constructed, consisting of a controller- and an environmental part. The behavior of the concurrent software is specified by the controller part. Whereas the components surrounding the controller are specified by the environmental part.

Generated test suites [9] can be compiled into a test software and then be automatically executed. Cover generates test cases by performing state-space exploration on-the-fly with reachability analysis of the timed automata. Coverage information is combined with the state-space.

The concurrent software under test is tested with aspect to specified coverage criteria. These coverage criteria is described by utilizing an *observer language*. An observer is a monitoring automaton that formally describes a coverage criteria.

Test case generation tools can be developed by using the framework of Cover. Cover uses the verifier of Uppaal.

[9] A set of test cases