State-based Coverage Analysis and UML-driven Equivalence Checking for C++ State Machines

Patrick Heckeler, Jörg Behrend,
Thomas Kropf, Jürgen Ruf, Wolfgang Rosenstiel
University of Tübingen
{heckeler, behrend, kropf, ruf, rosenstiel}@informatik.uni-tuebingen.de

Roland Weiss
Industrial Software Systems
ABB Corporate Research
roland.weiss@de.abb.com

Abstract: This paper presents a methodology using an instrumentation-based behavioral checker to detect behavioral deviations of a C++ object implementing a finite state machine (FSM) and the corresponding specification defined as a UML state chart. The approach is able to link the source code with the appropriate states and provides a coverage analysis to show which states have been covered by unit, system and integration tests. Furthermore, the approach provides statistical information about the distribution of covered lines of code among all included files and directories. As a proof of concept the presented approach has been implemented in terms of a C++-library and has been successfully applied to OPC UA, an industrial automation infrastructure software.

1 Introduction

In the area of safety-critical hardware-depended software (SCHDS) like automation and engine controllers, avionics or medical devices, it is necessary to follow heavy-weight development processes (HWP) in order to satisfy safety development standards like IEC 61508[Bel05]. This is necessary to achieve certain *Safety Integrity Levels* which are the base for safety certifications. Among other things those standards define guidelines for planning, developing and testing the whole system and its single components. Those standardized procedures minimize safety critical bugs and errors in software but lead to extremely high development costs. Today a major goal of industrial software development units is to optimize and change their heavy-weight processes to reduce costs and time-to-market to compete economically. As a first step for cost reduction, many software development units have chosen C++ (for years C was the dominating language) as their preferred programming language for safety-critical hardware-dependent systems. Reasons for that are the inherent language flexibility, its potential for portability across a wide range of micro controllers and other hardware and, in contrast to native C, the possibility to use object-oriented language constructs for better code maintenance. A second step for optimization is to push the development processes towards agile processes[Mar03] which are

much more flexible and less cost intensive than HWP. For the introduction of new development processes in the area of safety-critical software it is yet indispensable to create new approaches for verification and validation to be in accordance with guidelines of standards like IEC 61508 to get safety certificates. Because the above mentioned systems are mostly event driven (waiting for a time tick, a signal of a sensor or a key press) this method aims at validating the correctness of a FSM which is common practice to handle events in software. The behavior of state machines can easily be described by UML state charts, even on a high abstraction level (in contrast to a formal FSM). An additional benefit of object-oriented languages is that a well-known design pattern exists to implement state machines. Nevertheless, we are in need of a methodology to validate such a state machine even at early stages. This is necessary because the state machine affects the whole software architecture and is responsible to put the system in a safe state when an error occurs. To conform to IEC 61508 it is necessary to test a system intensively with unit, system and integration tests. Therefore, often test cases are generated automatically. To estimate test quality, metrics like *percentage of covered lines of code* or *percentage of executed statements* are used. But this metrics can lead to a major misjudgment: Even high coverage values cannot promise that all critical areas of code have been tested (chapter 2.1 provides an overview of common coverage techniques). The main goal of this work is to create a methodology which presents more suitable metrics to estimate test quality: State-based coverage. During coverage analysis a lot of profiling information is generated (distribution of source code among files and directories). The results are presented in chapter 4. Furthermore, a behavioral check between UML state chart and C++ implementation is executed. An invalid transition taken during test execution can be revealed. Details about the presented validation approach can be found in chapter 3.

2 Preliminary Work and State of the Art

In this chapter we present state of the art coverage analysis techniques. In addition we comment patterns and language constructs to implement state machines using C++ and we classify the most important C++ software defect classes which are relevant for the design of SCHDS. Furthermore we discuss the Simulink/Stateflow[1] (SL/SF) approach and point out weak points of agile software developement for safety-critical systems.

2.1 Coverage Analysis

Coverage tools[YLW06] are commonly used to measure quality and completeness of test runs based on unit, system and integration tests[MS04]. These tools are counting how often code fragments (single lines of code, basic blocks or statements), branches and decisions are executed. Also the calling of methods and functions can be measured. Most coverage tools instrument the source code with so called probes which are injected into the

[1] Simulink and Stateflow are trademarks of The MathWorks Inc.

executable to be examined. This additional code tracks whether lines of code, branches or statements have been reached. Another possibility is to embed the executable into a monitor system which tracks the execution information from outside. The information gathered during a coverage analysis can be used to identify weak points in the source code: At first dead source code can be identified. These parts have never been executed or reached. Such code is just overhead and unnecessarily increases the size of source code files and compiled executables. Furthermore, test quality can be estimated, no matter what kind of test has been used. The more lines of code have been executed the better the unit test was. Table 1 presents an overview of available coverage methods an their corresponding metrics to describe test quality.

Coverage Method	Metrics	Complexity
Line Coverage	$Q = \frac{ExecutedLines}{AllLines}$	low
Statement Coverage	$Q = \frac{ExecutedStatements}{AllStatements}$	low
Branch Coverage	$Q = \frac{ReachedBranches}{AllBranches}$	medium
Path Coverage	$Q = \frac{TakenPathes}{AllPathes}$	high
Condition Coverage	All conditions must be executed with true & false	high

Table 1: Overview of coverage methods

For our methodology we are in need of a coverage tool which provides line coverage, a command line interface (CLI) and the ability to dump coverage data during execution (this is necessary to connect the source code with the single states). In Table 2 we present a feature overview of all C++ coverage tools taken into account: Bullseye[Bul09] is not able to perform a real line coverage. It is limited to functional and decision coverage. In contrast to Bullseye all other examined tools provide the feature of line coverage which is one of the main criteria for our concept. Decision coverage is also supported by CodeTEST[Met09], Dynamic[Sys09] and C++test[Par09] whereas functional coverage is only integrated in Bullseye, Dynamic and Intels code coverage tool[Int09]. All listed coverage tools bring along a file reporting feature. This means, that all coverage results are stored into files and are not only presented in a GUI (Bullseye, CodeTEST and C++test also support GUI-based reporting). Except Dynamic all coverage tools use a probe-based approach to measure coverage. These so called probes are small code fragments which are inserted into the executable during compile time. When a probe is reached and executed it increases the corresponding line or function counter. Dynamic does not instrument the code at compile time. Instead it uses a dynamic approach where the code is instrumented on the fly during execution time. File reporting is supported by all mentioned coverage tools. But the ability to dump coverage data into a file during execution is supported only by the open source tool Gcov[GNU09] which is part of the GNU compiler collection. That is the reason why we have chosen Gcov for our methodology.

Tool name	Line coverage	CLI	File	Dump during exec.	commercial	open source
Bullseye		✓	✓		✓	
CodeTEST	✓		✓		✓	
Dynamic	✓	✓	✓		✓	
Gcov	✓	✓	✓	✓		✓
Intel	✓	✓	✓		✓	
C++test	✓		✓		✓	

Table 2: Criteria for coverage tools to be suited for the presented methodology

2.2 Simulink Stateflow

Simulink Stateflow (SL/SF)[Mat09] is currently the de-facto standard in the area of model-based development. Simulink models consist of connected blocks which represent operations. Stateflow is able to describe the system behavior using parallel and hierarchical state machines as well as flowcharts. SL/SF provides the possibility of rapid prototyping and easy testing. A key feature of this tool bundle is the simulation of the whole model. A graphical debugger helps to reveal unexpected system behavior. Approaches like [KAI+09] and [AKRS08] optimize the symbolic analysis of traces and simulation coverage. SL/SF is able to generate C code but a major drawback is the lack of C++ code generation: Object-oriented languages provide more modern constructs and patterns to describe and dispatch finite state machines (see chapter 2.5) which often eases system integration. Furthermore, the developer has the possibility to insert hand-written code into SL/SF models which makes a subsequent validation or verification indispensable.

2.3 Agile Methods and Heavy-Weight Plan-Driven Processes

Agile processes have gained tremendous acceptance in industrial software development over the past years. A lot of research projects have examined the quality assurance abilities (QAA) in agile processes to test their relevance for development of safety-critical systems[HVZB04]. It is hard to compare agile methods with plan-driven strategies[Boe02] due to the big differences in costs and team size (agile methods have a smaller team size and therefore lower costs and also usually deal with more restricted system complexities). Therefore other approaches try to combine agile and plan-driven development processes[BT03] to take advantage of their strengths. We believe that it is possible to evolve agile processes such that they will become suitable for industrial light-weight development of safety-related systems. But it is necessary to provide the right tools for cost-efficient early-stage validation of software components. Our presented methodology (see chapter 3) contributes to that topic.

2.4 Software Defect Classes

Software defects can be categorized into several different classes. Table 3 presents an overview of some possible software defects in C++ software[MIS08] for safety-related systems. In addition we show which errors can be avoided by using our FSM behavioral checker (FBC).

Defect Class	Solution Strategy
Mistakes in specification (state machine)	Hugo/RT
Misunderstanding the specification or programming mistakes	FBC
Misunderstanding the language; compiler errors; runtime errors	Misra-C++, FBC

Table 3: Overview of software defect classes and solutions to avoid these errors

To detect, clasify and track software errors, a failure mode and effects analysis [PA02] (FMEA) can be integrated into the development process. A few software defects occurring in C++ programs can be avoided by sticking to the guidelines of MISRA-C++[MIS08]. But the defect classes mentioned in Table 3 need some other solution concepts.

- **Mistakes in specification:** These mistakes, e.g. a transition which points to a wrong target state and therefore create deadlocks, could be avoided by using a model checker like Hugo/RT to check state charts defined in UML.[BBK+04].
- **Developer misunderstands the specification or makes mistakes:** When the developer team misunderstands the specification, for example the team misinterprets the modeled behavior or functionality of the state diagram, behavioral errors in the implementation will occur. It is also possible that, despite correct understanding, implementation mistakes concerning the general behavior of the software component are made. To detect these bugs FBC can be used. FBC can detect deviations between implementation and specification.
- **Misunderstanding the language; compiler errors; runtime errors:** The developer team may misunderstand the effects of some language constructs. There are a number of areas of the C++ language that are prone to developer-introduced errors. If the developers follow the Misra-C++ definitions, those errors can be avoided. There are some areas in the C++ language that are not completely defined. The behavior varies from one compiler to the other. Even the behavior can vary within the same compiler, depending on the context. Misra reduces the C++ language to a proven subset. Following these guidelines compiler errors can be reduced, but Misra-C++ can not ensure a correct compilation process. In addition C++ programs tends to be small and efficient but they have a lack of data-dependent runtime checks. To deal with these defects our FBC can come into action.

2.5 State Machine Implementations

In this chapter we discuss four common ways to implement state machines in C++[Sam08].

1. A **nested switch statement** is perhaps the most simple FSM implementation. It consists of enumerated transitions and states and only one state variable is necessary. But it does not promote code reuse (here it breaks with agile processes) and it can easily grow too large. It is also very error prone because there is no language construct to control whether a valid transitions was taken or not.

2. A **generic state-table event processor** can be used to implement a FSM based on a simple two-dimensional state table. This table lists events along the horizontal and states along the vertical dimension. Therefore all states are listed in a regular and easy to read data structure. Also we only need a simple dispatcher which maps events to source and target states and checks if they are valid or not. Another advantage is the dispatcher can be used for more than one software projects (This is more conformant to agile methods than the *nested switch statements*).

3. The **state design pattern** is the object-oriented approach to implement a state machine. It relies heavily on polymorphism and partitions the single states in different classes. Transitions are efficient because only a pointer must be reassigned. All states are derived from an abstract super state class. The performance is better than indexing in a *state table*. But the implementation is not generic and the code cannot be reused for other state machines.

4. A **single object instance** of a class can be described by a FSM. Methods represent the triggers and attributes represent the states. More than one state encoding variable is possible.

3 The Methodology

Our methodology faces two major challenges. At first we want to adjust the abstraction level of specification and implementation for coverage analysis data: Both become state-based. The second aspect is to detect behavioral deviations between specification and the corresponding object instance during runtime. As a side-product many profiling information occur. In this chapter we present our methodology and its corresponding implementation in terms of a C++ library. At first we explain the main stages of our approach (chapter 3.1). Afterwards we go into detail (chapter 3.2) by presenting the industrial automation infrastructure software which is used later on in chapter 4 for an empirical study.

3.1 Main Stages

Our methodology is divided into four main stages (see Figure 1): Input, instrumentation, runtime and coverage stage.

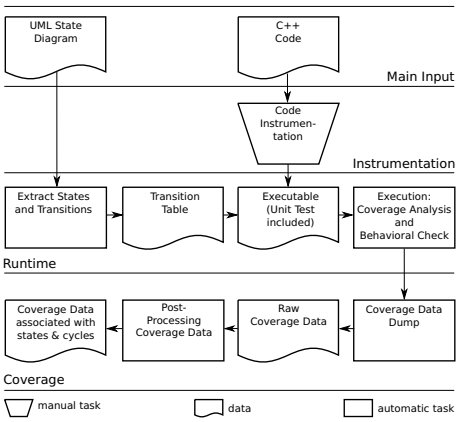

Figure 1: Flowchart of the methodology

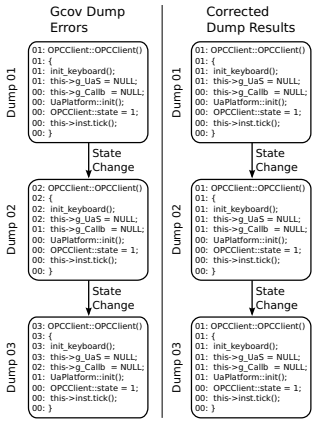

Figure 2: Post-processing of the coverage data

1. **Input stage:** Our methodology checks a state chart against its corresponding C++ implementation and associates the states with the corresponding source code. It is necessary to provide a *Class under Test* (CUT) and a UML state chart represented in XMI[2] (see Figure 1, Main Input).

2. **Instrumentation stage:** To apply our methodology to a C++ class it is important that the states of the FSM are encoded by one or more class members (attributes). Without this explicit encoding of states it is not possible to track the behavior of the object instance. Right now, our approach is based on a manual source code instrumentation. That means, we have to register all state encoding variables for monitoring and we have to integrate a timing reference in terms of a *tick()*-method to trigger the FSM (see Figure 1, Instrumentation).

3. **Runtime stage:** During runtime our library reads in the XMI file and creates a transition table. Based on this table our methodology is able to check whether a correct transition is taken or not. The CUT has to be linked against the library (see Figure 1, Runtime).

4. **Coverage stage:** During execution the behavior is checked (are only valid transitions taken?) and coverage data is dumped and stored in temporary files. Afterwards the coverage data is post-processed and connected with the corresponding states of the FSM (see Figure 1, Coverage).

[2]The most common use of XMI is as an XML-based interchange format for UML models

3.2 Details based on Industrial Automation Infrastructure Software

In this chapter we present the details of our approach by means of an industrial automation infrastructure software[MLD09] (IAIS). OPC UA consists of a client-server implementation (the focus in our experiments is on the client) which is used to access field data from field devices. In this example the client is used to monitor temperature data from an engine controller. The client connects to the server and is then able to receive data of the engine controller either with single reads or based on a subscription service which will deliver data in regular intervals. Furthermore, the client is able to browse the attributes provided by the engine controller and can also write data to the server. It consists of seven states which can be seen in Figure 3. Each state in the pseudo state (we want to keep the figure clear) can be reached by calling one of the following methods: *browse()*, *read()*, *write()* or *subscribe()*. The transitions are labeled with precise method names, including the return type, of the client class. Each call of one of these methods triggers the FSM to switch to the correct target state. The state encoding relies on the variable *int state*, written before a tick is executed. Of course the state encoding can rely on more than one variable. But to keep the example clear, we have chosen only one. The lines 2 and 9 in Figure 4 show the additional code caused by the necessary manual instrumentation. Figure 6 shows a small test scenario to stimulate our client without an error: All transitions are triggered in correct order. In contrast to this Figure 5 shows a test case which leads to an error: The transition *int OPCClient::shutdown()* is called before the transition *OPCClient::disconnect()* was executed. For our experiments we have embedded the client into the well-known Cppunit framework[Ope09]. Figure 3 shows the graphical representation of the error case (the dashed transition).

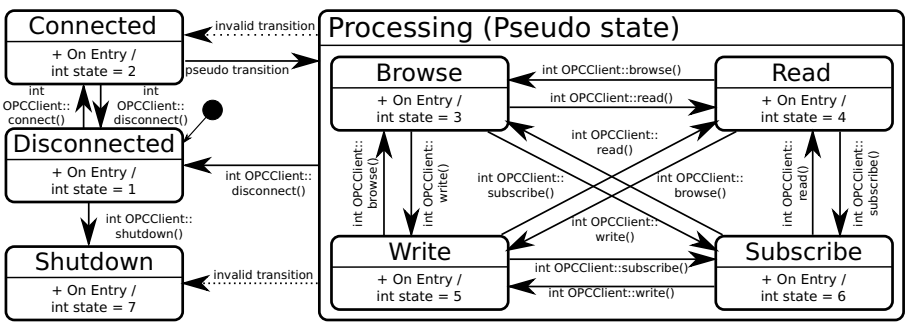

Figure 3: UML State machine representing the client behavior including invalid transitions (dashed arrows). Self transitions are valid for all states in the pseudo state. They have been omitted due to the lack of space.

To apply our validation approach to the client we have modeled the UML state diagram as seen in Figure 3 using Enterprise Architect (EA) in version 7.5[Spa09]. After that the FSM has been exported into a XMI file which must be stored in a directory together with all instrumented C++ source code files. An example of an instrumented file can be seen in Figure 4. In the default constructor the state encoding variable *state* must be registered for monitoring and the *tick()*-method must be called after each write access of this variable.

These ticks are used as a time reference for the FSM. Now the source code is ready to be compiled. After a successful compilation the CUT can be executed. During execution, the coverage data is dumped into a directory specified during the installation process of our library[3]. After a complete run the post-processing script is automatically triggered. This script processes all coverage data and presents a report as an HTML document. The flow sheet in Figure 1 provides an overview of the whole process. Following, we describe the single steps of the process in detail.

```
1  OPCClient::OPCClient() {
2    this->inst.regMonitorVar("state",
3                             "int",
4                             &this->state);
5    OPCClient::state = '1';
6  }
7  int OPCClient::browse() {
8    // browse the OPC Server
9    OPCClient::state = 3;
10   this->inst.tick(BOOST_CURRENT_FUNCTION);
11 }
```

Figure 4: Instrumented class Door: Default constructor and the *close()*-method

```
1  #include "OPCClient.hpp"
2
3  int main()
4  {
5    OPCClient client;
6    client.connect();
7    client.browse();
8    client.read();
9    client.shutdown();
10   return 0;
11 }
```

Figure 5: Unit test causing an error: An invalid transition is taken

```
1  #include "OPCClient.hpp"
2
3  int main()
4  {
5    OPCClient client;
6    client.connect();
7    client.browse();
8    client.disconnect();
9    client.shutdown();
10   return 0;
11 }
```

Figure 6: Unit test causing no errors: Only valid transitions are taken

- **UML State Diagram:** Modeled with EA the UML state diagram must follow some special design guidelines. Each transition must be labeled with the exact (return type, parameters) method name which triggers the corresponding transition. Each state must include one or more state variables representing the state encoding. These variables have to be modeled as an entry action. At the end the state diagram has to be exported into an XMI file.
- **Extract States and Transitions:** An XML parser reads in the XMI file and extracts all information of the modeled FSM.
- **Transition Table:** Based on the information obtained from the XMI file, the FBC creates a transition table. This table represents the core data of the behavioral checker. After triggering the state machine, our library checks if the transition as well as target and source state are valid.
- **C++ Code:** The C++ Code is the implementation of the FSM which we want to check (the CUT).
- **Code Instrumentation:** The state encoding variables have to be registered and the *tick()*-method has to be injected after each write access of these variables (see Figure 4 line 2).
- **Executable:** The CUT and its corresponding unit test have to be compiled with special compiler flags. We have chosen Gcov as our coverage tool. To make it work it is important to use the *-fprofile-arcs -ftest-coverage* flags supported by the g++

[3]We cannot use command line parameters because we have to avoid collisions with command line parameter needed by the CUT.

compiler. These flags control the code instrumentation for our coverage analysis based on Gcov. The manual code instrumentation mentioned above is only needed for the behavior check.

- **Execution:** The compiled source code can now be executed. Our library checks whether the whole course is correct or not (deviation detection).
- **Coverage Data Dump:** After each tick, the line coverage data is dumped into separate directories and is associated with the single states.
- **Raw Coverage Data:** The coverage data is distributed among subdirectories representing the sates.
- **Post-Processing Coverage Data:** After a complete execution of the CUT the library has to recalculate the coverage values. This correction is needed because Gcov adds up all line numbers of each dump (Gcov was not designed for dumping data during runtime). This leads to incorrect coverage values. Figure 2 shows two state changes and the line counter variance (the first column in the Figure). The second part of the Figure shows the recalculated line coverage values.
- **Coverage Data associated with States and Cycles:** At the end we receive an detailed graphical overview of all coverage results in HTML (the HTML generation is based on genhtml, a simple conversion tool provided with Gcov).

4 Results

We have successfully applied our methodology to the IAIS client presented in chapter 3.2. In this chapter we discuss the achieved results of a test run which has reached 9 states of the FSM. At first we have measured the runtime to show the scalability and overhead of our approach. Figure 7 shows the execution time of four test runs with 3, 5, 9 and 15 taken transitions and reached states (each valid taken transition triggers a data dump). We can see a constant growth relative to the number of reached states. Therefore our approach scales linear. Due to the heavy file I/O-operations caused by the coverage data correction the major part of computational time is spent for the post-processing of this data (see Figure 8. Each included source file is stored in a separate Gcov HTML file (generated with genhtml). In our test run with 9 reached states and therefore 9 taken transitions we have to parse 46 files per state. In addition to that we also have to correct the overview files produced by genhtml: One per included directory, in total 11 (see Table 4) and one for the whole overview. All in all our library had to post-process 453 files.

In our next two experiments we have provoked errors to test the behavioral checking feature of our library. In the first test run the *OPCClient::shutdown()*-method was called before the client has been disconnected from he server. In the second run we have called the *OPCClient::connect()*-method while one of the sub-states of the pseudo state has been active. Both errors have been detected correctly and the corresponding error traces have

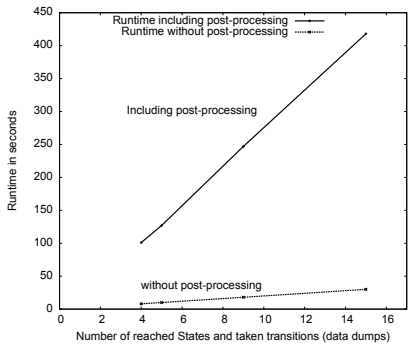

Figure 7: Runtime of 4, 5, 9 and 15 reached states: Constant growth

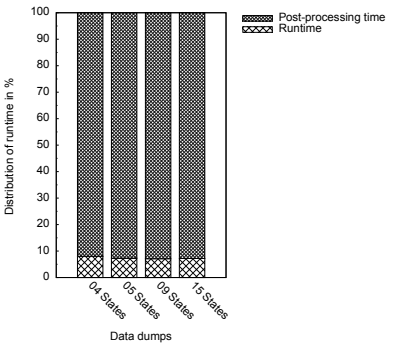

Figure 8: Runtime ratio of simulation time and post-processing time

been created. These traces include the execution order of all states until the error occurs. These traces are important for reproducing errors during the development process.

Directory Name	Disconnect (1)	Disconnect (2)
/usr/include/c++/4.3	3.03 %	0.0 %
/usr/include/c++/4.3/backward	28.57 %	0.0 %
/usr/include/c++/4.3/bits	20.73 %	0.0 %
/usr/include/c++/4.3/ext	4.35 %	0.0 %
/usr/include/c++/4.3/i486-linux-gnu/bits	0.0 %	0.0 %
/usr/local/include/cppunit	3.13 %	15.63 %
/usr/local/include/cppunit/extensions	92.00 %	0.0 %
examples/OPCClient/unittest	5.9 %	5.65 %
examples/utilities/linux	33.33%	0.0 %
include/uabase	7.14 %	45.24 %
include/uaclient	0.0 %	13.73 %

Table 4: Covered lines of code per directory in percent of the first and second hit of the state *disconnected*: The profile of the hits differ significantly.

Because the OPCClient implementation includes a huge number of source code files we cannot present the whole source code connected with its corresponding states. As an example we present an extraction of the file OPCClient.cpp including the main part of the client implementation. The code fragment can be seen in Figure 10. It presents the source code connected with the disconnect state when it is run through for the second time. Line 138 is marked as not covered. The reason for this is, that the tick()-method must be called before the return statement is executed. In chapter 5 we propose a method to avoid instrumentation and therefore lead to a correct coverage result by connecting also the return statement with the corresponding state.

In Figure 9 we present an overview of executed source code lines per state. Furthermore this figure shows, that all states has been covered. With metrics $Q = \frac{ReachedStates}{AllStates}$ we provide a formula to estimate test quality. In our test run we have achieved a state coverage

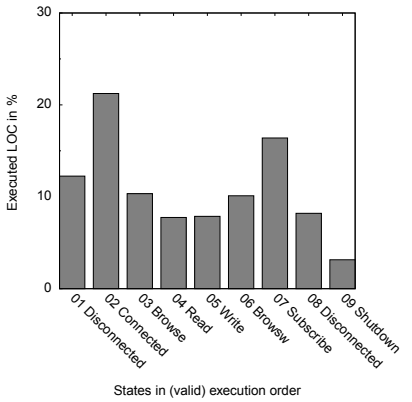

Figure 9: Executed code per state in percent

```
115| 1: int OPCClient::disconnect() {
116| 1:   printf("** Disconnect from Server\n");
117| 1:   if ( this->g_pUaSession == NULL ) {
118| 0:     printf("** Error: Server not connected\n");
119| 0:     return 1;
120|  :   }
121|  :
122| 1:   ServiceSettings serviceSettings;
123|  :   this->g_pUaSession->disconnect(
124|  :                       serviceSettings, // Use default settings
125| 1:                       OpcUa_True);     // Delete subscriptions
126|  :
127| 1:   delete this->g_pUaSession;
128| 1:   this->g_pUaSession = NULL;
129|  :
130| 1:   this->g_VariableNodeIds.clear();
131| 1:   this->g_WriteVariableNodeIds.clear();
132| 1:   this->g_ObjectNodeIds.clear();
133| 1:   this->g_MethodNodeIds.clear();
134|  :
135| 1:   OPCClient::state = 1;
136| 1:   this->inst.tick(BOOST_CURRENT_FUNCTION);
137|  :
138| 0:   return 0;
139|  : }
```

Figure 10: The *disconnect()*-method in *OPCClient.cpp* reached the second time (state *disconnected*)

of 100 %. We have shown that our test case has executed all safety relevant states. This will avoid the misjudgment of test quality based on common metrics presented in chapter 2.1.

5 Conclusion and Future Work

In this paper we have presented a methodology to connect states of a UML state chart, specifying the behavior of a C++ class, with the corresponding lines of code of all involved source files. This adjusts the abstraction level of specification and implementation (both are state-based now) and makes it much easier to estimate quality of test scenarios. Furthermore, our library is able to detect behavioral deviations between CUT and the state chart. Moreover we have successfully applied our library to an industrial automation infrastructure software. We have shown that the approach scales very well and that it has the potential to be a vital component to push HWP towards agile software development processes.
To simplify the use of our library we want to introduce a monitor mechanism, based on the GNU Debugger[SPS02] and Valgrind[NS07], which is able to observe trigger method calls and write accesses for state encoding variables. This will avoid the task of manual source code instrumentation. As a second extension we want to integrate a mechanism to execute all invalid (not available) transitions to perform stress tests.

6 Acknowledgments

We would like to thank ABB Corporate Research for co-funding our research and providing an industrial software to test our approach.

References

[AKRS08] Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pages 89–98, New York, NY, USA, 2008. ACM.

[BBK+04] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive Verification of UML State Machines. In *ICFEM*, pages 434–448, 2004.

[Bel05] Ron Bell. Introduction to IEC 61508. In Tony Cant, editor, *Tenth Australian Workshop on Safety-Related Programmable Systems (SCS 2005)*, volume 55 of *CRPIT*, pages 3–12, Sydney, Australia, 2005. ACS.

[Boe02] Barry Boehm. Get Ready for Agile Methods, with Care. *Computer*, 35:64–69, 2002.

[BT03] Barry Boehm and Richard Turner. Using Risk to Balance Agile and Plan-Driven Methods. *Computer*, 36(6):57–66, 2003.

[Bul09] Bullseye. Bullseye Coverage, 11 2009. http://www.bullseye.com/.

[GNU09] GNU. Gcov Coverage, 11 2009. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[HVZB04] Ming Huo, June Verner, Liming Zhu, and Muhammad Ali Babar. Software Quality and Agile Methods. volume 1, pages 520–525, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[Int09] Intel. Intel Code Coverage Tool, 11 2009. http://www.intel.com/cd/software/products/asmo-na/eng/219633.htm.

[KAI+09] Aditya Kanade, Rajeev Alur, Franjo Ivančić, S. Ramesh, Sriram Sankaranarayanan, and K. C. Shashidhar. Generating and Analyzing Symbolic Traces of Simulink/Stateflow Models. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 430–445, Berlin, Heidelberg, 2009. Springer-Verlag.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[Mat09] MathWorks Inc. http://www.mathworks.com/products/stateflow/, 07 2009.

[Met09] Metrowerks. CodeTEST, 11 2009. http://www.metrowerks.com/MW/Develop/AMC/CodeTEST/default.htm.

[MIS08] MISRA. *MISRA-C++:2008 Guidelines for the use of the C++ language in critical systems*. Motor Industry Research Association, Nuneaton CV10 0TU, UK, 2008.

[MLD09] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. Springer Publishing Company, Incorporated, 2009.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[Ope09] Open Source Community. CPPUnit, 11 2009. http://sourceforge.net/projects/cppunit/.

[PA02] Haapanen Pentti and Helminen Atte. Failure Mode and Effects Analysis of software-based automation systems. In *VTT Industrial Systems, STUK-YTO-TR 190*, page 190, 2002.

[Par09] Parasoft. C++TEST, 11 2009. http://www.parasoft.com/.

[Sam08] Miro Samek. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes, Newton, MA, USA, 2008.

[Spa09] Sparx Systems. Sparx Systems - Enterprise Architect, 11 2009. http://www.sparxsystems.de/.

[SPS02] Richard M. Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 9 edition, 1 2002.

[Sys09] Dynamic Memory Systems. Dynamic Code Coverage, 11 2009. http://dynamic-memory.com/.

[YLW06] Qian Yang, J. Jenny Li, and David Weiss. A survey of coverage based testing tools. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 99–103, New York, NY, USA, 2006. ACM.