CHALMERS

Strobilus

Test Application Generator for AUTOSAR systems

Master of Science Thesis in the Software Engineering Programme

ADAM HULIN
MARCUS JOHANSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2011

Strobilus
Test Application Generator for AUTOSAR systems

Examiner: Roger Johansson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2009

Abstract

Automatic test generation is a way to ensure that trivial problems and errors can be found without manual work. Automatic tests are a great complement to manual testing and one strong advantage is that tests can be run on every product release without much effort.

The goal of this thesis was to develop a framework for automatic test generation for AUTOSAR systems, called Strobilus. Strobilus features support for reading and modifying AUTOSAR configuration files and generation of script and SWC code through an intermediate language.

Strobilus was built using Java with XML-libraries. It is primarily intended to be a complement to the tool Picea Workbench, testing Picea Workbench's output of configuration files, but it can also be used as a stand-alone product.

The scope of Strobilus covers support for AUTOSAR version 3.1, code generation for C and script generation for CANoe. The design is very modular making it easy to add additional AUTOSAR versions, test modules or output formats.

In its current state, Strobilus is capable of testing signals connecting to CAN. The generated test evaluates the correctness of signal paths, signal timing and verifies that packets are transmitted properly with the correct contents. More tests do however need to be implemented for the Strobilus framework to reach its full potential.

Keywords: AUTOSAR*, CANoe, automatic test generation, Picea Workbench, Mecel, code generation*

Preface

This thesis was conducted at the company Mecel, where the current flora of products follows a common theme: Latin names of trees. Since the result of this thesis will be used in conjunction with Mecel's Picea Workbench it was clear that it should have a corresponding name. We therefore chose to name the result of this thesis Strobilus. Picea is the Latin word for spruce. Strobilus is cone in Latin, which makes sense since it's a small add-on to Picea.

This report will explain how and why the thesis project, *Strobilus - automatic test generator for AUTOSAR*, was created. After reading this document the reader will have a deeper understanding of why automatic tests are needed and how they can benefit the AUTOSAR community. The reader will also get an understanding of how Strobilus was created using general software engineering principles.

We would like to thank the staff at Mecel, especially Jörgen Tryggvesson, Mikael Pontusson and Samuel Sveningsson. We would also like to thank our supervisor at Chalmers, Roger Johansson.

Table of Contents

Glossary

AUTOSAR - **AUT**omotive **O**pen **S**ystem **AR**chitecture, is a standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers.

API - Application Programming Interface

Artop - The **A**UTOSA**R** **To**ol **P**latform is an implementation of common base functionality for AUTOSAR development tools. The Artop development process is based on a community driven approach by AUTOSAR members and partners.

ARXML – An XML file containing configuration on an AUTOSAR specific format.

BNF-converter – A tool for implementing programming languages specified in Backus-Naur Form (BNF).

BSW – Basic Software, the bottom layer of the AUTOSAR hierarchy.

Callback - The AUTOSAR software can be configured to call certain functions when for example a signal has been successfully sent or received.

CAN - Controller Area Network

CANoe – ECU development tool.

CAPL - CAN Access Programming Language, developed by Vector.

DOM - Document Object Model, a way of parsing XML code by interpreting the code as a tree structure.

ECU – Electronic Control Unit. An embedded computer inside a vehicle.

Front-end – In compilers the front-end is the part that translates the source code into an intermediate representation.

JIT – Just-in-time compilation, also known as dynamic translation, is a way to improve the performance of a program by dynamically compiling the running application to optimized machine code. Java uses a JIT compiler.

Lexer – A program, which converts character sequences in a text to tokens.

MCAL – Micro controller Abstraction Layer

OO – Object Oriented, a way of structuring applications during the programming phase. The name comes from that the code is split up in different objects making the code easier to separate in distinctive parts.

RTE – Runtime Environment, the upper layer of AUTOSAR. It is the layer that connects the SWC's to the rest of the AUTOSAR structure.

Runnable – A function within RTE or SWC. It could be a callback method for a network signal or a function that needs to be run within an SWC.

SAX - Simple API for XML, a more lightweight parsing compared to DOM. Generates events upon chosen XML strings.

SWC – Software component. An application that is running on top of the RTE. Strobilus creates an SWC that performs tests.

Task – The RTE has several tasks that run runnables. Several runnables can be mapped to the same task or to different tasks if they need to run at the same time.

Test Module - A test module contains the logic of a certain test, and describes what information needs to be read and modified from the ARXML configurations. Test modules are also responsible for writing the code for the SWC and scripts in the intermediate language.

XML – Extensible markup language. Basically a markup language that can be customized for ones own needs using schema files specifying rules for how the markup language should be written.

License

Strobilus is developed under the BSD, Berkeley Software Distribution license. There are several models of the BSD license and the one that Strobilus uses is the 3-clause version also known as new BSD license or modified BSD license. Simplified the 3-clause BSD license says that copyright information about the authors must be distributed together with the source code and binaries. The license also says that the authors of the code cannot be used in promotional material for modified version of the software without their permission.

1. Introduction

This thesis aims to develop testing framework for AUTOSAR configurations, called Strobilus. It will be created as a terminal based application for stand-alone use or integration within a system. Strobilus is to facilitate an extensible API for writing test modules capable of reading and modifying AUTOSAR configurations and generating test software and scripts.

This chapter will give an introduction to AUTOSAR, the problem description and how the thesis work will provide solutions to the problems described.

1.1 Background

1.1.1 Original Problem Description

The purpose of this thesis is to develop a framework for automatic testing of integrated AUTOSAR systems.

AUTOSAR is a global standard for software architecture in the automotive industry. An AUTOSAR system contains one or more SW-C's (application layer) and an underlying software platform (RTE and basic software) set up through a number of XML files (AUTOSAR configuration). The configuration is often complex and may contain hundreds of bus and IO signals. To verify the proper operation of the RTE and basic software test applications and corresponding test scripts are needed. These tests are time demanding to produce manually, and chances are that some test cases will be completely missed.

The thesis project therefore aims to develop a design and develop a tool that generates test applications and test scripts automatically based on an existing AUTOSAR configuration. The thesis work is also included to evaluate an appropriate testing environment, including hardware. (Mecel AB, 2010)

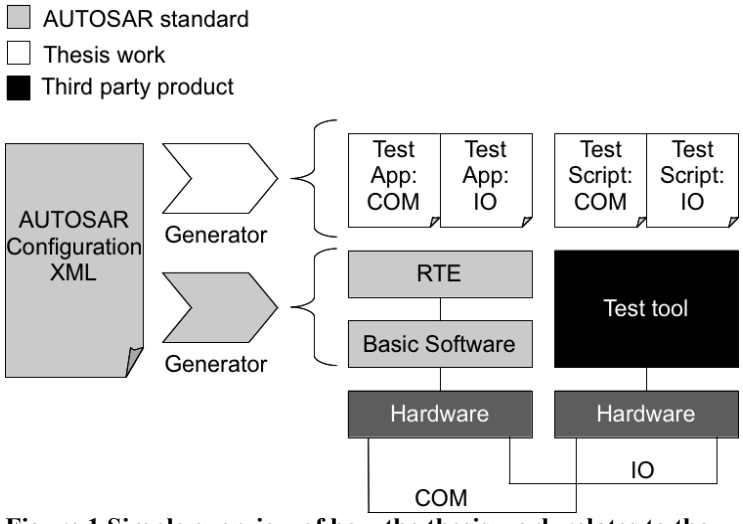

Figure 1 Simple overview of how the thesis work relates to the standard AUTOSAR generation procedures.

1.1.2 Interpretation of problem description

The interpretation has been done in cooperation with Mecel and the result is a list of requirements for the project (appendix A). An added requirement from the original problem description was that the tool should be created as a terminal based application for stand-alone use or integration within a system.

1.2 Purpose

Given the pressure to quickly release new products, developers might be tempted to skip testing and simply do trial runs of the product. Such behavior would not only mean a great risk of errors slipping through to product release, but also mean a lot of work to track down the cause of any problems found.

If tests can be run automatically, the AUTOSAR configuration can be tested nightly or even several times daily, without any loss of productivity. Errors can then be detected at an earlier stage and therefore corrected at a lower total cost (HP, 2010). In this way automated test procedures can both save time for the developers and potentially improve the quality of the product before a product release.

1.3 Objective

The objective of the thesis work is to design and develop a tool with an accompanying framework that allows users to easily write test modules for AUTOSAR configurations.

To run a test module from Strobilus, the user has to supply a test configuration specifying what module to run, locations of AUTOSAR files, what kind of outputs to produce and any test specific settings.

An example test module will be implemented with the purpose of testing that messages sent between the layers in an AUTOSAR system work as specified. An AUTOSAR software component (SWC) and a script controlling an external hardware will be generated and triggered to communicate with each other. The test can then find and report any potential problems with message contents, timing or periodicity.

1.4 Scope

Mecel limited the test generation framework to only support AUTOSAR version 3.1 and the CAN-bus (Controller Area Network) for external communication. This limitation meant that the code generators would be limited to C-code generator for SWC's and a CANoe script generator for scripts to manipulate the CAN-bus. For testing of Strobilus as a product, a single communication test module will be implemented.

The main focus is not to build an application capable of testing everything; building such a tool would not be possible with the given time constraints. Instead the goal was to build a modular framework for test generation, which Mecel could easily extend for their current and future test requirements.

1.5 A short description of AUTOSAR

To be able to fully understand Strobilus requirements, one must have an overview of the AUTOSAR layers. To simplify, AUTOSAR behaves much like any modern operating system for personal computers. It uses several abstraction layers to hide information and simplify system calls for the specific layer.

In figure 2, the different layers of AUTOSAR are visualized. At the bottom of the hierarchy is a layer called basic software or BSW in short. Within BSW is the hardware abstraction layer. In AUTOSAR the hardware abstraction layer is called Microcontroller abstraction layer (MCAL).

A hardware abstraction layer simplifies the communication between the operating system and the hardware. This is done using standardized calls from the rest of the operating system to the abstraction layer while the abstraction layer does the specialized calls to the hardware.

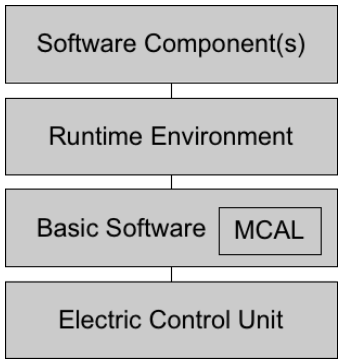

Figure 2 AUTOSAR layers.

Within the BSW is not only the MCAL but also the operating system and standardized interfaces that the Runtime Environment, RTE, connects to.

The runtime environment is a middle layer between the software components and the basic software. The runtime environment takes care about the different signals so the software component developer does not need to. For example if a software component is supposed to communicate to another software component using a signal called *my_signal*, the runtime environment makes sure that *my_signal* gets connected to the right interface regardless of whether *my_signal* connects the two software components internally or externally between two different ECU's.

To create an AUTOSAR environment, every ECU needs a basic software module specifically written for that ECU type. The manufacturers usually supply the specialized BSW modules. The runtime environment is automatically generated together with the software components. For this to work the runtime environment generator needs a set of configuration files describing the ECU and the basic software. These configuration files are written in an XML based format called AUTOSAR XML, ARXML.

2 Method

The development of Strobilus followed an iterative waterfall process involving the following distinctive phases; planning, requirements, analysis, design implementation and testing.

At the start of the project a rough time plan was made describing how much time would be spent on each phase and iteration.

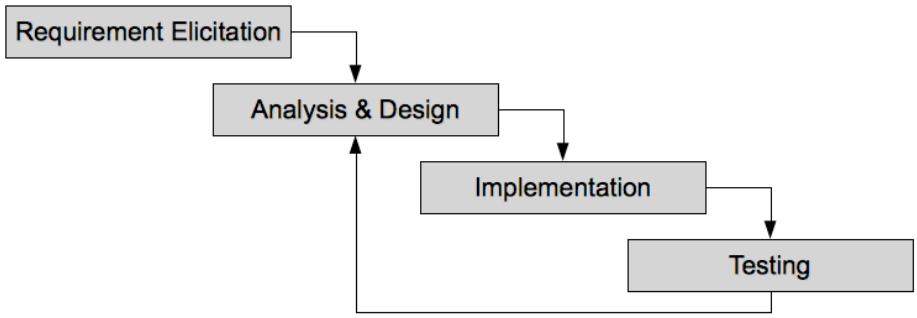

Figure 3 Strobilus development was based on an iterative waterfall model

During the requirement elicitation phase we had discussions with our supervisors at Mecel to discover the functional and non-functional requirements of the system. The result of this phase was the requirement specification document, appendix A.

The first analysis phase resulted in an initial design of the application. The subsequent implementation phase aimed to create prototypes for the various parts of the program. Finally these parts were interconnected and tested as a whole to find any misconceptions in the analysis phase and to validate the design.

During the final iteration, the design was revised with the knowledge gained from the prototyping, and the implementation of the final product began.

2.1 Analysis

Strobilus needs to be able to parse and modify ARXML code. The different versions of the AUTOSAR standard use slightly different notation for ARXML. Therefore time was spent investigating existing applications for AUTOSAR development, in order to see how they handled different versions and if parts of the code could be reused. There were two good candidates, Picea Workbench and Artop.

Picea Workbench is an in house tool by Mecel, used to build, manipulate and verify the correctness of ARXML code. Picea Workbench is built on Eclipse and has a similar user interface.

An analysis of Picea Workbench source code showed that each supported version of AUTOSAR had its own model for representing the ARXML data, and no means of translating between models could be found. The source code library consisted of over 100.000 files, making it cumbersome to filter out usable code. Our conclusion was that extracting the code handling ARXML would be a very complex task, and parts of that code would still need to be rewritten to fit our purposes.

The second candidate, Artop, is a tool actively being developed by the AUTOSAR user group and its source code is available to all AUTOSAR members. Artop is built on Eclipse like Picea Workbench, but it's not made to be a complete tool but a software platform on which tools like Picea Workbench can be built. The rationale behind this is that all tool vendors need to create the same XML manipulating base code, so having a common code base would minimize errors, speed up the development and making the process of creating new tools cheaper.

The analysis of Artop showed that it had different models for different AUTOSAR versions, much like Picea Workbench. Again, extracting code for reuse did not seem like a worthwhile task.

Having deemed the existing tools too complex for reuse, it was decided necessary to implement our own data model and handler for the ARXML code. The first step in designing an ARXML handler was to find the most suitable way of parsing XML code.

2.1.1 Choosing an XML parser

DOM (Document Object Model) and SAX (Simple API for XML) are two common ways to parse XML code.

SAX is a simple and lightweight implementation (Oracle, 2004) that parses an XML file and generates events when it locates specified tags. The events can then be used to find the desired data from the XML.

DOM on the other hand parses the entire XML file and generates a tree structure, consisting of nodes representing the data. The tree can easily be searched, manipulated and resaved to an XML file.

A SAX implementation would not need to keep a representation of all the XML data in memory; it would only go through the file and read the data requested, generate the events and close the file. On the other hand, it could be hard to design a SAX implementation extendable and flexible, as different test modules require different data from the ARXML code.

There are two big parts required to create a SAX parser. Firstly, data generated by the events must be stored in some kind of data structure. Secondly an event handler must be written so the data can be collected from the events and stored in the data structure.

A DOM implementation requires more start up time, as it needs to parse the XML file and generate a data model for it. This results in a higher memory footprint, as the entire model must be stored in memory. Once the model has been built however, it is simple and cheap to locate data using look up methods or search using the query language XPath. No special data structure is required, nor any foresight on what data is needed, making it a more flexible choice.

Parsing ARXML will be a significant part of the total execution time, so performance must be taken into consideration. A goal of Strobilus is however to reduce the time it takes a developer to test AUTOSAR configurations, so the time it takes to implement new test modules must also be considered.

A SAX implementation is usually faster and requires less memory (IBM, 2003). On the other hand Strobilus would need to have a very general SAX implementation in order to support all possible kinds of test modules. SAX searches are preferably done all at once instead of in sequence, this could make it more difficult for the test developers.

The conclusion of the analysis was that DOM would be a better choice for the developer as long as the performance is deemed acceptable.

2.1.2 Analysis of example test module

The generated test consists of an SWC in C code and a script controlling the network hardware, in this case generated as a CANoe script.

The software component uses special signal identifiers, fetched from the ARXML code to communicate with the network interface in the ECU.

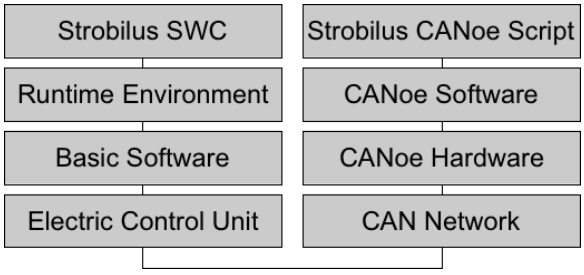

Figure 4 Example AUTOSAR test configuration.

The test hardware uses the script generated by Strobilus during runtime. The script can be configured to test received data, send data back to the software component or both.

2.2 Design

Strobilus is meant to be a framework for generating tests for AUTOSAR systems. Thus it is important that Strobilus is designed with extendability in mind. This means that strict object orientation will be necessary, and also that interfaces and API's will be used to make the code modular and decoupled so that new features can be added with minimal changes to existing code.

Strobilus has several requirements for extendability. The following requirements are those that affect the design decisions the most. A complete list of requirements can be found in appendix A.

Strobilus must be able to support different versions of AUTOSAR. Practically this means that Strobilus must be able to read and write different versions of AUTOSAR configuration files and generate different SWC codes depending on the AUTOSAR version.

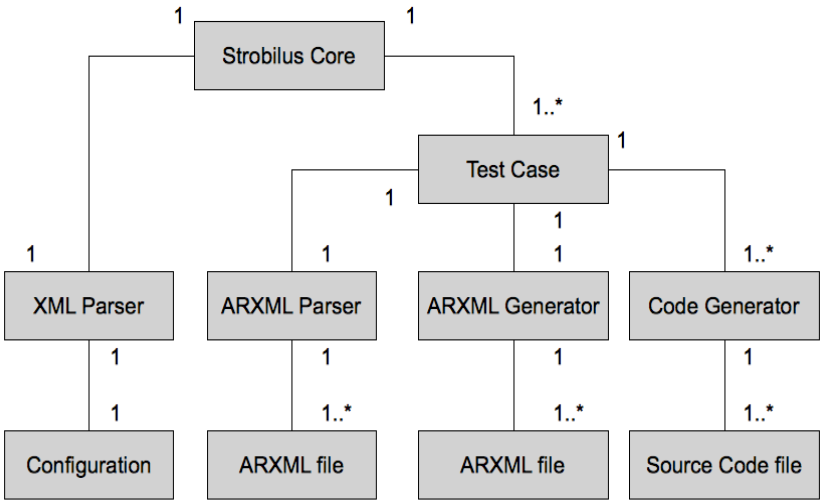

Figure 5 Strobilus domain model.

There should be support to add new test modules; this means that the code generator must be able to support new types of code for new tests. There should also be support for addition of other types of script generators than CANoe.

The approach to meet the above demands is to modularize as much as possible as well as designing the application in such a way that additions to the current code base should need a minimum of code addition. Time has also been spent on assuring that the easiness of creating modules does not limit their complexity.

2.2.1 Intermediate code representation

In order to support several programming languages and architectures modern compilers often translate source code in multiple languages into a single intermediate representation. The intermediate representation can then be optimized and used to produce machine code for multiple target systems.

AUTOSAR 4.0 has support for SWCs written in C, C++ or Java, and required test scripts can be written in any number or languages. In order to make Strobilus code output independent from its test modules, a similar approach to that of compilers was used. Modules do not produce their software components and scripts in C-code directly, but instead produce an abstract representation of what the tests should do in an intermediate language. A code generator then translates the intermediate code into source code for a specific language.

2.3 Implementation

Strobilus's modularity and extendibility was mostly achieved through extensive use of interfaces and generic types. Classes only interact with other classes through interface methods. Generic types are used to represent nodes in XML trees. This separates the Strobilus framework from the parser specific node implementation.

2.3.1 Test of parsers

During the early implementation phase, several XML parsers were tested to find one that best suited the project in both performance and usability. The performance was tested through a series of global searches using a benchmark class. The usability was evaluated by using the XML library, looking at the complexity to perform searches, creating new XML and so on.

VTD-XML

According to specifications and tests, VTD-XML looked very promising. The developer claimed it to have up to twelve times higher performance than the competitors while requiring less system resources, though no recent tests from independent sources could be found to support this. However, since performance was important, VTD-XML was chosen to be tested.

In practice VTD-XML was quite different from the competition, it was indeed very fast, but writing queries quickly became a complex task. Also, it was not possible to do local searches on smaller parts of the XML tree. Furthermore, in the specifications VTD-XML has support for writing XML, we could not get that feature to work; not even the supplied example files would produce correct results, regardless whether it was tested in Windows, OS X or Ubuntu.

Apache Xerces

Apache Xerces worked well, had lots of functionality, but there was no functionality that was needed for Strobilus that wasn't already inside the standard Java XML libraries. In our initial tests the performance was lower than the other candidates. We therefore did not see any reason to continue using the Apache Xerces implementation.

XOM

The developer of XOM stated it to be built with correctness and ease of use in mind rather than performance. Our prototyping concluded that it was very easy to both retrieve data from and write data to the XML using XOM. Performance was slightly lower than the competition, but not by much. A downside with XOM was that the XPath engine had trouble with the default namespaces used in the ARXML documents, thereby making it necessary to modify the documents for XOM to parse them properly.

Java XML Libraries

Java includes its own libraries for parsing XML, and since version 1.5 these libraries include support for XPath. The XPath implementation in Java had no problems using the default namespace in ARXML files like XOM did and would not bind the project to any additional dependency or license. Initial tests even showed the performance to be a bit better, so the choice fell on using XPath in the standard Java libraries.

2.3.2 Early Strobilus Prototype

For the prototyping of Strobilus a simplified set of AUTOSAR configuration files was used. This set is normally used for setting up a tutorial of AUTOSAR in Picea Workbench. Real configurations can be quite large, often containing over 200.000 lines of code, which was unnecessary complex to try out the prototype. The prototype aimed to test the ideas we had from the design phase.

Intermediate Representation

Strobilus test modules use an intermediate representation to describe the tests that should be produced. This intermediate representation is later on converted to script code and source code by code generator classes.

The first idea was to create this intermediate representation by using the BNF-converter. The BNF-converter is a tool created to enable developers to quickly write the front-end for a compiler. The tool takes a grammar describing the language and creates a lexer, a parser and an abstract syntax implementation. The abstract syntax implementation could then have been used as the intermediate representation for Strobilus.

The idea of using the BNF-converter was dismissed however, since there was no practical use for the front-end lexer and parser. Instead an intermediate representation was developed from scratch in Java. This approach made it much simpler to configure the various language constructs and configure how they were allowed to interact.

For the prototype a very basic syntax with very few constructs was created to evaluate the idea. Initial tests showed it to function as desired, and the same design was retained in the final implementation. The final version intermediate representation was inspired by the C syntax with a few additional constructs to represent AUTOSAR specific features, such as sending and receiving communication signals.

Problematic XML Parsing

During the test phase of the prototype, the simple ARXML-file was exchanged to a complete one, used for real vehicles. After the switch to the complex configuration file, the required searches took more than five minutes to perform. An analysis showed that the search speed was limited by the CPU, in this case a Core 2 Duo 2.33GHz.

Benchmarking showed that the XML parser wasn't multithreaded, but reducing the search time from five minutes to two and a half by utilizing both cores on the Core 2 Duo wasn't the solution to our problems.

Building a cache containing all paths to already found nodes seemed like a feasible way of improving the speed since many similar searches were performed. The idea was that the parser would be able to do searches on smaller parts of the XML tree, thus improving the performance.

Before implementing a search cache, we tried to optimize our search strategy. During the first XML tests it was noticed that global searches were expensive to perform. With this knowledge as many global searches as possible were converted into local searches. This was done either by doing smarter searches, relying on information from the AUTOSAR ARXML specification, or by reusing nodes from already performed searches.

It was discovered that converting global searches to local searches did not improve performance at all. The cause of the low performance turned out to be Java's XPath implementation. Java handles local searches almost the same way as global searches (GJ, 2006). This obviously had a very negative impact on performance.

The solution was to detach nodes from their parents before doing local searches. That way Java could not traverse any nodes beyond the local scope. The results of the fix were clear; what had taken five minutes before, now took less than ten seconds!

The search times were now considered as satisfactory and prototype testing of writing XML began. Quickly another issue was discovered. After changes had been made to an XML tree, Strobilus could write these to file, print it to screen but doing an XPath search within the changed XML would not display that anything had changed at all. The solution was to write our own XPath search engine, Strobiter. The search engine operated on Java's DOM implementation. The name Strobiter was chosen by combining the Latin word for path, iter, with the name of the application, Strobilus.

Comparing Strobiter to the other parsers

Since we had built our own implementation, we needed to do new benchmarks, to make sure that performance still was acceptable.

The benchmark tested VTD-XML, Java's XML, XOM and our own XPath implementation Strobiter. The test took the average time of ten repetitions of ten global searches, using a complex ARXML file with nearly 18000 lines of code.

The result was quite interesting, showing that Strobiter was about twice as fast as Java's XPath implementation. Strobiter was also 2.5 times faster than XOM. It did however take about 1.5 times longer time than VTD-XML. It was also noticed that none of the implementations were multi threaded.

This particular benchmark only tested global searches, which can be seen as the worst-case scenario. Strobiter has a huge advantage over VTD-XML in that it can perform very cheap local searches; something VTD-XML does not support. With these results we settled for our own implementation, Strobiter.

2.3.3 Final implementation

This part will describe what Strobilus does from execution until it stops after the relevant files have been modified or created.

During startup Strobilus main class reads the path to a configuration file from a command line argument. If Strobilus can't find an argument it uses a default path and default file name. The path will be passed to the constructor, which instantiates the main class.

The constructor uses the configuration file to read what tests modules, XML parser and code generators that needs to be loaded. The configuration file also contains information regarding what input and output paths that will be used.

The input path is used to find the AUTOSAR version. This is done by searching the input folder for ARXML files and parsing the first file found for its version information. The AUTOSAR version is in turn used to select the correct ARXML handler and connect it to the XML parser chosen from the configuration file.

Finally the class loader is fed with all information parsed, it instantiates test modules, code generators, XML parser, XML handler and executes the tests.

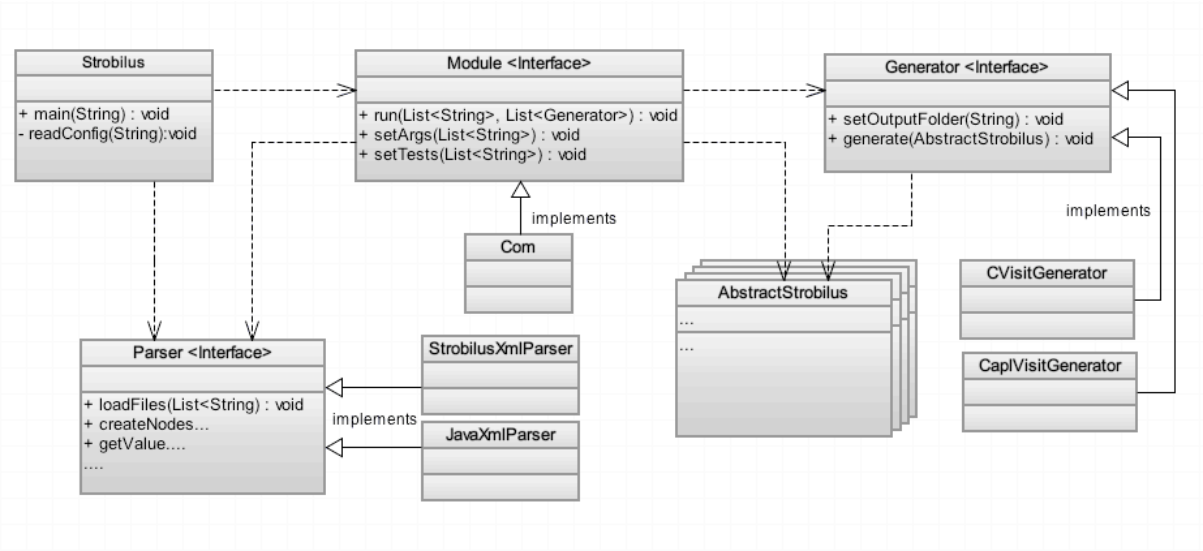

Figure 6 Simplified class diagram.

In this release of Strobilus, only a communication test module for the protocols COM – CAN is implemented. The communication test, or just COM in short, searches for signals and references in the ARXML files from the input folder. It finds a connection down from a CAN signal up to a system signal reference inside COM. It also parses the signal information for attributes such as size, data type and signaling mode. To keep execution time low, global searches were rewritten lookups with full search paths wherever possible.

With the signal information collected the COM test can now generate the XML required to implement the Strobilus software component. First the ARXML handling the basic software is modified. On this level the basic software configuration for RTE is changed to contain references to the Strobilus SWC. Callback support for the signals is added. Tasks are created and runnables are mapped to them. In this case there are two runnables, one for signal callbacks and one to run the SWC, and each of them is mapped to a different task.

On the runtime environment level, XML is added for the SWC and for the functions inside the SWC. Timers are also added in the RTE to trigger the functions in the SWC. Signal names are created and mapped to system signals, so that the C code in the SWC can access the communication paths.

All XML needed for the SWC is now created and the communication tests uses the information regarding incoming and outgoing signals to create intermediate script and SWC code. The intermediate code is then sent to the code and script generators. The generators will turn this into source code, one part that the build chain can compile to an SWC, and one part that can be used by CANoe.

To sum it up, at this point of time Strobilus has read and modified the ARXML files from an AUTOSAR project, it has created the C code needed to create an SWC and created the CAPL scripts for CANoe. The project is now ready to be built using the build chain bundled with Picea Suite. The build chain will generate the final code that is runnable on an ECU, this code will in turn be tested against the CAPL scripts that Strobilus has created.

3 Results

The results section contains a summary of the produced tool, our conclusions, a discussion section as well as a small section about the potential future developments of the Strobilus tool.

3.1 Implementation results

Main application
The main application basically consists of a file loader and a class loader that executes the rest of the classes based on the requirements in the configuration file. This has made the main class efficient and flexible.

XML parser / writer
Our first idea was to use a premade solution to reduce development time, but in the end we developed our own XPath implementation on top of Java's DOM parser to get the flexibility that we needed. The current solution for parsing and writing XML is complete and works very well.

Test modules
The only implemented test module is a simple communications test that measures periodicity and that the contents of packets does not get corrupted when signals are sent between software components and the CAN-bus.

Code / Script generator
Strobilus supports generation of C code as well as CANoe-scripts. The code generators do not currently support the full syntax of our intermediate representation, but enough to cover the needs of our single test module.

3.1.1 Example of code generation

This section contains some example code from Strobilus. For more detailed behavior of Strobilus and the communication test module, refer to chapter 2.3.3.

When the test module has discovered the relevant signals and modified the ARXML code, it must create the intermediate representation (IR) of the output code. Below you can see part of the code that creates the IR for the SWC, taken from the communications test module:

```
...
for (Signal signal : outgoingSignals) {
       Block body = swc.declareFunction(new VoidType(), Acess.PUBLIC,
                                                "test" + signal.getName, null);
       Variable ret = body.declareVariable(new StdRetType(), null);
       Variable i = body.declareVariable(new IntType(), null);

       body.print(”Testing ” + signal.getName());

       Block forLoop = body.for(i.setValue(0), i.lessThan(64), i.inc());
       forLoop.send(signal, i, ret);

       Block sigErr = forLoop.if(ret.equals(noDataConst));
       sigErr.print(“Error sending signal ” + signal.getName + “: %i”, i);

       forLoop.wait(10);
}
...
```

In an example run, there was an outgoing signal with the name *TxSig001*. When IR from the above snippet of code was sent to the C code generator, the following was produced:

```
...
void testTxSig001(){
       Std_ReturnType retVar_0;
       int intVar_1;
       printf(”Testing TxSig001”);


       for(intVar_1 = 0; (intVar_1 < 64); int_Var_1++) {
              retVar_1 = Rte_WriteTxSig001(intVar_1);

              if((retVar_0 == RTE_E_NO_DATA)) {
                     printf("Error sending signal TxSig001: %i", intVar_1);
              }
              Sleep(10);
       }
}
...
```

The above code is not the complete communication test; it is merely an example of what writing tests with the Strobilus framework can look like. The complete test contains more SWC code as well as code that creates IR for the script side. Developers can add their own tests to this module or create their own modules to test features of the AUTOSAR configurations.

3.2 Conclusion

As a concept Strobilus is a great tool, making developers able to generate new tests in just a couple of seconds. So the big question is, will Strobilus be used or not? To us that is a question of whether or not more tests will be implemented into Strobilus. With more tests Strobilus could do lot of tedious work for the developer and it could become a very helpful tool during development.

Even at the current state it eases the burden of implementing an SWC, just run the application and a C file with signaling for input and output to CAN will be generated. A developer can easily modify the C file before running the build chain. Since the signals will have the same name, a developer could hardcode a test that he pastes into the C file for every platform he wants to test. Thus Strobilus can make testing easier and faster even without implementing more tests, just by taking care of all the ARXML work.

3.3 Discussion

During the development of Strobilus there have been two big struggles. One of them has been AUTOSAR and the other one has been XML parsing, something that has been discussed throughout this report. In this section it will be discussed why AUTOSAR has been a problem and how Java's JIT made our XML problems even worse.

3.3.1 AUTOSAR

AUTOSAR is a complex standard, trying to learn AUTOSAR 3.1.4 in a very limited time proved to be a challenge. To raise the bar further, Strobilus is working with all of the three of the layers, SWC, RTE, BSW, at the same time. AUTOSAR is not made for the developers to work with several layers. This becomes apparent while trying to configure the XML-files or reading the documentation. Everything is written in a manner of working with one layer at a time.

Strobilus is built for AUTOSAR 3.1.4 but also has to support a later integration of AUTOSAR 4.0. Finding the differences between version three and four and making sure that later integration would work was not easy since the documentation is not written to show what's changed on an ARXML-level between the versions. Rather it's written on a much higher level to inform which new features that exist.

Finding a suitable set of AUTOSAR configuration files also led to a couple of problems. In the beginning of the project an easy and lightweight configuration file was needed for faster understanding and less confusion. Later on a real full configuration was needed to check that Strobilus actually worked for configuration files used in production. The switch between the simple and advanced configuration showed that we had drawn some incorrect assumptions from the simpler file, which led to a bit of extra work later on.

3.3.2 JIT

Java uses a just-in-time compiler. One of the benefits with a JIT compiler over static compiling is that JIT can optimize code during runtime of an application, thus having more information for optimization. Usually this is a good thing that can result in code even faster than it's static counterpart. One of the downsides is that it's harder to write synthetic benchmarks for a JIT compiler. For example, the compiler can optimize parts of the benchmark code in a way that skews the results.

Strobilus used benchmarking extensively to choose the most suited XML parser. The results looked normal and the parsers were relatively a like in execution time. The problem was that the JIT actually had done optimizations that skewed the results. The differences weren't enough to make it noticeable, but big enough to change the order of which parser that was the fastest.

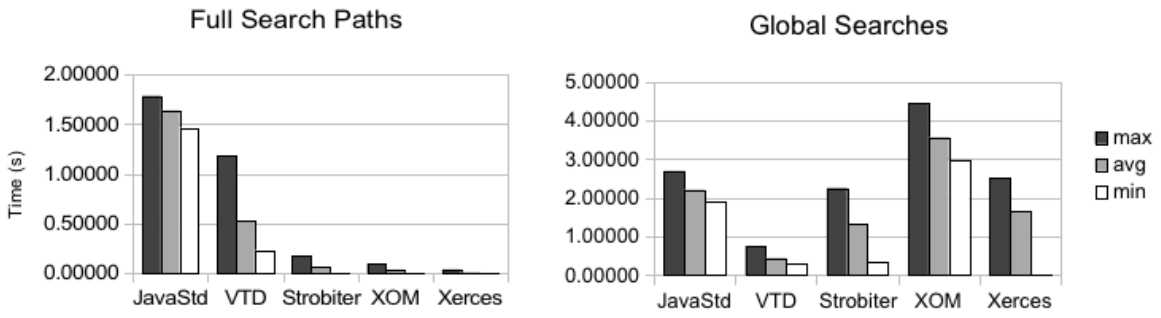

Figure 7 The max, average and minimum search times for different XML parsers in our test. To the left is a test where a full path to the element is provided and to the right the parser has to search through all elements for a specific one.

A final benchmark during the last test phase showed that Xerces would have been the fastest parser for Strobilus. This is not a big issue since the current parsing is fast enough. Still, lots of time would have been saved if we had chosen Xerces from the beginning and the performance would have been even better. It should be noted in the picture below that while VTD-XML might seem like a better choice with it's significantly lower global search time, the queries in Strobilus are almost exclusively in the full search path category.

3.4 Future development

As mentioned earlier in this report Strobilus is built very modular so that extending it with new AUTOSAR versions, tests, script- and code generators can be done as efficiently as possible without the need to rewrite any existing parts. We think that the most probable future development for Strobilus will be to implement more tests for the already AUTOSAR 3.1x core.

Even if AUTOSAR 4.0 support will be generated in form of a new XML handler and code generator, more test will be needed in order for Strobilus to reach its full potential as a testing tool.

As described in the problems section, our final benchmarks showed that we had dismissed Xerces for XML parsing without good reason. A further development could be to implement a parser using Xerces as that might give some performance gains.

Test modules write code in the intermediate language we created for the thesis project. A future development could be to create front-ends, which translates source code written in regular programming languages like Java or C into the intermediate language. This would help developers implement new test modules more quickly, they would simply import actual code or even the manual tests they used prior to Strobilus and get most of the test module for free.

References

GJ. (2006, 02 24). *Slow XPath evaluation for large XML documents in Java 1.5*.
Retrieved 06 01, 2011, from http://blog.astradele.com/2006/02/24/slow-xpath-evaluation-for-large-xml-documents-in-java-15/

HP. (2010, 04 29). *Test Automation ROI.*
Retrieved 06 01, 2011, from Harness the power of your application lifecycle:
http://h41112.www4.hp.com/events/emea/en/alm/AQM_event_2010_BreakoutSession2_Test_Automation_ROI.pdf

IBM. (2003, 06 29). *Understanding SAX*.
Retrieved 06 01, 2011, from IBM developer works:
http://www.ibm.com/developerworks/xml/tutorials/x-usax/section2.h

Mecel AB. (2010, 11 10). Examensarbete:
Testapplikationsgenerator för AUTOSAR system. Göteborg.

Oracle. (2004, 03 22). *Oracle® XML DB Developer's Guide*.
Retrieved 06 01, 2011, from Oracle:
http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10790/xdb_dbmstore.htm

Appendix A: Requirements

Bellow follows a list of the functional requirements of Strobilus.

SwRS01-01 Test Generation Output
In order to be a useful generator, Strobilus shall be capable of generating the following types of files:

- SW-C-descriptions (.h-files)
- SW-C-implementations (.c-files)
- Test scripts (e.g. CAPL)

These will together be used to test the target system as specified by the user in the input configuration.

SwRS01-02 Test Generation Input
Strobilus will require the following inputs:

- BSW configuration file(s), describing the target system
- A configuration file for Strobilus, describing for example input file version and locations, what tests to run, what scripts to generate and so on.

Input files shall be AUTOSAR .arxml files.

SwRS01-03 Configuration error reporting
If the selected test configuration contains tests that the generator does not have support for, an error will be printed.

SwRS01-04 Tests use the diagnostics protocol
The generated tests shall make use of the diagnostics protocol for communication.

SwRS01-05 Modular Design
Strobilus shall be created in a modular fashion so that support for different input and output formats can easily be added without modifying the application core.
AUTOSAR 3.1 will be the first supported version.

SwRS01-06 Command-line based
Strobilus shall be runnable from the command-line.

SwRS01-07 Autonomous testing
Once a batch of tests has been started, no further user input will be required. Should any test result in an error or otherwise fail, the next test will commence.
The user may be able to set Strobilus to abort on error in the configuration file.

Appendix B: Development tools

In this section the tools used for development of Strobilus will be described.

CANoe

CANoe is an application for controlling hardware tools connected to the CAN bus. In Strobilus the scripting part in CANoe is used to either verify the results sent from Strobilus or to send test data to Strobilus.

Checkstyle

Checkstyle is an eclipse plugin that has been used throughout the implementation phase to make sure that coding conventions are followed and that the code is easy to read.

Dropbox

Dropbox is a Web-based file hosting service that uses cloud computing to enable users to store and share files and folders using file synchronization. Dropbox was used for synchronization of project files that were not put in the code repository.

Eclipse

Eclipse is a software development environment comprising an integrated development environment and an extensible plug-in system. It is written mostly in Java and can be used to develop applications in Java and, by means of various plug-ins, other programming languages.

Gliffy

Gliffy is an online diagram software, that easily can create flowcharts, diagrams, floor plans, technical drawings, and more. Gliffy has been used to create several of the diagrams and drawings used throughout the project.

Google Docs

Google Docs is cloud based and made for collaborative work, which makes it perfect for rapidly changing documents in progress. Google Docs has been used for the project diary, notes regarding Strobilus and the early drafts of this thesis.

Java

Java is a programming language originally developed at Sun Microsystems, released in 1995. It is a general-purpose, concurrent, class-based, object-oriented language. Java derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities.

It was the wishes of Mecel that Strobilus be written in either C or Java. Java was chosen because object orientation was more suitable for Strobilus' modular and extendable design. On top of that, Java also has great built in libraries, is platform independence, and we were both more experienced in Java than C.

Notepad++

Notepad++ is a free source code editor that supports syntax highlighting for several programing languages. Notepad++ has been the main tool for manual editing of configuration files, especially ARXML files. Notepad++ proved very useful with its customizable plugins, complex search functions.

Microsoft Office 2010

Office 2010 is the default productivity suite at Mecel and was used for writing internal documents and presentations.

Open Office

OpenOffice.org is an open-source alternative to the Microsoft office suite. Open Office has been used for writing large parts of this thesis and for the creation of many of the illustrations in it, as well as some of the presentations held at Mecel.

Picea Suite

The Picea suite contains Picea Workbench and Picea Integration Package.

Picea Integration Package

The Picea Integration Package or in short PIP, consists of Picea BSW and Picea RTE together with a tool chain for generating the RTE and SWCs.

Picea BSW

The Picea BSW is the software layer below the RTE. It provides services like bus communication, memory management and operating system. The basic software is only accessible through the RTE. Each Picea BSW module comes with a standard software core and a code generator. The code generator is a tool that reads AUTOSAR configurations, validates it according to the AUTOSAR schema and generates C code.

Picea RTE

The Picea RTE is the interface between software components and the basic software. Picea RTE is optimized for a low memory footprint and minimal OS resource overhead.

Picea Workbench

The Picea Workbench is a tool made for ECU suppliers, it supports configuration of the basic software and the RTE. Picea Workbench was originally developed to meet Mecel's internal requirements of a tool that is capable of handling AUTOSAR ECUs in production programs where there can be several thousand ports and signals.

Picea Workbench is a tool that is both sold to other companies and used internally on Mecel. It's built on Eclipse using a similar user interface. For Strobilus Picea Workbench has been mainly used to browse ARXML code in a tree view manner and following references. It has also been used to validate our ARXML code and see how it merges with existing code.

Subversion

Apache Subversion is a version control system originally designed to be a better CVS. Subversion has since expanded beyond being a replacement of CVS, but still remain heavily influenced by that goal. Subversion has been the version control system of choice for Strobilus mainly because we've used it before and are familiar with it.

Subversive

The Subversive project provides Subversion integration for Eclipse. Subversive adds graphical features and dialogs inside Eclipse for SVN.

There are several subversion plugins for Eclipse, Subversive is the one that has worked best for us with Strobilus.

Testbench

Testbench is an internal Java based tool from Mecel for use together with Picea Integration Package. It can be used to send and receive virtual CAN signals from the PC version of Picea Integration Package.

Visual Studio 2008 express

Microsoft Visual Studio Express is a freeware integrated development environment, which is a lightweight version of the Microsoft Visual Studio product line. The idea of Express editions, according to Microsoft, is to provide streamlined, easy-to-use and easy-to-learn IDEs.

The free version of Visual Studio is used together with Picea Suite build script for compiling Strobilus SWC into the Picea Integration Package.

XMLSpy

XMLSpy is an XML editor for modeling, editing, transforming, and debugging XML-related technologies. XMLSpy was used for verifying the correctness of XML-files and their corresponding schema files.