A Pragmatic Approach For Debugging Parameter-Driven Software

Frank Ortmeier, Simon Struck, Jens Meinicke
{frank.ortmeier, simon.struck}@ovgu.de
Jochen Quante
jochen.quante@de.bosch.com

Abstract: Debugging a software system is a difficult and time consuming task. This is in particular true for control software in technical systems. Such software typically has a very long life cycle, has been programmed by engineers and not computer scientists, and has been extended numerous times to adapt to a changing environment and new technical products. As a consequence, the software is often not in an ideal condition. Additionally, such software often faces real-time requirements, which often makes it impossible to use dynamic techniques (e. g., single-stepping or tracing).

Technically, such software is often realized in C/C++ in a rather imperative programming style. Adaptation and extension is often done by adding configuration parameters. As a consequence, checking for correctness as well as debugging requires to consider the source code as well as its configuration parameters. In this paper, we propose a pragmatic approach to debugging such software. The approach was designed such that (a) it does not require any understanding of the software before starting, and (b) that it can be easily used by programmers and not only by experts. We evaluated the approach on an artificial but realistic case study provided by Robert Bosch GmbH.

1 Introduction

Over the last decades, computing power has risen dramatically. This lead to a completely new generation of software systems as well as new processes for software design and construction. Software development became professionalized – numerous, elaborate design processes and development tools made software construction very efficient. However, new processes and design methodologies often focus on information systems. They are very well tailored for (new) development of IT support for business processes of a company.

On the other hand, software is taking over more and more functionality in many technical applications. In a modern high-end car, the source code of the software now exceeds one million lines of code. Software development and maintenance sums up to almost 40% of the total development costs for a high-end car (according to a statement of a leading German automotive manufacturer).

Embedded software has now grown over more than three decades. As new versions are often implemented by adapting existing software controls, the set of parameters controlling the software has become very complex. In addition, documentation is often poor, and the programmers might even have retired. This makes maintenance, extension and debugging very expensive. For mid- and long-term improvements re-engineering such software systems (into modular and extensible architectures [SH04, KKLK05]) is a worthy goal. However, this requires quite some effort and time.

In this paper, we propose a pragmatic approach for locating bugs in parameter-driven software. It can be used during re-engineering (specifically during testing) as well as for locating bugs after the release. We specifically address the problem of locating bugs. To debug a software that heavily depends upon parameters, it is not enough to only inspect the source code – you also have to consider the used (fixed) parameters. Only a combined view of both will make debugging possible. The core idea of our approach for locating bugs is to make use of the fact that software often has been running successfully in many scenarios, and that variants of the software only differ in the values of the configuration parameters. This allows for a structured search during debugging. We explain and evaluate the approach on an industrial challenge supplied by Robert Bosch GmbH [BQ11]. We want to explicitly point out, that with this approach we do not vote against solid software engineering or re-engineering software for increasing maintainability. We fully support this idea. The presented approach is not meant as an alternative, but rather as a method which can be used during all stages of bug-fixing in parameter-driven software. This ranges from fixing bugs detected during development (maybe in conjunction with automatic builds/tests in a continous integration environment like JENKINS/HUDSON) as well as locating bugs after the software is released.

In the following section (Section 2), we give a brief introduction to parameter-driven software and explain the challenges. The proposed approach for debugging is explained in Section 3, followed by a description of the industrial challenge and an application of the approach to this example (Section 4). Related approaches are discussed in Section 5. A summary and an outlook is given in the final section 6.

2 Parameter-driven Software

In many technical domains – for example automotive industry or production automation – control software has grown over several years or even decades. Over the years, more and more functionality was added and/or existing control software has been adapted to new series of products. Typically, this calls for a modular software architecture. However, in industrial practice, evolution/adaptation is most often controlled by parameters. Software controls in technical domains usually rely on a large set of parameters for correct functioning with the concrete hardware and physical process that is to be controlled. Each set of parameters is designed for optimal results for a very specific type of product – for the product that the software shall control. A certain share of these parameters is usually not adjusted by the company that develops the software, but by a different company that integrates it into its own hardware. We call the software itself together with a set of values for all parameters a *variant* of a *parameter-driven software*.

For such software, the choice of parameters typically heavily influences the behavior of the whole control software. Some parameters might even be used to (de-)activate major parts of the code in a top level if-statement. Therefore, we call such a software "parameter-driven". A parameter driven software program allows the user to specify and modify specific operations by only changing parameter values. This allows for highly customizable software. Of course, this comes at a price: understanding the software does not only require understanding the source code, but also understanding the parameters and specific sets of parameter values for a given product. This also includes understanding dependencies between different parameters.

Note that a parameter-driven software system can also be understood as a *software product line* (SPL): parameters that are evaluated at runtime are one way of implementing variability. However, in this paper, we do not focus on reengineering the existing software to a maintainable software product line, but rather describe a pragmatic approach for systematically debugging such a software. The presented approach has only been applied to parameters that are treated by the compiler, but it can also be applied to parameters that are directly manipulated in the binary. It also seems possible to extend the approach such that parameters for pre-processors (for example those used in *#ifdef* statements) can also be treated. In contrast to the parameter-driven software, *#ifdef*s typically alter the source code. However, if *#ifdef* statements are only used in a structured, "disciplined" way (i. e., only to in-/exclude entire functions or type definitions or sequences of entire statements in the source code [LKA11]), they could also be realized by a fixed parameter and a normal *if* statement.

3 Parameter-driven Debugging

In the following, we describe a pragmatic approach for systematic debugging of a parameter driven software. We assume that the source code as well as a number of parameter configurations is given. We also assume that documentation, design specification and expert knowledge is no longer available – at least not to such an extent that debugging is a trivial task. We further assume that a bug has been observed in one specific variant of the software, and that there exists at least one variant that behaves as specified.

Debugging is the process of reducing errors in computer software. This process is usually split in two separate tasks. First, the affected code lines need to be identified. Then the source code must be modified in such a way that the error disappears. Fixing the affected lines of code strongly depends on the source code itself and possibly requires much more insight knowledge. In our approach we thus concentrate on the bug localization step. However, knowing a limited number of erroneous lines of code greatly assists the subsequent correction of the bug.

Locating an error is a difficult task that can be tackled with various methods. In the case of parameter-driven software we make use of the fact that the software has typically been running in numerous variants for a long time and the bug only appears for a newly created variant. The core idea is to compare parameter sets of variants that are working correctly with those of the faulty variant. This is comparable to searching for an error in a new version of a previously updated software by looking at the differences in the source code made during the update. This process is applicable even if the developer has only less knowledge of the source code.

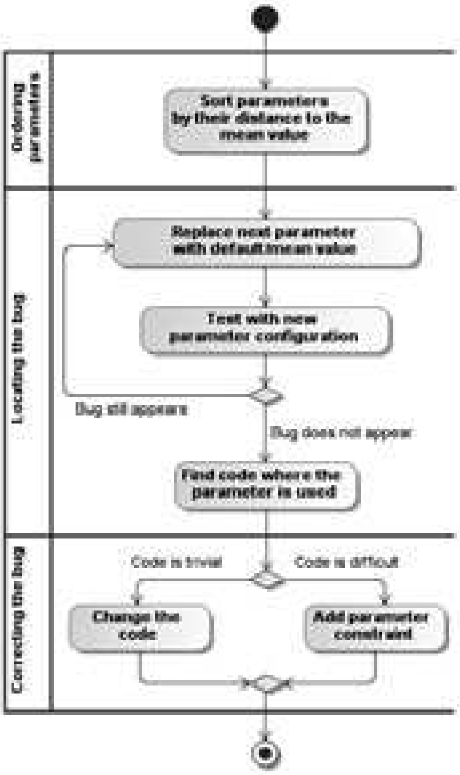

Ordering parameters
Sort parameters by their distance to the mean value
Locating the bug
Replace next parameter with default/mean value
Test with new parameter configuration
Bug still appears
Bug does not appear
Find code where the parameter is used
Correcting the bug
Code is trivial
Code is difficult
Change the code
Add parameter constraint

Figure 1: Overview of the approach for systematic debugging of parameter-driven software

Figure 1 shows the proposed approach. It can basically be split into three steps. In a first step, we try to identify candidates of possibly critical parameters. The good thing of parameter-driven software is that the source code itself is the same for all variants. So it is obvious that changes in the parameters are a good starting point for the search. The next step is to identify the “responsible” parameter(s) and the corresponding parts of the source code. Finally, the bug must be corrected.

3.1 Ordering parameters

In practice, a parameter-driven software will have hundreds or thousands of parameters. For a given variant and a given parameter, there most probably exists at least one other variant that differs in this parameter. Therefore, an effective search strategy is needed to identify the parameter(s) that cause the faulty behavior. A search strategy basically is an ordering on the set of parameters, which defines which parameter shall be analyzed first. In our approach, we want to present a generic concept. Thus, we may not rely on any application specific background knowledge.

An effective search strategy – if no background knowledge is available – is to start with the most "differing" parameter. There are different possible definitions of "differing". In this paper, we only use two intuitive definitions – (numerical) deviance from the mean value of the parameter and a quantile-based metric. The idea of the first one is that it makes sense to first look at parameters with a very large or very small absolute value (compared to the average). This is inspired by the fact that errors often occur if boundary values are used. The quantile-based "differing" metric basically follows the idea that if a parameter has only very rarely been used with a specific value, then it is more probable to cause an error than a parameter that is frequently used with the same value.

There may exist various other metrics like ordering by "runaway values" or even metrics that rely on domain-specific information. As each of them could be plugged in easily into the proposed approach, we do not go into more detail here. However, we found that even the two used, simple metrics allowed to quickly locate bugs in a third-party parameter-driven software of moderate complexity.

3.2 Locating the bug

After the identification of the most likely error causing parameters, we can substitute the parameters one by one with a default value and run the software. If the bug still appears, we proceed *incrementally* with the next parameter. Here, we follow the ordering of the parameters we obtained in the previous step. Typically, this will terminate at some point as we assume that the configuration of the software with all parameters set to default values behaves correctly.

Once the error disappears, we start the second step of the localization process[1]. Now that we have identified parameters that might be causing the error, we need to understand their effect. The first step is to find the place(s) in the source code where this parameter is used. Locating can be done easily with most IDEs. For example, in the Eclipse IDE, a function called "Call Hierarchy" solves this problem automatically. The "Call Hierarchy" view shows callers and callees for a selected Java member. It is also available for some other languages – in particular for C, which is interesting for the following real world example. The "Call Hierarchy" of a Field, Method or Class will show all places in the code that use it. Applied to the selected parameter, this will lead us to places in the source code where this parameter is used. Of course, the parameter could be used at numerous locations. However, as the software is parameter-driven, it is quite probable that the parameter is either only used a very limited number of times, or that the found locations may be re-

[1]Note that the debugging process depends on the order introduced by the concept "differing". Although it does not specifically address pairs of parameters, all such combinations are implicitly treated as if the parameters are set to default values *incrementally*. This means if the first parameter did not remove the bug, then we leave the first parameter at the default value and proceed with the second one.

duced significantly (e. g., by removing locations where it is only passed through from one function call to another).

Note that in theory, error localization might of course become hard if two parameters are far from each other according to the used metric *and* are used at very different locations in the code. However, it proved very efficient in practice as such situations seem to be rare. If such situations are common, the approach could be extended in various ways. We discuss such limitations and experiences after the case study.

3.3 Correcting the bug

The above steps reduced the whole source code to a few lines of code that are candidates for causing the error of the actual configuration. To fix the bug, one may now use any debugging strategy or code visualization technique of common software development you know. One well-suited technique for this step might be program slicing to find those pieces of the code that are influenced by the parameter.

Note that, after debugging the variant, in the worst case, some or even all other variants might have become faulty. So it is strongly suggested to (at least) run unit tests on some or all working variants. During bug fixing, the programmer usually focuses on the (single) not working variant and might easily forget about all the other perfectly working variants. So *before* fixing, the programmer must be made explicitly aware of this situation. A combination with documentation of changes and reasons for adaptation is also strongly suggested but not in the scope of this paper.

4 Case Study

This section describes the application of the proposed approach on a small system and scenario that was provided by Bosch and Microsoft Research as an “industrial program comprehension challenge” for ICPC 2011 [BQ11]. The team from University of Magdeburg successfully participated in this challenge, using the technique that is presented in this paper.

4.1 An Industrial Challenge

The task for the participants of the ICPC challenge was to find and fix a number of bugs in a robot leg control software. The (artificial) code contained typical elements of embedded control software, such as filters, ramps, and curves. It also showed the typical high variability that makes embedded software adaptable to different variants of hardware settings and customer requirements without changing the code. This was in particular reflected by a high number of application parameters.

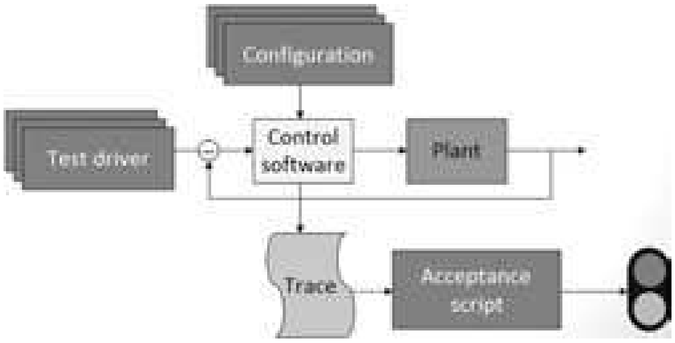


Figure 2: Test environment setup, as provided to the participants.

The core functionality of the software was to move a robot leg to a user-requested position. Certain limitations of the engine had to be obeyed for optimal performance of the leg. The software worked in general, but there were a number of bugs in the code. In particular, the following three bug reports were sent in by different (fictitious) customers:

1. The robot leg sometimes moves too slowly. In the example test case, one particular movement takes about 70 seconds, where it should only take 10 seconds. Other movements in the same test case are fast enough.
2. When the robot leg reaches its target position, it jiggles around instead of stopping.
3. Sometimes when it is moving, the robot leg explodes, sending shrapnel everywhere. After destroying 3 or 4 robots, the engineers believe the cause is due to the voltage that is sent to the engine being inverted too quickly. Also, sometimes the engine voltage drops much too quickly.

The participants got the following artefacts to work on[2]:

- The robot leg controller code (about 300 lines of C code, containing about 35 application parameters),
- a test environment, consisting of a simulation of the robot leg and a test driver (about 600 lines of C code),
- three bug reports, along with the concrete parameter configurations and test cases that show the erroneous behaviour,
- documentation (i. e., description of general idea and application parameters), and
- an acceptance test script to validate correct controller output.

For this challenge, the parameters were provided in separate header files. In reality, these parameters are usually directly changed in the binary (using special editors).

Figure 2 shows the overall setup of the test environment. The control software is only complete in combination with one of the configuration parameter sets. Different test drivers

[2]The complete challenge description and material is available at `http://icpc2011.cs.usask.ca/conf_site/IndustrialTrack.html`, accessed on 28 November 2012

simulate different scenarios, and the “plant” simulates the behaviour of the real-world robot leg. Finally, an acceptance script decides whether a test run was successful or not. In summary, we have a complete automated test environment for the robot leg controller software.

4.2 Applying the Approach

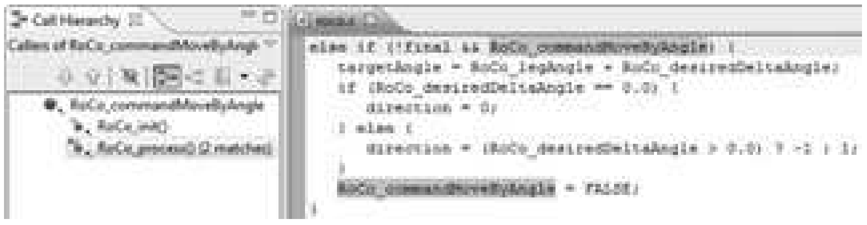
In this section, we describe the application of our approach to the ICPC industrial case study. Only one of the four variants of the product that were provided by the challenge was probably correct. The other three variants where faulty. As we will explain below, we were able to locate and fix all three bugs with our approach.

As we described above, we assume that no or only little knowledge or understanding of the code is available – and in fact, we (University Magdeburg) had absolutely no understanding of the code in the challenge when we started.

We start with step one of the approach, which is to find outliers in the parameters’ values. Note that we did not implement an automatic ordering in the prototype, but rely on visualizing the metric to the programmer. This semi-automatic approach helps a lot if one is not sure which metric to use first. Technically, automatization would be straight forward. However, choosing the right metric, or even deciding how long to proceed with one metric and when to use another metric, is an interesting question.

4.2.1 Locating and fixing error 1

When comparing the parameter configuration of this variant to the configuration of the correct variant (and also to the other variants that showed different bugs), we found that the Boolean parameter *RoCo_commandMoveByAngle* was only used in the faulty variant (quantile-based metric). All other three variants used this parameter with a value “0”, while in this variant, it was defined as “1”. Changing the parameter to the default value “0” made the bug disappear. Examining the code – i. e., applying the *Call hierarchy* method – found only the single code sequence shown in Figure 3 using the parameter.


Call Hierarchy
Callers of RoCo_commandMoveByAngle
RoCo_commandMoveByAngle
RoCo_init()
RoCo_process() (2 matches)

```
else if (!final && RoCo_commandMoveByAngle) {
   targetAngle = RoCo_legAngle + RoCo_desiredDeltaAngle;
   if (RoCo_desiredDeltaAngle == 0.0) {
      direction = 0;
   } else {
      direction = (RoCo_desiredDeltaAngle > 0.0) ? -1 : 1;
   }
   RoCo_commandMoveByAngle = FALSE;
}
```

Figure 3: The snipped of code which causes the first error

In this code snippet, we see a frequently used communication pattern in embedded software. To satisfy real-time constraints, embedded software is usually organized in time slices and uses a blackboard architecture [SCTQ09]. This means that messages between components – in particular in different time slices – are realized by writing to and reading from global variables. In the example, different types of commands are controlled by different values of variables. More concretely, the Boolean parameters control whether the robot leg will move to an absolute position or move relatively to its current position.

In order to fix the error, one still has to understand the source code. However, this becomes much easier now. For example, breakpoints or debug output can be introduced at exactly this part of the code (or even at all locations where the parameter is used). In this example, it is relatively easy to see that the assignments to the *direction* variable is wrong.

4.2.2 Locating and fixing error 2

For the next error, the quantile-based ordering of parameters did not show any significant parameters. Therefore, we chose to use the "deviance-to-mean" metrics.

Please note that two different metrics can be beneficial for visual analysis. One can either use the metric

$$\frac{\text{parameter-value}}{\text{mean-of-parameter-value}} \tag{1}$$

or its reciprocal. The first one is the standard, where one would plot the percentages of deviation to the mean. So if one parameter has a relatively high value, it will result in a peak in this plot. However, if a parameter has a very low (absolute) value, it will only be plotted around 0 and might thus be overseen. If the second metric is used, then both situations are switched: small values will become peaks and large ones will be plotted around 0. In this example, the second metric is more useful, as obviously very small values are used in this variant.

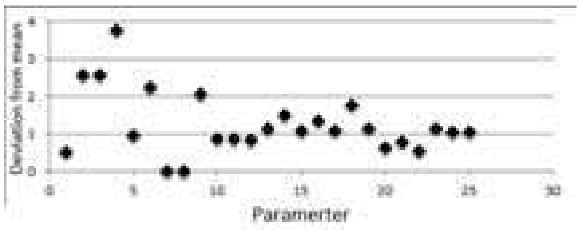
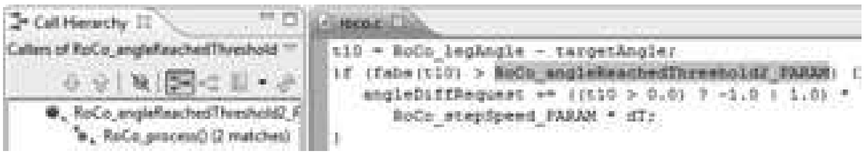


Figure 4: Deviation of parameters of variant 2 from mean values. The horizontal axis shows the 25 parameters. The vertical axis shows the corresponding result of the metric for small values.

In Figure 4, the deviation of parameter values of variant 2 are shown[3]. It does not contain Boolean parameters and parameters representing curves, because the metric needs numerical values. So there are only 25 parameters left in the diagram.

[3]Note that we did not include the Boolean parameters in Figure 4, as this metric does not make sense for them. Therefore, only 25 of 35 parameters are shown.

We see that the fourth parameter *RoCo_angleReachedThreshold2_PARAM* differs the most from its average value. It is almost four times smaller compared to the average. We start with this parameter and replace it with a default value. Now, we see that the error does not appear anymore. This means that the error is caused by this parameter. We then search for the usage of this parameter in the source code, which leads us to the lines of code shown in Figure 5.


Call Hierarchy
Callers of RoCo_angleReachedThreshold
RoCo_angleReachedThreshold2_F
RoCo_process() (2 matches)
roco.c

```
t10 = RoCo_legAngle - targetAngle;
if (fabs(t10) > RoCo_angleReachedThreshold2_PARAM) {
    angleDiffRequest += ((t10 > 0.0) ? -1.0 : 1.0) *
        RoCo_stepSpeed_PARAM * dT;
}
```

Figure 5: Code causing error 2.

Fixing the error requires understanding the code again. Here, the error is caused by an invalid combination of the threshold and step speed parameters. Considering the error description “jiggling” we decided to fix the bug, by implementing an adaptive dampening depending on the *RoCo_angleReachedThreshold2_PARAM* value.

4.2.3 Locating and fixing error 3

For finding the cause of the third error, we look at the deviation from mean values again (see Figure 6).

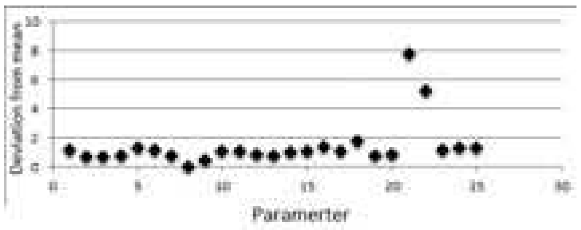


Figure 6: Deviation of parameters of variant 3 from mean values.

It is easy to see that two parameters – 21 and 22 – differ a lot from their average: they have very low absolute values. These parameters are called *RoCo_TimeSlopeNeg_PARAM* and *RoCo_TimeSlopePos_PARAM*. If we replace their values by default ones, we see that the voltage no longer drops rapidly, and the bug disappears. This means that these parameters are responsible for the faulty behaviour. The “Call Hierarchy” applied to both parameters leads us to the call of the method *Ramp_out()* in the *RoCo_process()* method.

The affected lines of code include relatively difficult mathematics (which is used for calculating velocity profiles). It is now again an application-specific task to correct this bug. If the mathematics may not be correctly re-designed such that they can deal with the given parameters, then the it should be explicitly noted what acceptable ranges for parameters are[4]. For example, this could be made visible in the parameter description by a note and could also be integrated into the source code by range checking at runtime, which might for example trigger an exception indicating that the parameter has been used with incorrect values[5].

4.3 Experiences

While applying the approach to the industrial challenge, we were surprised how easy it was to locate the source code snippets that were responsible for the bug (even for us who had no understanding of the code before). Although we only used relatively simple metrics, they quickly led us to the bugs. This is even more interesting as only a very limited number of parameter sets was available. We did all the computations on the parameters in an Excel worksheet. However, it would be easy to integrate this functionality as a plug-in to Eclipse or Visual Studio. We also found that – theoretically possible – complex combinations of parameters values that are scattered throughout the code are rare. Currently, we are examining in another case study if this hypothesis also holds in larger applications.

Note, that the presented approach is about *locating* bugs in parameter-driven software. *Correcting* the bug is still a difficult task that cannot be solved uniformly. It also requires a good amount of program understanding that still has to be accomplished manually or by using adequate tools (such as a slicer). The advantage of the approach is that it quickly points the developer to those spots in the code that might be responsible for the bug. This reduces the overall required effort significantly.

Although we used this approach in an a posteriori scenario, where variants already existed and were deployed, we think that it will be of equal use during (re-)engineering of such applications. In particular a combination with highly iterative builds in a continous integration environment might be promising.

5 Related Work

In this section we first take a broader look at alternative approaches. After that the other submissions to the ICPC 2011 challenge are discussed.

A feature model depicts the hierarchy and relation between features of a software product line in a formal way [Bat05]. Thus, there is a lot of research on debugging product lines based on the feature models. For example Batory as well as Benavides et al. propose the usage of a constraint satisfaction problem to decide between valid and invalid variants with respect to the feature model [Bat05, BMAC05]. White et al. also continue this approach by determining the most minimalistic changes to a given invalid configuration to solve all conflicts [WSB^{+}08]. Of course, these techniques perform well if there is a feature model

[4]Of course, such situations must not happen and therefore we should note it explictly in the specification
[5]This could of course also be combined with other source code annotation methods like SPEC# or JML.

and/or a precise specification of the product line. Due to the missing of a specification for the case study used in this paper their approaches are not applicable here. The absence of a feature model is not uncommon. Either a simple software product evolved to a product line over the time and thus has no product-line specific design. Or under strong time-to-marked requirements there simply might not be enough time or motivation to develop a complete feature model. Our approach performed well even without further specifications or domain specific knowledge.

Testing is an well known technique to detect failures before deploying a product. This holds also for software product lines. We believe that an elaborated test process could point out at least some of the demonstrated failures. But as well as for the feature model based debugging approaches the process of testing starts in a very early stage of the software design [PM06, MvdH03]. In addition, testing also requires detailed knowledge of the domain and/or the implementation. Thus, locating the bugs in this case study based on an elaborated test process was not feasible.

Statistical analysis of the run-time behaviour of a program can also lead to the buggy core regions [LYF$^+$05, CLM$^+$09]. A clear benefit of these approaches is the assumption that there is no further knowledge about the semantics of the program. The statistical data about the program execution is collected via program instrumentation. Due to limited resources (memory and computational) or strict real-time requirements in embedded systems the insertion of additional code might be impossible. Also these approaches do not exploit the information that the bugs might depend on certain parameter configurations.

In the original ICPC challenge [BQ11] there were four other submissions that applied different techniques to identify and fix the bugs:

- A Frama-C[6] based approach, using interpretation, slicing, and tracing. This approach is technically very complex, requires a strong infrastructure and good knowledge of the tool, but identified all the faults in the code. It also provided a good level of program understanding.
- Control and data flow visualization based approach. This one provided the weakest results, probably due to wrong interpretation of the extracted flow graphs.
- Spectrum-based fault localization [AZvG07] (two submissions). This technique identified some of the bugs, but completely missed others.

In comparison to our approach, all other submissions either require an advanced infrastructure and a high level of knowledge about it (Frama-C), or they were much less successful in the challenge. Our approach is by far the most lightweight and easy to learn, and therefore also the one that is closest to applicability in daily practice. Additionally it provides a well-balanced combination of automation and manual understanding by pointing to parts of the code that one should look at in more detail without causing much effort.

[6]`http://frama-c.com/`

6 Conclusion and Outlook

We presented a lightweight approach for locating bugs in parameter-driven software. The approach is designed according to two simple requirements from industrial practice: (i) It should be easy to apply, and (ii) it should be as generic as possible. We achieved this by structuring the debugging process into two steps. First the the responsible parameter is identified, and then the locations in the source code are identified. To meet both requirements, we only rely on standard functionality that is available in most IDEs and a simple methodology for locating the responsible parameter. As a consequence, the approach can be applied in many domains by practitioners without much additional training or software. This distinguishes the approach from most others. However, the approach does not provide a recipe for fixing the code. This part still requires a good level of understanding of the program. Manually gaining such an understanding is usually very expensive. Therefore, an additional strategy or tool is needed to support this process. One probably well-suited approach is program slicing, which can indicate those parts of the code that are influenced by an identified parameter.

We successfully applied our approach to an industrial case study provided by Robert Bosch GmbH. The case study was presented as an industrial challenge on the International Conference on Program Comprehension 2011. It is important to note that the case study firstly stems from an industrial practitioner and secondly was defined completely independent of the presented approach. Nevertheless, we where able to quickly locate and fix three different bugs. Efficiency and practicability was judged by the evaluation board of the challenge to be very good. Only a single other approach was rated better (Frama-C), but this one relies on very elaborate static analysis methods. It is further worth to note that the complete analysis was done by a bachelor student who had no a priori experience in parameter-driven software. Therefore, we believe that the approach is easily usable in industrial practice. In addition, it is easy to adapt the approach to specific needs of various domains or companies. This ranges from domain specific orderings and/or integration of automatic tools for calculating the orderings.

An extension of the proposed approach could be a combination with a dynamic coverage differencing approach (similar to the feature location technique by Wong et al. [WGHT99]): the program would be executed twice, first with the faulty configuration, and after that with the configuration where only the faulty parameter is changed. In the next step, the executed lines of the two runs are compared to each other. This could help to identify parts of the code that are only executed with certain parameter settings. Similarly, a combination with tools for static analysis (which can for example extract control flow graphs) seems promising.

In future work, we will investigate how good the presented orderings work for other parameter driven programs, and if the approach can be applied to other types of software (e. g., software with many *#ifdef* statements) as well. In particular, we also plan to evaluate it on a large scale software system from the automotive industry.

References

[AZvG07] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART'07)*, pages 89–98, 2007.

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer Berlin / Heidelberg, 2005.

[BMAC05] David Benavides, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. Automated Reasoning on Feature Models. In *Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 381–390. Springer Berlin / Heidelberg, 2005.

[BQ11] Andrew Begel and Jochen Quante. Industrial Program Comprehension Challenge 2011: Archeology and Anthropology of Embedded Control Systems. In *Proc. of 19th Int'l Conf. on Program Comprehension (ICPC)*, pages 227–229, 2011.

[CLM$^+$09] T.M. Chilimbi, B. Liblit, K. Mehra, A.V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Proc. of the 31st International Conference on Software Engineering (ICSE)*, pages 34–44, 2009.

[KKLK05] K.C. Kang, M. Kim, J. Lee, and B. Kim. Feature-Oriented Re-engineering of Legacy Systems into Product Line Assets–a Case Study. In *Proc. of 9th International Software product lines conference (SPLC)*, pages 45–56, 2005.

[LKA11] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM Press, March 2011.

[LYF$^+$05] C. Liu, X. Yan, L. Fei, J. Han, and S.P. Midkiff. SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[MvdH03] H. Muccini and A. van der Hoek. Towards testing product line architectures. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 82(6):99–109, 2003.

[PM06] Klaus Pohl and Andreas Metzger. Software product line testing. *Communications of the ACM (CACM)*, 49:78–81, December 2006.

[SCTQ09] Vincent Schulte-Coerne, Andreas Thums, and Jochen Quante. Challenges in Reengineering Automotive Software. In *Proc. of 13th Conf. on Software Maintenance and Reengineering (CSMR)*, pages 315–316, 2009.

[SH04] M. Staples and D. Hill. Experiences adopting software product line development without a product line architecture. In *Proc. of 11th Asia-Pacific Software Engineering Conference (APSEC)*, pages 176–183, 2004.

[WGHT99] W. Eric Wong, Swapna S. Gokhale, Joseph R. Horgan, and Kishor S. Trivedi. Locating Program Features using Execution Slices. In *Proc. of the IEEE Symposium on Application-Specific Systems and Software Engineering & Technology*, pages 194–203, 1999.

[WSB$^+$08] J. White, D.C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated diagnosis of product-line configuration errors in feature models. In *Proc. of Intl. Software Product Line Conference (SPLC)*, pages 225–234, 2008.