2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

# How to achieve verification closure of configurable code by combining static analysis and dynamic testing

Antonello Celano, STMicroelectronics, Milan, Italy (antonello.celano@st.com)

Alexandre Langenieux, MathWorks, Grenoble, France (alexandre.langenieux@mathworks.com)

***Abstract*— Software is becoming more and more important in the semiconductor industry to further differentiate between vendors. Silicon companies are developing software to 1) provide a software stack to leverage their silicon capabilities and 2) enable their customers to quickly jump-start their designs by providing a basis for ready-to-use software and middleware. Using software variants is becoming a popular method of supporting the diversity of possible hardware configurations, preventing source code duplication, and minimizing the footprint of the executable firmware. Due to the heavy use of software variants, the exponential increase in possible software configuration is reaching the limits of traditional verification and validation (VnV) methods. This paper proposes a novel framework that selects and analyzes only a subset of all software variants while providing the same guarantees as if all combinations had been analyzed, thereby reducing verification efforts without losing quality. This framework leverages results from structural code coverage to select the subset of software variants on which tests are executed and static code analysis performed. Before releasing the software, developers can detect bugs and violations of coding guidelines for every possible software configuration, which would be impossible with either static analysis or dynamic test in isolation.**

***Keywords—Software Variant, Configurable Code, Software Verification, Static Code Analysis, Code Coverage***

## I. INTRODUCTION

The Software Engineering Tools team at STMicroelectronics (STM) has developed the methodology described in this paper to support the developers of the AUTOSAR software team. This Team develops MCU drivers and CAN Module drivers for the Power PC and ARM STM device families. Many industries, including the automotive industry, use these microcontrollers, for which the embedded software must comply with safety and security standards such as ISO 26262 and ISO/SAE 21434. These standards define a strict development process, with many verification and validation requirements. The required methods for the highest level of safety (ASIL) or security (CAL) include various dynamic, functional, robustness, and structural testing activities. They also require strict software design and implementation, usually enforced using static code analysis techniques and tools. It usually implies that the software code follows coding standards such as MISRA-C, AUTOSAR-C++-14, or CERT-C.

MCU drivers and CAN Module drivers are highly configurable through preprocessing directives in C or C++ code (such as `#ifdef` preprocessor statements). Many software variants are generated from configurable software according to the preprocessing configuration. The MCU driver and the CAN Module studied in this paper have respectively 58 and 50 different Boolean preprocessing directives. Considering that Boolean directives are only a subset of all directives and that the number of possible variants is even larger than those computed just for these directives, the number of different software variants derived from the configurable code is considerable.

That massive number of software variants makes it challenging to verify and validate the software for all the possible configurations. How can engineers ensure thorough testing of all the software code? For safety and security-critical software, how can engineers ensure that all the code is covered by tests for every possible software configuration and that the entire software source code complies with coding standards? Current VnV tools, both commercial and open-source, such as structural coverage testing or static analysis tools, do not provide a solution.

To support the developers in addressing this combinatoric challenge, the Software Engineering Tools team at STM has developed a methodology supported by tools and processes to leverage the structural code coverage results to optimize the testing activities and check for code adherence to guidelines.

## II. THE CHALLENGE OF THE VERIFICATION AND VALIDATION OF CONFIGURABLE CODE

To support its automotive customers, the AUTOSAR Software Team (AS Team) at STMicroelectronics must develop basic software that (1) complies with the AUTOSAR architecture, (2) complies with safety and security standards such as ISO 26262 and ISO/IEC 21434, and (3) is extensively configurable to support the unique ECU specificities required by different Automotive OEMs. To fulfill these requirements, the AS team has set up an architecture relying on AUTOSAR components consisting of configuration parameters and metacode that is added to STM-developed applicative C code. The set of metacode, AUTOSAR variant parameters, and source code is referred to later as the "STM Configurable Code".

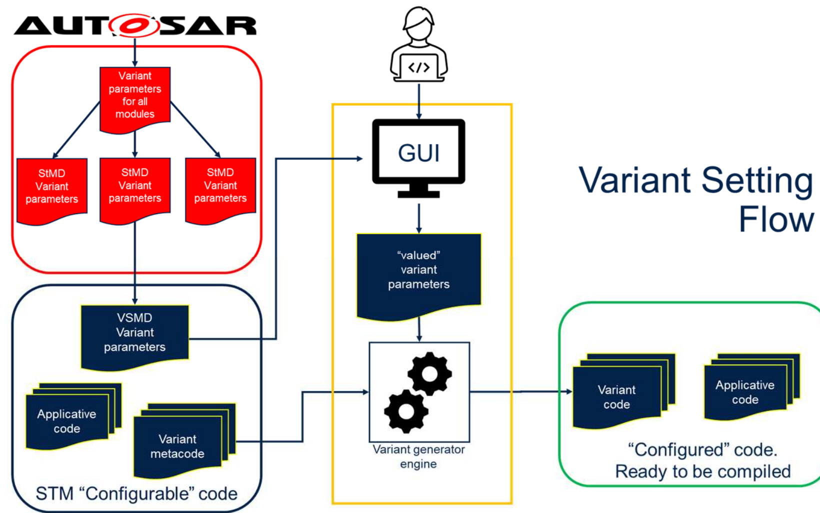



Figure 1 - Variants settings workflow

To generate the final configured code specific to a customer from the STM Configurable Code, engineers of AS Team set the value of the parameters to define the code's behavior for the customer project. The setting of the parameters defining a variant is an ad-hoc process. These parameters are used as input of the variant metacode to generate C code (referred to later as "Variant Code," see figure 2) that will be compiled with the STM applicative source code to create the software fitting the customer requirements.

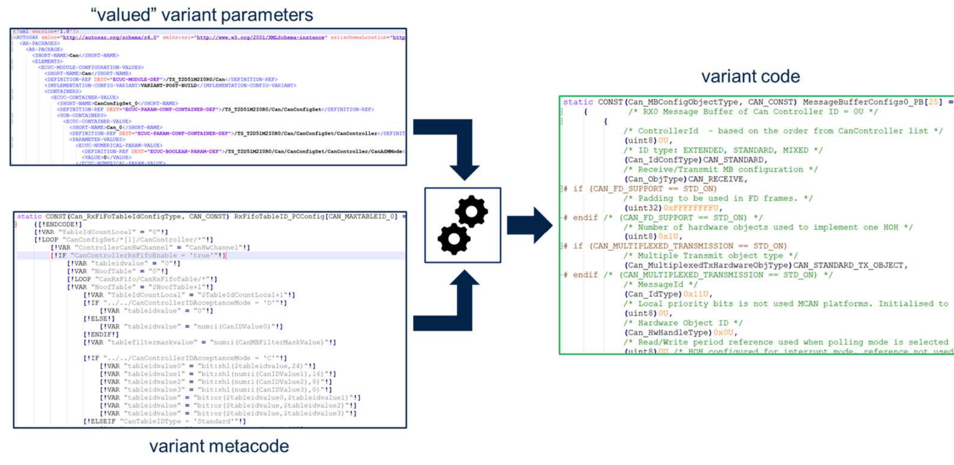



Figure 2 – Process to generate variant code from metacode and variant parameters

The number of all possible software variants that can be generated from the STM configurable code is extremely high (see *Table 1*), making the exhaustive verification and validation of all of them impossible to achieve. Before adopting the methodology described in this paper, the AS Team focused only on verifying and validating the software variants that their customers requested. As the variant selected by each customer was unknown during the software development, the verification and validation of the specific software variant were done afterwards, leading to the discovery of issues after the software was deployed. The cost of fixing those issues was high because they were detected too late, and the confidence in the developed software quality was low due to the lack of completeness of the VnV.

| Project | Number of variant parameters | Number of boolean preprocessor macro (#define) | Number of software variants considering only boolean parameters |
|---|---|---|---|
| MCU Driver | 357<br>(208 booleans, 78 enumerations, 103 integers, 9 strings) | 58 | $2^{58} = 288*10^{15}$ |
| CAN Module | 97<br>( 32 booleans, 25 enumerations, 39 integers, 9 strings) | 50 | $2^{50} = 10^{15}$ |

Table 1 - Number of software variants estimated considering only boolean preprocessor macros

To improve its efficiency and the quality of the delivered software, the AS team has defined a new methodology and developed a toolchain to support it. This methodology consists of selecting a subset of variant parameters that enable the full structural code coverage through functional testing and then performing static code analysis on the same subset of software variants. This methodology optimizes the VnV effort of the AS team developers while increasing the quality of the delivered software by providing a broader VnV scope (not only on customer variants) and faster feedback on potential software issues to be fixed by developers.

## III. DEFINING MINIMAL VARIANT SUBSET TO ACHIEVE THE VERIFICATION CLOSURE OF A CONFIGURABLE CODE

The AS Team's new methodology is iterative and based on developer expertise. It includes a specific process to be followed by all developers. This process is supported by a dedicated toolchain developed and maintained by the AS Tool team.

The starting point of the process is the selection of a first subset of variants that aims to achieve the full structural coverage of the configurable code. The variant parameter selection is based on the expertise of the developer, who knows which variant parameter will include specific code segments into configured code and how the variant parameters impact the functional behavior of the software. Once this first set of variants is selected, the developer runs the structural coverage tests on the hardware target to evaluate the amount of reachable code in the overall

configurable code. Statement coverage, branch coverage, and modified condition/decision coverage (MCDC) are used to identify which statements and branches have been executed for each variant.

This dynamic analysis provides two types of results:

- Coverage scores are the percentage of code executed during all the tests on the executable code of the subset of variants over the complete source code. The complete source code is an overset of the executable code of each variant, as the execution of some pieces of code depends on the preprocessor macro and could be unreachable for some variants. For example, in figure 3, the entire code contains 15 lines of code (including the function header and brackets). If the preprocessing macro CAN_DEV_ERROR_DETECT is defined as STD_OFF, then eight lines of code are executable (underlined in green) and seven lines of code are not activated in the software variant (surrounded by red boxes).

```
1  #if (CAN_VERSION_INFO_API == STD_ON) || defined(__DOXYGEN__)
2
3      FUNC(void, CAN_CODE) Can_GetVersionInfo( P2VAR(Std_VersionInfoType, AUTOMATIC, CAN_APPL_DATA) versioninfo)
4      {
5
6      #if (CAN_DEV_ERROR_DETECT == STD_ON)
7          /*
8          (CAN177) If development error detection for the Can module is enabled: The function Can_GetVersionInfo shall
9                      raise the error CAN_E_PARAM_POINTER if the parameter versionInfo is a null pointer.
10         */
11         if ( NULL_PTR == versioninfo )
12         {
13             /* polyspace +1 MISRA-C3:17.7 [Justified:Unset] "Det APIs return always E_OK: test has no added value" */
14             Det_ReportError( (uint16)CAN_MODULE_ID, (uint8)CAN_INSTANCE, (uint8)CAN_SID_GET_VERSION_INFO, (uint8)CAN_E_PARAM_POINTER);
15         }
16         else
17         {
18     #endif /* (CAN_DEV_ERROR_DETECT == STD_ON) */
19             /*
20             (CAN105) The function Can_GetVersionInfo shall return the version information of this module. The version information
21                         includes: Module Id, Vendor Id, Vendor specific version numbers.
22             */
23             /* polyspace +5 MISRA-C3:D4.14 [Justified:Unset] "Beside the implemented NULL-pointer check, no further assumptions can be done
24             versioninfo->vendorID          = (uint16)CAN_VENDOR_ID;
25             versioninfo->moduleID          = (uint16)CAN_MODULE_ID;
26             versioninfo->sw_major_version = (uint8)CAN_SW_MAJOR_VERSION;
27             versioninfo->sw_minor_version = (uint8)CAN_SW_MINOR_VERSION;
28             versioninfo->sw_patch_version = (uint8)CAN_SW_PATCH_VERSION;
29     #if (CAN_DEV_ERROR_DETECT == STD_ON)
30         }
31     #endif /* (CAN_DEV_ERROR_DETECT == STD_ON) */
32     }
33
34 #endif /* (CAN_VERSION_INFO_API == STD_ON) */
```

Figure 3 – Example of coverage results of Configurable Code depending on preprocessing directive

Three coverage scores are computed and specific to the coverage type:

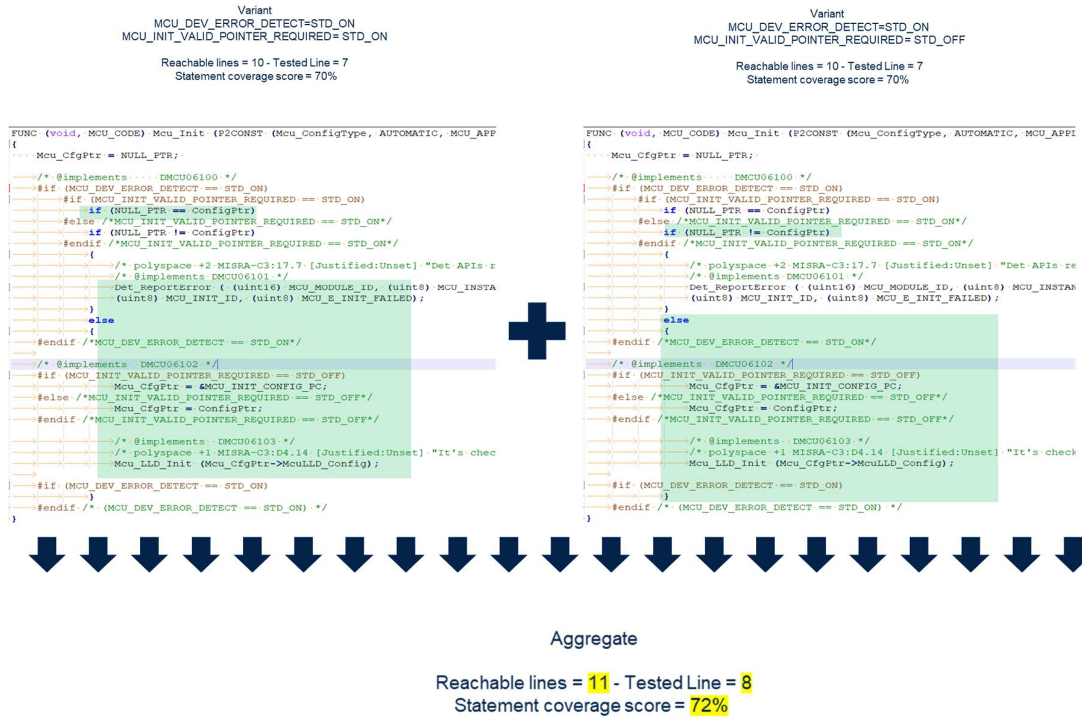
- The statement score is the ratio between tested lines of code and the total number of executable lines of code.
- The branch score is the ratio between executed branches (also called the DD-path) and the number of executable branches. In the example above, the number of the executable branches depends on the preprocessor macro: it can contain two branches if CAN_DEV_ERROR_DETECT is set to STD_ON or no branch if CAN_DEV_ERROR_DETECT is set to STD_OFF.
- The MC/DC score is the ratio between the executed condition and the executable condition considering only the ones affecting a decision's outcome independently by varying just that condition while holding all other possible conditions fixed. In the table below, the number of independent conditions is four.

**MC/DC test cases for A && B && C**

| Test Case | A | B | C | Outcome |
| --- | --- | --- | --- | --- |
| 1 | True | True | True | True |
| 2 | True | True | False | False |
| 3 | True | False | True | False |
| 5 | False | True | True | False |

Figure 4 - Modified Condition and Decision Coverage example

- Coverage Map is the graphical and analytic part of the report highlighting which lines and statements of the code have been executed and which have not. In figure 4, the executed part is highlighted in green. Developers use this part to understand how to improve the score and select additional variants.




```
FUNC (void, MCU_CODE) Mcu_Init (P2CONST (Mcu_ConfigType, AUTOMATIC, MCU_APPL_CONST) Cc
{
    Mcu_CfgPtr = NULL_PTR;

    /* @implements     DMCU06100 */
    #if (MCU_DEV_ERROR_DETECT == STD_ON)
        #if (MCU_INIT_VALID_POINTER_REQUIRED == STD_ON)
            if (NULL_PTR == ConfigPtr)
        #else /*MCU_INIT_VALID_POINTER_REQUIRED == STD_ON*/
            if (NULL_PTR != ConfigPtr)
        #endif /*MCU_INIT_VALID_POINTER_REQUIRED == STD_ON*/
            {
                /* polyspace +2 MISRA-C3:17.7 [Justified:Unset] "Det APIs return alway
                /* @implements DMCU06101 */
                Det_ReportError ( (uint16) MCU_MODULE_ID, (uint8) MCU_INSTANCE_ID, \
                (uint8) MCU_INIT_ID, (uint8) MCU_E_INIT_FAILED);
            }
            else
            {
    #endif /*MCU_DEV_ERROR_DETECT == STD_ON*/

    /* @implements  DMCU06102 */
    #if (MCU_INIT_VALID_POINTER_REQUIRED == STD_OFF)
                Mcu_CfgPtr = &MCU_INIT_CONFIG_PC;
    #else /*MCU_INIT_VALID_POINTER_REQUIRED == STD_OFF*/
                Mcu_CfgPtr = ConfigPtr;
    #endif /*MCU_INIT_VALID_POINTER_REQUIRED == STD_OFF*/

                /* @implements  DMCU06103 */
                /* polyspace +1 MISRA-C3:D4.14 [Justified:Unset] "It's checked when DE
                Mcu_LLD_Init (Mcu_CfgPtr->McuLLD_Config);

    #if (MCU_DEV_ERROR_DETECT == STD_ON)
            }
    #endif /* (MCU_DEV_ERROR_DETECT == STD_ON) */
}
```

Figure 5 – Snapshot of a Coverage Map

To support the developers in this process, the AS team developed a tool that automatically generates a report from the coverage tests to aggregate the coverage results for each software variant. From consolidated results, the tool provides to the developer:

- The aggregated Coverage Score of the test campaigns on the subset of software variants.
- The Coverage Map highlights which part of the configurable code has been executed from the variants.

Using the Coverage Score, the developer can determine if the subset of software variants is enough to achieve the desired quality level. To increase the number of variants, the developer can use the coverage map to identify the lines of code that the coverage tests have not exercised and use that as a guide for the selection of additional variant parameters.

The developer is iterating on this process until the Coverage Score reaches a level that complies with the expected score for the configurable software, as defined by the STM quality team. Using that methodology guarantees that each statement and branch of the STM Configurable code will be included in at least one version of the variant software. Moreover, it guarantees that each statement and branch is exercised by at least one test and that each statement within the source code can be analyzed using static analysis, enforcing the possibility of the verification closure of all possible variants.

The applicative source code must be identical across variants to consolidate coverage scores between the subset of variants. The variant code must contain only C constants, or C preprocessing constants must not include any statement to guarantee that the applicative code remains the same for all variants (see merge Figure 5).

## IV. VERIFICATION CLOSURE OF CONFIGURABLE CODE USING THE SOFTWARE VARIANTS SUBSET

Once the minimal subset of software variants has been established using the methodology described in the previous section, the developer uses this subset of variants to verify any changes in the STM Configurable code.

After modifying the STM Configurable Code, a developer must verify that changes in the source code are fulfilling the quality requirements defined for the project. If changes extend or change the set of possible variants for the STM Configurable Code (through a change in the metacode for example); the developer must add a subset of the new possible variants to the existing set (obtained using the methodology described in the previous section) to ensure that the verification closure is preserved.

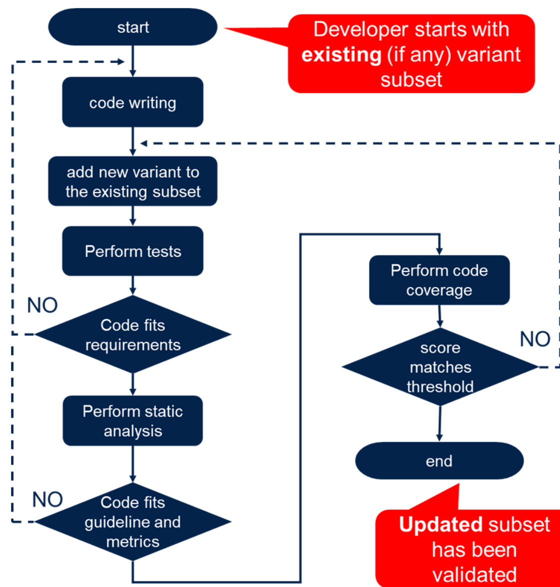



Figure 6 - Verification process - Ensure the continuity of the verification closure of the STM Configurable Code

Then the verification steps include checking that (1) the software is functionally correct and robust and (2) the source code fulfills the quality requirements related to coding standards, code metrics, and safety and security vulnerability avoidance.

The functional and robustness testing activities ensure that the software matches the functional and robustness requirements. Testing is performed on the target hardware using the existing subset of variants plus eventually the additional new variants related to the modification in the source code.

Like functional testing, static code analysis is performed on the same subset of variants impacted by the code change. This step includes the source code quality checking with two types of analysis:

- Coding guidelines (MISRA-C:2012, CERT-C, Polyspace Bug Finder defects) enforce the STM software components' code safety, security, portability, and reliability. A guideline violation implies either a code modification or adding a comment in the source code to ignore it.
- Code Metrics, such as cyclomatic complexity or comment density, provide code complexity measures. Those measures are compared to a threshold defined by the quality team. Any value above or below the threshold should be fixed, which may imply a significant code reshuffling.

As any issue detected during those steps can imply a source code change, testing and code checking are performed before the coverage testing, as this necessary code change may invalidate coverage results obtained on the previous version of the code.

As a final step, the developer will run the coverage analysis after performing testing and code-checking activities to provide results aligned with quality expectations. The developer will ensure that changes in the STA Configurable Code and the updated subset of variants still achieve the expected Coverage Score. The coverage reporting tool (described in the previous chapter) will provide consolidated test results, a list of coding guidelines deviation, and consolidated code metrics based on the worst value for each measure per variant. If the quality level expected is reached, the developer can submit the code changes and the updated subset of variants for future updates.

## V. Results and Future Enhancements

Before using the methodology described in this paper, the STM AUTOSAR software team used to verify only software variants specified by their customers once their software was released. The Team had to test the software in the context of the customer, adding unnecessary steps to the customer's adoption of the STM solution. This former method was empiric and based on a small subset of software variants. Issues were found late in the development process, sometimes several months or years after the software was released, making it complex to fix.

With the methodology described in this paper, the STM AUTOSAR team can now achieve the verification closure on the STM Configurable Code. This improved methodology optimizes the VnV of all the possible software variants possible. Leveraging the result of structural code coverage to define the minimal subset of software variants makes it possible to verify the entire software code on that subset through functional and robustness testing and static code analysis.

| Project | Number of software variants considering only boolean parameters | Subset of variants used with the described methodology | Coverage (Statement, branch, MCDC) thresholds | Achieved Coverage Score |
|---|---|---|---|---|
| MCU Driver | $2^{58} = 288*10^{15}$ | 177 | 100% | 100% |
| CAN Module | $2^{50} = 10^{15}$ | 179 | 100% | 100% |

Table 2 - Achieved results using the described methodology

In the described methodology, the process to achieve the full source code coverage is empirical and relies on the engineer's experience of the application structure. It requires running the tests incrementally to get the subset of variants that will exercise all the source code statements. With required connectivity to the hardware, it is a long process for which it is difficult to predict the effort that will be necessary. To improve this approach, a tool to analyze the source code for preprocessing directive would be needed. Such a tool should automatically provide the

smallest subset of variants to ensure the closure of the source code on which testing and checking will be performed. Christian Kästner et al. work on variability-aware analysis can be used as an approach to creating such a tool.

The AS Team is considering improving how code metrics are consolidated for verification closure. The current tool is reporting the metrics with the worst value. It is enough to ensure the customer will have a software version that fulfills code metrics requirements. The AS Team is considering updating the reporting tool to measure the Configurable Code's maintainability by providing a range of metrics values across all variants. It would help measure the Configurable Code complexity due to the dangling of the preprocessor directives since large ranges between two variants on code metrics values, such as the number of instructions or the number of paths, would highlight a function that is highly modifiable depending on software variants, hence harder to maintain.

To increase the confidence in the software robustness, the described methodology can be extended to formal verification of the source code. The static analysis should be extended to using Polyspace Code Prover from MathWorks, a tool based on formal methods that analyze the source code exhaustively to complement coding rules checking and code metrics computation. The tool verifies the correctness of C and C++ statements and identifies code that can lead to an undefined behavior at runtime under certain conditions. It would bring additional confidence in software robustness regarding safety and security-critical components.

## VI. CONCLUSION

The verification and validation of highly configurable software are challenging as the number of possible variants increases exponentially to support many configurations. Using structural coverage analysis, the STM AUTOSAR development team lowered its VnV efforts for dynamic testing and coding guidelines enforcement while ensuring that the entire configurable code has been exercised with testing and static code checking. It led to an increase in software quality and improved productivity.

The framework described in this paper can be applied to any software that relies heavily on software variants beyond the semiconductor industry, such as the automotive industry. It can be improved to rely less on developer experience by automatically computing the minimal software variants to achieve completeness of structural code coverage. Finally, other kinds of software verification and validation methods that could benefit from this framework have been identified, such as formal code verification to ensure that all source code has been analyzed for robustness.

## REFERENCES

H. Post, C. Sinz, "Configuration Lifting: Verification meets Software Configuration," 23rd IEEE/ACM International Conference on Automated Software Engineering, September 2008.

A. Knüppel, T. Thüm, I Schaefer, "GUIDO: Automated Guidance for the Configuration of Deductive Program Verifier," 2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering, May 2021.

F. Ferreira, M. Viggiato, M. Souza, E. Figueiredo, "Testing configurable software systems: the failure observation challenge," Software Product Line Conference, October 2020.

R. Zhang, S. Huang, Z. Qi, H. Guan, "Combining Static and Dynamic Analysis to Discover Software Vulnerabilities," 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, June 2021.

C. Artho, A. Biere, "Combined Static and Dynamic Analysis," Electronic Notes in Theoretical Computer Science, May 2005.

P. Jalote, V. Vangala, T. Singh, P. Jain, "Program partitioning: A framework for combining static and dynamic analysis," 2006 international Workshop On Dynamic systems Analysis, May 2006.

C. Kaestner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, T. Berger. "Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation," 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 2011.

AUTOSAR Specifications, "Specification of ECU Configuration," November 2011.

Polyspace Bug Finder, "Enforcing MISRA rules and Code Metrics," March 2022.