Static Data and Control Coupling Analysis

Daniel Kästner, Laurent Mauborgne, Stephan Wilhelm, Christoph Mallon, Christian Ferdinand

▶ **To cite this version:**

Daniel Kästner, Laurent Mauborgne, Stephan Wilhelm, Christoph Mallon, Christian Ferdinand. Static Data and Control Coupling Analysis. 11th Embedded Real Time Systems European Congress (ERTS2022), Jun 2022, Toulouse, France. hal-03694546

HAL Id: hal-03694546

https://hal.science/hal-03694546v1

Submitted on 13 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Static Data and Control Coupling Analysis

Daniel Kästner, Laurent Mauborgne, Stephan Wilhelm, Christoph Mallon, Christian Ferdinand
AbsInt Angewandte Informatik GmbH. Science Park 1, D-66123 Saarbrücken, Germany

Abstract

All current safety norms require determining the data and control flow in the source code and making sure that it is compliant to the intended control and data flow as defined in the software architecture. In traditional static code analysis, data accesses via pointer variables and control flow by function pointer calls might be missed. Using sound static analysis based on abstract interpretation, it is possible to guarantee the absence of run-time errors that could cause memory corruption and control flow corruption. Furthermore, it is possible to guarantee that in the analysis, all data and function pointer targets are considered and that the possible data and control coupling is fully captured. In this article we propose a comprehensive methodology for statically computing a safe approximation of the data and control coupling between software components. Our approach incorporates global static data and control flow analysis, taint analysis and program slicing. It can detect critical data and control flow errors and allows to complement traditional code coverage criteria by the degree of data and control coupling covered by the testing process, helping to identify relevant previously untested scenarios. It can also demonstrate freedom of spacial interference between software components at the source code level.

Keywords: data coupling, control coupling, DO-178C, static analysis, taint analysis, program slicing, abstract interpretation, interference analysis, software architecture

1 Introduction

A failure of a safety-critical system may cause high costs or even endanger human beings. With the unbroken trend towards growing software size in embedded systems more and more safety-critical functionality is implemented in software. Preventing software-induced system failures becomes an increasingly important task. Contemporary safety norms from all industry domains – including DO-178B, DO-178C, ISO-26262, IEC- 61508, the FDA Principles of Software Validation, IEC62304, and EN-50128 – require to identify potential hazards and to demonstrate that the software does not violate the relevant safety goals.

Functional safety implies demonstrating the functional correctness: the functional software requirements have to be satisfied. Demonstrating functional correctness can be addressed by requirements-based testing, in particular automatic and model-based testing, and by formal methods such as model checking or theorem proving. In addition, safety-relevant quality requirements, the so-called non-functional requirements, have to be addressed. Examples of safety-relevant non-functional software requirements are adherence to resource bounds, especially worst-case execution time bounds and stack size, as well as freedom of run-time errors. Runtime errors are typically caused by undefined or unspecified behaviors of the programming language used. In the case of the programming language C they include faulty pointer manipulations, numerical errors such as arithmetic overflows and division by zero, as well as data races, deadlocks, and further synchronization errors in concurrent software.

The data and control flow of the software is of crucial importance for the verification of functional and non-functional correctness properties. In the DO-178C, the verification goal 6.3.3.b (Consistency) demands that "a correct relationship exists between the components of the software architecture. This relationship exists via data flow and control flow." It is complemented by the verification goal of Sec. 6.3.4.b (Compliance with the software architecture) which demands to "ensure that the source code matches the data flow and control flow defined in the software architecture". Obviously the data and control flow of the implemented software must match the intended data and control flow as specified in the software architecture, and unintended data and control flow must be avoided. This also implies demonstrating freedom of interference between software components in mixed-criticality software. According to DO-178C, Sec 2.4.1, if partitioning and independence between software components cannot be demonstrated, all of them are subject to the highest criticality level assigned to any of them. Similar requirements can also be found in ISO-26262 (cf. Sec. 7.4.9 and Annex D of [8]) and other safety norms.

Furthermore, the data and control flow also determines the required effort for functional testing. In DO-178C, Objective 8 of Annex A Table A-7 requires that "Test coverage of software structure (data and control coupling) is achieved", referencing Sec. 6.4.4.2.c which states that structural coverage analysis should "confirm that the requirements-based testing has exercised the data and control coupling between code components". These terms are defined as:

Data coupling The dependence of a software component on data not exclusively under the control of that software component.

Control coupling The manner or degree by which one software component influences the execution of another software component.

The term "software component" is not precisely defined for these requirements. As stated in the CAST19 report [2],

there is currently no established understanding of the granularity of "component", it depends on the architecture being implemented. As a consequence, [2] suggests that certification projects should define what they mean by "component" in their specific architecture for demonstrating compliance to DO-178C.

The intent of structural coverage analysis is to provide a measure of the completeness of the testing process to ensure that requirements-based testing adequately exercised the software program under test [2]. Statement coverage, decision coverage and modified condition/decision coverage (Objectives 5-7 of Table A-7 of [20]) can be addressed at the module level by reviewing test cases and executing requirements-based tests of that module in isolation from other program modules. In contrast, Objective 8 of Table A-7 is primarily intended to be a verification of the integration activity: The intent of the structural coverage analysis of data coupling and control coupling is to provide a measurement and assurance that the software modules/components affect one another in the ways in which the software designer intended and do not affect one another in unintended ways, thus resulting in unplanned, anomalous or erroneous behavior. Hence, satisfying this objective is intended to provide a measure of the completeness of integration verification. As software increases in size and complexity, the quality of data and control coupling analysis is a growing concern for certification authorities [2].

In general, the data and control flow of the software can be determined by semantical static analysis. Semantics-based methods can be further grouped into unsound and sound approaches. Abstract interpretation is a formal method for *sound* semantics-based static program analysis which provides assurance that there are no false negatives with respect to the classes of defects under consideration. For data and control flow analysis the soundness of the analysis ensures that all potential targets of data and function pointers are taken into account.

This article is structured as follows: In Sec.2 we will give a brief overview of abstract interpretation and illustrate its application to runtime error analysis with the example of the analyzer Astrée. As discussed in Sec. 3 runtime error analysis can be seen as a prerequisite for data and control flow analysis, since it aims at detecting code defects that can corrupt the intended data and control flow. Sec. 4, Sec. 5 and Sec. 6 give a general overview of the core methodologies used in our work: sound global data and control flow analysis, sound taint analysis, and semantically refined program slicing. Based on those methodologies, Sec. 7 presents a novel approach for static data and control coupling analysis and interference analysis, that builds on a specification mechanism for software components appropriate for automatic static code analysis. It augments global data and control flow analysis by the software component level and presents a scalable and automatic taint analysis to determine data and control dependences between software components. Sec. 9 concludes.

2 Sound Static Source Code Analysis

The term static analysis is used to describe a variety of program analysis techniques with the common property that the results are only based on the software structure. Purely syntactical methods can be applied to check syntactical coding rules as contained in all relevant coding guidelines, including MISRA C/C++ [19, 15], or SEI CERT C/C++ [21]. A deeper understanding of the code such as knowledge about variable values, pointer targets, etc. requires semantical static analysis. It can be applied to check semantical coding rules which are also contained in the coding guidelines mentioned above, or to identify semantical code defects.

The semantics of a programming language is a formal description of the behavior of programs. The most precise semantics is the so-called concrete semantics, describing closely the actual execution of the program on all possible inputs. Yet in general, the concrete semantics is not computable. Even under the assumption that the program terminates, it is too detailed to allow for efficient computations. *Unsound* analyzers may choose to reduce complexity by not taking certain program effects or certain execution scenarios into account. A *sound* analyzer is not allowed to do this; all potential program executions must be accounted for. Since in the concrete semantics this is too complex, the solution is to introduce a formal abstract semantics that approximates the concrete semantics of the program in a well-defined way and still is efficiently computable. This abstract semantics can be chosen as the basis for a static analysis. Compared to an analysis of the concrete semantics, the analysis result may be less precise but the computation may be significantly faster.

Abstract interpretation is a formal method for sound semantics-based static program analysis [3]. It supports formal correctness proofs: it can be proved that an analysis will terminate and that it is sound, i.e., that it computes an over-approximation of the concrete semantics. Imprecisions can occur, but it can be shown that they will always occur on the safe side. Abstract interpretation-based static analyzers provide full control and data coverage and allow conclusions to be drawn that are valid for all program runs with all inputs. Nowadays, abstract interpretation-based static analyzers that can detect stack overflows and violations of timing constraints [22] and that can prove the absence of runtime errors and data races [4][12], are widely used for developing and verifying safety-critical software [9].

Runtime Error Analysis

At the source code level, the data and control flow of a program might be accidentally affected by unintended behavior, including unspecified and undefined behaviors of the programming language. Hence, a safe analysis of data and control flow must be embedded in a runtime error analysis that captures such undefined/unspecified behaviors.

In runtime error analysis, *soundness* means that the analyzer never omits to signal an error that can appear in some execution environment. If no potential error is signaled, definitely no runtime error can occur: there are no false negatives. When a *sound* analyzer does not report a division by zero in `a/b`, this is a proof that b can never be 0. If a potential error is reported, the analyzer cannot exclude that there is a concrete program execution triggering the error. If there is no such execution, this is a false alarm (false positive).

Throughout this article, we will focus on the Astrée analyzer as an example of sound static runtime error analyzer [13][18]. Astrée's main purpose is to report program defects caused by unspecified and undefined behaviors in C/C++ programs. The reported code defects include integer/floating-point division by zero, out-of-bounds array indexing, erroneous pointer manipulation and dereferencing (e.g., buffer overflows, null pointer dereferencing, dangling pointers), accesses to uninitialized variables, and further sequential programming defects. In addition, Astrée's sound thread interleaving semantics enables it to also report concurrency defects, such as data races, lock-/unlock problems, and deadlocks. Hence, Astrée not only determines the data and control flow within one thread of control, but can also capture interferences between different threads and their effects on the data and control flow within those threads.

Practical experience on avionics and automotive industry applications are given in [13][17][14]. They show that industry-sized programs of millions of lines of code can be analyzed in acceptable time with high precision for runtime errors and data races.

3 Data and Control Flow Errors

The purpose of data and control coupling analysis is to determine the effective data and control flow between software components which might be desired or undesired, depending on the properties of the software architecture. In addition, cases where there is an actual defect in the data or control flow behavior with respect to the semantics of the programming language, must be reported as an error.

In general, Astrée reports defects related to undefined or unspecified behavior of the programming language. Since their behavior is undefined or unspecified, all of them might have an effect on data or control flow – an example is a division by 0, which can cause the program to stop with a trap, obviously causing an unexpected change in control flow. In the following we will give an overview of the alarm classes of Astrée that specifically address memory safety or control flow behavior of the program.

Data Flow Errors

- *Out-of-bounds array access:* The value of the index used to access an array can be outside the feasible index range.
- *Invalid pointer dereference and manipulation:* This defect category includes dangling pointer accesses, invalid pointer dereferences and arithmetics, null pointer dereferences, misaligned dereferences, buffer overflows, etc.
- *Invalid dynamic memory allocation:* Allocation size is negative or too large.
- *Memory leak:* Memory may not be freed after dynamic allocation.
- *Uninitialized variable access:* This category includes read accesses of uninitialized local variables and read accesses to global/static variables without explicit initializer or prior assignment.
- *Data race:* Write-write or read-write data race, i.e. access to the same variable from at least two threads without proper synchronization.
- *Spectre vulnerability:* Occurrences of Spectre V1, V1.1, or SplitSpectre vulnerabilities.
- *Writes to constant memory:* Attempts to write to a constant.
- *Pointer aliasing:* Two pointer variables may alias which have been declared as distinct by the directive `__ASTREE_check_separate`.

Control Flow Errors

- *Non-returning functions:* Functions that may never return, e.g. due to infinite loops, calling `exit`, etc.
- *Incompatible function calls:* This category includes function calls with wrong number or incompatible types of parameters, incompatible return types, etc.
- *Deadlocks:* A deadlock occurs when two threads wait for each other indefinitely, e.g. due to blocked resources.
- *Recursions:* Recursive function calls are reported.
- *Infinite loops:* This category includes alarms about loops which definitely never terminate, and alarms about loops that might never terminate (e.g. only terminate upon reading a particular value from a truly volatile variable).
- *Lock/unlock problems:* This category includes attempts to unlock a mutex variable that has not been locked, locks without unlocks, locks acquired by a wrong task, etc.
- *C++ Exception:* The alarm reports C++ statements that can raise a C++ exception.
- *Pure virtual function call:* A pure virtual function is called in a specific context (C++).

It is apparent that these defects may invalidate any assumptions about the data and control flow behavior of the program, and hence, must also be considered for data and control coupling analysis. The same may also hold for other cases of undefined/unspecific behavior which are reported by Astrée. Hence sound static runtime error analysis can be seen as prerequisite for data and control coupling analysis. As emphasized in Sec. 2 the defect classes are not limited to sequential program execution but also include program defects induced by concurrent thread execution.

4 Data and Control Flow Analysis

Classical global data and control flow analysis determines the variable accesses and function invocations throughout program execution. It is centered on concepts included in the programming language, as compared to data and control *coupling* analysis that focuses on software components which are not expressed by programming language constructs. It constitutes the required basis for data and control coupling analysis.

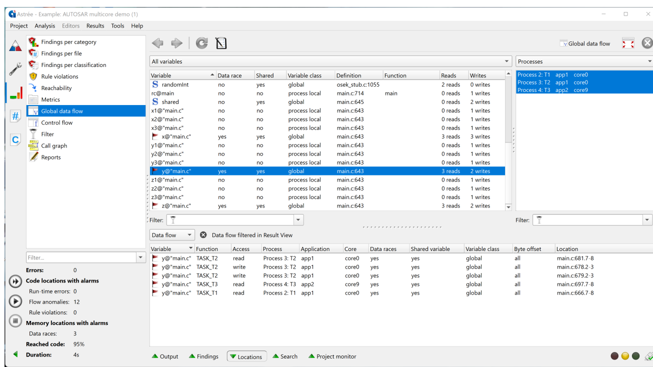

Figure 1: Astrée's Data Flow View.

In its basic data and control flow analysis module, Astrée tracks accesses to global variables, static variables, and local variables whose accesses are made outside of the frame in which the local variables are defined (e.g., because their address is passed into a called function). All data and function pointers are automatically resolved. The soundness of the analysis ensures that all potential targets of data and function pointers are taken into account. Astrée's data and control flow reports show the number of read/write accesses for every global, static, and out-of-frame local variable, lists the location of each access and shows the function from which the access is made. All variables are classified as being thread-local, effectively shared between different threads, or subject to a data race (cf. Fig. 1). Variable accesses can be interactively explored, e.g. , by selecting a variable and filtering for accesses from a particular thread or function, or for a given access type.

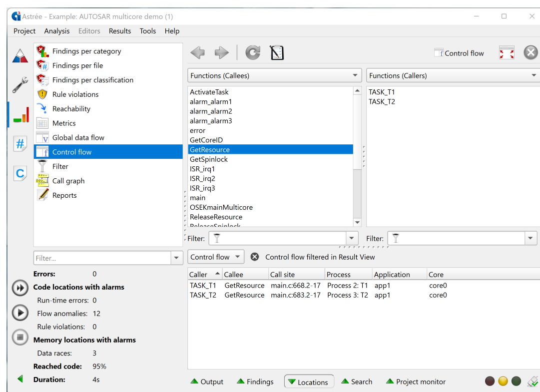

Figure 2: Astrée's Control Flow View.

The control flow is described by listing all callers and callees for every C function along with the threads in which they are called. In AUTOSAR projects, additionally the application and the core to which the executing thread belongs is listed. An example is shown in Fig. 2. The control flow can be interactively explored, e.g. , when selecting a function and filtering for its callers or callees or for threads in which it is called, all relevant call sites are displayed. There is also a call graph visualization which can be interactively explored as well (cf. Fig. 3).

In case of C++ programs, a class graph visualization shows a selected class with all its fields and methods as well as its sub- and superclasses with the relevant template instantiations in a

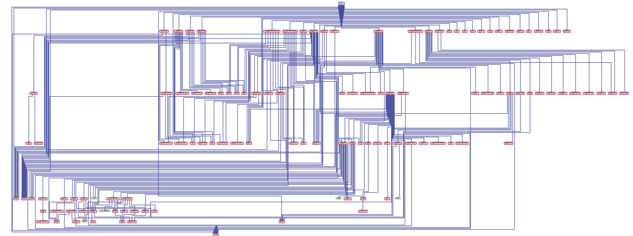

Figure 3: Astrée's Call Graph Visualization.

graph depicting the inheritance hierarchy (cf. Fig. 4).

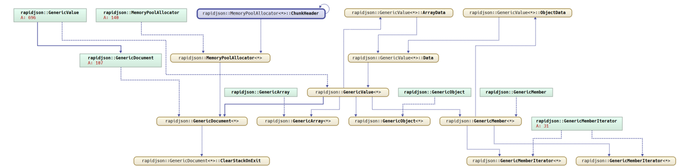

Figure 4: Astrée's Class Graph Visualization.

More sophisticated information about selected flows of values can be provided by two dedicated analysis methods: taint analysis and program slicing. *Taint analysis* is a forward analysis and can answer questions about program parts affected by reading corrupted input values. *Program slicing* is a backward analysis which can answer questions about the program parts which might influence the value of a particular variable at a particular program point.

All data and control properties described in this section are agnostic of the definition of software components; they are limited to native concepts of the programming language. Lifting this information to the level of software components is the topic of Sec. 7 which leverages the techniques of taint analysis and program slicing described in Sec. 5 and Sec. 6.

5 Taint Analysis

Taint analysis was first introduced as a dynamic analysis technique (e.g., in PERL), to try to find out which part of a code could be affected by some inputs. The original technique consisted in flipping normally unused bits, that would be copied around by operations and assignments. The same idea can be extended to static analysis by enhancing the concrete semantics of programs with tainting, the formal equivalent of the unused flipped bit in the dynamic approach. In the context of abstract interpretation, it is easy to abstract this extra information in an efficient and sound way, using dedicated abstract domains. Conceptually, taint analysis consists in discovering data dependencies using the notion of taint propagation. Taint propagation can be formalized using a non-standard semantics of programs, where an imaginary taint is associated to some input values. Considering a standard semantics using a successor relation between program states, and considering that a program state is a map from memory locations (variables, program counter, etc.) to values in $\mathscr{V}$, the *tainted* semantics relates

tainted states, which are maps from the same memory locations to $\mathscr{V} \times \{\text{taint}, \text{notaint}\}$, and such that if we project on $\mathscr{V}$ we get the same relation as with the standard semantics.

To define what happens to the *taint* part of the tainted value, one must define a *taint policy*. The taint policy specifies:

- **Taint sources** which are a subset of input values or variables such that in any state, the values associated with that input values or variables are always tainted.
- **Taint propagation** describes how the tainting gets propagated. Typical propagation is through assignment, but more complex propagation can take more control flow into account, and may not propagate the taint through all arithmetic or pointer operations.
- **Taint cleaning** is an alternative to taint propagation, describing all the operations that do not propagate the taint. In this case, all assignments not containing the taint cleaning will propagate the taint.
- **Taint sinks** is an optional set of memory locations. This has no semantical effect, except to specify conditions when an alarm should be emitted when verifying a program (an alarm must be emitted if a taint sink may become tainted for a given execution of the program).

A sound taint analyzer will compute an over-approximation of the memory locations that may be mapped to a tainted value during program execution. The soundness requirement ensures that no taint sink warning will be missed by the analyzer.

Astrée provides a generic abstract domain for taint analysis that can be freely instantiated by the users. It augments Astrée's process-interleaving interprocedural code analysis by carrying and computing taint information at the byte level. Any number of taint hues can be tracked by Astrée, and their combinations will be soundly abstracted. Tainted input is specified through directives (`__ASTREE_taint((var;hues))`) attached to program locations. Such directives can precisely describe which variables, and which part of those variables is to be tainted, with the given taint hues, each time this program location is reached. Any assignment is interpreted as propagating the join of all taint hues from its right-hand side to the targets of its left-hand side. In addition, specific directives may be introduced to explicitly modify the taint hues of some variable parts. This is particularly useful to model cleansing function effects or to emulate changes of security levels in the code.

The result of the analysis with tainting can be explored in the Astrée GUI, or explicitly dumped using dedicated directives. Finally, the taint sink directives may be used to declare that some parts of some variables must be considered as taint sinks for a given set of taint hues. When a tainted value is assigned to a taint sink, then Astrée will emit a dedicated alarm, and remove the sinked hues, so that only the first occurrence has to be examined to fix potential issues with the security data flow.

The main intended use of taint analysis in Astrée is to expose potential vulnerabilities with respect to security policies or resilience mechanisms. Thanks to the intrinsic soundness of the approach, no tainting can be forgotten, and that without any bound on the number of iterations of loops, size of data or length of the call stack. Based on its taint analysis, Astrée provides an automatic detection of Spectre-PHT vulnerabilities [10].

6 Program Slicing

The following definitions introduce the basic principles of static program slicing.

Definition 1 (Slicing Criterion) *Let P be a program. A slicing criterion in P is a tuple (s,V) which consists of a statement s and a set of variables V from P*

Definition 2 (Slice) *A slice S is a subprogram of P that exhibits the same behavior with respect to the slicing criterion (s,V).*

Computing a *statement-minimal* slice is an undecidable problem. However there are well-established algorithms for computing non-minimal, but still useful slices. A common approach is to compute a *System Dependence Graph* (SDG), which contains all data and control dependences of the program. Then a slice can be expressed as a reachability problem in this graph [7]. The precision of the slice directly depends on the precision of the SDG. However, computing precise system dependency graphs is a non-trivial task since it requires deriving intricate program properties. These may include points-to information for variable and function pointers, code reachability, context information or possible variable values at certain program points. As an example, over-approximating the set of possible destinations of a pointer variable blows up the size of the system dependence graph as it may add false dependences to statements which contain variables that would otherwise not be included in the slice. This may cause a drastic transitive increase in the number of dependences and vertices.

Astrée provides a novel concept of program slicing, that can be termed *sound semantically refined slicing*: its slicer can be run in the sequel of a finished Astrée analysis run, which makes it possible to leverage the invariants computed by its main fixpoint analysis. The system dependence graph computed by our approach is a sound abstraction of the data- and control dependences of a computer program. This follows from the soundness of the Astrée core analysis. As a consequence, the resulting slices are also sound. Minimizing false alarms is an important design goal of Astrée, which mandates a highly precise point-to analysis. Furthermore, Astrée detects code which is guaranteed to be unreachable for any possible program execution, and which, consequently, can be ignored when computing the slices. Hence, compared to slicing without leveraging Astrée invariants, a significant precision and efficiency gain is achieved by reducing the amount of vertices and the amount of data- and control dependences in the system dependence graph. This efficiency improvement makes it possible to compute precise slices for very large programs in feasible time.

Semantically refined slicing can be run in context-insensitive mode (considering all possible call contexts) and context-sensitive mode (considering exactly one call context). To compute context-sensitive slices we enhance the slicing algorithm of [7] with a description of call contexts (stacks). In each step of the reachability analysis we additionally check that the de-

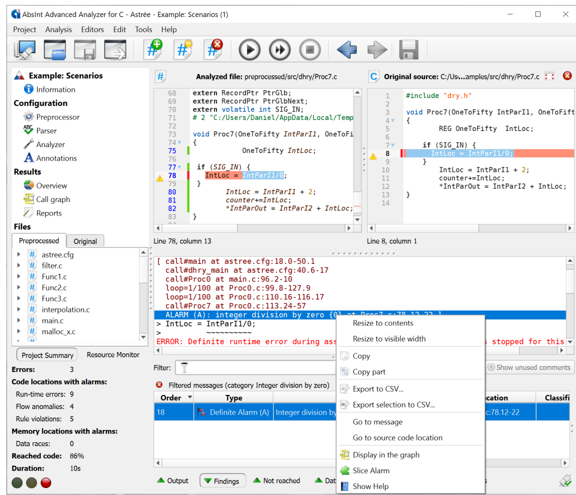

Figure 5: Alarm Slicing in Astrée

pendences under examination match the relevant stacks. Dependences which do not match are discarded. When following a dependency edge which represents a function call, the topmost function is removed from the stack. For one given program point a context-insensitive slice is identical with the union over all context-sensitive slices.

In contrast to context-insensitive slices, context-sensitive slices do not capture all possible behaviors of the original program which influence the slicing criterion. Instead, the behavior described by the slice is restricted to execution paths which are in accordance with the set of considered call contexts. Context-sensitive slices tend to be significantly smaller than context-insensitive ones.

The different slicing modes presented in this section are relevant for demonstrating safety and security properties. Sound slices can be computed by *context-insensitive* analysis-enhanced slicing. With these slices it is possible to show that certain parts of the code or certain input variables might influence or cannot influence a program section of interest. They yield a global overview of these properties for the entire program.

In contrast to that, *context-sensitive* analysis-enhanced slicing, which only considers a subset of possible contexts, is more suitable for investigating the influence of a certain code section, e.g. a function, or a module, on the program location of the slicing criterion. Hence it is perfectly suited as a basis for automatic alarm slicing, which is available in Astrée. To give an example, the critical situation for a division by zero alarm, which is reported for a given context, is precisely the context where the denominator becomes 0. Therefore the alarm slice is a partial slice for the variables used in the denominator, which only considers program paths leading to this particular context.

This concept of alarm slicing has been implemented in Astrée and is available from the GUI: from the context menu of an alarm in the Astrée graphical user interface the computation of a slice for the alarm can be automatically triggered, as shown in Fig. 5.

A detailed experimental survey of Astrée's semantically refined program slicer with programs from automotive and avionic industry is given in [11]. It demonstrates that semantically refined slicing (termed analysis-enhanced slicing in [11]) can be applied to industry-size code with high precision and with feasible memory and computation time requirements.

7 Data and Control Coupling

The CAST19 report [2] reviews the data and control coupling requirements of the DO-178C and provides clarifications about the intended use. It emphasizes that the purpose of data coupling analysis includes (among others) identifying data dependences, verifying the data interfaces between modules/components through testing and analyses, identifying inappropriate data dependencies, evaluating the need for and accurate use of global data, and evaluating input/output data buffers.

The purpose of control coupling is stated to include identifying control dependencies, identifying inappropriate control dependencies, verifying correct execution call sequence, defining and evaluating the extend of interface depth, and assisting in WCET analysis.

In addition, as outlined above, data coupling and control coupling together aim at providing a completion check of the integration testing effort. The CAST19 report further clarifies that the "Data Coupling and Control Coupling Analyses" objective of the DO-178C (Objective 8 of Annex A, Table A-7 [20]) may be satisfied as a static activity, a dynamic activity, or a combination.

In the following we propose a concept for sound static data and control coupling analysis that builds on Astrée's data and control flow analysis as described in Sec. 4, and which satisfies all requirements mentioned above. First, in Sec. 7.1, we present a flexible and extensible concept for specifying software components and critical data and control flow interactions between them. Sec. 7.2 outlines static data and control flow analysis augmented by the concept of components. Sec. 7.3 presents an automatic taint analysis that efficiently tracks the flow of values between components, and automatically reports undesired data flow and undesired control dependencies. In case of taint sink alarms indicating undesired data or control flow, alarm slicing can be used to track down the cause of the undesired behavior.

7.1 Specifying Software Components

Since there is no established understanding of the granularity of "component", a specification mechanism is needed that allows users to specify their concept of software components and the "interesting" data or control flow between them. For the purposes of data and control coupling, the full power of special-purpose architecture description languages (ADL) is not needed (cf. AADL [6], ArchiMate [1], SysML [23], or UML-based architecture specifications), since the aim is to ascertain the desired properties by automatic static analysis from the source code.

A simple starting point is to define a software component as a collection of all variables and functions defined in a set of source files. This can be provided by (conceptual) annotations of source files or functions:

```
"xfile1.c" insert :
__ASTREE_attributes((component("trusted")));
```

```
"yfile1.c" insert :
__ASTREE_attributes((component("non-trusted")));
```

This schema can be easily extended to more complex component definitions, e.g., based on individual functions, all files in a sub-directory, etc.

In addition to defining the elements of a software component, the specification also allows to declare component interactions that should be "observed" during the analysis, i.e., specifically tracked and reported. To this end, the analyzer can be instructed to report control and data flow from the component "trusted" to the component "non-trusted", and vice versa:

```
<observe>
  <item key="trusted">"non-trusted"</item>
  <item key="non-trusted">"trusted"</item>
</observe>
```

A dedicated view in the graphical user interface of Astrée allows to conveniently create the software component specification, and then export them to an XML file. Alternatively, the XML file may also be generated automatically from an existing ADL specification.

The specification of component interactions to be observed mostly aims at explicitly reporting unexpected or forbidden interactions. By placing components with expected interactions under observation, Astrée can also address intended interactions, however, in that case, the result of Astrée only contributes potential interactions due to its sound over-approximation: a reported component interaction can also be a "false alarm". A dedicated tag to support simultaneous tracking of unexpected and intended interactions is currently not available but could be easily added.

7.2 Augmented Data and Control Flow Analysis

The first step towards detecting and reporting data and control flow relations between software components is to enhance the standard data and control flow analysis by taking the component definitions into account, which we term *augmented* data and control flow analysis.

The augmented data and control flow analysis of Astrée will thus report any interaction between two software components of the following classes:

- Calling a function of a given component, including indirect calls through pointer dereferences.
- Writing to a variable defined in a given component, which can be a global variable that belongs to the component or a local variable defined in a function of the component. That includes indirect writes through pointer dereferences.
- Reading from a variable defined in a given component. That includes indirect reads through pointer dereferences.

These interactions already capture most interferences between components, including indirect ones, such as scenarios when the address of a variable of component A is stored in component C, and then read from C by component B which writes to it through dereference.

They will not capture more subtle value dependencies, such as component C copying the value of a global variable of component A, store it in one of its variables and then give the value to component B that may take different control flow based on that value. To track such interferences, we need to follow the flow of values, and we can do that using taint analysis techniques – cf. Sec. 7.3.

The data and control flow views in the Astrée GUI presented in Fig. 1 and Fig. 2 of Sec. 4, and the corresponding data and control flow reports will be augmented by additional columns which make the component assignment explicit. Each access to a variable X then is reported with the following information in the data flow view:

- variable name
- component to which the variable belongs
- location of access (may be pointer dereference)
- access type (read/write)
- function containing the access
- component to which the accessing function belongs
- task in which the accessing function is executed
- application of task
- core to which application has been assigned
- access locality (thread-local, effectively shared, data race)
- notification for components under observation

Each function call is reported with the following information in the control flow view:

- caller
- component of caller
- callee
- component of callee
- call site (may be function pointer call)
- task in which the call is executed
- application of task
- core assigned to application
- notification for components under observation

The data and control flow reports can be generated in various open formats that support post-processing, so it is easy to query for any component interaction of interest. If a flow from a component X to a component Y has been introduced with an `observe` tag in the component specification, a dedicated notification is generated about any observed flow from X to Y.

Also the visualizations of Fig. 3 and Fig. 4 will be augmented by component information so that it will be possible, e.g., to highlight the nodes for selected components, or show different components in different colors.

This mechanism gives a sound overapproximation of all variable accesses and function calls, and establishes their link to the software component definition. It makes it easy to spot flows under observation, hence call attention to flows that should be investigated.

7.3 Data and Control Coupling Analysis

As outlined in Sec. 5, taint analysis allows to track the flow of values in a project. One solution to compute data dependencies between components is to assign a distinct taint hue to each component, and use all global and local variables of a component as taint source with the component hue. To be automatically notified about all components where these values flow, all reads in each component are declared as taint sinks for all

other components hues. The required __ASTREE_taint and __ASTREE_taint_sink directives can be generated automatically. To focus on particular component interactions, the automatic generation of taint source and taint sink directives can be restricted to only those component flows placed under observation, which reduces the number of taint sink alarms about component dependences. Of course, it is also possible to manually select specific component flows to observe, or perform the taint analysis on a per-variable base for individual variables deemed critical.

This approach not only detects additional interferences compared to the augmented data and control flow analysis, it also allows to account for authorized interactions in a fine-grained way. One example is that data flow from component Y to X is forbidden, unless the access is made by specific gateway functions. Such interactions can be modeled by taint-cleaning operations which remove the taint in those gateway functions (sanitization points), reducing the number of legitimate interferences that need to be examined. To further facilitate this examination, automatic alarm slicing on taint sink alarms can be used to help identify the program regions responsible for undesired component interactions.

In addition to the data flows reported by the automatic tainting as described above, taint analysis also allows to focus on control coupling. The generation of taint sink directives can be adapted, so that only values used in guards (for conditional statements, while loops or switch statement) and in function pointer dereferences are considered as taint sinks.

Hence, the taint analysis of software components can be performed in data coupling and control coupling mode, satisfying the requirements of the CAST19 report as described above.

It should be noted that taint analysis is a complement to the augmented data and control flow analysis (cf. Sec. 7.2), but cannot replace it, since it focuses on the data flow aspect and does not report invocations of functions from other components. Its data flow results are more powerful: taint analysis can keep track of call-by-value parameters of functions and of function return values and hence report dependences with respect to constants or local variables. It not only shows that at a certain location a given variable is accessed, but also provides the corresponding call context – as an example, if a function is called with a pointer argument, and only in one call site an address of a variable from another component is passed, then the alarm context will show precisely this call site. Furthermore, it makes it possible to narrow the focus on selected component interactions and tracks the relevant data flows also through code that does not belong to one of the components under observation. Both of them together, i.e. , the combination of augmented data and control analysis and the taint analysis for software components provide a sound interference analysis. They report all interactions between software components executed in one or multiple concurrent threads, pinpoint interactions under observation, hence enable users to find undesired interactions, and browse the code locations where they happen. They also provide a basis for checking intended interactions and allow computing metrics about the data and control flow between the software components.

As an example, consider the following code sequence:

```
/* file main.c */
void main(void) {
  int x;
  x = F_a(0);
  g_b = x;
  F_c(g_b);
}

/* file A.c */
__ASTREE_attributes((component("A")));
int g_a;
int F_a(int x) {
  int y=x;
  return x + 2;
}

/* file B.c */
__ASTREE_attributes((component("B")));
int g_b;

/* file C.c */
__ASTREE_attributes((component("C")));
int g_c;
int F_c(int x) {
  g_c=g_a;
  if (g_c < 0) g_c = F_a(3);
}
```

The augmented data and control flow analysis provides the information that component C depends on A through variable read and function call, since F_c reads variable g_a and calls function F_a. Component tainting raises four taint sink alarms: first it reports that component C depends on A at the assignment g_c=g_a in F_C, which is information also available in the augmented data flow view. It also reports a dependence from A to C at g_c = F_a(3); in F_C because the return value of F_A is assigned to g_c. This is more precise information than provided by the augmented control flow view which just reports a call to a function from C at this location. Third, the taint analysis reports a dependence from C to A at the assignment y=x in F_a, since C passes a constant value to F_a, which then is assigned to a local variable of A. Finally, the component tainting reveals that component B depends on component A through variable x: a taint sink alarm is raised at the assignment g_b = x in main.c, which is outside of components A, B, and C. Note that in the example, all variables are directly accessed, however, the analysis results would not change if all variable accesses and function calls were made via pointers.

8 Experimental Results

The augmented data and control flow analysis is part of the sound runtime error analysis, hence, it is not associated with additional runtime or memory overhead.

To assess the performance of the taint-based components analysis we investigated four industry projects of various sizes, two from the avionics domain and two from the automotive domain. We partitioned the code in software components and determined the increase in analysis time and memory consumption caused by component tainting. Tab. 1 shows the character-

istics of the projects, the results obtained with component tainting are summarized in Tab. 2. Column S of Tab. 1 gives the size of the projects in million physical lines of code[1] after preprocessing, column N_C indicates the number of software components. The large number of software components in project AE2 results from declaring every application source file as an own component; in the other projects the components consist of larger code parts. Column T and M show the analysis time and memory consumption in their original configuration.

Project	S [$MLOC$]	N_C	T	M [GB]
AE1	0.14	11	33m	2.43
AE2	1.08	4309	8h 27m	15.53
AU1	5.46	11	7h 1m	39.81
AU2	5.03	35	10h 44m	31.52

Table 1: Project Characteristics

Column T_t and M_t of Tab. 2 show the analysis time resp. memory consumption with component tainting. The increase in analysis time and memory consumption associated with component tainting is given in column Δ_T resp. Δ_M. The results show that the overhead of component tainting is low. The increase in memory consumption is below 1% for all test cases except AE2. For AE2 an extremely high number of components has been defined, which also entails a large number of component interactions: there are 78283 alarms about variable accesses creating component dependencies (cf. Tab. 3). However, even in that scenario the memory overhead is only 4.76%.

The increase in analysis time is between 4.93% and 8.07% for the larger projects. The largest increase of 8.07% occurs in project AU2, where the analysis time increases by 45 min from a total analysis time of 10 hours 44 minutes. On the smallest project, AE1, there is no measurable difference in memory consumption.

Project	T_t	M_t [GB]	ΔT	ΔM
AE1	33m	2.43	0%	0%
AE2	8h 52m	16.27	4.93%	4.76%
AU1	7h 24m	40	5.5%	0.48%
AU2	11h 36m	31.78	8.07%	0.82%

Table 2: Results with Component Tainting

Taint sink alarms for component dependences can be limited to flows under observation. However, in the experiments, we configured Astrée to generate taint sink alarms for every cross-component flow, since the goal was to assess performance on large projects with many component interactions. The number of cross-component variable accesses derived from the augmented data flow view is listed in column N_V of Tab. 3, column N_C gives the number of cross-component function invocations, and the number of taint sink alarms denoting cross-component data flows is listed in column N_{TA}. All numbers indicate the number of *code locations* which exhibit a cross-component interaction, e.g., all accesses to a variable g_A of component A from component B are separately counted.

[1] I.e., comment lines and empty lines are not counted.

The difference between columns N_V and N_{TA}, on the one hand, is due to additional dependences discovered by tainting, e.g., due to values passed through library functions not assigned to a specific components, call-by-value function parameters and function return values. On the other hand, as discussed in Sec. 7.3, component tainting focuses on data dependences and tracks the effect of all function calls, but does not separately report the calls themselves. The effect of infrastructure code without component assignment is particularly visible in project AE2. The components do not explicitly invoke one another, the invocations are made in an infrastructure layer we did not assign to an own component. Dependencies carried through that infrastructure layer, e.g., by copying component variables via infrastructure variables, are visible with component tainting, but not in the augmented data flow view.

Project	N_V	N_C	N_{TA}
AE1	147	289	415
AE2	40929	0	78283
AU1	9569	1695	10285
AU2	8485	4242	16852

Table 3: Code locations with cross-component interactions

9 Conclusion

At the source code level, the data and control flow of a program might be affected by behavior unintended by the programmer, including unspecified and undefined behaviors of the programming language. Hence, a safe analysis of data and control flow must be embedded in a runtime error analysis that captures such undefined/unspecified behaviors. Determining the data and control flow in the source code and making sure that it is compliant to the intended control and data flow as defined in the software architecture is a common requirement in all contemporary safety norms. The aim of data coupling and control coupling analysis objective of DO-178C is to provide a measure of the completeness of integration verification by ensuring that the software components affect one another only in intended ways. Furthermore all safety norms require demonstrating the freedom of interference between software components of different criticality levels.

In this article we have presented a novel approach for data and control coupling analysis and interference analysis, that builds on a specification mechanism for software components appropriate for automatic static code analysis. It is based on the sound static analyzer Astrée that allows to prove the absence of critical runtime errors that could cause memory corruption and control flow corruption. Our approach augments sound global data and control flow analysis by the software component level, proposes a scalable and automatic taint analysis to determine data and control dependences between software components, and incorporates semantically refined program slicing to help identify the program regions responsible for undesired component interactions. It enables a sound approximation of data and control coupling and a sound interference analysis that help demonstrating the correctness of the data and control flow of the program and contribute to satisfying the data and control

coupling and freedom of interference verification goals of contemporary safety norms. The experimental results show that our approach is highly efficient and can be applied to industry-size software projects.

References

[1] The ArchiMate Enterprise Architecture Modeling Language. https://www.opengroup.org/archimate-forum/archimate-overview[retrieved: Jan. 2021].

[2] Certification Authorities Software Team (CAST). Position Paper CAST-19. Clarification of Structural Coverage Analyses of Data Coupling and Control Coupling, 2004.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.

[4] D. Delmas and J. Souyris. ASTRÉE: from Research to Industry. In *Proc. 14th International Static Analysis Symposium (SAS2007)*, number 4634 in LNCS, pages 437–451, 2007.

[5] C. Faure and V. Delebarre. Automatic proof of freedom from interference with iffree. In *Proccedings of the 10th European Conference on Software Architecture Workshops, Copenhagen, Denmark, November 28 - December 2, 2016*, page 36, 2016.

[6] P. Feiler, D. Gluch, and J. Hudak. Technical Note CMU/SEI-2006-TN-011. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Software Engineering Institute, Carnegie Mellon University, 02 2006.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[8] ISO 26262. Road vehicles – Functional safety, 2018.

[9] D. Kästner. Applying Abstract Interpretation to Demonstrate Functional Safety. In J.-L. Boulanger, editor, *Formal Methods Applied to Industrial Complex Systems*. ISTE/Wiley, London, UK, 2014.

[10] D. Kästner, L. Mauborgne, C. Ferdinand, and H. Theiling. Detecting Spectre Vulnerabilities by Sound Static Analysis. In R. F. Anne Coull, Steve Chan, editor, *The Fourth International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2019)*, volume 4 of *IARIA Conferences*, pages 29–37. IARIA XPS Press, 2019. Archived in the free access ThinkMind^{TM} Digital Library, http://www.thinkmind.org/download.php?articleid=cyber_2019_3_10_80050.

[11] D. Kästner, L. Mauborgne, N. Grafe, and C. Ferdinand. Advanced Sound Static Analysis to Detect Safety- and Security-Relevant Programming Defects. In J.-C. B. Rainer Falk, Steve Chan, editor, *8th International Journal on Advances in Security*, volume 1 & 2, pages 149–159. IARIA, 2018.

[12] D. Kästner, A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, A. Schmidt, H. Hille, S. Wilhelm, and C. Ferdinand. Finding All Potential Runtime Errors and Data Races in Automotive Software. In *SAE World Congress 2017*. SAE International, 2017.

[13] D. Kästner et al. Finding All Potential Runtime Errors and Data Races in Automotive Software. In *SAE World Congress 2017*. SAE International, 2017.

[14] D. Kästner et al. Analyze This! Sound Static Analysis for Integration Verification of Large-Scale Automotive Software. In *Proceedings of the SAE World Congress 2019 (SAE Technical Paper)*. SAE International, 2019.

[15] M. Limited. MISRA C++:2008 Guidelines for the use of the C++ language in critical systems, June 2008.

[16] A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):63, Mar. 2012.

[17] A. Miné and D. Delmas. Towards an Industrial Use of Sound Static Analysis for the Verification of Concurrent Embedded Avionics Software. In *Proc. of the 15th International Conference on Embedded Software (EMSOFT'15)*, pages 65–74. IEEE CS Press, Oct. 2015.

[18] A. Miné et al. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, Jan. 2016.

[19] MISRA (Motor Industry Software Reliability Association) Working Group. *MISRA-C:2012 Guidelines for the use of the C language in critical systems*. MISRA Limited, Mar. 2013.

[20] Radio Technical Commission for Aeronautics. RTCA DO-178C. Software Considerations in Airborne Systems and Equipment Certification, 2011.

[21] Software Engineering Institute SEI – CERT Division. *SEI CERT C Coding Standard – Rules for Developing Safe, Reliable, and Secure Systems*. Carnegie Mellon University, 2016.

[22] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[23] OMG Systems Modeling Language (OMG SysML™) Version 1.6. https://www.omg.org/spec/SysML/1.6/PDF[retrieved: Jan. 2021].

[24] B. Zimmer, C. Dropmann, and J. U. Hanger. A systematic approach for software interference analysis. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, pages 78–87, 2014.