Verifying Fortran Programs with CIVL

Wenhao Wu[1]✉, Jan Hückelheim[2], Paul D. Hovland[2], and Stephen F. Siegel[1]

[1] University of Delaware, Newark DE 19716, USA
{wuwenhao, siegel}@udel.edu
[2] Argonne National Laboratory, Lemont IL 60439, USA
{jhueckelheim, hovland}@anl.gov

Abstract. Fortran is widely used in computational science, engineering, and high performance computing. This paper presents an extension to the CIVL verification framework to check correctness properties of Fortran programs. Unlike previous work that translates Fortran to C, LLVM IR, or other intermediate formats before verification, our work allows CIVL to directly consume Fortran source files. We extended the parsing, translation, and analysis phases to support Fortran-specific features such as array slicing and reshaping, and to find program violations that are specific to Fortran, such as argument aliasing rule violations, invalid use of variable and function attributes, or defects due to Fortran's unspecified expression evaluation order. We demonstrate the usefulness of our tool on a verification benchmark suite and kernels extracted from a real world application.

Keywords: Fortran · verification · static analysis · model checking

1 Introduction

Fortran is a structured imperative programming language with a unique set of features, such as common data blocks and array reshaping and sectioning, that support efficient numerical computing. Many scientific applications, especially those requiring high performance, are written entirely in Fortran; others have core subroutines or rely on external components written in Fortran. A 2018 report from the European Performance Optimisation and Productivity Centre states that over half of the 151 HPC programs the centre had analyzed over a two-year period were written in pure Fortran or a combination of Fortran and C or C++ [10]. Likewise, 12 of the 33 HPC benchmark applications in the U.S. Department of Energy's widely-used CORAL suite have components written in Fortran [17].

The Fortran language has been used and revised for decades, and it has had many standard versions. Early versions of Fortran employed a fixed-form coding style, well-suited for punch cards and with strict positional constraints. Beginning with Fortran 90, a free-form style was introduced, enabling more structured programs, eliminating limits on line lengths, and providing more flexibility with

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 106–124, 2022.
https://doi.org/10.1007/978-3-030-99524-9_6

character positioning by removing the restriction that the first six columns could be used only for labels and continuation characters. Modern Fortran programs tend to use the free-form style, but programs derived from a Fortran 77 predecessor or relying on legacy components may rely on fixed-form style or a mix of both styles.

Fortran is used to implement applications such as Nek5000 [21] or Flash [32] that are used for critical tasks such as nuclear reactor licensing reviews or to answer important scientific questions. These applications are often computationally demanding, requiring hours of computation on millions of execution units. Because of the critical importance and high resource requirements of these applications, one would like to verify their correctness.

The Fortran language itself provides little support for verification—not even assertions. Compilers can check certain simple syntactic and semantic properties, and static analyzers such as Coverity [35] can detect standard violations and other anomalies. But there are very few tools that can be used to specify and verify deeper functional correctness properties of programs, and nothing like the rich ecosystem of formal verification tools for C.

One might approach Fortran program verification by using a source-to-source translator such as f2c [11] to convert to C and then applying a C verifier. Unfortunately, even if the translator provides a completely valid translation, defects in the original code may not be preserved in the translated code; an example is given in Section 3.1. In addition, the C verifier may not be able to access translator support libraries, or defects that manifest themselves via the library may be difficult to map back to the original program.

A second approach is to use a compiler front end to convert Fortran code into an intermediate form such as the LLVM [16] Intermediate Representation (IR), and then apply a verifier for the IR. This is more difficult than it appears: most verifiers that consume LLVM IR are tuned to a specific source language and front end and cannot be easily modified to effectively verify multiple languages. This issue is explored in [13] in the case of SMACK, a C-via-LLVM verifier that has been extended to provide limited support for other languages, including Fortran. Moreover, as with source-to-source translators, the front end may translate away a defect in the original program; this is discussed in Section 3.2.

In this paper, we present an approach to extending the CIVL [33] verification framework so that it can be directly applied to Fortran source code. CIVL is a model checker that uses symbolic execution to verify correctness properties and was originally designed for programs written in C with a set of parallel programming language extensions such as OpenMP [26]. In our extended framework, summarized in Section 2, a new Fortran front end with a static analyzer has been integrated into the system. In Section 3 we describe the sequence of defect-preserving transformations that convert the Fortran source to the CIVL intermediate verification language, CIVL-C. Proper handling of arrays is a special concern, discussed in Section 4. The Fortran extension supports a subset of the major features defined in the language standard, focused on those features necessary to verify code excerpts from real world applications.

In Section 5, we evaluate our approach by verifying several examples of Fortran code, including (1) a custom Fortran benchmark suite designed to test CIVL's ability to verify programs using unique Fortran features such as array slicing and reshaping, (2) a published micro verification benchmark [13], and (3) a set of code excerpts from Nek5000 [21]. The evaluation employs both of CIVL's verification modes on Fortran programs. The first uses assumptions and assertions inserted in the program to specify the desired correctness properties. The second compares two programs with the same input-ouput signatures to determine whether they are functionally equivalent.

Related work is discussed in Section 6, and conclusions and future work are summarized in Section 7.

2 Overview of CIVL Extension

The Concurrency Intermediate Verification Language (CIVL) platform was developed to verify C programs that use various concurrency language extensions [33]. CIVL has two primary components: a front end and a back end verifier. The front end consumes a set of source files, which, prior to this work, had to be written in C or CUDA-C, possibly using certain CIVL extensions to C. These source files may use one or more concurrency language extensions, including MPI [18], OpenMP [26], Pthreads [25], and CUDA-C [24]. The input is parsed, analyzed and merged to create a single abstract syntax tree (AST) representing the whole program. This AST then undergoes a sequence of transformations to replace all of the concurrency primitives with equivalent CIVL-C primitives, and to simplify the AST in other ways, resulting in a "pure" CIVL-C AST.

The back end first converts the pure AST to a lower-level representation in which each procedure is represented as a program graph. A node in this graph represents a program counter value, i.e., a location in the procedure body. An edge represents an atomic transition, and is decorated with a guard expression that specifies when the transition is enabled, and a basic statement, such as an assignment. The verifier then performs an explicit enumeration of the reachable states of the program ("model checking"). This is carried out by depth-first search, while saving the seen states in a hash table. Each state maps variables to symbolic expressions and includes a path condition—a symbolic expression of boolean type that records the guards that held along the explored path ("symbolic execution"). An interleaving model of concurrency is used, and processes can be created and destroyed dynamically.

During the search, automated theorem provers are invoked to determine whether the path condition has become unsatisfiable (in which case the search backtracks) and to check assertions. CIVL checks both explicit assertions appearing in the program and implicit assertions (a divisor is not 0, a pointer deference is valid, and so on). The supported provers include Z3 [20], CVC4 [4], Why3 [5], and a number of additional provers invoked by Why3.

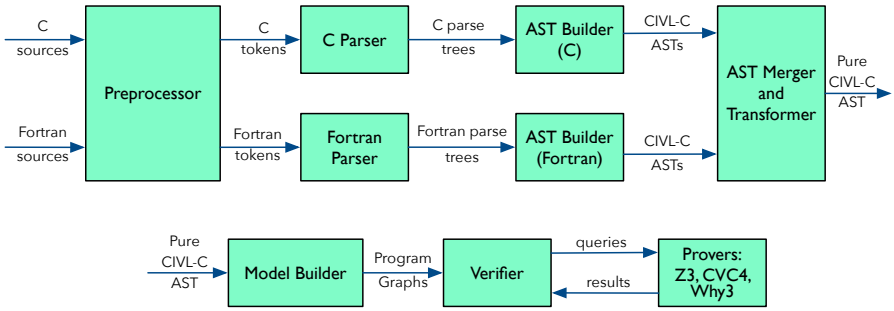

Fig. 1. CIVL architecture: front end (top) and back end (bottom)

Figure 1 shows the tool prior to this work and highlights the extensions developed as part of this paper. Modifications were made to both the front and back end to enable the direct application of CIVL to Fortran source code.

The CIVL preprocessor was generalized to accept a superset of C and Fortran: it is common practice to use C preprocessor directives in Fortran programs and Fortran compilers can invoke the preprocessor as a first pass. The tokens emanating from the preprocessor have a type specific to the source language—C or Fortran—which is determined by the file suffix or a command line option. It is possible to invoke CIVL on a mix of C and Fortran source files—each will be preprocessed separately and yield a separate stream of tokens in the correct language.

Each Fortran token stream enters the Fortran parser. This was produced by the parser generator ANTLR [28] using a grammar derived from the Open Fortran Project (OFP) [31]. We extended the grammar by adding support for CIVL primitives, such as assertions and assumptions, which can appear as structured comments in the Fortran source. The parser produces a parse tree, which is then converted to a CIVL-C AST. Each C token stream follows a similar path, and also results in a CIVL-C AST. Finally, the individual ASTs, together with ASTs generated from any libraries, are merged into a single AST, analogous to the linking phase in a standard compilation flow. The supported Fortran subset is listed below:

- **program units**: main programs, subroutines, and functions
- **statements**: allocate, assignment, call, computed goto, data, dimension, do, exit, goto, if, implicit, intent, parameter, pointer assignment, print, return, stop, target, type declaration, and write
- **expressions**: variable references, function calls, operators for scalar types
- **intrinsic functions**: mod, max, abs, sin, cos, atan, and sqrt
- **extended features**: CIVL preprocessor directives and CIVL primitives.

The transformation from a Fortran parse tree to a CIVL-C AST is quite involved because the languages differ substantially. In almost every case, we were able to find a way to represent a Fortran statement—in a semantics-preserving

and defect-preserving way—using existing CIVL-C AST nodes; in a few cases we had to add new fields to the AST node. Issues include the Fortran "intent" specification for a procedure parameter (in, out, or in/out); pass-by-reference semantics; and advanced array operations. Details for some of these translations are described in Section 3.

The verifier was also upgraded to check specific Fortran runtime constraints during state exploration. For example, the verifier normally uses short-circuit semantics for evaluating *and* and *or* expressions. This is appropriate for C, but Fortran does not mandate short-circuiting or the order in which subexpressions are evaluated. Since evaluation can result in error, a verifier which assumes short-circuiting semantics could miss defects in a Fortran program. By default, our modified verifier turns off short-circuiting for Fortran code.

3 Defect-Preserving Translation

When used for verification, it is crucial that all translation phases preserve defects. This is in contrast to translation and lowering phases in a compiler, which generally are allowed to narrow the semantics of a program or choose arbitrarily from multiple interpretations. In this section, we first demonstrate with small examples that an approach relying on existing source-to-source translation tools such as f2c, or compiler front ends such as Flang, is bound to miss certain defects in the Fortran input, since these defects are removed by these tools. One might be tempted to argue that defects which disappear during translation or compilation are not really important. However, these defects are still present in the original source code and may manifest themselves when a different compiler is used or when other seemingly innocent changes are made to the code or the translation/compilation tool chain.

3.1 Translation from Source to Source

Figure 2 shows a procedure in Fortran 77 and its C translation produced by f2c. The example extracts a value x from an array at the given index, and computes $\max(x, 0)$. An array bounds check is performed in the same boolean expression in which the array is accessed. The C code is certainly valid, because it uses short-circuiting when evaluating logic expressions. Thus, the evaluation of the second part of the boolean expression is skipped if the first part is false. Fortran, on the other hand, does not define the order in which the subexpressions are evaluated,

```
1 if (idx .le. size_arr .and.
2     arr(idx) .ge. 0) then
3   relu = arr(idx)
4 else
5   relu = 0
6 end if
```

```
1 /* Function Body */
2 if (*idx <= *arr_size__ && arr[*idx] >= 0.f) {
3   *relu = arr[*idx];
4 } else {
5   *relu = 0.f;
6 }
```

Fig. 2. Applying f2c to Fortran *and* operator removes a defect

and the compiler may choose an evaluation order that causes an out-of-bounds access in the second half of the expression. The implementation-defined order chosen by f2c happens to remove this defect during translation, which makes it difficult to detect for a verifier that is only provided with the C program. Nevertheless, the Fortran program may break when a different translator or compiler tool chain is used to execute it.

Besides the lack of defect preservation, there are other drawbacks when using a source-to-source converter, including the fact that some of them introduce hard-to-verify external headers or libraries to simulate Fortran behaviors. Further, by verifying translated code, source file information (e.g., file and identifier names, code locations, etc.) can be harder to communicate to the user, and translation tools may actually introduce new errors, leading to another potential source of unreliable verification results.

3.2 Translation for Compilation

A popular approach for verifying source code is to build a verifier based on a mature compiler tool chain (e.g., LLVM [16]). This allows verification researchers to spend more of their time on research and less time on maintaining language front ends, and allows robust support of a variety of languages. We argue that such an approach, while also very valuable, achieves a different outcome than what we present in our work. Compiler front ends such as Clang or Flang are not developed with the goal of preserving defects, and defective programs may be lowered into correct LLVM intermediate representation (IR). Furthermore, the compiler may in rare cases introduce new defects due to compiler bugs. In the absence of such compiler bugs, verification based on the IR will ensure that the input program is correct *if compiled with the same compiler and settings that were used for verification*. With our approach, we instead aim to verify that a program adheres to the language standard.

Figure 3 shows the LLVM-IR produced by Flang (version 1.5 2017-05-01) for the Fortran code snippet in Figure 2. Similar to the case with f2c, it first checks the array bounds by comparing %15 (element index) with %17 (array size). If the index is out of bounds (i.e., %18 is evaluated as *true*), then the control flow skips the block that accesses the array elements, and the second subexpression

```
1 L.LB1_339: ; preds = %L.entry
2 %14 = bitcast i64* %idx to i32*, !dbg !18
3 %15 = load i32, i32* %14, align 4, !dbg !18
4 %16 = bitcast i64* %arr_size to i32*, !dbg !18
5 %17 = load i32, i32* %16, align 4, !dbg !18
6 %18 = icmp sgt i32 %15, %17, !dbg !18
7 br i1 %18, label %L.LB1_313, label %L.LB1_349, !dbg !18
8 ..
9 L.LB1_313: ; preds = %L.LB1_349, %L.LB1_339
10 %41 = bitcast i64* %relu to float*, !dbg !21
11 store float 0.000000e+00, float* %41, align 4, !dbg !21
12 br label %L.LB1_314
```

Fig. 3. Result of applying Flang to Fortran code of Figure 2

```
subroutine intent_bad(i)
  integer, intent(out) :: i
  i = i + 1
end subroutine
```

```
subroutine intent_good(i)
  integer, intent(inout) :: i
  i = i + 1
end subroutine
```

```
void INCR(int* __OUT_I) {
  int I;
  I = I + 1;
  *__OUT_I = I;
}
```

Fig. 4. Fortran routine that fails to conform to specified intent; one that conforms; and CIVL translation of the non-conforming code

in the condition expression is omitted. This means that the defect in the original Fortran code is undetectable in the IR.

Figure 4 is another case where Flang translates an incorrect program into valid IR. A Fortran subroutine may use the `INTENT` attribute in an illegal way, for example by declaring an argument as `INTENT(OUT)` and subsequently reading from it. This is problematic since the value of such a variable is undefined at the entry of the subroutine, even if it was initialized in the caller. Flang nevertheless generates identical LLVM IR for the two subroutines, the first of which violates the Fortran standard, the second of which declares the same argument as `INTENT(INOUT)` and hence correctly passes the variable into and out of the subroutine.

3.3 Translation for Verification

Based on these observations, we extended CIVL with a front end to translate Fortran to CIVL-C ASTs in a way that is designed to preserve defects. The front end avoids AST simplifications and optimizations that may introduce or remove defects, or that may hide violations of the Fortran language standard. The short-circuit evaluation of logic expressions is disabled by default when verifying Fortran source. When processing the code of Figure 2, the CIVL-C AST builder thus keeps both subexpressions in the condition, and all parts of the expression are evaluated in the verification phase. The model checker consequently reports an out-of-bounds access in the array.[3]

We also developed a static analyzer to detect certain defects before the program is even translated to a CIVL-C AST. The analyzer mainly checks constraints on variable attributes or procedure specifications. For example, variables in Fortran may have the `ALLOCATABLE`, `POINTER`, or `TARGET` attribute. It is legal to pointer-assign a variable with the `POINTER` attribute to a variable with the `TARGET` or `POINTER` attribute, but not to a variable without any of these attributes. Both sides of each pointer assignment are statically checked for required attributes by the analyzer. When all constraints of a specific attribute are verified for each associated variable, that attribute information is not passed to the model checker. Similarly, a subroutine or function is only allowed to be recursively called if it has the `RECURSIVE` attribute, and our analyzer checks this by searching for loops in the call graph and checking if subroutines or functions

[3] It is also possible (however unlikely) that a defect may manifest only when short-circuiting is *enabled*. A strictly conservative solution could use nondeterministic choice to decide, at each logical expression, whether to short-circuit. We plan to add such an option to CIVL.

that are part of a loop have the required attribute. As a result, this kind of constraint is checked by the analyzer and it is not necessary to include certain attributes in the CIVL-C AST.

The defect-preserving translation is mainly performed by the Fortran AST builder shown in Figure 1. For properties that can not be verified by the analyzer, the translation phase inserts auxiliary structures into the CIVL-C AST for verification in a later phase. For example, the subroutines in Figure 4 have distinct CIVL-C AST structures. A formal parameter having `INTENT(OUT)` attribute is initialized with a value representing "undefined." This allows the model checker to find and report a violation (reading an undefined value) during the transition executing the assignment statement. The CIVL translation of the incorrect routine is shown in Figure 4(right).

In summary, our extended front end focuses on preserving defects and translates source code into a CIVL-C AST specifically designed for verification. Violations of variable attributes and function specifications are guaranteed by performing specialized analysis or by inserting auxiliary information into the AST that is analyzed in a later phase.

4 Fortran Array Modeling

Fortran arrays are more powerful than arrays in most other languages, and require special handling during the translation to CIVL-C. Section 4.1 will briefly discuss some of the features of Fortran arrays, before we discuss how these features are modeled in Section 4.2.

4.1 Fortran Array Semantics

Fortran natively supports multi-dimensional arrays. For example, `b` and `c` in Figure 5 are two-dimensional arrays. Fortran stores arrays in column major style, unlike C arrays, which are stored in row major style.

```
1 REAL:: b(6,3), c(0:9,-3:3), u(3)
2 REAL, POINTER, DIMENSION(:) :: p
3 INTEGER, DIMENSION(3) :: idx
4 ! copy columns -1, 0, 1 from every other row of c into the first 5 rows of b
5 b(1:5,:) = c(::2,-1:1)
6 ! fill the array idx with constant values 1, 4, 17
7 idx = (/1, 4, 17/)
8 ! use the array idx as indices into a. This will copy a[1,4,17] into u[1,2,3]
9 u = a(idx)
10 ! associate the pointer p with column 1 in array c
11 p => c(:,1)
12 b = 42.0
```

Fig. 5. Examples of Fortran array usage: a 2-dimensional array of size 6×3, a 2-dimensional array with non-default index ranges, a pointer to a one-dimensional array, and two one-dimensional arrays of size 3, one for integers and one for reals, are declared. Following that, several data copy operations and pointer associations are performed.

Arrays in Fortran are 1-based by default, just like in Matlab or Julia, but unlike in C and many other languages. However, Fortran allows the base to be specified for each array dimension. For example, `c` in Figure 5 represents a two dimensional array whose row dimension of size 10 is 0-based and whose column dimension ranges from −3 to 3. Array sizes and index ranges can be either defined statically or calculated from parameters or function and subroutine arguments.

Fortran programs can in most situations determine the size of arrays using the intrinsic `size` or `shape` functions. It is also possible to modify an entire array, or an array along an entire dimension, without explicitly referring to its size. For example, one can assign a scalar value to an entire array though a simple assignment as shown in line 12 of Figure 5. Fortran compilers usually implement this behavior using an array descriptor that is embedded in the generated program and contains the array size and shape information.

Furthermore, Fortran supports the extraction of slices from an array by specifying a *subscript triplet* for each dimension, which specifies a lower and upper bound on the index as well as a stride. It is possible to omit the lower (and/or upper) bound, in which case the start (and/or end) of the array is used. An optional stride n can be specified to extract only every n-th element. For example, line 5 in Figure 5 extracts even rows, and of those, only the columns from −1 to 1, from `c`. These values are then copied into the first five rows of `b`. Instead of subscript triplets, one can also use an integer array as an index for another array. This is shown in line 9. Fortran provides other ways to modify or reinterpret arrays, including the `reshape` function that can change the number of dimensions and the size in each dimension, and has optional arguments to pad or reorder an array.

When an array is passed to a function or subroutine as an argument, it may be accessed with a different index scheme inside that function or subroutine. For example, a three-dimensional array with index ranges $[0:8][0:2][0:2]$ could be passed to a subroutine that internally declares this argument as an array with ranges $[1:9][1:3][1:3]$ or $[0:8][0:8]$ or any other number of dimensions or index ranges, as long as the array within the callee has at most as many overall entries as the array within the caller. This essentially provides a *view* of the original array, and because Fortran uses the call-by-reference paradigm, any changes to this re-interpreted array within the callee will also affect the original array in the caller. Depending on the situation, the Fortran compiler may implement this using an array descriptor and suitable index expressions, or by transparently copying data to and from an array that is used within the callee.

A similar situation occurs when Fortran pointers are used. Despite their similar name with C pointers, their behavior and features differ significantly. Fortran pointers can represent a view into a multi-dimensional array, and contain size and shape information. For example, a pointer can be associated with an array slice that represents column 1 across all rows in an array, as shown in line 11 of Figure 5. In this case, writing to the first element in `p` will also modify the first row in `c`'s column 1. The `size` and `shape` functions can be used on `p` and

will return the size and shape of the portion of `c` that `p` is associated with. The pointer itself can be accessed with a subscript triplet or index array, and the pointer can be passed to a subroutine or function that may reinterpret it with a different dimensionality or index range.

There are a number of details regarding the use of arrays and pointers in Fortran that we do not discuss in this paper for brevity. We refer to [1] (particularly Sections 5.4, 5.6 and 12.6.4) for a more thorough discussion.

4.2 Modeling Fortran Arrays for Verification

Arrays in CIVL-C always have indices starting at 0 and do not support strides, sectioning, or reshaping. To handle the features described in the previous subsection, each Fortran array is modeled by a CIVL-C array that is augmented with a recursive data structure called `FORTRAN_ARRAY_DESCRIPTOR`. This allows CIVL to model the rich Fortran array semantics using only CIVL-C language features. As Figure 6 shows, the descriptor stores metadata for an array instance, and contains the kind, rank, index upper and lower bounds and strides, as well as a pointer.

When a Fortran program creates a new array from scratch, CIVL will create a CIVL-C array whose length is the total number of elements in the Fortran array. This array is then augmented with an array descriptor whose kind is `SOURCE` and whose pointer holds the memory address of the CIVL-C array. The bounds and stride in the descriptor are set according to those set by the Fortran program. In essence, the descriptor provides a mapping from the Fortran array index (which may be strided or non-zero-based) into the CIVL-C array index (which is dense and zero-based). This mapping is used by the CIVL-C program whenever the Fortran program accesses the array.

If a Fortran array instance is created by reshaping or sectioning an existing array, no new CIVL-C array is created. Semantically, the new array instance in Fortran provides a view into the existing array, which we model by creating a new array descriptor with appropriate bounds and stride whose kind is

```
1 typedef struct FORTRAN_ARRAY_MEMORY *farr_mem;
2 typedef struct FORTRAN_ARRAY_DESCRIPTOR *farr_desc;
3 typedef enum FORTRAN_ARRAY_DESCRIPTOR_KIND {
4   SOURCE, // A var. decl. w/ an array type or a dimension attr.
5   SECTION, // An array section
6   RESHAPE // An array, whose indices are reshaped w/ no cloning
7 } farr_kind;
8 struct FORTRAN_ARRAY_DESCRIPTOR {
9   farr_kind kind; // The kind of a Fortran array descriptor
10  unsigned int rank; // The rank or the number of dimensions.
11  int *lbnd; // A list of index left-bounds for each dim.
12  int *rbnd; // A list of index right-bounds for each dim.
13  int *strd; // A list of index stride for each dim.
14  farr_mem memory; // Being non-null iff kind is 'SOURCE'
15  farr_desc parent; // Being non-null iff kind is NOT 'SOURCE'
16 };
```

Fig. 6. Implementation of the CIVL-C array descriptor.

```
1 PROGRAM ARRAYOP
2   INTEGER :: A(0:8)
3   CALL SUBR(A(1:7:2))
4 ! A: {0,1,0,2,0,3,0,4,0}
5 END PROGRAM ARRAYOP
6
7 SUBROUTINE SUBR(B)
8   INTEGER :: B(-1:0, 2:3)
9   B(-1, 2) = 1
10  B(-1, 3) = 2
11  B( 0, 2) = 3
12  B( 0, 3) = 4
13 END SUBROUTINE SUBR
```

```
1 int main() {
2   fa_desc A = fa_create(sizeof(int), 1, {{0},{8},{1}});
3   fa_desc __arg_A = fa_section(A, {{1},{7},{2}});
4   subr(__arg_A);
5   fa_destroy(__arg_A); ! pop section descriptor
6   fa_destroy(A); ! free array descriptor and data storage
7 }
8 void subr(fa_desc __B) {
9   fa_desc B = fa_reshape(__B, 2, {{-1,2},{0,3},{1,1}});
10  *(int*)fa_subscript(B, {-1,2}) = 1;
11    ...
12  *(int*)fa_subscript(B, {0,3}) = 4;
13  fa_destroy(B); ! pop reshape descriptor
14 }
```

Fig. 7. Transformation of array section and reshape operations

set to `SECTION` or `RESHAPE`, and whose pointer stores the location of the array descriptor for the existing array. This new descriptor now provides a mapping from indices of the new array instance into indices of the existing array instance. Such an array section or reshaped array can itself be reshaped or sectioned by the Fortran program, which will result in a stack of array descriptors. Whenever the Fortran program accesses an array at a given index, CIVL will recursively use the mappings provided by the descriptors until the index in the underlying CIVL-C array is resolved by a descriptor of kind `SOURCE`. Figure 7 shows how some basic Fortran array operations are translated to CIVL-C using the array descriptor and associated utility functions.

5 Evaluation

The first goal of this evaluation is to determine whether CIVL correctly verifies or finds defects in a suite of synthetic Fortran programs that use various language features peculiar to Fortran. The second goal is to investigate how CIVL performs on Fortran code from an existing production-level HPC application.

5.1 Compute Environment and Experimental Artifacts

All CIVL executions were conducted on a TACAS 2022 Artifact Evaluation Virtual Machine (AEVM) with Ubuntu 20.04; the version of CIVL is 1.21. All SMACK executions were conducted on a TACAS 2020 AEVM provided by the authors of [13]; the version of SMACK is 1.9.1. Both virtual machines were deployed by Oracle VirtualBox 6.1 on a laptop running MacOS 11.6.2 on a 2.5 GHz Quad-Core Intel Core i7 CPU with x86_64 architecture and 16 GB memory. The CIVL program and all experimental artifacts can be downloaded from https://vsl.cis.udel.edu/tacas2022.

5.2 Specification and Verification Approach

As shown in Figure 8, CIVL primitives are inserted as structured comments for verifying a Fortran code, which have no effect on the normal build process. Similar directives exist for C. These primitives have two major kinds: type qualifiers

```
1 PROGRAM civl_primitive_example
2 !$CVL $input
3   INTEGER :: arg
4   INTEGER :: x
5 !$CVL $assume(-1 .LE. arg .AND. arg .LE. 1);
6   x = arg
7 !$CVL $assume(x .LT. 0);
8     x = ABS(x)
9 !$CVL $assert(0 .LE. x .AND. x .LE. 1);
10 END PROGRAM civl_primitive_example
```

Fig. 8. Example illustrating CIVL Fortran primitives.

and verification statements. `$input` specifies that the variable in the following declaration is to be initialized with an unconstrained value of its type. The value can be subsequently constrained with an assumption statement. Alternatively, an input variable may be given an exact concrete value on the command line. Input variables are read-only.

The `$output` qualifier declares a variable to be write-only. Output variables are used for functional equivalence verification. When two programs have the same input and output variables, they can be compared to determine whether, given the same inputs, the two programs will produce the same outputs. This is carried out by CIVL's *compare* command, which merges the two programs into a single program with a new driver. The driver invokes the two programs in sequence on the same input variables, and then asserts that the corresponding outputs agree.

A CIVL assumption statement has the form `$assume(expr);`. It is used to constrain the set of executions that are considered to be valid. If an assumption is violated, no error is reported; instead, the execution is ignored and the search backtracks immediately. `$assert(expr);` reports an assertion violation if the argument expression does not hold. This statement provides the capability of checking desired properties in Fortran, which has no intrinsic assertion procedure. All primitives must be preceded by the prefix `!$CVL`.

5.3 Fortran Verification Benchmark Suites

Our suite incorporates the 22 synthetic examples from the SMACK suite [13]. These examples cover basic Fortran structures ranging from expressions to functions and subroutines. The only change made is to switch SMACK-style assertions and symbolic value assignment to CIVL primitives. SMACK uses calls of the form `assert(`*`expr`*`)` to check desired properties, which is similar to CIVL's `$assert` primitive. With SMACK, symbolic values are generated by calling `__verifier_nondet_int()` and assigning the result to a variable, while CIVL uses the `$input` qualifier.

To these, we added 13 examples we created ourselves, exercising different language features, including argument intent specification, array sectioning, and boolean expressions that might lead to different results if short-circuiting is or is not used. We include a parallel example that uses an OpenMP for loop, executed

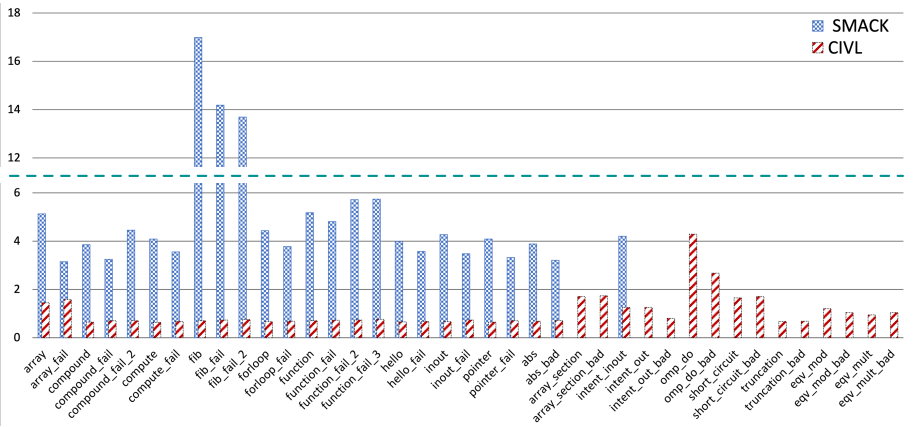

Fig. 9. Total verification time (in seconds) for CIVL and SMACK on benchmarks. Each time is the mean over 5 of 7 executions after dropping the shortest and longest.

with 4 threads. Finally, we constructed 4 pairs of programs each of which can be compared for functional equivalence.

The programs are listed on the x-axis in Figure 9. Where the name includes "fail" or "bad", a negative verification result is expected; otherwise, a positive result is expected. The figure also shows the average verification execution time printed by CIVL and SMACK on each example. CIVL has correct results in all cases, while SMACK encounters exceptions or has incorrect results for some of the CIVL Fortran examples. Thus, the figure only reports timing results when the verification results are correct.

5.4 Verifying Nek5000 Components

Nek5000 [21] is a computational fluid dynamics code for simulating unsteady incompressible two- or three-dimensional fluid flow. Nek5000 has hundreds of industrial and academic users and won a Gordon Bell prize for its scalability on high performance compute clusters.

The code contains many Fortran subroutines that perform a numerical computation that can be easily expressed in a formal way. For example, there are various implementations for matrix multiplication, each optimized for best performance on a particular matrix size. We use CIVL to verify that these subroutines indeed compute matrix multiplications, by showing their equivalence with a straightforward un-optimized textbook implementation.

Furthermore, Nek5000 contains subroutines to numerically approximate the integral of a function, a process known as quadrature. Quadrature rules typically define carefully chosen locations, known as quadrature points, at which the function in question is evaluated. The results are then each multiplied with a weight, and summed to obtain the overall integral. The quality of a quadrature rule is often evaluated by quantifying its order of accuracy, where a higher order

```
N Points  2 Degree  2
Ref soln  2*AF_SIN(2)
Quadrature  2*AF_SIN(2)
Expected error: ZERO

  .. Program Output Message ..

=== Source files ===
util.f  (util.f)
driver_speclib.f  (driver_speclib.f)
speclib.f  (speclib.f)


=== Command ===
civl verify -checkMemoryLeak=false
util.f driver_speclib.f speclib.f

=== Stats ===
   time (s)            : 11.64
   memory (bytes)      : 3393191936
   max process count   : 1
   states              : 54336
   states saved        : 50392
   state matches       : 0
   transitions         : 54335
   trace steps         : 35239
   valid calls         : 148085
   provers             : cvc4, z3, why3
   prover calls        : 10

=== Result ===
The standard properties hold for all
executions.
```

```
Violation 0 encountered at depth 3244:
CIVL execution violation in p0
(kind: ASSERTION_VIOLATION, certainty:
MAYBE)
at driver_speclib_bad.f:103.6-12

!$CVL $ASSERT(DIFF .EQ. MINDIFF)
      ^^^^^^^


 .. Detailed Violation Info ..

=== Source files ===
  ..

=== Command ===
  ..

=== Stats ===
   time (s)            : 4.19
   memory (bytes)      : 2587885568
   max process count   : 1
   states              : 4973
   states saved        : 4585
   state matches       : 0
   transitions         : 4974
   trace steps         : 3244
   valid calls         : 13662
   provers             : cvc4, z3, why3
   prover calls        : 7

=== Result ===
The program MAY NOT be correct.  See
CIVLREP/util_log.txt
```

Fig. 10. CIVL output for verifying correct and erroneous Nek5000 examples

quadrature rule yields the exact result for polynomials of a higher degree. The Gauss-Lobatto Legendre quadrature rules are a unique set of weights and points that are known to be optimal under certain conditions, and are used in Nek5000. We use CIVL to verify that the quadrature implemented in Nek5000 indeed has the claimed order of accuracy, by verifying that the quadrature is exact for polynomials with symbolic coefficients of the claimed degree. Due to its uniqueness properties, this also proves that Nek5000 indeed uses Gauss-Lobatto Legendre weights and points.

We also seeded some of these implementations with defects and confirmed that CIVL reports the defects. Figure 10 shows the output from CIVL on a correct and incorrect example from Nek5000. Table 1 shows the verification results for the Nek5000 excerpts for various parameter values. The expected result is obtained in all cases, at modest cost (at most 12 seconds).

6 Related Work

Fortran has been the focus of early program verification research. One of the first papers on symbolic execution dealt with Fortran [8], and one of the earliest verification condition generation tools was for Fortran [6]. More recently, several Fortran static analyzers have been developed, including ftnchek [19], Cleanscape FORTRAN-Lint [9], and FORCHECK/Coverity [35]. These tools detect certain

Name	LoC	Result	Scale	Time	States
speclib	560	True	$2 \leq \texttt{NP} \leq 2$; $2 \leq \texttt{DEG} \leq 3$	5.14s	10857
speclib	560	True	$2 \leq \texttt{NP} \leq 3$; $2 \leq \texttt{DEG} \leq 5$	12.08s	55908
speclib_bad	560	False	$2 \leq \texttt{NP} \leq 2$; $2 \leq \texttt{DEG} \leq 3$	4.67s	6011
speclib_bad	560	False	$2 \leq \texttt{NP} \leq 3$; $2 \leq \texttt{DEG} \leq 5$	4.27s	3223
mxm_unroll	458	Eqv	3×3	5.49s	26867
mxm_unroll	458	Eqv	4×4	8.51s	59914
mxm_unroll_bad	458	NEq	3×3	5.48s	26865
mxm_unroll_bad	458	NEq	4×4	8.56s	59912
mxm_pencil	458	Eqv	2×2	5.83s	9264
mxm_pencil	458	Eqv	3×3	7.38s	26893
mxm_pencil	458	Eqv	4×4	10.14s	59968
mxm_pencil_bad	458	NEq	2×2	6.01s	9262
mxm_pencil_bad	458	NEq	3×3	7.53s	26891
mxm_pencil_bad	458	NEq	4×4	10.48s	59966

Table 1. Results of verifying Nek5000 code excerpts at various scales

pre-defined generic defects, such as variables that are read but never written, unused variables and functions, and inconsistencies in common block declarations. They do not allow one to specify and verify functional correctness properties.

Other tools use dynamic analysis (or a combination of static and dynamic analysis) to check such generic properties. One example uses the PIPS compiler to detect forbidden aliasing in subroutines [22]. The NAG Fortran compiler can also insert checking code to catch many defects at runtime [29].

In contrast, CamFort [27] implements a lightweight specification and static analysis approach. The user annotates the Fortran program with comments in a domain specific language for specifying array access patterns (stencils) or associating units of measurements to variables. CamFort, which is written in Haskel, parses the code, constructs an AST, and verifies conformance to the properties using Z3. This approach strikes a balance between the generality of program verifiers such as CIVL, which can specify arbitrary assertions in a general purpose assertion language, and the more tractable static analysis tools.

Several tools have been developed to translate Fortran to other languages. These include f2c [11] (which translates to C) and Fable [14] (C++). In addition to the issues discussed in Section 3.1, the potential of these tools as front ends for verifiers is limited by the fact that the translated code is often considerably more complex than the original or involves complex libraries which the verifier must also understand. It should be noted that Fable's approach to modeling Fortran arrays is similar to ours in that it defines a class that bundles a reference to the data with meta-data describing the "view" of the array.

A number of verification tools work off of the LLVM compiler's low-level intermediate language, LLVM IR. These include SMACK [30], Divine [2], LLBMC [34], and SeaHorn [15]. In theory, this should allow one to chain together any of the many compiler front ends that generates LLVM IR with a general LLVM

IR verifier. In practice, this is very difficult, and most of these verifiers accept only a subset of LLVM IR generated by a particular front end from a particular source language—usually C or C++ [13]. To the best of our knowledge, only SMACK has been applied to Fortran [13], using the Flang front end [12]. However, the subset of Fortran accepted and the example codes themselves are small. A more significant concern, discussed in Section 3, is that a front end may "compile away" defects in the source program by choosing one of several acceptable ways to translate a construct with unspecified behavior, or assuming the absence of undefined behaviors.

In this work we have translated Fortran to the intermediate verification language (IVL) CIVL-C. Other, more widely-used, IVLs include Boogie [3] and Why3 [5]. Among these languages, CIVL-C stands out for its robust support for pointers and concurrency, which simplifies much of the modeling effort.

The CIVL verifier analyzes a CIVL-C program using symbolic execution, a widely-used technique for test-case generation and verification. Other mature symbolic execution tools include KLEE [7] (for C programs, via LLVM) and Symbolic PathFinder [23] (for Java byte code).

7 Conclusion and Future Work

We presented a Fortran extension to CIVL, a novel model-checking approach that preserves and reveals defects in source code written in Fortran. Compared with compiler-based verifiers, this tool parses and analyzes source programs from a verification perspective. In doing so, it mitigates against the risk of missing defects that are eliminated via legal but non-defect-preserving compiler optimizations.

The extension includes a data structure and associated algorithms for describing Fortran array metadata and tracking complex array transformations. This method of handling Fortran arrays could be adopted by other verification tools. The extension also supports a set of CIVL verification primitives which can be introduced into Fortran programs as structured comments.

Evaluation results show that our tool performs correctly and quickly (compared to previous work) on a range of synthetic benchmarks and some kernels extracted from real world applications. In the future, we plan to enlarge the supported subset of Fortran language features and to enhance support for verifying Fortran programs with OpenMP directives. The resulting CIVL extension is expected to cover the DataRaceBench [36] suite, including both the C and Fortran examples.

Acknowledgements

This material is based upon work by the RAPIDS Institute, supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, and by contracts DE-AC02-06CH11357 and DE-SC0021162. Support was also provided by U.S. National Science Foundation award CCF-1955852.

References

1. Adams, J.C., Brainerd, W.S., Hendrickson, R.A., Maine, R.E., Martin, J.T., Smith, B.T.: The Fortran 2003 Handbook: the Complete Syntax, Features and Procedures. Springer Science & Business Media (2008). https://doi.org/10.1007/978-1-84628-746-6
2. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: Automated Technology for Verification and Analysis. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_14
3. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium (FMCO 2005). Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011), http://dl.acm.org/citation.cfm?id=2032305.2032319
5. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wrocław, Poland (August 2011), http://proval.lri.fr/publications/boogie11final.pdf
6. Boyer, R.S., Moore, J.S.: A verification condition generator for Fortran. Tech. Rep. CSL-103, SRI International, Computer Science Laboratory, Menlo Park, CA (June 1980), https://apps.dtic.mil/sti/pdfs/ADA094609.pdf
7. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08) (2008)
8. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng. **2**, 215–222 (May 1976). https://doi.org/10.1109/TSE.1976.233817
9. Cleanscape Software International: FORTRAN-lint: a pre-compile analysis tool, https://stellar.cleanscape.net/docs_lib/data_F-lint2.pdf, accessed 13-Oct-2021
10. Dingle, N.: Not only Fortran and MPI: POP's view of HPC software in Europe, https://pop-coe.eu/blog/not-only-fortran-and-mpi-pops-view-of-hpc-software-in-europe, accessed 14-Oct-2021
11. Feldman, S.I.: A Fortran to C converter. SIGPLAN Fortran Forum **9**(2), 21–22 (Oct 1990). https://doi.org/10.1145/101363.101366
12. Flang Fortran language front-end. https://github.com/flang-compiler/flang, accessed 09-Oct-2021
13. Garzella, J.J., Baranowski, M., He, S., Rakamarić, Z.: Leveraging compiler intermediate representation for multi- and cross-language verification. In: Beyer, D., Zufferey, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 90–111. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39322-9_5
14. Grosse-Kunstleve, R.W., Terwilliger, T.C., Sauter, N.K., Adams, P.D.: Automatic Fortran to C++ conversion with FABLE. Source Code for Biology and Medicine **7**(5) (2012). https://doi.org/10.1186/1751-0473-7-5

15. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: A framework for verifying C programs (competition contribution). In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 9035, pp. 447–450. Springer Berlin Heidelberg, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_41
16. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO'04). pp. 75–86. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665
17. Lawrence Livermore National Laboratory: CORAL benchmark codes (2014), https://asc.llnl.gov/coral-benchmarks, accessed 14-Oct-2021
18. Message Passing Interface Forum: MPI: A Message-Passing Interface standard, version 3.1 (Jun 2015), https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
19. Moniot, R.K.: ftnchek: a static analyzer for Fortran 77, https://www.dsm.fordham.edu/~ftnchek/, accessed 09-Oct-2021
20. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
21. NEK5000: a fast and scalable high-order solver for computational fluid dynamics (2021), https://nek5000.mcs.anl.gov, accessed 14-Oct-2021
22. Nguyen, T.V.N., Irigoin, F.: Alias verification for Fortran code optimization. Electronic Notes in Theoretical Computer Science **65**(2), 52–66 (2002). https://doi.org/10.1016/S1571-0661(04)80396-7, COCV'02, Compiler Optimization Meets Compiler Verification (Satellite Event of ETAPS 2002)
23. Noller, Y., Păsăreanu, C.S., Fromherz, A., Le, X.B.D., Visser, W.: Symbolic Pathfinder for SV-COMP. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 239–243. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_21
24. NVIDIA: CUDA Toolkit Documentation, v11.4.2, https://docs.nvidia.com/cuda/, accessed 14-Oct-2021
25. Open Group: IEEE Std 1003.1: Standard for information technology—Portable Operating System Interface (POSIX(R)) base specifications, issue 7: pthread.h (2018), https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html
26. OpenMP Architecture Review Board: OpenMP Application Programming Interface (Nov 2020), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, version 5.1
27. Orchard, D., Contrastin, M., Danish, M., Rice, A.: Verifying spatial properties of array computations. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 1–30 (Oct 2017). https://doi.org/10.1145/3133899, article no. 75
28. Parr, T.: The Definitive ANTLR4 Reference. The Pragmatic Bookshelf, Dallas, TX (2013), https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference/
29. Polyhedron Solutions: Linux Fortran compiler diagnostic comparisons, https://www.fortran.uk/fortran-compiler-comparisons/intellinux-fortran-compiler-diagnostic-capabilities/, accessed 13-Oct-2021

30. Rakamarić, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementation. In: Biere, A., Bloem, R. (eds.) Proceedings of the 26th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 8559, pp. 106–113. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_7
31. Rasmussen, C.E., et al.: OFP: Open Fortran Project, https://sourceforge.net/p/fortran-parser/wiki/Home/, accessed 14-Oct-2021
32. Rosner, R., Calder, A., Dursi, L., Fryxell, B., Lamb, D., Niemeyer, J., Olson, K., Ricker, P., Timmes, F., Truran, J., Tufo, H., Young, Y.N., Zingale, M., Lusk, E., Stevens, R.: Flash code: Studying astrophysical thermonuclear flashes. Computing in Science and Engineering **2**, 33–41 (2000). https://doi.org/10.1109/5992.825747
33. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: CIVL: The Concurrency Intermediate Verification Language. In: SC15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, New York (Nov 2015). https://doi.org/10.1145/2807591.2807635, article no. 61, pages 1–12
34. Sinz, C., Merz, F., Falke, S.: LLBMC: A bounded model checker for LLVM's intermediate representation. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 7214, pp. 542–544. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_44
35. Synopsys: Synopsys static analysis (Coverity) Fortran syntax analysis, https://community.synopsys.com/s/article/Synopsys-Static-Analysis-Coverity-Fortran-Syntax-Analysis, accessed 13-Oct-2021
36. Verma, G., Shi, Y., Liao, C., Chapman, B., Yan, Y.: Enhancing DataRaceBench for evaluating data race detection tools. In: 2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness). pp. 20–30. IEEE (2020). https://doi.org/10.1109/Correctness51934.2020.00008