Crossover: Towards Compiler-Enabled COBOL-C Interoperability

Mart van Assen
mart@vanassen.info
Computer Science, University of Twente
Enschede, The Netherlands

Manzi Aimé Ntagengerwa
m.a.ntagengerwa@gmail.com
Computer Science, University of Twente
Enschede, The Netherlands

Ömer Faruk Sayilir
o.f.sayilir@student.utwente.nl
Computer Science, University of Twente
Enschede, The Netherlands

Vadim Zaytsev
vadim@grammarware.net
Formal Methods & Tools, University of Twente
Enschede, The Netherlands

Abstract

Interoperability across software languages is an important practical topic. In this paper, we take a deep dive into investigating and tackling the challenges involved with achieving interoperability between C and BabyCobol. The latter is a domain-specific language condensing challenges found in compiling legacy languages — borrowing directly from COBOL's data philosophy. Crossover, a compiler designed specifically to showcase the interoperability, exposes details of connecting a COBOL-like language with `PICTURE` clauses and re-entrant procedures, to C with primitive types and `struct` composites. Crossover features a C library for overcoming the differences between the data representations native to the respective languages. We illustrate the design process of Crossover and demonstrate its usage to provide a strategy to achieve interoperability between legacy and modern languages. The described process is aimed to be a blueprint for achievable interoperability between full-fledged COBOL and modern C-like programming languages.

CCS Concepts: • **Software and its engineering → Interoperability**; *Maintaining software.*

Keywords: legacy languages, integration, compilation

ACM Reference Format:
Mart van Assen, Manzi Aimé Ntagengerwa, Ömer Faruk Sayilir, and Vadim Zaytsev. 2023. Crossover: Towards Compiler-Enabled COBOL-C Interoperability. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23), October 22–23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3624007.3624055

GPCE '23, October 22–23, 2023, Cascais, Portugal

1 Introduction

Many organisations from the public and private sectors rely on legacy software systems written in the last century for their critical day-to-day operations: 43% of all banking systems are built on COBOL [11], 44 out of top 50 banks [24] and 92 out of top 100 [4] use mainframes. 71% of Fortune 500 companies are relying on legacy systems [4]. The most prominent legacy languages in the 2020s are COBOL (42%), HLASM (37%), PL/I (22%) and various Fourth Generation Languages, or 4GLs (22%–32%) [16]. Among mainframe-using companies, 75% are concerned about having access to the right IT talent to maintain and manage their mainframes [16].

A major part of the challenge of understanding, maintaining and renovating software written in legacy languages, is related to the different data philosophy adapted by them [2, 38, 39]. Mainframe languages normalise mixing the form and the representation, leading developers into writing code that relies on knowing precise byte sizes of data and operating with ad hoc data structures with constraints varying per byte and reuse provided by lexically reimporting data definitions. Modern languages tend towards separation of concerns, leading developers to define reusable types encapsulating and hiding exact implementation details.

In this paper, we introduce Crossover: a compiler that bridges the gap between COBOL's and C's data philosophies. Since real legacy programming languages such as COBOL, FORTRAN or PL/I are too large to implement in a similar way as a proof-of-concept [22], Crossover showcases our strategy for language interoperability on BabyCobol [45] (see Section 2.1). This programming language is meant to be quickly implementable and yet still offer many of the challenges that arise from processing legacy languages, the same way Featherweight Java is used for experimental features in language semantics instead of Java.

Crossover allows for the calling of procedures between BabyCobol and C. Since these languages have vastly different ways of representing data, marshalling of parameters and return values is necessary. To account for this difference in data representation, we designed the "BabyCobol

standard library" (BSTD); a shared C library used by the compiler for data representation, marshalling and manipulation during runtime. By also making this library available to the programmer we bring some BabyCobol semantics to C.

To illustrate the impact of the compiler, the following code snippet shows how BabyCobol could use a function call to a C program to calculate the value of the seventh Fibonacci number. First, the variables `N` and `RESULT` are defined in the `DATA DIVISION` (the part of the COBOL program containing data definitions). In the `PROCEDURE DIVISION` (the code-containing part), the value of N gets set to 7. Then the `CALL` statement calls the function "fib" of a C program called "fibonacci" and passes `N` as an argument to indicate the N-th Fibonacci number must be calculated. When reentering BabyCobol, the return value of the C function is parsed into the `RESULT` variable. The BabyCobol program then displays the result and terminates.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FIB.
DATA DIVISION.
01 WORKING-STORAGE-AREA.
  02 N PICTURE IS 99.
  02 RESULT PICTURE IS 9999999999.
PROCEDURE DIVISION.
  MOVE 7 TO N.
  DISPLAY "Calculating the " N "-th Fibonacci
    number: "
  CALL fib OF fibonacci
      USING BY VALUE N AS PRIMITIVE
      RETURNING RESULT.
  DISPLAY RESULT.
  STOP.
```

Listing 1. Calculating the 7th Fibonacci number by calling a C function

Crossover is created using the readily available compiler construction tools ANTLR [28] (a parser generator) and the LLVM Core libraries [21]. At runtime, it also requires the Clang compiler to be installed on the system. Complete engineering details about dependencies and installation requirements can be found in the project repository [33]. Crossover currently implements a subset of the BabyCobol language [45]. Its novel features include:

- The instantiation and modification of BabyCobol data types in C code.
- The invocation of C functions from a BabyCobol codebase, and the invocation of BabyCobol paragraphs from C.
- The automatic generation of C composite structs from BabyCobol data definitions.

Systems similar to Crossover have been created before, as we will see in the next section. However, all of them are tightly integrated parts of proprietary frameworks. Crossover is meant to be open source, it is focused primarily on solving the interoperability problem, and it has a palatable size, suitable for use as an example both in an industrial context of developing similar bridges between languages, as well as for code reading in software evolution education. Furthermore, Crossover is designed to support programmers by abstracting away the dichotomy of data representation between BabyCobol and C.

The paper is structured as follows: Section 2 dives into the background and related work. In Section 3 we formulate the problem statement and pose a number of research questions, to which Section 4 proposes a solution. In Section 5 we describe the evaluation of the proposed solution. Finally, Section 7 discusses the strategy we used, concludes the paper and proposes future work directions.

2 Background

We focus on COBOL as our subject for investigating a legacy data philosophy, since it is one of the largest and by far the most used legacy programming language nowadays [16]. For practical reasons, we substitute COBOL with BabyCobol [45] which is significantly smaller yet following an identical data philosophy.

The choice for modern data philosophy falls on C [17]. The motivation is based on C being an immensely popular language itself, as well as a solid representative for the class of C-like languages which are prevalent in the software engineering practise today.

2.1 BabyCobol

BabyCobol, also known as "the software language engineers' worst nightmare" [45], is an experimental language specifically designed with an intention to highlight challenges of implementing legacy languages. It has a tiny size (1 data type and 18 statements), paling in comparison to real legacy languages like COBOL (43 statements and 87 functions, most statements being split into up to 8 variants) [14] or IBM's mainframe assembly language HLASM (3296 directives, instructions and mnemonics) [3, 44], which makes its implementations compact and focused. Yet, it is uncomfortable and perpetually challenging: for example, it combines Fortran-style computable `GO TO`s (where the name of the target label is a result of a runtime computation) with the COBOL ability to `ALTER` them (reassign an existing control flow transferring statement to another target at runtime), REXX-like error handling with `SIGNAL` (comparable to aspect weaving [19]) and non-reserved keywords inspired by PL/I (where variables are allowed to have names identical to keywords).

BabyCobol was meant as a playground for experimenting with various techniques of compilation, software analysis and transformation, both for educational purposes and in the research context. Thus, instead of investing years [22] into building a parser for some extensive legacy language, and then a compiler covering all variations and subclauses of

one of its dialects, we have built a minimal implementation of the language core and extended it with one new `CALL` statement (for calling external programs written in C or BabyCobol) and one subclause of the `PROCEDURE DIVISION` (for specifying return values of the program).

BabyCobol's data philosophy is a strict subset of that of COBOL. The language offers only one primitive type of data: the `PICTURE`. A field (variable) refers to a single piece of information of a fixed size. Fields are defined by a template, determining how the system interprets the data stored in them. For example, a field can be defined with the template `"XX99"`, which specifies that it is four bytes long and may contain two alpha-numeric characters followed by two decimal digits. The exact semantics will be explored in Section 2.5.

Similarly to COBOL, each BabyCobol program consists of divisions. In BabyCobol there are three divisions: the `IDENTIFICATION DIVISION`, `DATA DIVISION` and the `PROCEDURE DIVISION`.

The `IDENTIFICATION DIVISION` is mandatory for each BabyCobol program and contains identifying information of the file. This division is made up of name-value pairs and may contain clauses like `PROGRAM-ID`, `AUTHOR`, `DATE-WRITTEN` and `DATE-COMPILED`, which are typically found in real COBOL programs. In BabyCobol, programmers are not limited by predefined clauses and can extend this division with their own keys. An `IDENTIFICATION DIVISION` is always the first division in a (Baby)Cobol program.

All explicitly defined variables in a BabyCobol program are declared in the `DATA DIVISION`. This division is optional: in case of is absence, the program is restricted to the use of constants and implicitly typed variables. If present, the `DATA DIVISION` is found after the `IDENTIFICATION DIVISION` and above the `PROCEDURE DIVISION`. Data structures in BabyCobol are hierarchical, such that fields can be declared at the top-level or as part of a record. A record is a composite data structure which may contain fields and/or other records. Records and fields can also be turned into arrays with the `OCCURS` clause and a number representing the desired fixed length. In COBOL, `DATA DIVISION`s are split into sections, but this is abstracted from in BabyCobol.

The `PROCEDURE DIVISION` follows the other divisions, and is the last division in a BabyCobol program. It consists of paragraphs, which are comparable to functions in C. However, unlike C, BabyCobol executes statements sequentially from the start of the division, fall through from the end of one paragraph directly to the start of the next paragraph below it until the end of the file or a `STOP` statement is encountered.

In Table 1 we align the terminology to describe elements and concepts of both BabyCobol and C.

2.2 Foreign function interface

Foreign function interfaces (FFIs) are mechanisms that allow one *host* language to invoke procedures written in another *guest* language, using the call semantics of the host language. An FFI bridges the differences in calling conventions and semantics between the two languages. Many modern languages and industrially strong compilers provide FFIs (we will give some examples in Section 6), and there exist libraries like `libffi` [9] which facilitate developers in making their own FFI. Since FFIs make it possible to seamlessly integrate differently typed code, their generalisation can be classified as a form of gradual typing [5, 37]. Practical implementations are often substantially simpler since legacy languages offer only very basic non-parametric forms of polymorphism, if any at all.

2.3 Application Binary Interface

An Application Binary Interface (ABI) is a specification which defines object and executable file formats, as well as calling conventions between applications and the platform on which they run. By specifying how compilation units should be linked together, ABIs play a crucial role in FFIs. The source files for the host and guest language are often processed by different compilers. Linking together these compilation units requires them to adhere to the same ABI. Additionally, at runtime the parameters and return values of procedures must follow the same memory/register layout defined in the ABI. The data types of the passed parameters are not defined in the ABI, and bridging them requires specific steps in the case of BabyCobol and C. We discuss this further in Section 2.5, and expand upon this in Section 4.2.

The Crossover compiler targets the UNIX System V ABI specification [43], which came about in 1998 and went through several minor backward-compatible changes until 2013. It consists of two parts; a generic part (gABI), which is the same for all implementations of System V, and a processor-specific supplement (psABI). The System V ABI comprises the Executable and Linking Format (ELF) [36], which defines the format of object files and how these are linked together to form executable files.

Because ABIs work at a very low level of abstraction, and there exist many different computing platforms [12], writing an application for any specific ABI greatly limits its portability. However, since Crossover is an LLVM-based compiler, it does have a certain degree of portability by construction. Crossover is designed to link ELF objects in its compilation pipeline. Because the ELF format is part of the generic System V ABI (gABI), and LLVM has backends for multiple processor architectures (psABIs), the compiler can target multiple System V-derived platforms (notably modern Linux).

2.4 Standard Libraries and Language Runtimes

A *standard library* [10] can contain language functionality not directly built into the language as a native construct. Functionality from standard libraries can either be expanded into the source code as a macro, or they can be compiled in their entirety and linked with the program binary.

Table 1. Terminology across language domains

Term	BabyCobol	C
variable	field or record	any non-constant of a basic data type or struct
string	a field containing alphabetic or alphanumeric data	a C-style string (`char*`)
number	a field containing numeric data (with a `PICTURE` clause containing only {9,S,Z,V})	`char`, `int`, `long`, `float`, `double` (any modifications)
procedure	a paragraph	a function

A *language runtime* [42] is a compiler component that needs to be present during the execution of a compiled program written in the language for the executable to function properly. A language runtime could take the form of one or more standard libraries or a virtual machine.

The BabyCobol standard library (Section 4.2) is a central component of our solution and is a hybrid between a standard library and a language runtime.

2.5 BabyCobol vs. C

There are many differences between BabyCobol and C, since they represent different schools of thought and directions of programming language evolution. But despite obvious syntactical differences, there are some similarities to be found. For example, both are compiled imperative languages. They are procedural, thus allowing for the definition of reusable paragraph or function blocks. In BabyCobol, these procedures may have a certain degree of isolation from the main scope of the program, up to being solely dependent on the procedure parameters passed to it by a caller. Similarly, C functions may be solely dependent on their function parameters.

There is no functional purity in either language, but besides minor side effects on the global state, we can identify a class of *isolated* procedures. Such isolated procedures may still have side effects in a system through file or database access, but from a functional perspective they are idempotent.

These characteristics offer a good starting point in developing interoperability between the two languages. All parameters we send over an FFI when invoking a procedure must be available on the other side. If the invoked procedure returns a result, that must be sent back over the FFI and be available on the invoking side.

In BabyCobol and C, we have different ways of representing data. We refer to the representation of data, and the constraints on that representation, as its data type. BabyCobol only has one inclusive data type constrained by a `PICTURE` clause. Essentially it gives a pattern to which the possible values have to conform, where each symbol mandates the contents of one letter or digit. Patterns are constructed freely (with some minor variations of what combinations trigger a warning or an error, which varies wildly across COBOL dialects — BabyCobol allows any combination to simplify this issue), but there are some that are more useful and thus more commonly used. For example, `S99` means a signed two-digit numeric value, `999V99` defines a fixed point decimal with five digits, two of which define the hundredths and the others the integer part, and `XXXXX` allows any five-character string.

COBOL has some more complex rules for representing enumerations and redefining the same memory area with alternative representations, but BabyCobol only has two kinds of data entries: a primitive one (a *field* defined with a `PICTURE` clause as explained above) and a composite one (a *record* that combines several fields or other records). Either of those can be *occurring*, essentially turning them into arrays. In the absence of the `REDEFINES` and `FILLER` clauses, all data structures are fairly straightforward and can be represented by a tree structure.

2.5.1 Fields and Basic Data Types. C is similar to (Baby)COBOL in the sense of having byte representation aware types, but it comes with predefined basic data types such as *int* which directly represent data in memory, and has `structs` which aggregate such basic data types and other `structs`. Unlike BabyCobol, `structs` in C are named and reusable by those names. The most significant dissonance in data representation may occur at the level of fields in BabyCobol and basic data types in C. In BabyCobol, fields are position-based, which used to align better with its business-oriented nature and punchcard-oriented implementation. For instance, we may define a field to consist of three decimal digits. This allows for an intuitive description of what data a field holds. The programmer does not need to be concerned with the bit-width of their data type, nor with the intricate concepts of floating-point numbers and rounding errors. BabyCobol's fixed-point numbers are incompatible with C's floating-point data types.

In C, we find that each basic data type represents only a single type of data. In contrast with BabyCobol, C does not let the programmer define a type per character. Instead, its basic data types describe a region in memory with a certain bit-width. Where BabyCobol can distinguish several data types in one Field by specifying the type of each individual character, C's basic data types treat the specified region in

memory as one homogeneous data type. There are some obvious "easy" cases such as `S99` from the example above that are representable as `signed char` in C (with some additional machinery for decimal overflows), but the general case is complicated and has no universal performant solution.

2.5.2 Records and Structs. Looking at composite data types, BabyCobol records have much in common with C structs. They can define a type by the composition of other types. This composition can, in BabyCobol, always be modelled as a tree, where a field is always a leaf vertex. In C, a struct may contain a reference to any declared type, even its own type. Modelling structs as graphs, we can form trees in C, much like in BabyCobol, but we may also create graphs containing loops. Therefore, we can not model all C structs as BabyCobol records. For interoperability to work, we must find a set of composite data structures that work in both languages. For BabyCobol, the set of possible composite data structures S_b is restricted by $\forall d \in S_b.isTree(d)$. The set of possible composite data structures in C S_c is only "restricted" to graphs: $\forall d \in S_c.isGraph(d)$. It follows that $S_b \subset S_c$. Therefore, we can achieve interoperability for a parameter p over our FFI iff $p \in S_b$:

A data structure is valid under the FFI if and only if it can be modelled as a tree.

All BabyCobol records are valid under this assumption, but not all C structs are.

One thing that should be noted is that BabyCobol records do not define a named type but a variable of a certain anonymous structure. In this sense, they are similar to unnamed structures in C. The type of a record can however be used to define another variable of the same structure through `LIKE` clauses. One crucial role of data types in strongly-typed languages such as BabyCobol and C is for the compiler to determine what operations are allowed on certain data. For example, in COBOL and BabyCobol, it is not allowed to *add* two strings together (see Section 4.5). A C compiler may provide warnings or fail on specific implicit (automatic) type conversions.

3 Problem Statement

From the related work and background we identify certain challenges which must be overcome to achieve interoperability between BabyCobol and C.

Functions need to be linked across compilation units. The challenge is matching symbols and implementing call semantics at the ABI level.

BabyCobol's `PERFORM` statement does not have the linguistic means to invoke procedures in other compilation units. It lacks the possibility of specifying arguments and return values. Furthermore, the BabyCobol language must be extended in order to support the different parameter forms that are possible in C function definitions.

Data type constraints and data integrity between BabyCobol and C must be aligned and respected across the boundaries of the FFI. This requires the marshalling of variables when crossing the bridge between the two languages.

To achieve interoperability, there must be mechanisms provided for the reliable manipulation and evaluation of data beyond the language boundary, consistent with unified semantics.

We pose the following research questions:

1. ***How can a Foreign Function Interface between BabyCobol and C be implemented?***
2. ***How can the key differences in data philosophy between BabyCobol and C be addressed?***

4 Design and Implementation

Crossover is a BabyCobol implementation that is designed specifically with interoperability between BabyCobol and C in mind. Alongside BabyCobol source files, users can provide compiled C object files to the compiler, providing the procedures they wish to invoke in BabyCobol.

As mentioned, C and BabyCobol vary greatly in how they handle data. Crossover's BSTD runtime library helps to bridge the gap between the two languages. The library contains a C implementation of the native BabyCobol data types and has the functionality to create, convert to and -from, and modify variables of these data types. The BSTD library plays a central role in this implementation: it is used not only for achieving the interface with C but is also extensively used for compiling pure BabyCobol programs. The compiler uses the interface with C to generate calls to the library whenever data needs to be created or modified, making this library effectively the language runtime. The design of the BSTD is discussed further in Section 4.2.

Besides the BSTD, the compiler also provides the option to automatically generate C headers from BabyCobol identification divisions. These files allow the user to synchronise data structures between the two languages.

The implementation relies heavily on two existing compiler construction tools: ANTLR [28] and LLVM [21]. ANTLR is used for generating a parser and a parse tree visitor from the provided BabyCobol grammar definition, and LLVM is used as a backend for code generation. The compiler is written in C++. This language was chosen due to it being the native language of the LLVM compiler backend, ANTLR allowing for the generation of parsers in this language, and it being interoperable with C code, allowing us to incorporate parts of the BSTD library into the compiler.

4.1 Toolchain overview

Figure 1 visualises the compilation of a mixed BabyCobol/C codebase using Crossover. During the compilation of a BabyCobol program, a parse tree is generated by ANTLR. When visiting a node of that parse tree, LLVM is used to

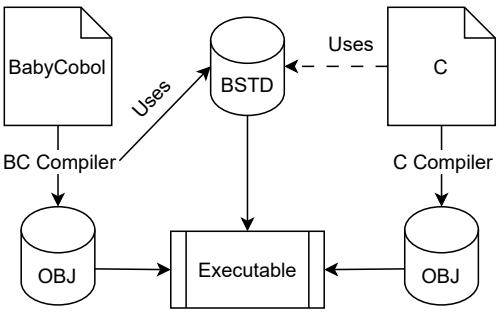

Figure 1. Crossover Compilation Overview

generate the appropriate instructions for that node. We leverage the fact that calls to external procedures get resolved during linking to create the FFI. This approach is also used to generate calls to the BSTD library for the runtime creation, conversion and modification of BabyCobol variables. This eliminates the need of having to implement this functionality again on a compiler level and guarantees that operations on BSTD data types from C code behave exactly the same as they do in native BabyCobol. When the compilation of the BabyCobol sources is done, the compiler invokes `clang` to link the newly created object files, the BSTD runtime library, and the C object files into an executable.

4.2 BabyCobol Standard Library: BSTD

The BSTD is the core of our approach to interoperability. Its key responsibility is resolving the differences between fields in BabyCobol and basic data types in C. It can also guarantee data integrity by allowing for the use of the same operation semantics on its data types in both BabyCobol and C (see Section 4.5). BabyCobol has only one data type — the *Field*. The BSTD contains two data types: the `Number`, defined in Table 2, and the `Picture`, shown in Table 3. The reason for this distinction is that certain operations are exclusive to BabyCobol Fields which represent numbers. Specialising a data type for `Numbers` and *everything else* allows for type checking at compile time whether an operation may be performed on the operands' data types. It also allows for a more specialised representation of data, simplifying its modification.

The fields in Table 2 and 3 are not packed. The C language specification [17] dictates that the size of `Numbers` and `Pictures` must both be 24 bytes due to inserted padding. This padding is inserted at the end of the structs and does not affect the offsets shown in Table 2 and Table 3.

The values stored in the BSTD data types create a complete representation of their `PICTURE` clause specification in BabyCobol.

The BSTD `Picture` contains an array of `bytes`, a `mask` character array and a `length`. The `length` property describes the dimensions of the mask- and byte array. For any `Picture` p with byte array B_p, mask M_p and length l, the lengths of these arrays are equal: $l = |B_p| = |M_p|$.

Masks are character based, and the i^{th} mask character applies to the i^{th} byte. A masking function $mask(byte, mask)$ creates an interpretation of a byte given its corresponding mask character, such that for a `Picture` p value $v_p^i = mask(B_p^i, M_p^i)$. For example, $mask(0, 9) = 0$ and $mask(65, X) = A$. Truncating may occur for values outside the permitted range for a mask character; $mask(65, 9) = 5$.

The BSTD `Picture` data type can be converted to a C-style string by iterating over each byte-mask character pair and mapping it to a character using the *mask* function.

The BSTD `Number` data type stores a base `value`, a `scale` and the `isSigned` property. Additionally, it stores the permitted `length` of the `Number` and the `positive` property. The data type can represent integer and fixed-point decimal values. Converting these data types to C integers or floating point numbers is done at runtime. The conversion function for any `Number` n with base value b and scale s, is $v(n) = sign(n) \cdot b \cdot 10^{-s}$, where $sign : \texttt{Number} \to \{-1, 1\}$.

Conversely, C basic data types can not trivially be converted to BSTD `Numbers` because they lack the constraining properties inherent to the `Number` type. Through this lack of information, we are restricted to the *assignment* of C basic data types to instances of BSTD `Numbers` which contain the otherwise missing constraints. This forces the programmer to be explicit in defining these constraints.

C integers are assigned to `Numbers` using the following formula: $b = \left(|k| - \left\lfloor \frac{|k|}{10^{l-s}} \right\rfloor \cdot 10^{l-s}\right) \cdot 10^{s}$, where an integer k is assigned to a `Number` with the base value of b, length l, and scale s. Additionally, we set the `positive` property of the `Number` such that $sign(n) = -1 \iff isSigned(n) \wedge k < 0$. Note that assigning a C data type does not affect any of the other properties of the `Number`.

Table 2. The `Number` Data Structure

Offset	Size	Field	Description
0x00	8 bytes	value	The base value of the `Number`.
0x08	8 bytes	scale	The scale of the `Number`.
0x10	1 byte	length	The length of the `Number`.
0x11	1 byte	isSigned	Set if the `Number` is signed, unset if not.
0x12	1 byte	isPositive	Set if the `Number` is positive. -0 and +0 are treated the same.

BSTD `Numbers` are representations of integer or fixed-point decimal numbers. They are equivalent to BabyCobol Fields with a `PICTURE` clause consisting of the characters 9, S, Z and V. With the set of characters $N = \{9, S, Z, V\}$, for any mask M, $isNumberMask(M) \iff \forall c \in M.[c \in N]$.

Table 3. The `Picture` Data Structure

Offset	Size	Field	Description
0x00	8 bytes	bytes	A pointer to the bytes array belonging to this `Picture`.
0x08	8 bytes	mask	A pointer to the `Picture` clause specification string.
0x10	1 byte	length	The length of the `Picture`.

Because `Number` data structures are distinct types from `Picture` data structures, we do not need to specify a mask when we construct one — all mask characters are in N and are represented in the `Number` data structure and their semantics are fully encapsulated in the struct.

Similarly, the Crossover compiler will represent any BabyCobol field with a `PICTURE` clause containing non-number mask characters by a `Picture` data type. For any field with mask M, $isPictureMask(M) \iff \exists c \in M.[c \notin N] \iff \neg isNumberMask(M)$. Circling back, `Numbers` are numbers, and `Pictures` are *everything else*.

4.3 Linking

We compile both C and BabyCobol code to ELF object files. After compilation, we will have lost many details about the original implementation language of the particular object file. Both Crossover and `clang` do not do name mangling and list functions as symbols by their name. We link these object files without any knowledge of their implementation languages, and thus we can use `clang` to opaquely create an executable binary including originally both C and BabyCobol functions. This step can be extended to any language which compiles to an ELF object file (see Section 4.6).

During the compilation of BabyCobol source files the compiler checks if any internally unknown CALL target procedure exists in the externally compiled object files. This is done by using the `nm` tool from the GNU Binutils package [7] to read the symbols in the object files. The `nm` output is then parsed to construct a symbol table of procedure names found in the external object files.

To link C objects and BabyCobol objects, we must have compiled them both. This creates a problem: How do we get the compiler to understand data structures defined in the other language's source? It seems to be the setup for a chicken-and-egg scenario. To compile one, the other must have already been compiled. To address this, we offer two tools:

- The BSTD library makes BabyCobol Fields accessible from C. It exposes a struct-representation of Fields and utility functions that allow for converting to and from C basic data types. To a programmer using these utility functions, the BSTD guarantees that BabyCobol semantics apply to all operations performed on such data types. More on this in Section 4.2.
- There is an integrated tool which parses BabyCobol source files and extracts their defined data structures from their `DATA DIVISION`s. It then outputs these data structures as C `structs` in a header file. This header file can be included in C source code and used like any other `struct`. We can then build and compile against the BabyCobol source code using these data types.

4.4 Language extensions

We introduce four optional modifiers as extensions to the BabyCobol specification, which allow for more fine-grained control over the way in which parameters are passed over the FFI:

- `BY VALUE` specifies that the parameter must be copied before being passed over the FFI. Changes made to the copy do not change the original data structure.
- `BY REFERENCE` specifies that the parameter must be passed over the FFI as a reference to that data structure. The invoked function has direct access to the parameter, and changes on the other side of the FFI are reflected in the data structure.
- `AS PRIMITIVE` specifies that the parameter or return value must be passed as a C basic data type. We distinguish three basic data types: the integer, the double (floating point), and the character pointer (C-style string). This modifier may only be applied to Fields.
- `AS STRUCT` specifies that the parameter or return value must be passed as a C struct. This is a predefined BSTD data type for Fields and a generated C struct for Records.

The programmer can choose to specify one of the leading modifiers (either `BY VALUE` or `BY REFERENCE`) and/or one of the trailing modifiers (either `AS PRIMITIVE` or `AS STRUCT`), as seen on Figure 2. The `BY VALUE` and `BY REFERENCE` modifiers are applied to all following parameters until a new modifier is encountered. Conversely, the `AS PRIMITIVE` and `AS STRUCT` modifiers are applied to all preceding parameters until a new modifier is encountered. If no optional modifiers are present the implicit behaviour for Fields is `BY VALUE` and `AS PRIMITIVE`, respectively. For Records, these defaults are `BY VALUE` and `AS STRUCT`, since being an composite data structure, records cannot be mapped to any C basic data type.

This BabyCobol language extension allows a programmer to create calls compatible with almost all possible function definitions in C. `BY VALUE` results in a non-pointer parameter, whereas `BY REFERENCE` results in a pointer parameter. Similarly, `AS PRIMITIVE` results in a C basic data type and `AS STRUCT` in a BSTD data type or `struct`. The compiler also ensures the correct marshalling during invocation and

re-entry as shown in Figure 3. At the start of a `CALL` the BabyCobol data is marshalled to the appropriate format for the function based on the parameter modifiers. Upon re-entry, the parameters are marshalled again, as well as the return value. Parameters passed `BY REFERENCE` can be modified on the C side, and variables defined in the BabyCobol `DATA DIVISION` must be updated to reflect their new values. In the case of a `BY REFERENCE` and `AS PRIMITIVE` clause, the marshalled parameter values need to be inversely marshalled and assigned to their original field. Similarly, return values may need to be marshalled to be assigned to their target Field or Record as defined in the `DATA DIVISION`.

4.5 Data Type Constraints and Data Integrity

Data defined using Crossover obeys the type constraints that are defined in the BabyCobol specification [45]. This means that the format of each variable has to abide by its description defined in the data division of a BabyCobol file. For instance, if a user tries to `MOVE` the value `42` to a variable defined as `9V99`, the value of the variable will be set to `2.00`. Likewise, when a variable is defined with the `PICTURE` clause `AX` and the user attempts to `MOVE` the string `"HI!"` to the variable, it will be set to `"HI"` (Note the missing exclamation mark).

While BabyCobol does not specify a limit on the number of characters of a `PICTURE` clause, our implementation currently limits the length of a `Number` to nine digits, and the length of a `Picture` to 255. The length of `Numbers` is limited because our implementation uses 64-bit integers to store the properties of the `Number` struct. The length of a `Picture` is also limited due to the usage of an 8-bit integer to store the length property.

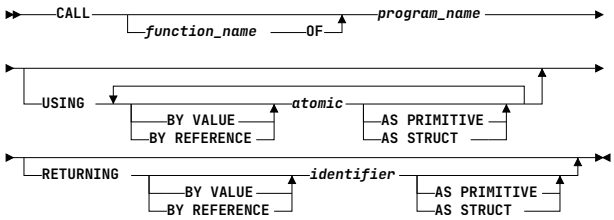

Figure 2. The railroad diagram of the `CALL` statement, including extensions

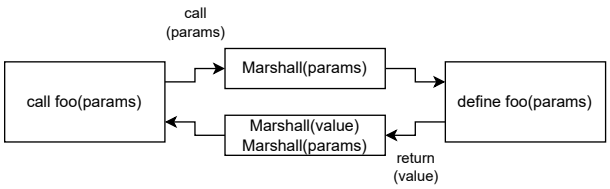

Figure 3. Marshalling When Invoking a Foreign Function

Because the BSTD is used by Crossover itself, using this library in C guarantees that the implementation of operations on `Numbers` and `Pictures` is one and the same in either environment. This provides exact BabyCobol data semantics to C, allowing for safe manipulation of BabyCobol variables within C programs.

The Crossover compiler will adhere to the invariant $\forall z \in F.[isNumber(z) \iff isNumberMask(M_z)]$, where F is the set of all possible Fields in BabyCobol, and M_z is the mask of Field z. This means that any Field with a numeric `PICTURE` clause will be instantiated as a `Number` data type.

Outside the Crossover compiler, the BSTD *does* allow for an instance of the `Picture` data structure to be created with a mask M such that $isNumberMask(M)$. This does not have a negative impact on data integrity, as the type system will prevent `Pictures` from being used where a `Number` is expected. Because some operations are exclusive to `Numbers`, `Picture` instances with a mask M such that $isNumberMask(M)$ cannot be used as parameters to arithmetic functions because of a mismatch in type.

On re-entry Crossover checks if the passed `Picture` still has the same `length` and `mask`, as this can never be changed. Similarly, the `length`, `scale` and `isSigned` properties of a `Number` should never be changed and are checked on re-entry.

4.6 C as a Bridge

The C language is a particularly interesting candidate for a language to interface with because many other languages also provide an interface with it. One could use C as an intermediate language to invoke functionality written in a third language using a wrapper and calling that wrapper over the FFI that Crossover provides. This process is illustrated in Figure 4.

Considering the two data types the BSTD provides, the FFI can be extended to any language that can interpret the data structures in Table 2 and 3, and that has a compiler that can create ELF object files with non-mangled function names as linkable symbols.

5 Evaluation

In this section, we present the evaluation of our proposed solution, which consists of three parts: unit testing of the BSTD library, performance analysis of the FFI and the demonstration of running examples showcasing how the interoperability of Crossover can be used.

5.1 Testing

To evaluate the stability of our solution, we thoroughly tested the BSTD library, the central piece of our architecture. This was done by writing unit tests using the techniques of equivalence class partitioning and boundary value analysis [15].

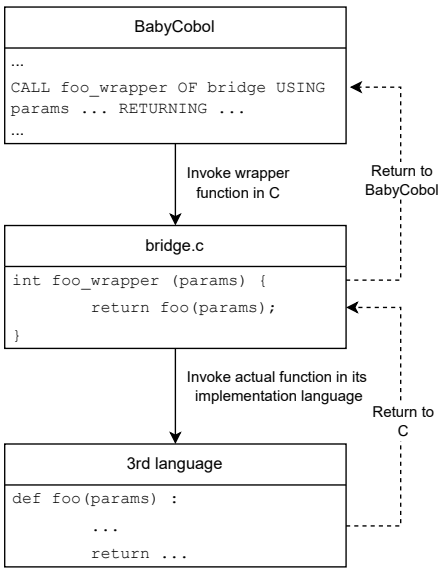

Figure 4. Using C as a Bridge

Equivalence class partitioning is a method of testing where the inputs of the tested functions are divided into valid and invalid equivalence classes. Under the assumption that each member of an equivalence class would be processed in the same way by the function, one would only need to write one test per equivalence class. Boundary value analysis expands upon the idea of equivalence class based testing by introducing test cases directly on, above or below the boundaries of equivalence classes. The rationale for these additional test cases is that bugs usually exist on and around the boundaries of the equivalence classes.

Table 4 illustrates how equivalence class partitioning was performed for the function `bstd_number_is_integer`. This function was tested by creating cases and creating inputs combining as many valid equivalence classes as possible. After these cases, additional cases are added that test the boundary values. Since behaviour for values outside the valid bounds is undefined, no test cases exist. Only behaviour within valid bounds is tested.

These methodologies helped us to validate the BSTD library and find and fix bugs in our implementation. Some examples of bugs found during our testing include:

- Mismatched types, for example, when a large `int64_t` value is used as an input for a function that takes a 32-bit `int` causing the `int` to overflow.
- Insufficient floating-point value precision; while the default six decimal digits of precision of C is sufficient to pass most of our BabyCobol `Number` type to C string tests, it proved insufficient once we tested the boundary of the number of decimals a BabyCobol `Number` could have.

Table 4. Equivalence class partitioning for function bstd_number_is_integer

Condition	Valid	Invalid
Value of value	$0 \leq$ value ≤ 999999999	value < 0, value > 999999999
Value of scale	$0 \leq$ scale ≤ 9	scale < 0, scale > 9
Value of length	$1 \leq$ length ≤ 9	length < 1, length > 9
Value of isSigned	isSigned = true, isSigned = false	
Value of positive	positive = true, positive = false	
Relation	isSigned = true $\wedge$ positive = false, isSigned = false $\wedge$ positive = true, isSigned = true $\wedge$ positive = true	isSigned = false $\wedge$ positive = false

Currently, our test suite consists of 261 tests, covering 87.8% (200/228) of the lines and 91.7% (22/24) of the functions in the BSTD library. The gaps in the coverage are internal auxiliary functions that are not exposed for use outside of the BSTD library.

5.2 Performance Evaluation

In this subsection we evaluate the overhead created by the different permutations of argument modifiers in the `CALL` statement. We created a set of four test programs — one for each possible permutation. Each program runs a loop of 100 million iterations, calling a C function with an empty body (to maximise the overhead) with a single argument under one set of modifiers. We took measurements on the execution time, the results of which are plotted in Figure 5. To obtain a statistical mean, each program was run back-to-back 100 times.

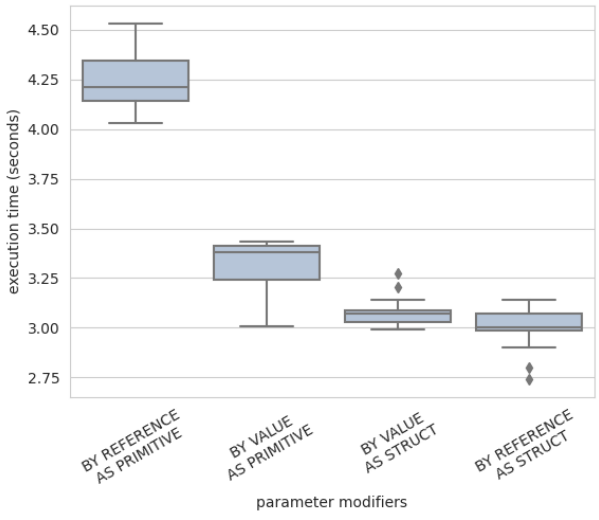

Figure 5. Result of the FFI Performance Analysis

Figure 5 shows that the combination of modifiers {BY REFERENCE, AS PRIMITIVE} takes the longest time to execute. This is in line with our expectations, and can be explained by the fact that when these modifiers are used, the compiler inserts code for marshalling before the reference of the argument is passed to the C function. During the re-entry to BabyCobol the inverse of this happens: the C primitive is marshalled back into the original BabyCobol variable. This ensures that changes to the marshalled argument on the C side are reflected in the original variable. The insertion of marshalling and unmarshalling code has a performance overhead of **40%** as compared to the fastest running variant ({BY REFERENCE, AS STRUCT}).

With the combination {BY VALUE, AS PRIMITIVE}, the compiler also inserts code to marshall the argument to a C primitive. However, with the BY VALUE modifier, changes to the argument should (and can) not be reflected upon re-entry. Inserting marshalling code only once implies a lesser performance overhead of **10%**.

The cases {BY VALUE, AS STRUCT} and {BY REFERENCE, AS STRUCT} are both faster than the previous two cases because no marshalling is involved. The parameter is passed to the C function as a copy of or reference to a BSTD structure respectively, which is its internal representation in Crossover. There is a slight performance penalty to passing value copies, it being **2%** percent slower than passing references.

5.3 Demonstration

This subsection features three examples that demonstrate how the new CALL statement can be used for BabyCobol-C interoperability.

5.3.1 Calculating a square root. The first demonstration is a simple program that calculates the square root of a fixed-point number. The BabyCobol code is shown in Listing 2. The program defines the variables *X* and *Y* as fixed-point numeric values of two integer digits, and one decimal digit. In the PROCEDURE DIVISION, *X* is first assigned the value 12. Then, the C standard library function *sqrt* is called, passing *X* as the argument. No leading modifier is specified, so the default BY VALUE semantics apply. The AS PRIMITIVE modifier dictates that the argument is to be marshalled into its C basic data type counterpart. The *sqrt* function in C accepts a double-precision floating point parameter. It returns the square root of this parameter. In the CALL statement, the RETURNING clause specifies that the result of the *sqrt* function is a C basic data type which should be marshalled into the variable *Y* (respecting Y's type constraints). Lastly, the program DISPLAYs the value of *Y*, which shows *03.4*.

5.3.2 Scanner Example. This example shows how a BabyCobol program can use the FFI to accept user input. The code can be seen in Listing 3. First, a four digit numeric variable

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. "radical".
DATA DIVISION
  01 X PICTURE IS 99V9.
  01 Y PICTURE IS 99V9.
PROCEDURE DIVISION.
  MOVE 12 TO X.
  CALL sqrt USING X AS PRIMITIVE
              RETURNING Y AS PRIMITIVE.
  DISPLAY "The square root of " X " is " Y.
```

Listing 2. Calculating a Square Root

READ is defined in the DATA DIVISION. In the PROCEDURE DIVISION, the program then asks the user to enter a value of up to four digits and calls the *scanf* function from the C standard library. The *scanf* function accepts two parameters. The first is a formatting string, and the second is a character buffer. We supply the following two arguments:

1. A string literal with the modifier AS PRIMITIVE, specifying that it is marshalled and passed to *scanf* as the C equivalent of a string (char*).
2. The *READ* variable, marshalled to a char* is the buffer to which *scanf* writes. It is passed BY REFERENCE, such that changes to the parameter are reflected in the BabyCobol variable *READ*.

Upon re-entry, the contents of this buffer are aligned with the *READ* variable's type specified by its PICTURE clause. Finally, the program DISPLAYs this value back to the user.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. "scanf_demo".
DATA DIVISION
  01 READ PICTURE IS 9999.
PROCEDURE DIVISION.
  DISPLAY "Please enter a number of up to four
    digits: ".
  CALL scanf USING "%4d" AS PRIMITIVE
              BY REFERENCE READ AS PRIMITIVE.
  DISPLAY "You entered " READ.
  END.
```

Listing 3. Scanner in BabyCobol

5.3.3 Banking Application. The final example, Listing 4 (in BabyCobol) and Listing 5 (in C) illustrate a toy banking application and API where a certain amount of currency is withdrawn from a balance. The DATA DIVISION specifies the variables *BALANCE* as a signed fixed point decimal number, *AMOUNT* as an unsigned fixed-point decimal number and *SUCCESS* as a single digit number. In the PROCEDURE DIVISION, the two MOVE statements first assign values to the *BALANCE* and *AMOUNT* variables. The CALL statement invokes the *withdraw* function of the *banking_api* with arguments *BALANCE* and *AMOUNT*, both AS STRUCT. These

arguments are thus passed as BSTD number struct pointers. The *BALANCE* argument is passed `BY REFERENCE`. This specifies that changes to the argument in C code are reflected in the BabyCobol *BALANCE* variable. In Listing 5, a check is first done to ensure the *withdraw* amount is not greater than the *balance*. If it is, false is returned and no funds are withdrawn. If the balance is sufficient, the amount to withdraw is subtracted from the *balance*, and *true* is returned. Upon re-entry, the return value of the *withdraw* function is marshalled and assigned to the variable *SUCCESS*. A check is done to either inform the user that the *BALANCE* was insufficient, or that the withdrawal was successful, in which case the new *BALANCE* is displayed. Note that the BSTD functions used in C allow for data evaluation and manipulation which implement the exact BabyCobol semantics.

```
IDENTIFICATION DIVISION.
    PROGRAM-ID. "withdraw".
DATA DIVISION
    01 BALANCE PICTURE IS S99V99.
    01 AMOUNT  PICTURE IS 99V99.
    01 SUCCESS PICTURE IS 9.
PROCEDURE DIVISION.
    MOVE 42,50 TO BALANCE.
    MOVE 3,50 TO AMOUNT.

    CALL withdraw OF banking_api USING
        BY REFERENCE BALANCE
        BY VALUE AMOUNT AS STRUCT
        RETURNING SUCCESS AS PRIMITIVE.

    IF SUCCESS = 0 THEN
        DISPLAY "Your balance (" BALANCE ") is too
     low! No funds were withdrawn."
    ELSE
        DISPLAY "Withdrawal successful. Your
    balance is now " BALANCE
    END.
```

Listing 4. BabyCobol banking application

```
#include <bstd/numutils.h>
#include <bool.h>

bool withdraw(bstd_number* balance, bstd_number
    amount) {

    if (bstd_greater_than(withdraw, balance))
        return false;

    bstd_subtract(balance, amount);
    return true;
}
```

Listing 5. Banking API implementation

6 Related Work

We have previously mentioned the existence of prior work on FFIs. For example, CFFI [31] is a foreign function interface for Python that allows users to call C code. A similar FFI exists for R and allows for the use of C libraries in R code [1], and uFFI for Smalltalk, providing C foreign function interface for Pharo [29]. Some approaches are more formal, such as MiniML$^+$ [20], for which Larmuseau and Clarke define secure operational semantics for combining a subset of ML with C. Others are more practice-driven, like Clasp [34, 35], which analyses C++ code and generates a Common Lisp interface with a performant garbage collector to be used with an LLVM backend. Languages like Julia [18] and Rust [32] have built-in foreign function interfaces which commoditise Fortran, C or C++ calls. MATLAB also has an extensive family of external language interfaces [23], allowing bidirectional integration with Fortran, C, C++, Java, Python and .NET languages.

On the legacy side, both traditional vendors as well as their competitors provide comprehensive integration functionality. For instance, IBM provides PL/I InterLanguage Communication (ILC) and Java Native Interface (JNI) [13], foreign function interfaces that allow C and Java code to be called from PL/I. Micro Focus has a C and C++ FFI in their commercial ACUCOBOL-GT modernisation portfolio, and interoperability between COBOL and Java and .NET [25]. Similarly, Raincode offers interoperability with C# in their compiler [30], and Fujitsu with Visual Basic in theirs [8].

The only open source implementation of COBOL, with or without FFI, available to the general public, is GnuCobol, also known as OpenCobol [26]. Essentially it compiles COBOL to C, and that fact can be used for interoperability purposes if one is sufficiently familiar with the internals of the compiler. Its FFI functionality is built on top of another open source library called `libffi` [9]. GnuCobol is an interesting subject of study for academics, but due to its (L)GPL licensing it often cannot be considered in a commercial setting. Additionally, GnuCobol is also essentially its own dialect of COBOL, so migrating to this compiler implies code changes.

Commercial compilers with a C-COBOL interface are sometimes based on external function prototypes — besides being proprietary, they expose many compiler internals to initialise, stop, manipulate runtime data, and leave data conversion to programmers to code explicitly. Unlike them, Crossover brings COBOL semantics to C where desired, with its bespoke runtime conversion between data representations across the language bridge without loss of data validity.

Deep considerations for COBOL data semantics are scarce in the academic literature, but occasionally discussed in smaller consortia like IFIP Working Groups. For instance, Andersson [2] provides an excellently professional overview of the data philosophy of COBOL and its implications on reverse engineering endeavours, providing solution sketches

for particularly problematic features like implicitly modelled references between records, and internally unspecified fields. Data model discovery, as the process of extracting reusable conceptual data (meta)models from entangled mazes of partially duplicate `PICTURE`-based definitions, has also sometimes been described as a part of larger renovation packages like COBOL/SRE [6]. Such activities are often going way beyond interoperability and tend to include pragmatic transformations that are provably and obviously wrong in the formal sense, but "good enough" to aid modernisation efforts. For example, Ueda and Ohara from IBM proposed to merge records with similar data layout and somewhat similar field names [40].

In contrast to our FFI/ABI based solution, some take an alternative path with a CORBA-based approach, which relies on a middleware architecture for communication among different languages and platforms. On that path, examples of relevant standards and specifications could be CORBA [27, 1991–2021] or SOAP [41, 1998–2007]. One can also design solutions based on RESTful APIs or Remote Procedure Calls (RPC). A compiler-based solution like the one we proposed, might seem more complex to implement, but it eliminates the need for complex middleware layers, potentially reducing overhead and enhancing performance, while providing greater flexibility and allowing for fine-tuned control over the interaction between COBOL and C components.

7 Concluding Remarks

COBOL/C interoperability has not been satisfactorily solved in decades of language coexistence, despite industrial need, which shows the issue being at least somewhat challenging. In this paper, we have examined how interoperability between the languages BabyCobol and C could be achieved. The resulting Crossover compiler can serve as a guiding blueprint for developers implementing similar bridges in the context of real large legacy languages such as COBOL or PL/I. Its implementation is released as open source for the sake of reproducibility [33].

There were two research questions to be answered: the first one on how a foreign function interface between BabyCobol and C could be realised, and the other one about how the key data differences between BabyCobol and C could be addressed.

We address the first question by proposing a solution using LLVM as the framework for our compiler, allowing us to generate ELF object files. This is the same binary format that C code is usually compiled to on Linux systems. Having the source code of both languages compiled into ELF files allows them to be linked into one executable. Both C and our implementation of BabyCobol (supported by our language extensions) allow for external procedure calls. This allows for calling foreign functions that are defined in other compilation units, including those originating from other languages.

To address the second research question, we proposed a solution where data creation, modification and marshalling within Crossover is handled by a library that functions both as a standard library and a runtime library; the BSTD. The BSTD is used on both sides of the communication, either by developers directly or by the compiler generating appropriate glue code. On the side of C, the library introduces BabyCobol data types and semantics to the C environment and provides the developer with tools to create and modify variables of these data types, as well as functionality to convert data into instances of native C data types. On the BabyCobol side, the BSTD is implemented in such a way that Crossover makes use of the foreign function interface to construct function calls to the runtime library. The functions are then invoked at runtime whenever data is created or modified. This bridges the gap between BabyCobol fields and records, on one side, and C basic data types and `structs`, on the other.

Crossover is a minimal implementation of BabyCobol, which mainly implements features that directly impact interoperability, such as data definition, manipulation, and function invocation. By limiting the scope of the implementation, we could focus on the challenges of implementing interoperability instead of the rest of the language, accelerating the development of this solution.

The strategy described in this paper, is meant for interoperability between BabyCobol and C. The concepts of BabyCobol which are important for interoperability(such as the data definitions in the `DATA DIVISION` and data exchange in the `CALL` statement), are a strict subset and simplification of their counterparts in COBOL. Hence, the strategy proposed in this paper could be applied similarly to achieve interoperability between COBOL and C. However, since COBOL has more features and higher complexity than BabyCobol, there may be issues that have not arisen in our project thus far that could come up when applying the interoperability strategy between COBOL and C. From our prior experience with COBOL, we foresee some minor additional implementation challenges dealing with `REDEFINES` and `FILLER` clauses and their interaction with the `INITIALIZE` statement. These challenges, albeit non-trivial, are purely engineering in nature, and no further research challenges should arise. For instance, when `REDEFINES` gives two alternative representations of the same memory fragment, C can handle it with several `structs` and manipulating pointers to them.

Finally, an interesting research direction would be to apply the approach introduced in this paper to implement interoperability with other languages that compile to ELF object files. This would allow users to incorporate their code written in these languages without using C as a bridge as described in Section 4.6. The major challenge in undertaking this would be to write or generate a BSTD-like library for the target

language to allow for marshalling and manipulating data at runtime.

Designing Crossover required much careful consideration, data semantics alignment, and ample experimentation. We hope that opening its design contributes to the field of software maintenance by exposing the general public to the problem instead of keeping it in the ever shrinking legacy developer pool.

Acknowledgements

The authors express their gratitude to the GPCE programme committee members for their efforts and expertise in reviewing this document. We are especially grateful to Julia Lawall for shepherding the last steps. We also appreciate advice, encouragement and interest in this project from Raincode engineers Johan Fabry and Darius Blasband.

References

[1] Daniel Adler. 2012. Foreign Library Interface. *The R Journal* 4, 1 (2012), 30. https://doi.org/10.32614/rj-2012-004

[2] Martin Andersson. 1998. Searching for Semantics in COBOL Legacy Applications. In *Data Mining and Reverse Engineering: Searching for semantics. IFIP TC2 WG2.6 IFIP Seventh Conference on Database Semantics (DS-7) 7–10 October 1997, Leysin, Switzerland*, Stefano Spaccapietra and Fred Maryanski (Eds.). Springer, Boston, MA, 162–183. https://doi.org/10.1007/978-0-387-35300-5_8

[3] Volodymyr Blagodarov, Yves Jaradin, and Vadim Zaytsev. 2016. Tool Demo: Raincode Assembler Compiler. In *Proceedings of the Ninth International Conference on Software Language Engineering (SLE)*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). ACM, Amsterdam, 221–225. https://doi.org/10.1145/2997364.2997387

[4] David Cassel. 2017. COBOL Is Everywhere. Who Will Maintain It? https://thenewstack.io/cobol-everywhere-will-maintain/.

[5] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.

[6] A. Engberts, W. Kozaczynski, E. Liongosari, and J.Q. Ning. 1993. COBOL/SRE: A COBOL System Renovation Environment. In *Proceedings of Sixth International Workshop on Computer-Aided Software Engineering*. IEEE, Singapore, 199–210. https://doi.org/10.1109/CASE.1993.634821

[7] Free Software Foundation. 1998. GNU Binutils. https://sourceware.org/binutils/

[8] Fujitsu. 2015. FUJITSU Software NetCOBOL V11.0. Getting Started. Integrating COBOL Programs with Visual Basic. https://software.fujitsu.com/jp/manual/manualfiles/m150009/b1wd3354/01enz200/b3354-00-03-02-00.html.

[9] Anthony Green, Richard Henderson, Josh Triplett, Zachary Waldowski, Landon Fuller, et al. 1996. libffi. https://github.com/libffi/libffi.

[10] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J. H. Jacobs, and Koen G. Langendoen. 2012. *Modern Compiler Design* (second ed.). Addison-Wesley, New York, NY, USA. https://dickgrune.com/Books/MCD_2nd_Edition/

[11] Travis Hartman. 2017. COBOL blues. Reuters Graphics, http://fingfx.thomsonreuters.com/gfx/rngs/USA-BANKS-COBOL/010040KH18J/index.html.

[12] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture: A Quantitative Approach. Appendix K: Survey of Instruction Set Architectures* (sixth ed.). Morgan Kaufman Publishers, Waltham, MA, USA.

[13] IBM. 2019. Enterprise PL/I for z/OS Programming Guide Version 5 Release 3.

[14] IBM Library. 1987. *SX26-3721-05: VS COBOL II Application Programming Reference Summary, Release 4.* IBM. https://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/IGYR1101/CCONTENTS.

[15] Burnstein Ilene. 2003. *Practical Software Testing: A Process-Oriented Approach.* Springer, New York, NY, USA.

[16] Feargus Illingworth et al. 2022. *2022 Mainframe Modernization Business Barometer Report.* Technical Report. Advanced. https://modernsystems.oneadvanced.com/en/reports/modernisation2022/.

[17] ISO/IEC JTC 1/SC 22. 2018. `ISO/IEC 9899:2018`*: Information Technology—Programming Languages—C.* International Organization for Standardization. https://www.iso.org/standard/74528.html.

[18] Julia Programming Language. 2023. Calling C and Fortran Code — The Julia Language. https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/.

[19] Ralf Lämmel and Kris De Schutter. 2005. What Does Aspect-Oriented Programming Mean to Cobol?. In *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/1052898.1052907

[20] Adriaan Larmuseau and Dave Clarke. 2015. Formalizing a Secure Foreign Function Interface. *LNCS* 9276 (2015), 215–230. https://doi.org/10.1007/978-3-319-22969-0_16/COVER

[21] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the Second IEEE / ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, San Jose, CA, USA, 75–88. https://doi.org/10.1109/CGO.2004.1281665 https://llvm.org.

[22] Vadim Maslov. 1998. Re: An Odd Grammar Question. http://compilers.iecc.com/comparch/article/98-05-108.

[23] MathWorks. 2023. External Language Interfaces — MATLAB & Simulink — MathWorks. https://www.mathworks.com/help/matlab/external-language-interfaces.html.

[24] Joris Mertens. 2020. IBM zSystems fundamentals: An introductory Q&A. https://developer.ibm.com/articles/what-is-ibm-z/.

[25] Micro Focus. 2014. A Guide to Interoperating with ACUCOBOL-GT Version 9.2.5. https://www.microfocus.com/documentation/extend-acucobol/925/GUID-0617E68F-C102-4A3B-9797-279F653777D7.html.

[26] Keisuke Nishida, Roger While, Edward Hart, Sergey Kashyrin, Ron Norman, Simon Sobisch, et al. 2002. GnuCOBOL: A free/libre COBOL compiler. https://gnucobol.sourceforge.io.

[27] OMG. 2012. *Common Object Request Broker Architecture (CORBA) Specification* (3.3 ed.). Object Management Group. https://www.omg.org/spec/CORBA/3.4/

[28] Terence Parr. 2023. ANTLR—ANother Tool for Language Recognition, release 4.12.0. http://antlr.org.

[29] Guillermo Polito, Stéphane Ducasse, Pablo Tesone, and Ted Brunzie. 2017. *Unified FFI — Calling Foreign Functions from Pharo*. Pharo.org, CC-BY-SA. https://books.pharo.org/booklet-uffi/

[30] Raincode. 2023. How Can I Migrate My COBOL Applications to the Cloud? https://www.raincode.com/cobol/.

[31] Armin Rigo and Maciej Fijalkowski. 2018. CFFI 1.15.1 documentation. https://cffi.readthedocs.io/en/latest/.

[32] Rust Programming Language. 2021. FFI — The Rustonomicon. https://doc.rust-lang.org/nomicon/ffi.html.

[33] Ömer F. Sayilir, M. Aimé Ntagengerwa, and Mart H. van Assen. 2023. Crossover. https://github.com/Crossover-Compiler/Crossover.

[34] Christian A. Schafmeister and Alex Wood. 2018. Clasp: Common Lisp Implementation and Optimization. In *Proceedings of the 11th European Lisp Symposium (ELS)*. European Lisp Scientific Activities Association, Marbella, Spain, Article 8, 6 pages.

[35] Christian E. Schafmeister. 2015. Clasp — A Common Lisp that Interoperates with C++ and Uses the LLVM Backend. In *Proceedings of the Eighth European Lisp Symposium (ELS)*. ELS, London, UK, 90–91.

[36] SCO Developer Network. 2014. System V Application Binary Interface. Chapter 4. https://www.sco.com/developers/gabi/latest/ch4.intro.html.

[37] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP) (LNCS, Vol. 4609)*. Springer, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

[38] Harry M. Sneed. 1992. Migration of Procedurally Oriented COBOL Programs in an Object-Oriented Architecture. In *Proceedings of the Eighth International Conference on Software Maintenance*. IEEE, Orlando, FL, USA, 105–116. https://doi.org/10.1109/ICSM.1992.242552

[39] Andrey A. Terekhov and Chris Verhoef. 2000. The Realities of Language Conversions. *IEEE Software* 17, 6 (2000), 111–124. https://doi.org/10.1109/52.895180

[40] Yohei Ueda and Moriyoshi Ohara. 2014. *Refactoring of COBOL Data Models Based on Similarities of Data Field Name*. Technical Report. IBM.

[41] W3C. 2007. *Simple Object Access Protocol (SOAP)* (1.2 ed.). World Wide Web Consortium. https://www.w3.org/TR/soap/

[42] David S. Wile. 2001. Supporting the DSL Spectrum. *Journal of Computing and Information Technology* 9, 4 (2001), 263–287.

[43] Xinuos Inc. 2013. Developers | SCO Developer Network. https://www.sco.com/developers/gabi/

[44] Vadim Zaytsev. 2020. Modelling of Language Syntax and Semantics: The Case of the Assembler Compiler. *Proceedings of the 16th European Conference on Modelling Foundations and Applications (ECMFA)* 19 (July 2020), 22 pages. Issue 2. https://doi.org/10.5381/jot.2020.19.2.a5

[45] Vadim Zaytsev. 2020. Software Language Engineers' Worst Nightmare. In *Proceedings of the 13th International Conference on Software Language Engineering (SLE)*. ACM, New York, NY, USA, 72–85. https://doi.org/10.1145/3426425.3426933

Received 2023-07-14; accepted 2023-09-03