 Acta Univ. Sapientiae Informatica **13,** 1 (2021) 134–179

DOI: 10.2478/ausi-2021-0007

Improving productivity in large scale testing at the compiler level by changing the intermediate language from C++ to Java

Izabella Ingrid FARKAS
Eötvös Loránd University,
Budapest, Hungary
email: Ingrid.Farkas@inf.elte.hu

Kristóf SZABADOS
Ericsson Hungary Ltd.,
Budapest, Hungary
email:
Kristof.Szabados@ericsson.com

Attila KOVÁCS
Eötvös Loránd University,
Budapest, Hungary
email: Attila.Kovacs@inf.elte.hu

Abstract. This paper is based on research results achieved by a collaboration between Ericsson Hungary Ltd. and the Large Scale Testing Research Lab of Eötvös Loránd University, Budapest. We present design issues and empirical observations on extending an existing industrial toolset with a new intermediate language[1].

Context: The industry partner's toolset is using C/C++ as an intermediate language, providing good execution performance, but "somewhat long" build times, offering a sub-optimal experience for users.

Objective: In cooperation with our industry partner our task was to perform an experiment with Java as a different intermediate language and evaluate results, to see if this could improve build times.

Computing Classification System 1998: D.2.2, D.2.3, D.2.6, D.2.9, D.3.4
Mathematics Subject Classification 2010: 68N20
Key words and phrases: C/C++, Java, IDE, compiler, Titan, developer productivity, performance, efficiency, incremental compilation, industrial experience, TTCN-3

[1]An intermediate language is a language which the input program is translated to, by using the already existing compilation toolchain, reaching the executable state.

Method: We extended the mentioned toolset to use Java as an intermediate language.

Results: Our measurements show that using Java as an intermediate language improves build times significantly. We also found that, while the runtime performance of C/C++ is better in some situations, Java, at least in our testing scenarios, can be a viable alternative to improve developer productivity.

Our contribution is unique in the sense that both ways of building and execution can use the same source code as input, written in the same language, generate intermediate codes with the same high-level structure, compile into executables that are configured using the same files, run on the same machine, show the same behaviour and generate the same logs.

Conclusions: We created an alternative build pipeline that might enhance the productivity of our industry partner's test developers by reducing the length of builds during their daily work.

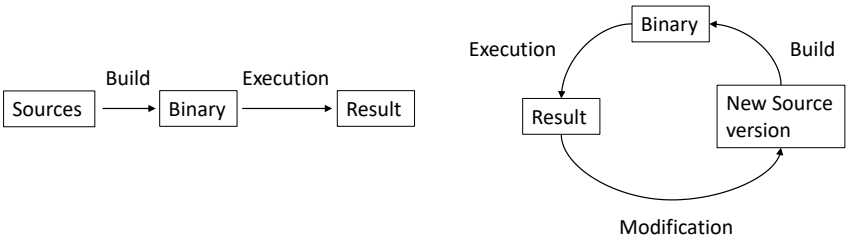
1 Introduction

Nowadays, the usage of software – developed by 11 million professional software developers ([4]) – belongs to the everyday life of our society. Software helps in navigating to destinations, communicating with other people, driving the production, distribution and consumption of energy resources. Software drives companies, trades on the markets, takes care of people's health. In order to support the growing demand for assuring quality ETSI[2] designed the TTCN-3[3] ([16]) standardised notation specifically for testing. TTCN-3 is important in many industrial domains. It is used for testing telecommunication systems ([13, 14]), IoT[4] systems ([45]), ITS[5] systems ([15, 34]), oneM2M systems ([25]), security in the industry ([3]), smart grids ([35]), etc.

As the products to be tested grow, so did their test systems written in TTCN-3, growing to millions of lines of code in size ([5]), having complex architectures ([36, 39]), showing code quality patterns ([27, 28, 37]) and evolution trends ([38]) very similar to those present in other programming languages. The size and complexity of these huge test systems lead to long lasting build and development iterations (also seen for various projects ([23]) using other programming languages). At our industry partner in a full build scenario – Figure 1(a) – building some of these large-scale test systems could require

[2]European Telecommunications Standards Institute
[3]Test and Test Control Notation 3
[4]Internet of Things
[5]Intelligent Transportation Systems

approximately 20 minutes from zero even on server-grade, 28 core machines allocated for such tasks only. In an incremental, iterative scenario – Figure 1(b) – even if every developer worked on such a dedicated server, it would still take a few minutes to check the effect of a code change. Clearly, neither scenario means "fast feedback" for the developers and allocating a separate build server for each developer is not cost-efficient.



(a) (b)

Figure 1: Building from source code. In the full build scenario 1(a) the full code base is built from zero so that the developer is able to execute the produced binary to see how it works. In the incremental build scenario 1(b) developers perform some modifications to the code, re-build the binary (incrementally, using the already built parts), execute the produced binary to see how it works and based on the result produce further modifications to the source starting the loop again. The length of these loops (build and execution time together) highly impact the development speed. Please note that applying a Continuous Integration machinery the full build can also be a loop, determining how fast/often developers can get feedback on their changes.

In this paper we present how a build process was created for our industry partner's tool that uses Java as the intermediate language. This new process is based on the existing one that used C/C++. We were tasked by our industry partner for performing and experimenting on the effects of adding a new intermediate language to our industry partner's toolset. We found that our solution is a good way to support the process of building large-scale TTCN-3 test systems by enabling the developers to get faster feedback after a code change.

Some of the reasons why our industry partner (at around 2000 as shown in [40]) chooses to create a build process that uses C/C++ as an intermediate language:

- TTCN-3 defines itself to be a language operating on an abstract level, creating abstract test suites, offering standardised mapping of its runtime ([12]) and control ([17]) interfaces to Java, ANSI C, C++[6], C#.
- Translating from TTCN-3 to C/C++ and lets platform dependent C/C++ compilers generate the binary, reduces development cost by supporting several platforms (operating system) handled by the C/C++ compilers.
- In case of a new C/C++ compiler version, used in the build pipeline on a platform, improves its code optimisation mechanisms (as part of the build process) then users of our industry partner's tool will also benefit from it on that platform.
- Translating from TTCN-3 to C/C++ not just offers high execution speeds, but as the code generated is statically typed, the type correctness of the operations is checked by the C/C++ compiler as well.

While for the purpose of the experiment presented in this article Java is just another programming language, in practice, our industry partner chooses Java for its added benefits:

- As the part of the toolset running on Eclipse was already written in Java (syntax and semantic analysis, code quality checking, refactoring, architecture visualisation), generating Java code keeps their systems consistent, and builds on already existing expertise.
- Eclipse users have access to a free and mature Java Development Environment that can be used to compile the generated Java code to an executable format, supports users in executing these projects, offers extensive support for developing code (for example developing test ports), etc...
- Opposed to C/C++ codes, that require to build separate binaries for the different operating systems, Java is running on virtual machines available on most platforms, and only one build is needed during the release, and CI procedures to support those systems (and maybe even ones not yet supported) automatically.

Our contribution is unique in the sense that with our extension there are two different build and runtime environments (see Figure 2) available for the

[6]Our industry partner uses a proprietary mapping better suited for their specific needs

developers. The input to both can be the same TTCN-3 source code. In order to unburden the conversion of the already existing platform dependent code parts, we kept the new runtime architecture, generated code and conversion methods as close as possible to the original one. The execution of the built "binaries" can be configured with exactly the same configuration files, performed on the same machine, creates same log messages, and in parallel mode, connects to and is directed by the same main controller via the same internal protocols. This puts us in the position of being able to study the effects of both environments on the development of the same user product.

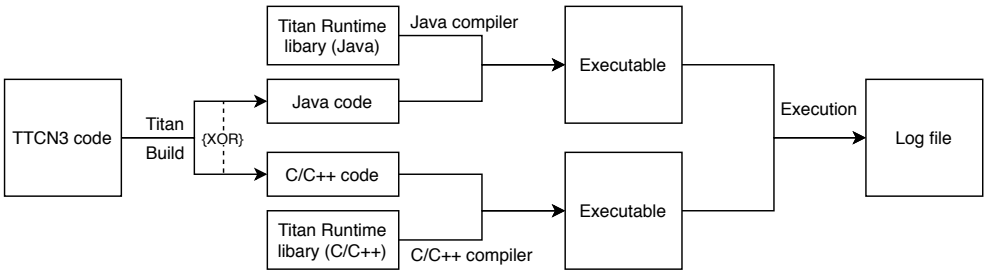


Figure 2: The figure presents the relationship of the 2 build and runtime environments. The input is the same for both the Java and C/C++ build pipeline and the execution of the binaries produce the same log files. The runtime libraries are the implementation of the base types, additional functions, etc. of the TTCN-3 language.

Our findings indicate that while C/C++ still might have performance advantages over Java in some situations, Java enables faster development iterations, and at least in our case, provides acceptable execution performance.

This paper is organised as follows. In Section 2 we present earlier work related to our subject. Section 3 presents our design and Section 4 some general implementation details. Section 5 shows our measurements and comparisons on build times and execution performance on both the C/C++ and Java sides. Section 6 deals with the limitations and Section 7 deals with the validity of our results. Finally, Section 8 summarises our findings.

In Section A in the Appendix we provide a short introduction to TTCN-3 and in Section B we show more details to our measurements.

2 Related Work

The examined toolset, Titan[7], has actively been developed and used in the last 20 years. During this time it has already gone through the usual steps of build time improvements: optimisation, experimenting with Continuous Integration configurations, better hardware, incremental build support, caching, etc. Our industry partner's experience showed that in the case of Titan's build process the full length of the compilation mostly came from using C/C++ as the intermediate language (in the projects shown in sections 5.2 and 5.5 this is measured to be approx. 96-97%). In order to improve the build process our university laboratory were asked to perform an experiment for compiling to a different intermediate language and for creating an alternative build process for the users of our industry partner's tool.

Although it could be said that we had to create a cross-compiler translating from TTCN-3 to Java and the runtime library it links to, this is only a part of the build process that creates value for the users. The users, in this case, write automated tests and execute them. From their point of view all of the operations required to make their tests run are "necessary technical limitations" that should be done in the background as fast as possible, as it engrosses time from the productive work. Also the term "cross-compilation" might induce misunderstandings: In the everyday usage of the term, a cross-compiler is a compiler that is compiling to a different hardware/software architecture that is not available, or does not support direct compilation on itself (like embedded devices). Compared to this, our solution takes the same input source code and creates an output executable that will be executed on the same machine (see figure 2). As such, we also find it better to look at our work as creating an alternative side of the toolset (instead of cross-compilation) as that better captures the scope and goal of the work.

When we tried to elicit information on the build time of different programming languages, we found that plenty of people engage in rather unhealthy discourses on how one programming language is "better" than the other, without any measurements for practical use cases. The general internet community does not seem to have easily available and thorough information about that. There also seems to be little academic interest on this topic. Maybe this is coming from the lack of deep connection between industry and academia. Maybe it is hard to find funding to run a 3 year-long experiment purely on

[7] https://projects.eclipse.org/projects/tools.titan

the academic side to reach functional equivalence in the implementation, while keeping industrial level code quality and coding practices.

At the same time long compilation times seem to be a widespread problem. While studying GitHub repositories of Java and Ruby projects Ghaleb et al. found ([23]), that 40% of the builds lasted more than 30 minutes. They suggest that developers should properly configure their Continuous Integration systems and use tools to identify cacheable spots. Others, like Abdalkareem et al. [1], tried to combat long-running builds by trying to identify changes for which the CI build might be skipped. While their results might make it possible to skip on average about 5% of the CI builds, in the iterative working scenarios we are investigating, this would still mean too many, too long-running builds, that hold back the productivity of the developers.

Reinholtz predicted ([33]) that Java not only can, but also will have better general performance than C++. He reasoned that Java can compile and recompile the program as it executes and has access to runtime information not available to a traditional C++ compiler allowing to achieve better execution performance. Our measurements show that C++ still has an edge during the initial executions, however, for longer executions, Java can reach a measurably better performance in some situations.

Although they directly migrated a software from COBOL to Java, the closest to our task we could find was the experience of De Marco et al. ([8]). They explained that an earlier attempt at redeveloping the same application from scratch has failed. So they choose to migrate the application to a new platform and programming language using code and data translation approaches. They reported delivering an application functionally equivalent to the original and provided some useful remarks: (1) the new Java application still had to follow COBOL idioms and mainframe concepts, (2) they observed performance bottlenecks during the conversion, and (3) mentioned that even with the translation they needed to develop deep application understanding.

Batyuk et al. found ([6]) that native C applications running on the Linux layer of Android can be up to 30 times faster than identical Java applications running on the Java Virtual Machine of Android. For their measurement they sorted arrays of random integers with different sorting algorithms. Although this can be a way for performance measurement it is not something most people use their phones for.

Some researchers have already observed that even if Java is not always "better" than C++, it has other features which might make it as a reasonable

choice. Amedro et al. ([2] compared the performance of a Fortran + MPI[8] and a Java implementation of the NPBs[9]. They found that "the overhead of Java is now acceptable when performing computationally intensive tasks", but they also found that (in 2008) Java had scalability problems in communication-intensive benchmarks.

Gherardi et al. found ([24]) that using the server compiler option, Java was only 1.09 to 1.91 times slower in their tests aiming at robotics-related scenarios. They concluded that for their use cases, also taking into account the additional benefits offered by the language, "Java can be considered a valid alternative to C++".

Taboada et al. analysed ([41]) the applicability of Java for High-Performance Computing. They also started with the assumption that Java lacks thorough evaluation on its performance, therefore they provided their implementation. They concluded that Java may achieve similar performance to natively compiled languages. Their most favourite features of the language (platform independence, portability, type safety, etc.) seem to them reasonable for the trade-off in the performance overhead.

Cook found ([7]) that a Java implementation of OpenMP[10] can be faster than a C implementation. They hypothesised that maybe the GCC implementation of OpenMP had deficiencies.

Nanz and Furia analysed ([31]) programs from the Rosetta Wiki, a site that collects solutions to programming tasks, implemented in various programming languages. This repository allowed them to compare several solutions, written in different programming languages, to the same programming tasks. Using statistical analysis (p-values, effect sizes) they found that "C is the king on computing-intensive workloads", but in the case of "everyday" workloads "languages may be able to compete successfully regardless of their programming paradigm". Later Furia et al. reanalysed ([21]) the same data with Bayesian techniques allowing them to draw more detailed conclusions.

While their works ([31, 21]) are consistent for their purposes, from our point of view their performance measurement had a bias for languages that compile into machine code: a) the compilation times were not measured/disclosed, which effectively gives an edge to pre-compiled languages over Just-In-time languages; b) as each execution was a standalone execution, this still might have penalised the Just-In-Time compiled languages where optimisations might only happen after a code part was executed several times (to have

[8] Message Passing Interface
[9] Numerical Aerodynamic Simulation Parallel Benchmarks
[10] http://www.openmp.org

profiling information for the optimisation). This way, pre-compiled languages could spend as much time as they wish on compilation and code optimisation at no cost in this setup, while Just-In-Time compiled languages might have been penalised for it if this starts during execution (because their compilation happens during execution), or have optimisations effectively disabled in some cases.

The papers described above presented some advantageous features of Java over C++, and in special cases showed that they have similar runtime performance. However, we could not find any recent academic studies which would present us the experiments on full build and execution times.

While working on this article it came to our attention that some hardware testing youtube channels like Gamers Nexus ([29]) started to publish code compilation test results for their hardware tests, in order to be able to measure the performance differences in massively multi-core modern CPUs. Their work offers useful, publicly available and detailed information for developers on what hardware to buy as their development equipment to improve their productivity. At the same time, by the very nature of their tests (building the same code with the same build system on several different CPUs), they will miss the opportunities presented in this article (where we reduce the build time on the same hardware, by using a different intermediate language, to reach a functionally equivalent executable).

3 Design

In this section we go into the design of our solution. We introduce the general context in which the design had to fit in. We explain the general design rules that we had to follow to be able to handle the long-lasting development process.

We don't detail the mapping of TTCN-3 elements to C/C++ or Java for two reasons: 1) as we tried to follow the mapping already existing on the C side, we dare claim as original ideas/observation only those cases where we had to differ in some way, 2) to save space in this article, as the mapping of TTCN-3 elements, runtime elements and API are already described in detail in the reference guide of the C side ([11]) and the reference guide of the Java side ([10]). We also made an example package available ([20]) containing a "Hello World" and some simple types in TTCN-3, together with the C/C++ and Java codes they compile into.

3.1 Context

To the best of our knowledge, at present TTCN-3 is the only internationally standardised notation designed specifically for testing which is also used frequently in the industry. Originally, Titan was able to translate TTCN-3 using C/C++ as the intermediate language ([40]). We were tasked to extend the part of Titan that runs on Eclipse to be able to generate Java code and write the necessary runtime libraries. As the examined tool stands behind many applications our solution might have a high impact, and at the same time, is also limited in how far we can drift from the current setup incentivising us to keep the architecture as close as possible to the original one.

Compared to De Marco et al. ([8]), we already had a deep understanding of the behaviour of the already existing code. Our team had a profound experience with Titan, either being its system architect, or being the member of the Large Scale Testing Research Lab.

Almost all of the needed tests had already been available for us before starting the development. Titan uses TTCN-3 code to test how well it can build and execute TTCN-3 code. As the new extension had to support the same input language, resulting in the same behaviour and output, and needed to conform to the same tests, our industry partner selected a set of their already existing tests to test our extension. We had to be able to successfully build and execute these tests, generate the same runtime logs during execution to show that the result of our experiment can be functionally equivalent to the existing solution.

We would like to highlight that we consider functionally equivalent the C/C++ and Java side if the logs, of the executions of the same tests, are the same since the logs contain detailed information together with the result (none, pass, inconc, fail, error) of the test cases. Keeping the architecture and the Java implementation as close as possible to the C/C++ architecture and implementation also contributed to guarantee this functional equivalence.

3.2 General design approach

It was clear from the beginning, that this experiment was going to take years, so we had to establish some general design rules that would govern our work and accept that certain limitations were reasonable to keep the timeframe.

Instead of inventing a new concept, we decided to stick to the already existing build and runtime structures. The following design decisions were made: (1) maximize the speed of work, (2) minimize the risk of running into major

blocking issues, (3) provide the users of our industry partner with the option to choose between tools that are as much as possible identical in their functionality, (4) minimize the cost of converting the already existing platform dependent code (testports and external functions). We also decided to leave possible Java-specific optimisations (that were not necessary to provide the same functionality) for later.

3.3 Performance optimisation shifted to the end

It is important to understand that when building a compiler, most of the infrastructure must be in place and language support must be implemented, before any kind of relevant performance measurement, needed for optimisation, can be done. For example:

- The concept of the compiled code, the runtime libraries and how they relate to each other must exist.
- The compiler needs to be able to generate the classes that represent the TTCN-3 modules.
- The compiler must also be able to generate code for functions and function calls.
- The runtime must have a decent implementation of the classes representing the used types, so that they could be used in the generated code.
- The compiler must be able to generate code for variables of these types, for their instantiation, for checking their values, etc...
- The compiler must also be able to support statements that represent loops, assignments, variable definitions, etc...

Already, reaching the minimum level to do some performance measurement (e.g. simple operation in a loop) requires substantial work, and this might still not be indicative of the performance characteristics of the final tool, as that will need to support much more features with possibly different structures. Hence, as the first step, we decided to "just rewrite" the existing C/C++ code to a functionally equivalent Java code, and optimise it further only after thorough testing.

3.4 Handling long development timeframe

It took us 3 years to reach a stage where we could claim to have reached functional equivalence on the tests we received from our industry partner. This is of course not complete functional equivalence to the industrial tool, but a transition point when our experiment can be turned into industrial product to reach complete functional equivalence.

To make sure we are on the right track we used parts of the tests we received from our industrial partner. Throughout the development we have seen that the build times were showing promising results (for the limited set of elements supported at those times) and that we did not run into problems that could have blocked this project.

To handle the challenge of not yet supported features and missing code parts, we used FIXME and TODO tags in the compiler and Dynamic Testcase Errors (DTE) in the runtime. Initially, we generated a syntactically erroneous FIXME string into the code for every statement, type and definition telling what is missing. Also, functions in the runtime libraries reported DTE when executed. Putting them together:

1. When a new branch was implemented in the code generator or the runtime, the FIXME tag was moved into the still missing branch. In this way, we could effectively reduce the scope of the FIXME tags and allow testing the executable paths.

2. When a feature was first implemented in the compiler to generate valid code, the library function still reported a DTE during execution reducing the scope of the FIXME tag effectively.

3. When a design element for a type, statement or a feature was fully implemented, all tests had to be run to pass.

This strategy enabled us precise tracking of what was still missing, since those tests didn't compile or run into errors.

The TODO tags were used similarly as the FIXME tags, marking improvement ideas that will be implemented later.

4 Implementation

In this section we go into the technical implementation details of which features/concepts of the original C/C++ conversion could be reused, and which needed to be done differently in Java to succeed. Finally, we show a way

how a compiler in an IDE might offer better performance than a traditional command-line compiler.

4.1 Concepts that could be reused from the C/C++ side

As we tried to stay as close to the original architecture as possible, there were several points/concepts that could be reused on the Java side without any problems.

- On the C side, each module was generated as a separate `.hh` and `.cc` file with a name generated from the module's name. The Java side kept the concept, generating a separate `.java` file from each module, using the same name conversion procedure.
- On both the C and Java sides non-synonym types were generated as a class representing the value version of the type and a class (with `_template` postfix in its name) representing the template version of the type.
- The runtime libraries of Titan on both sides contained the implementation of the same built-in types, runtime functionalities and additional functions.
- The implementation of each type in the runtime library of Titan on both sides followed the same abstract logic, had similar abstract interfaces, access patterns, etc...

4.2 Concepts of the C side that needed to be adapted

While many of the concepts could be reused, we had to make some changes.

- In the generated `.hh` and `.cc` files of the modules the code was located in a namespace generated from the module's name. On the Java side, we generated a class from each module, and the definitions of the module would be members and static nested classes of this class.
- Synonym types were mapped on the C side onto `typedef` -s. On the Java side we either used the referenced type's name instead of the new type's in the generated code or generated new classes to extend the referenced ones (if the user requested it or additional properties made it necessary).

- While the runtime libraries of Titan contained the same abstract contents on both sides, there were some minor differences: all code had to be placed into a package, the additional functions into a class of their own (on the C side they were outside namespaces), etc...
- While the implementations of each type in the runtime library of Titan on both sides were similar on an abstract level, in the actual implementations they differed slightly: Operator overloadings from the C side were translated as functions on the Java side (`operator[]` → `get_at()`, `operator==` → `operator_equals()`, etc...). Also, const and non-const access operators are important in the runtime, so they have been mapped to functions following the naming convention `get()` / `const_get()`, `get_at()` / `const_get_at()`, etc...
- On the C side each TTCN-3 `import` statement was translated to be a C/C++ `include` statement. As this statement is transitive in C/C++, it automatically ensures that all needed types are present. However, imports are not recursive in Java. As such, we decided not to generate code for TTCN-3 `import` statements, but whenever the code generator found that it needed to use a definition that was not defined in the TTCN-3 module (we are currently generating code for) it would generate the necessary Java `import` instead for the class that represents the module of that definition and generate the "usage" prefix with the name of its encompassing class (that represents the module it is located in). Please note, that this way the dependency hierarchy in the generated Java code can be different from that present in the generated C code. When the code compiled does not have unnecessary imports, all directly used modules will be referenced in both generated codes.
- In the generated code the C side used implicit type conversions[11] heavily. In the Java generated code, we had to use two different methods: (1) in the general case we generated the type conversions as static functions, or calls to constructors, or (2) we create several versions of the functions in the runtime library with both native and generated/runtime represented types, allowing to save the costs of the conversion.
- The Java side had a unique challenge not present on the C side. Nested classes in the .java files are translated into separate .class files. On MS-

[11] E.g. assignments between different types when the receiving type has a constructor with a single parameter of the type to be assigned to it or the assigning type has an implicit cast operator to the target type.

Windows however, where the file system is not case sensitive by default, this created a problem when the names of two or more embedded classes differ only in small/capital letters, as the file system will not allow generating the class files properly. To solve this problem an extra translation step checking the names of TTCN-3 types in the modules, before code generation, and the problematic names were postfixed with the starting offset of the definition (to make them unique). As this might create inconveniences for the users accessing these definitions from Java, a code smell checker was implemented in Titanium[12] that warns the user during semantic checking about similar type definition names in the same module.

- While C/C++ compilers can choose to ignore unreachable codes (and so it was also allowed in TTCN-3 in Titan), this is a semantic error in Java. Since we already had a code smell checker for such situations, we decided that this issue can be fixed best by the users: they can remove (the already reported) unnecessary statements from the TTCN-3 code.

- On the C side, the internal representation of string types used reference counting. This enabled efficient TTCN-3 assignment operations since the left-hand side could be set to reference the same internal structure used on the right-hand side (for example unsigned char array for hexstrings) and delay the actual copy till the variable of a string type needs to be modified and its internal structure has a reference count > 1. When we implemented this feature on the Java side we did not measure any significant performance benefit (outside of artificial use-cases) and decided to leave it out to keep the code as simple as possible.

- On the C side, static values of string types were generated into the code only once, as static objects, that are unique, initialised at startup time and only referenced during execution. On the Java side, each occurrence of such a string created a new object in the code. Once implemented, such an optimisation should improve the runtime speed of the Java generated code.

- The internal representation of some types on the C side used `unsigned char` array to store the elements (for example, `hexstring`, `octetstring`). As Java does not support such an unsigned type we had to use other

[12] The code quality analyser of Titan for checking for code smells, measuring code metrics and having support for extracting and visualizing architecture.

signed types, for example `byte` arrays instead. This also meant using additional code (for example using `& 0xFF` to make sure the value was understood correctly as an unsigned value) in certain operations.

- Java also has the limitation that functions cannot contain more than 65535 bytes of code ([43]). We run into this in two situations: (1) Some of the functions generated for a TTCN-3 union type with more than approx. 1670 alternative fields (in which case we had to generate helper functions each of which could handle 200 alternatives), or (2) we also ran into a 2.800+ lines long TTCN-3 function (where we had to ask the users to partition it into smaller parts as we could not find an easy way to automatically generate helper functions to it so far).

- When a new PTC[13] was created on a HC[14] the C side does a fork, both resulting in a parallel process for the new PTC and also copying the memory contents[15]. Java does not support this behaviour, so we decided to use threads instead. This way, while every PTC was a new process on the C side, they were new threads on the Java side. Plus, every static variable inside the runtime library on the C side was mapped to be a thread-local variable on the Java side. Thus, we do not need to start a new JVM and configure it for every PTC created, as the C side algorithms had already been designed to run in their own process, and the TTCN-3 language also does not allow direct access between two PTCs. This introduced minor slowdowns during access. As thread-local variables are currently implemented inside the JVM using map data structures with the threads as keys, accessing the value that should be seen in the current thread has the overhead of an extra data access.

For a detailed description of the mapping of TTCN-3 elements, runtime elements and API turn to the reference guide of the C side ([11]) and the reference guide of the Java side ([10]). To demonstrate the mappings we made a short example package available ([20]). This package contains a "Hello World" module and some simple types in TTCN-3, together with the C/C++ and Java codes they compile into, to illustrate the mappings between these language elements. For a more complete example turn to the regression tests ([19]) we

[13] Parallel Test Components are TTCN-3 concept for components created by the Main Test Component and running in parallel to each other.
[14] Host Controllers are instances of the executable test program.
[15] Actual implementations might use a copy-on-write solutions for performance.

used for our measurements, which can also be used to check the mappings of all supported language elements.

4.3 The benefits of being an IDE

In this subsection, we show one more element of our work, that goes beyond simple cross-compilation, to improve the performance of the build process. As the Java side was built into an IDE, that was able to track changes on a fine grain level to optimise semantic checking, we could use this information to improve the build times even before the compilation to Java happens.

On both the C and the Java sides the codes were generated on a per-module base. That is, first the code generator compiled the `string char*` and `StringBuilder` that represents the module in the target language (called S further), then compared it to the content of the already existing file. The string S was only written out if the target file did not exist, or had a different content. The mentioned process created a potentially large saving in compilation time: since most of the TTCN-3 modules do not change from one compilation to the next, their generated code will also not change, not even be touched. During the build process the C/C++ and Java compilers might notice that their input file has not been changed, and skip its compilation. It can be a large improvement, as comparing the contents of strings and files is negligible compared to compiling the file again (in larger systems the generated code can be several hundreds of thousands or millions of lines of code).

On the Java side, the compilation process could further be improved since it was part of the TTCN-3 IDE of Titan. As presented in the work of Oláh ([32]), the Designer part of Titan could analyze the code of TTCN-3 projects incrementally. Once the project was analyzed, and when the user edited something, the IDE was able to discover easily (and on a fine grain) the code parts whose status might have changed (being correct or erroneous in some way, or referring to a different target, etc...) and used this information to minimize the amount of code whose semantic information needed to be re-checked.

There is no need to generate the string S for modules which had already been compiled and their status has not been changed. Adding this technique (further referred to as IDE mode) to the new compiler, it had a substantial effect on the speed characteristics of compilation: as most daily changes can be reasonably assumed to affect the status of only a few code parts, the (same) file overwriting process can be saved for most of the dependent modules. For example, while a full compilation might have compiled all code, changing the signature of a function required recompiling only that module and the modules

from where that function is called directly. While a change inside a function only required compiling the module it is located in.

Please note that while in this article we analyse this feature as something that can be turned on/off, in the product it will be always active.

5 Performance measurements

In the first part of this section we describe the input on which we performed the measurements together with the testing machines. In the second and third part, we present the build and execution performance. In the last part, we show the information we have on the scalability of our implementation.

5.1 Performance measurement environment

To measure and compare the performance of the C and Java sides we used the subset of the regression tests of Titan provided to us by our industry partner ([19]). These tests included:

- Tests for handling the values of all
  - Base types (`boolean`, `integer`, `float`, `objid`, `bitstring`, `hexstring`, `octetstring`, `charstring`, `universal charstring`);
  - Complex types (`record`s, `set`s, `record of`s, `set of`s, `enumeration`s, `union`s, the `anytype` type).
- Tests for template matching for all base types and complex types.
- Tests for basic statements.
- Two ASN.1 modules and two TTCN-3 modules that import them (to test importing from ASN.1).
- Tests for the predefined functions of TTCN-3.
- Tests for timers within testcases and used as a testcase timer to limit their execution duration.
- Tests for the `all from` language construct (for its special handling of values).
- Tests for the "RAW" encoding supported by the Java code generator.

We note that although the verdict tests were excluded from our measurements, they could also be compiled and executed. The reason was that when test verdicts other than pass are reported, the replication studies can be confusing.

Altogether 90 source files were used in the measurement of compilation and execution speed: 88 TTCN-3 files and 2 ASN.1 files. The package also contained configuration files, set to execute all control parts in the modules at a given amount of times. This package can be downloaded from [19].

From our point of view, these tests were input. Examining the output we could conclude that the implementation was functionally equivalent with the original compiler.

For the measurements, we used two different architectures (laptops).

1. Laptop 1 used for measurements was a HP EliteBook 840 G5, with an Intel® Core™ i5-8350U CPU, 16 GB RAM, using a SSD, running Windows 10 and the 3.0.7-1 version of Cygwin ([44]), GCC 7.4.0, Eclipse 4.4.2, Java SDK 1.8.

2. Laptop 2 used for measurements was a Lenovo ideapad Y700, with an Intel® Core™ i7-6700HQ CPU, 8 GB RAM, using a HDD, running Windows 10 and the 3.0.7-1 version of Cygwin, GCC 7.4.0, Eclipse 4.10.0, Java SDK 1.8.

Laptop 1 was expected to be the target architecture at our industry partner, Laptop 2 was used as a control platform to check our observations in a different setup ([22]). We had to use a control platform to account for the potential issues coming from Laptop 1 being a machine used at our industry partner. Users (test automation developers in this case) have limited control over it (the firewall software could not be stopped/deactivated/reconfigured for the duration of measurements, operators might trigger minor refreshes remotely, that would run in the background, etc.).

The source code used for the measurements is part of the "CRL 113 200/6 R6A (6.6 pl0)" version of Titan ([46]), with the "6.6.0" tag on github.

5.2 Build speed

To get a picture of the speed of the build processes for both the C and Java sides we set up and measured 4 uses cases.

1. Full build. In order to measure the speed of a full build we cleaned all generated files and ran the build process from scratch.

2. Incremental build with minor change. In order to measure how the system might perform in daily operations we also measured the build times after a small change in an already built project. For this reason, we changed in the 22nd line of the `TboolOper.ttcn` file the `true` value to `false` and back. This modification triggered the re-generation of the code for this module but did not re-generate code for any other modules. Also it made the "boolAssign" test report a fail verdict, but this was not an issue for measuring the compilation time.

3. Incremental build with inserting/deleting a testcase. In this use case we simulate a scenario when users write or delete a small testcase in a single file resulting in only one incremental build after this operation. For this purpose in the `TboolOper.ttcn` file we commented out the execution call of the `boolAssign` testcase from the control part, deleted the code of the testcase for one change and reverted this modification as another change.

4. Incremental build, refactoring in 5 files at the same time. In this use case we simulate a scenario, when users perform refactoring operations that create changes in 5 files at the same time. For this, we extended the `TcharOper.ttcn`, `TbitstrOper.ttcn`, `ThexstrOper.ttcn`, `ToctetstrOper.ttcn` and `TucharstrOper.ttcn` files with a new testcase called "dummy_test()" having an empty body and using the search & replace features of Eclipse we renamed it "dummy2_test" for one change and reverted it for another.

Full builds are expected to happen rarely (as they take a long time), but when developers wish to make sure that everything works from zero, they have to do full builds and it gives an upper bound for our measurements.

Incremental builds are expected to happen more frequent in practice. In fact, in Eclipse one of the default settings for builds is to "Build Automatically". That is, whenever a developer saves a file after a change, Eclipse starts an incremental build in the background. For builds on the C side users had to turn this off, as the duration of incremental builds was a lengthy and resource intensive process. Based on the numbers we have measured for incremental builds, on the Java side users should be able to leave this feature on and work interactively in the environment.

As the extended tool was an IDE running on Eclipse (supporting both C/C++ and Java code generation), we measured the compilation speed via

the Eclipse's interface[16]. This method allowed us measuring the entire build duration, as the user would experience it. At the same time, it applied a common interface for both the build process generating the C/C++ code and the build process generating the Java code.

Internally, the build on the Java side used the mechanisms provided by Eclipse to build the generated Java files. Here we assumed that the process used all available processing power for a fast compilation. We considered the following settings as a form of configuration that might affect the build:

- The Eclipse in which the builds were done was launched with the "-Xms5178m" and "-Xmx5178m" options. In this way we could minimise the chance for garbage collections.
- Inside the Eclipse instance where the builds were done, we set the "Compiler compliance level" of the "JDK Compliance" options to 1.6 (compiling for Java version 1.6), as that is the minimal version of Java supported by the generated Java code.

On the C side, we used the processes and tools Titan offers for its users. The "-O2" flag was used to control the optimisation used by g++, make was called with "-j8" to enable parallel compilation, and inside the generated makefile the following rule was used to call g++ to translate the generated .c and .cc files into .o files[17]:

```
.cc.o .c.o:
        g++ -c $(CPPFLAGS) -Wall -O2 -o $@ $<
```

Where CPPFLAGS listed the files to be included from the runtime libraries of Titan.

To do statistically rigorous performance evaluation ([22]) we measured 50 compilations for all scenarios.

We could make several observations from the results presented in Tables 1(a) and 1(b).

The full build (case 1) was measured to be much faster on the Java side than on the C side, 14∗ faster on laptop 1 and 8∗ faster on laptop 2. Considering case 2 from the table the Java side was 35∗ faster on Laptop 1 and 21∗ faster

[16]To run our plugins from source we already had "Eclipse application" style launch configurations set up. We selected the "build/invoke" and "debug" options for the measurements on the launch configuration's "Tracing" tab. This way, for every build invoked in the launched Eclipse instance, we got debug logs on the duration of the build in the Eclipse instance the source code was located in.

[17]Please note that these are the default and recommended settings of our industry partner

	Tool	case 1	case 2	case 3	case 4
C side					
	avg	153.40	34.51	35.15	35.20
	min	150.25	29.06	33.44	32.63
	max	159.49	41.64	37.60	36.99
Java side					
	avg	10.95	0.98	0.86	0.87
	min	9.51	0.88	0.82	0.83
	max	18.31	1.30	1.16	0.97
Java side + IDE					
	avg		0.15	0.18	0.34
	min		0.08	0.15	0.31
	max		0.24	0.22	0.39

(a) Laptop 1

	Tool	case 1	case 2	case 3	case 4
C side					
	avg	224.52	25.73	25.74	26.38
	min	221.29	25.09	24.65	25.52
	max	234.08	26.72	27.47	27.26
Java side					
	avg	28.12	1.23	1.29	1.64
	min	26.43	1.09	1.18	1.59
	max	29.94	1.52	1.44	1.74
Java side + IDE					
	avg		0.3	0.32	0.74
	min		0.25	0.29	0.7
	max		0.38	0.44	0.82

(b) Laptop 2

Table 1: Build speeds measured on Laptop 1 1(a) and Laptop 2 1(b) in seconds. Case 1 is Full build, Case 2 is Incremental build with a minor change, Case 3 is Incremental build with inserting/deleting a testcase, Case 4 is Incremental build with refactoring in 5 files at the same time

on Laptop 2. When the IDE mode was used, the speedup increased to $230*$ on Laptop 1 and $85*$ on Laptop 2.

The speedups were different, but were present on both laptops in all scenarios used for measurements. It would be reasonable to assume that the HDD in Laptop 2 might have created the overhead in both cases (access time and read/write speed), compared to the SSD in Laptop 1.

Our observations shows that the effort for reducing the build time was successful. We could also see how users benefit from the IDE mode: knowing what the user has edited it is possible to calculate the set of code pieces needed for the re-analysis.

In order to understand better the importance of the measurements it is beneficial to look at the development activity from the viewpoint of the user ([26, 42]):

- 0.1 second is the limit where the user still feels the interface to be reacting instantaneously.

- $0.1 - 1$ second is a "stammer", when the user's flow of thought is not interrupted, but the delay is noticeable. User's do loose the feeling of operating directly with the interface.

- > 10 seconds is "disruption", this is the limit of keeping the user's attention focused. User's might wish to do something else while the computer is working, leading to even longer non-productive periods of time.

Based on the above classification we found that the user experience of builds on the C side fall into the "disruption" category, on the Java side into the "stammer" category, keeping user's attention un-interrupted. In best case scenarios the user experience can even be instantaneous on the Java side.

5.2.1 Some interesting facts and observations

1. On the C side for this project, in full build, the translation from TTCN-3 to C/C++ files took approximately 6 seconds, the translation of C/C++ files into .o files approximately 132 seconds, linking the object files took approximately 15 seconds. Altogether, approximately 96% of the build on the C side was coming from using C/C++ as the intermediate language.

2. On the Java side for this project, in full build, the translation from TTCN-3 to Java files took approximately 1.2 seconds. Here, approximately 89% of the build on the Java side was coming from using Java as the intermediate language.

Although we lack the necessary skills to analyse the internal procedures done during build in GCC and Eclipse's Java toolset, we can still offer some intuitive reason for this difference in build speeds:

- In C/C++ the platform specifics of the target system has to be taken into account during build time. In Java this is not done during build, as the actual execution platform is only known during execution.

- In C/C++ all optimisations have to take place during build, before execution. As Java is a Just In Time compiled language, most of the optimisation work happens during execution.

5.3 Execution speed

The execution speed of the generated C/C++ binary was measured on both laptops in a Cygwin bash shell instance using the `time` command.

To measure the execution speed of the Java code, as it was executed from within Eclipse, we had to extend the generated code of the main class used for

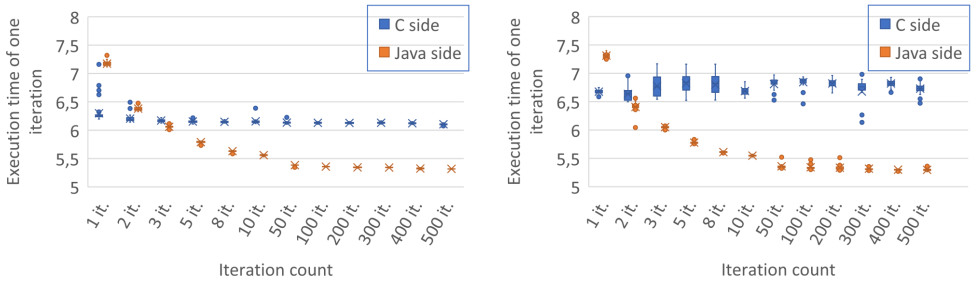
single-mode execution `Single_main`. The very first Java line to be encountered was `long absoluteStart = System.nanoTime()` (this was executed before any test-related code) and the last line calculates the elapsed time as `(System.nanoTime - absoluteStart) * (1e-9)` and outputs to the standard output (this was executed after all test-related code is properly terminated).

To see how each side performed on a longer run we decided to run and measure different amount of iterations of the same test set. A single iteration means that in the configuration file each module's control part was executed. Each control part in a module executed the testcases in that module, resulting in each testcase was executed exactly once. The 500 iteration means that we had a configuration file which listed the contents of the single iteration 500 times. This was not a good simulation for complex tests running for days, simulating complex network components and user, while sending millions of messages per seconds, but the best option we had to understand how the Java side we created might perform in such situations (the effects of Just In Time compilation, garbage collection, whether we had memory leaks, etc...).

Since these tests also contained tests on how Titan was handling timers, we decided to perform two different measurements. The "with timers" means that we were executing the tests as they were. The "without timers" means that the testcases testing timers were commented out from their module's control parts. More precisely, in the `TcontrolTimer.ttcn` and `TlostTimer.ttcn` files we put in comment the content of the control part. This eliminated the part of the execution that was not beneficial for performance testing and also reduced the number of features to be tested. For each iteration size, both with and without timers, we measured the execution time 50 times.

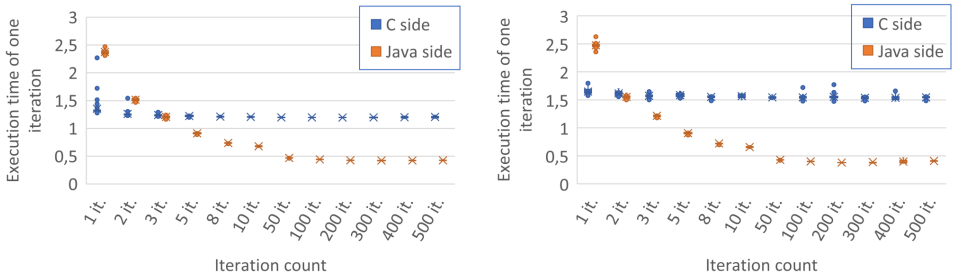
Performance measurements (see Figures 5 and 6 in the appendix) showed similar results on both machines for high iteration counts, in the same measurement modes. For low iteration counts Laptop 1 executed the C side code and Laptop 2 the Java side code somewhat faster at identical settings (see data in [18]).

Figures 3 and 4 (showing the per iteration times on Laptop 1 and 2) help emphasise some interesting observations. In the first iteration, we could see that the C side is usually faster in both measured ways. The C side also had an approximately constant per iteration execution time for all iteration scenarios. The Java side seemed to "warm-up" as the number of iterations increased reaching a constant per iteration execution time after approximately 50 iterations. On Laptop 1, without timers, the iteration time (initially 2.37s) decreased to approx. 0.42s on average which is 83% reduction. Java finished the



(a) Laptop 1 (b) Laptop 2

Figure 3: Box & Whisker chart of the Per Iteration Execution times with Timers.



(a) Laptop 1 (b) Laptop 2

Figure 4: Box & Whisker chart of the Per Iteration Execution times without Timers.

500 iterations in 390 seconds which was 65% faster than the C side. On Laptop 1, the Java side was running faster than the C side from the 3rd iteration, on Laptop 2 from already the second iteration.

We analysed our measurements with the statistical package R (see table 2). We applied t-tests (Welch two-sample) and Bayesian analysis (BESTmcmc described in [30]). Both showed a clear difference between the C and Java sides for the same iteration counts. BESTmcmc also showed how the C side was faster in one iteration, but fell behind the Java side for iteration counts larger than two for both laptops (see tables 2(a), 2(b)). For the 2-iteration case there was a difference between Laptop 1 and Laptop 2: the Java side was faster on Laptop 2 but slower on Laptop 1, compared to the C side.

Measurement \ Iteration		1, 2	3	5 - 500
with timers				
	p	$< 2.2e-16$	$< 2.2e-16$	$< 2.2e-16$
	muDiff	0.0	100.0	100.0
without timer				
	p	$< 2.2e-16$	$< 1.523e-10$	$< 2.2e-16$
	muDiff	0.0	100.0	100.0

(a) Laptop 1

Measurement \ Iteration		1	2	3 - 500
with timers				
	p	$< 2.2e-16$	$< 2.2e-16$	$< 2.2e-16$
	muDiff	0.0	100.0	100.0
without timer				
	p	$< 2.2e-16$	$< 2.756e-14$	$< 2.2e-16$
	muDiff	0.0	100.0	100.0

(b) Laptop 2

Table 2: t-test results on execution times (p value is reported by t.test in R doing Welch Two Sample t-test, muDiff is the $>compval value of muDiff reported by the summary of BESTmcmc result)

We note that, although so far we have only aimed at providing the necessary functionality, not execution performance, according to our measurements our tool might be also useful in long-running testing scenarios to improve the runtime performance of tests.

Recall that both Java and C sides were built from the same TTCN-3 code, had to parse the same configuration file, run on the same machine and generated the same amount of log files, so the measured difference in speed could not be explained by a different workload on the abstraction level of executing the testcases. However, while the static compilation of the C side might have resulted in faster code loading-up, the Java side might have also needed to start the JVM and/or perform the first iteration in interpreted mode (adding the JIT compilation as an extra overhead) [18].

5.3.1 Some interesting facts and observations

1. We could see the largest standard deviation of the measured execution times in case of the 1-iteration executions. In case of Laptop 1, the worst 1-iteration execution times of the C side were similar to the best Java execution time for the same scenario.

2. In the case of 1-iteration on Laptop 2, there were no huge extremal values. The deviation increased having multiple iterations compared to Laptop 1 in case of "with timers" (Figure 3).

[18] One more technical difference was that the C side used flex+bison for parsing the configuration files, the Java side ANTLRv4. The startup time could also be explained by the different performance characteristics of LALR(1) parsing used by Bison and LL* used by ANTLRv4, but as the input in our case was very simple, this was unlikely.

3. We could see that the per iteration time on Laptop 2 was larger than on Laptop 1 for the C side, however, on Java side, they were very close to each other.

4. A 500-iteration long execution was executing 882,500 testcases.

5. A 500-iteration long execution generated a log file of 1.7GB[19].

6. The execution with timers was executing 1765, while the execution without timers executed 1757 testcases.

While in our research we did not do a deep analysis of why we observed such different performance curves we can still offer some intuitive reason that can explain them:

- In C/C++ the optimisations happen during build time, the built executable does not undergo changes during execution. As Java is a Just In Time compiled language, the performance of the application can be optimised during execution time, possibly leading to better performance over time.

- Parallel hardware architectures might also play a role. As the software architecture used by our industry partner is single threaded, on the C side memory deallocation has to be done by the same thread doing the business logic. In Java, even though the business logic is still single threaded, garbage collection can happen at any time after a resource is no longer in use delaying the cost of deallocation and the garbage collector itself might run on a CPU core different from the business logic, eliminating all work that CPU core would have to do to free up memory.

5.4 Build + Execution iterations

We have already presented our observations separately on the improvements in build time and the execution speeds on our test project. It is also interesting to take a look at these observations from the point of developer iterations, where we are interested in how long a single build + execute iteration takes. This is the duration developers need to wait from changing the source code, to being able to analyse the effects of the change.

[19] This cost was present on both sides during the execution.

For this we decided to check how long it would take for a build and execution to last, for 4 different build scenarios[20] and all iteration numbers measured so far, by adding together the average build time of the given build type and the average execution time of the given execution with a given iteration number, presented in the Appendix in Table B.

Please note that these are only theoretical durations, the real world durations could be much bigger. In the case of the incremental build and single iteration execution combination, on the Java side this operation is so fast (2,49 seconds) that simply the act of moving the mouse pointer after invoking the build to the toolbar of Eclipse to start the execution could substantially increase it. At the other end of the spectrum, full build and 500 iterations on the C side, we see approximately 53 minutes where the developer is bound to start doing something else either leaving their computer, or slowing it down with some other activities.

5.4.1 Some interesting facts and observations

1. For all build scenarios and iteration number combinations (and on both laptops) we observe that the Java side performs better. Having the shortest theoretical duration between a change by the developer and having the execution results ready to be analysed.

2. In case of full builds and executions with timers the improvement is at least approximately 140 seconds, increasing to approximately 535 seconds for the largest iteration number on Laptop 1 (respectively 88% to 16% improvement).

3. In the case of incremental builds and executions with timers, the improvement is at least approximately 33 seconds, increasing to approximately 424 seconds for the largest iteration number on Laptop 1 (respectively 82% to 13% improvement).

5.5 Scaling up for larger projects

In order to see if our solution would scale up for larger projects we asked for some help from our industry partner. They have provided us with another large scale project and one of their system architects could spend a day on providing us with some measurements on this project on a build server used by

[20] We leave out the incremental build on the Java side without the IDE mode, as the users will not be able to turn off IDE mode.

our industry partner. The aim in this exercise was purely to see if our solution would be able to scale up. We disclose both our numbers and the numbers we got from our industry partner, but note that at that point in time and implementation completeness, this was not a core part of our observations, only a look ahead. The 3 years of development was not enough for our group to reach full functional equivalence and for many of these numbers, we had to rely on the information given to us by our industry partner, where we had little control on the way the measurements were done.

The project used had 1275 modules (1143 TTCN-3 and 132 ASN.1), 1.24 million LOC at the time of measurements[21]. Our Java code generator generated 930 MBs of .java files from it. Since it was a large codebase, this project seemed to be appropriate to see if our method would work on large scale projects. There were 2 issues unique to this measurement, that we could not overcome for the amount of work needed:

1. At that time, our implemented compiler supported only the "RAW" encoding, while the examined project on the C side used several others. This means, that on the C side somewhat more code was generated. Approx. 18.93% of the code was used to support the other encodings.

2. To compile the project, we created dummy implementations (empty function bodies) of all required platform-dependent testports and external functions (proper implementation would have required a solid background on protocols and a large amount of time). The creation was made by hand and constitutes only a small amount of code compared to the 930MB generated codes. Hence, we didn't expect them to greatly impact the scaling of the build performance.

We performed 3 measurements for full build and 10 for incremental on Laptop 1, with the Java side only, as this project did not compile on Cygwin on the C side due to missing libraries.

The full build of the project took approximately 741 seconds (see Table 3). The time needed to incrementally re-build the project was approximately one to five seconds, depending on how many Java files had to be re-generated for the change. All 8 logical processors were heavily used during the builds.

To be able to compare build times of the C and Java sides of this project, a system architect of our industry partner (Eduard Czimbalmos) was asked to perform the measurements on one of their industrial build servers. The build

[21] More information is available on this project at [38, 39].

Scenario	build time
Laptop 1 full build	741.70
Laptop 1 incremental build (4 files are re-generated)	3.27
Laptop 1 incremental build (1 file is re-generated)	1.28

Table 3: Large industrial project build times on Laptop 1 (Java only, in seconds)

server characteristics used for the measurements were: Intel® Xeon® E5-2450v2 CPU @ 2.50Ghz, 64 GB RAM, running GCC version 4.8.5 (Ubuntu 4.8.5-4Ubuntu2) and Java HotSpot™ 64-Bit VM (build 25.201-B09) in server mode.

We received the numbers presented in Table 4 from our industry partner, who also told us that they were measuring a typical working scenario and there was no other task running on the build server during the measurements. Please note as these measurements were not done by our group, we had no direct control on the measurements and don't have more detail about them. These numbers should only be regarded as showing the possibility of scaling, not as exact figures.

Scenario	build time
Build Server full C build	1150
Build Server full Java build	640
Build Server incremental C build	61-130
Build Server incremental Java build	12

Table 4: Large industrial project build times on the Build Server (in seconds)

The full builds on the build server took 1150s on the C side and 640s on the Java side (see Table 4). The incremental build took approximately 61-130 seconds on the C side and 12 seconds on the Java side.

5.5.1 Some interesting facts and observations during the measurements

1. During the C build all 28 cores were used to almost 100% for the whole duration of the build (maximum or close to the maximum potential parallelisation of build).

2. During the Java build for most of the build time 1-3 cores were used and jumped to 4-8 cores only for a few seconds (as they could observe it using the output of `top` command).

3. On this project, doing a full build, the translation from TTCN-3 to C/C++ files took approx. 33 seconds, leaving 97% of the build on the C side coming from using C/C++ as the intermediate language.

5.5.2 Consequences

(1) While the C side required a 28 core build server for the development and build scenarios, the Java side might have made it possible to develop this project on a laptop; (2) the full build time of the Java side was half, the incremental build was 1/5th to 1/10th to the C side; (3) the scaling of the Java build in terms of build duration seemed to break in the many-core situations, as it was not able to take advantage of all cores to build faster; (4) at the same time, from the resource usage point of view, the Java side had about 50 times more efficient resource usage. While the C side put the build server under heavy load for approximately 20 minutes to build this project, using Java it might have been possible for approximately 20-28 developers to work on the same machine at the same time and still finish the full build in half the time.

6 Limitations

This paper presents a work that was an experiment with limited scope. We had a list of features not yet supported by our extension, compared to the industrial tool, but they were not used in the examples provided by our industry partner. It is also possible that there might be programming issues in the generated Java code. The C side had been in production and use for about 20 years, long enough for most issues to manifest. Although our extension was new, we were able to show functional equivalence (as far as the tests provided by our

industry partner go). This makes us confident to say that we don't expect issues resulting in larger architectural changes in our solution.

As a limitation, there might be problems with scaling. The Java class file format is known to have some strict limitations ([43]) that might cause further problems in the generated Java code for larger projects.

Due to the very specific context of our work, our results might not generalise to other systems. Compiling from TTCN-3 to Java in itself might be a limited context and measuring build times and execution times of test systems might also not be a priority for other companies with smaller test systems.

We followed the approach presented by Georges et al. ([22]) to make our performance evaluation statistically rigorous for the compilation and execution measurements done on Laptop 1 and 2, but not for the build server as we had limited and indirect access to it.

7 Threats to validity

During our measurements we had to turn on some debug printouts, which might have added some overhead, that will not be present during the normal usage of the tool.

We had limited control over Laptop 1, as it was used at our industry partner (the firewall software could not be stopped/deactivated/reconfigured for the duration of measurements, etc.). We tried to make sure that no resource intensive operation was running in the background while performing our measurements, but there is still a possibility of some background tasks specific to our industry partner's systems affecting our measurements in ways that might not be present when reproducing this research elsewhere.

There is also a question of fairness in our measurements: we do not have intimate understanding of the internal implementations of either the C/C++ compiler or the Java builder used. That is, in our experiments we tried to reach the best possible performance on both sides. On the C side we used "-O2" to get optimised binaries and "-j8" to use all processing power available in our laptops. On the Java side, in Eclipse, we could not find such options, but we made sure to give enough memory for the build process to minimise the need for garbage collection.

Also at the time our work was accepted by our industrial partner we had limited access to their internal test systems: information on how certain equipment is being tested could raise potential security concerns.

As we had no control over the measurements our industry partner did, checking whether our solution is able to scale might not be perfectly valid and reproducible. However, our industry partner used the information gained from those measurements, and decided that our solution can be turned into an industrial product, we believe that it was in their best interest to gain valid and reliable information.

8 Summary

In this paper we presented our work and empirical observations on extending an industrial compiler to support a new intermediate language.

We have shown how we created an alternative of the existing build procedures of our industry partner by creating an extension that uses Java as the intermediate language besides the already supported C/C++. We described how much of the architecture could be kept to proceed fast and how we overcame the difficulties coming from the differences of the languages.

Our contribution is unique in the sense that both the build and the execution used the same source code (as input) written in the same input language, generated intermediate codes with the same abstract hierarchy, built into executables that were configured using the same files, performed the same behaviour on the same machine and generated the same logs. Evidenced by the test files provided to us by our industry partner, selected from their regression tests, to decide if our solution could be functionally equivalent to their existing system.

Our measurements showed that using Java as an intermediate language might improve build times significantly, providing users with better development iteration times, that might lead to improved productivity. While C/C++ still has better performance in some situations, at least in longer testing scenarios, Java can be a viable alternative as an intermediate language.

In contrast to De Marco et al. ([8]), our development from scratch can be considered a successful migration as we couldn't identify any performance bottleneck and is more maintainable since the knowledge was transferred by programmers using Java idioms where it was possible without changing the architecture. Our observations both supported and contradicted the findings of Nanz and Furia ([31]) and Furia et al. ([21]). Our 1-iteration measurements supported that the same input compiled to C/C++ could usually execute faster compared to Java (although at the cost of much longer compilation). But our measurements with larger iteration counts showed a reversing situ-

ation, where compiling to Java might lead to executables that finish faster. Adding the overhead of the build to the execution time (to measure build-execute workflows) would, in our situation, lead to Java finishing earlier in all investigated situations. As such in contrast to Taboada et al. ([41]), we found that for us, Java was a good alternative also from a performance point of view.

Thanks to the help of our industry partner we could check (using one of the largest test systems of our industry partner) that our method scaled up well and kept its benefits, even on large scale test systems. This will enable our industry partner to provide a unique solution for their customers. The developers of these large industrial test systems will be able to use the Java-based build system to work from an IDE with fast feedback cycles (eliminating the negative effect on productivity coming from the C sides build times). At the same time, they will also be able to build the same test systems using the C/C++ based build system for situations where the Java-based executable would not be able to provide the necessary runtime performance.

To support the reproduction of the results of this study all information was made publicly available. The code of the toolset is part of the open-source Titan ([9]) toolset (the Eclipse plugins), and the tests used for testing compilation and execution speed are available at [19].

9 Further work

Even with so limited time we could reach a point where we can confidently say that it is desirable to continue this work.

First of all, there were several features still missing from our solution, that were present in Titan already. We hope they will be added by our industry partner while turning our solution into an industrial product. So that their users can enjoy the benefit of this research.

As described in section 3.2, the 3 years of development was only enough to reach the level of functional equivalence our industry partner required from us. As a follow-up research, it would be desirable to investigate opportunities for performance optimisations in both the runtime and build time. For example: we generated the Java files from TTCN-3 files sequentially. In theory, all of these files could be generated at the same time, in parallel, leading to further build time improvements.

This analysis could be extended to involve more and different hardware into the research and go into more detail on how different hardware features impact performance. It would also be interesting to see how different compiler

versions (GCC, Java, etc...) perform on the same source codes in build time and execution performance. For us, the target platforms we were aiming to support were given by our industry partner.

It would be interesting to look into how to improve CPU utilisation on massively multi-core CPUs. According to our industry partner, the build on the Java side did not seem to use all cores to provide maximum build performance.

It might also be a good idea to think of our solution as a platform for further research. A side effect of our solution is that we are transforming large scale industrial and standardised test codes, that are in active use, into a large amount of Java code. This could allow researchers to experiment with how different Java features and usage patterns impact both compilation and runtime performance. Such an experiment would just need to change how our solution translates TTCN-3 code into Java code to have a large amount of source code, from complex and in use systems, for measurements.

Acknowledgements

The authors would like to thank the Test Competence Center of Ericsson Hungary for the financial support of this research and for providing access to their in-house tools. These proved to be invaluable to our measurements.

We would also like to thank Andrea Pálfi, Gergő Újhelyi, Árpad Lovassy, Attila Balaskó and the Titan team for their help in the implementation and measurements. We would also like to thank Eduárd Czimbalmos for his help in our measurements and his feedbacks on the tool.

The research of the first author was supported by the ELTE-Ericsson Laboratory, for the third author by the Project no. TKP2020-NKA-06 (Application domain specific highly reliable IT solutions) with the support from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme.

References

[1] R. Abdalkareem, S. Mujahid, E. Shihab, A machine learning approach to improve the detection of ci skip commits, *IEEE Transactions on Software Engineering*, pp. 1–1, 2020. ⇒140

[2] B. Amedro, V. Bodnartchouk, D. Caromel, C. Delbe, F. Huet, G. L. Taboada, Current State of Java for HPC, Technical Report RT-0353, *INRIA*, 2008. [accessed Apr-2020] ⇒141

[3] ARMOUR, Test generation strategies for large-scale IoT security testing – v1, 2016. [accessed Apr-2020] ⇒135

[4] A. Avram, IDC Study: How Many Software Developers Are Out There?, 2014. ⇒135

[5] N. Bartha, *Scalability on IT projects*, Master's thesis, 2016. ⇒135

[6] L. Batyuk, A.-D. Schmidt, H.-G. Schmidt, A. Camtepe, S. Albayrak, Developing and Benchmarking Native Linux Applications on Android, in J.-M. Bonnin, C. Giannelli, T. Magedanz, *MobileWireless Middleware, Operating Systems, and Applications*, volume **7**, pages 381–392, Berlin, Heidelberg, 2009. Springer. ⇒140

[7] R. P. Cook, An OpenMP library for Java, In *2013 Proceedings of IEEE Southeastcon*, pp. 1–6, April 2013. ⇒141

[8] A.De Marco, V. Iancu, I. Asinofsky, COBOL to Java and Newspapers Still Get Delivered, in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 583–586, Sep. 2018. ⇒140, 143, 166

[9] Ericsson Telecom AB., Source code of titan, version 6.6.0. `https://github.com/eclipse/titan.EclipsePlug-ins/releases/tag/6.6.0`, 2019. [accessed Apr-2020]. ⇒167

[10] Ericsson Telecom AB., Programmers' Technical Reference Guide for the Java side of the TITAN TTCN-3 Toolset, `https://github.com/eclipse/titan.core/blob/master/usrguide/java_referenceguide/JavaReferenceGuide.adoc`, 2020. [accessed Apr-2020]. ⇒142, 149

[11] Ericsson Telecom AB., Programmers' Technical Reference Guide for the TITAN TTCN-3 Toolset, `https://github.com/eclipse/titan.core/blob/master/usrguide/referenceguide/ReferenceGuide.adoc`, 2020. [accessed Apr-2020]. ⇒142, 149

[12] ETSI, Methods for Testing and Specification (MTS);The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI), `https://www.etsi.org/deliver/etsi_es/201800_201899/20187305/04.08.01_60/es_20187305v040801p.pdf`, 2017. [accessed Apr-2020.] ⇒137

[13] ETSI, 3GPP test suites, `http://www.ttcn-3.org/index.php/downloads/publicts/publicts-3gpp`, 2020. [accessed Apr-2020]. ⇒135

[14] ETSI, 5G;5GS; User Equipment (UE) conformance specification; Part 3: Protocol Test Suites, `https://www.etsi.org/deliver/etsi_ts/138500_138599/13852303/15.00.00_60/ts_13852303v150000p.pdf`, 2020. [accessed Apr-2020]. ⇒135

[15] ETSI, Intelligent Transport Systems (ITS) Test Suites, `http://www.ttcn-3.org/index.php/downloads/publicts/publicts-etsi/65-publicts-its`, 2020. [accessed Apr-2020]. ⇒135

[16] ETSI, Methods for Testing and Specification (MTS);The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, `https://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.12.01_60/es_20187301v041201p.pdf`, 2020, [accessed Oct-2020]. ⇒135, 172

[17] ETSI, Methods for Testing and Specification (MTS);The Testing and

Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI), https://www.etsi.org/deliver/etsi_es/201800_201899/20187306/04.12.01_60/es_20187306v041201p.pdf, 2020, [accessed Oct-2020]. ⇒137

[18] I. I. Farkas, K. Szabados, A. Kovács, Measurement data with configuration, http://compalg.inf.elte.hu/~attila/materials/Measurements_Laptop1.xlsx, http://compalg.inf.elte.hu/~attila/materials/Measurements_Laptop2.xlsx, 2019. ⇒157

[19] I. I. Farkas, K. Szabados, A. Kovács, *Regression test data*, http://compalg.inf.elte.hu/~attila/materials/RegressionTestSmall_20190724.zip, 2019. ⇒149, 151, 152, 167

[20] I. I. Farkas, K. Szabados, A. Kovács, An example containing a "Hello World", some simple types in TTCN-3, and the compiled C/C++ and Java codes, http://compalg.inf.elte.hu/~attila/materials/Example_package.zip, 2020. ⇒142, 149

[21] C. A. Furia, R. Feldt, R. Torkar, Bayesian Data Analysis in Empirical Software Engineering Research, *IEEE Transactions on Software Engineering*, pp. 1–1, 2019. ⇒141, 166

[22] A. Georges, D.Buytaert, L. Eeckhout, Statistically Rigorous Java Performance Evaluation, *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pp. 57–76, New York, NY, USA, 2007. ACM. ⇒152, 154, 165

[23] T. A. Ghaleb, D. A. da Costa, Y. Zou, An empirical study of the long duration of continuous integration builds, *Empirical Software Engineering*, 24(4):2102–2139, Aug 2019. ⇒135, 140

[24] L. Gherardi, D. Brugali, D. Comotti, A Java vs. C++ Performance Evaluation: A 3D Modeling Benchmark, I. Noda, N. Ando, D. Brugali, J. J. Kuffner, editors, Simulation, Modeling, and Programming for Autonomous Robots, pp. 161–172, Berlin, Heidelberg, 2012. Springer. ⇒141

[25] IoTKETI, oneM2MTester, https://github.com/IoTKETI/oneM2MTester, 2016. Last visited: April, 2020. ⇒135

[26] J. Nielsen, Response times: The 3 important limits https://www.nngroup.com/articles/response-times-3-important-limits/, 1993. Last visited: October, 2020. ⇒155

[27] A. Kovács, K. Szabados, Test software quality issues and connections to international standards, *Acta Universitatis Sapientiae, Informatica*, **5**, pp. 77–102, 05 2013. ⇒135

[28] A. Kovács, K. Szabados, Advanced TTCN-3 Test Suite validation with Titan, *In Proceedings of the 9th International Conference on Applied Informatics*, volume **2**, pp. 273–281, 02 2014. ⇒135

[29] P. Lathan, S. Burke, K. Gallick, A. Coleman, New cpu test methodology 2020: Code compile, updated gaming, transcoding, & more https://www.youtube.com/watch?v=sg9WgwIkhvU, 2020, [accessed May-2020]. ⇒142

[30] M. Meredith, J. Kruschke, Bayesian Estimation Supersedes the t-Test, https://cran.r-project.org/web/packages/BEST/vignettes/BEST.pdf, 2018, [accessed Apr-2020]. ⇒158

[31] S. Nanz, C. A. Furia, A Comparative Study of Programming Languages in Rosetta Code, *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume **1**, pp. 778–788, May 2015. ⇒141, 166

[32] P. Oláh. *Improving the semantic analysis in TTCN-3 environment*, Master's thesis, *Eötvös Loránd University, Budapest*, Hungary, 2016. ⇒150

[33] K. Reinholtz, *Java will be faster than C++*, SIGPLAN Not., **35**(2):25–28, Feb. 2000. ⇒140

[34] A. Ruano, G. Réthy, Developing an Open Source conformance testing environment for ITS communications, UCAAT, 2016. ⇒135

[35] S. Salmons, M. Arnaud, X. Zeitoun, C. Bouattour, *Model-based platform for smart grid interoperability testing using TTCN-3* In UCAAT, 2018. ⇒135

[36] K. Szabados, *Structural Analysis of Large TTCN-3 Projects* In M. Núñez, P. Baker, M. G. Merayo, editors, Testing of Software and Communication Systems, pages 241–246, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ⇒135

[37] K. Szabados, A. Kovács. *Technical debt of standardized test software* In 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), pages 57–60, Bremen, Oct 2015. ⇒135

[38] K. Szabados, A. Kovács, *Internal quality evolution of a large test system–an industrial study* In Acta Universitatis Sapientiae, Informatica, 8(2):216–240, 12 2016. ⇒135, 162

[39] K. Szabados, A. Kovács, G. Jenei, D. Góbor, *Titanium: Visualization of TTCN-3 system architecture* In 2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), pages 7–11, May 2016. ⇒135, 162

[40] J. Szabó, T. Csöndes, *TITAN, TTCN-3 test execution environment*, https://www.hiradastechnika.hu/data/upload/file/2007/2007_1a/HT_0701a-6.pdf, 2007. Last visited: October, 2020. ⇒136, 143

[41] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, R. Doallo. *Java in the High Performance Computing arena: Research, practice and experience*, Science of Computer Programming, 78(5):425 – 444, 2013. Special section: Principles and Practice of Programming in Java 2009/2010 & Special section: Self-Organizing Coordination. ⇒141, 167

[42] U.S. General Services Administration (GSA) Technology Transformation Service, *Interaction Design Basics*, https://www.usability.gov/what-and-why/interaction-design.html, 2020. Last visited: October, 2020. ⇒155

[43] ∗ ∗ ∗. *Chapter 4. The class File Format*, https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html, 2020. last visited: April, 2020. ⇒149, 165

[44] ∗ ∗ ∗. Cygwin, 2020. last visited: January, 2020. ⇒152

[45] ∗ ∗ ∗. Eclipse IoT-Testware, 2020. Last visited: April, 2020. ⇒135

[46] ∗ ∗ ∗. Titan, 2020. last visited: January, 2020. ⇒152

Appendices

A Short introduction to TTCN-3

TTCN-3[22] is a high level standardised language ([16]) designed for testing. Mostly used for functional testing (conformance testing, function testing, integration, verification, end-to-end and network integration testing) and performance testing. TTCN-3 can be used (1) to test reactive systems via: message based communication, (2) API based and analog interfaces and systems.

The language is governed by a strict, internationally accepted specification. Each language construct, allowed by the syntax and semantics of the standard, has a well specified behaviour. Tests written in TTCN-3 can be transferred to other vendor's tools without modification. Some standards of reactive systems (for example communication protocols) offer their specifications together with a set of tests written in TTCN-3. This provides an easy and automated way for tool vendors and users to check the conformance of the implementation.

TTCN-3 offers platform independent abstract data types (see listing 1). There is no value range restriction for integers, no precision restriction for floats, and no length restriction for string types. String types are differentiated based on their contents (bitstring, hexstring, octetstring, charstring, universal charstring). Creating new types is supported by building structured types with fields (record, set) or by list of an element type (record of, set of). It is also possible to create new types with restriction (for example length restriction on strings). This rich type / data constructs can easily be extended by importing other data types / schema (ASN.1[23], IDL[24], XSD[25] and JSON[26]) without need for manual conversion.

The templates of TTCN-3 merge the notions of test data and test data matching into one concept (see listing 2). This enables the specification of expected responses in a concise way. Matching rules can be for example: single value ("Budapest"), list of alternatives ("Monday", "Tuesday"), range (1 .. 5), ordered and unordered lists of values, sub- and supersets of unordered values, string patterns (pattern"* chapter"), permutations of values. When declaring templates for structured data types, these matching rules can be declared for each field and element individually or for the whole tem-

[22] Test and Test Control Notation 3
[23] Abstract Syntax Notation One
[24] Interface Definition Language
[25] XML Schema Definition
[26] JavaScript Object Notation

Listing 1: data types example

```
var boolean v_boolean:=true;
const integer c_i:= 123456789101112131415;
const float c_f1:=1E2;
const float c_f2:=100.0;
var bitstring v_bits:='01101'B;
var charstring v_chars:="ABCD";
var hexstring v_hexs := '01A'H;
var octetstring v_octs:='0BF2'O;
var universal charstring v_uchars := "F" & char(0, 0, 0, 65)

type record recordOper_trecord {
        integer x1 optional,
        float x2 };
type record of octetstring recordOper_trecof;
type set recordOper_tset {
        integer x1,
        float x2 optional };
type set of charstring recordOper_tsetof;

type integer templateInt_subtype (0..1457664);
type record length (3)
  of record length (3)
  of record length (3) of integer threeD;
```

Listing 2: templates example

```
template integer t_i := 123456789101112131415
var template float vt_f := (1.0 .. 2.0);
template mycstr t_mycstr := pattern "ab" & "cd";

template templateCharstr_rec templateCharstr_tList :={
        x1:="00AA", //specific value
        x2:=("01AA","01AB","11AC"), // value list
        x3:=complement ("11","0A","1BC0"), // complement list
        x4:=? length(2..4), //any string with a length of 2 to 4
        x5:= pattern "10*" //any string matching the pattern
};
```

Listing 3: Example for receiving message

```
testcase tc_HelloWorld() runs on MTCType system MTCType
{
  timer TL_T := 15.0;
  map(mtc:MyPCO_PT, system:MyPCO_PT);
  MyPCO_PT.send("Hello, world!");
  TL_T.start;
  alt { //branching based on events
    [] MyPCO_PT.receive("Hello, TTCN-3!") {
        TL_T.stop;
        setverdict(pass);//receiving the right message
       }
    [] TL_T.timeout {
        setverdict(inconc); // the test timed out
       }
    [] MyPCO_PT.receive {
        TL_T.stop; // some other message was received
        setverdict(fail);
       }
  }
}
```

plate. Checking whether a data value matches to the template is as easy as "match(value, templateValue)". Other constructs offer additional functionality, e.g. "∗.receive(templateValue) → value" activates only if a value matching to the provided template is received, in which case the value of the message is saved in "value" for further processing.

TTCN-3 can also be viewed as a "C -like" procedural language with testing specific extensions. The usual programming language features (function, if, while, for, etc.) are extended with other constructs needed for testing: test cases as standalone constructs, sending/receiving messages, invoking remote procedures and checking the content of the received data structures (messages/results/exceptions), alternative behaviors depending on the response of the tested entity, handling timers and timeouts, verdict assignment and tracking, logging of events (see listing 3) are all built in.

Creating distributed test cases and test execution logic is easy as well. A TTCN- 3 test may consist of several parallel test components which are distributed on a set of physical machines, able to work in tandem to test all interfaces of the tested system, or able to create high load. Test components, communication ports to the tested entity and to other test components are

defined in TTCN-3. The number of test component instances and their connections are controlled from the code of the test case dynamically using various language features (see listing 4). Deploying and controlling the test component also happens in an abstract and platform independent way. The user does not need to work with the implementation details. It is the tools responsibility to utilize the available pool of machines, possibly running on different operating systems.

Listing 4: multiple components example

```
testcase commMessageValue() runs on commMessage_comp2 {
var commMessage_comp1 comp[5];
var integer xxint;
for (var integer i:=0; i<5; i:=i+1)
{ log(i);
  comp[i]:=commMessage_comp1.create;//creating component
  comp[i].start(commMessage_behav1(i));//start remote behavior
  connect(self:Port2[i],comp[i]:Port1);//connect to component
  xxint:=5;
  Port2[i].send(xxint);//send message on port
  Port2[i].receive(integer:?) -> value xxint;//receive response
  if (xxint==5+i) {setverdict(pass)}
    else {setverdict(fail)};
}
for (i:=0; i<5; i:=i+1) {comp[i].stop};//stop the components
};
```

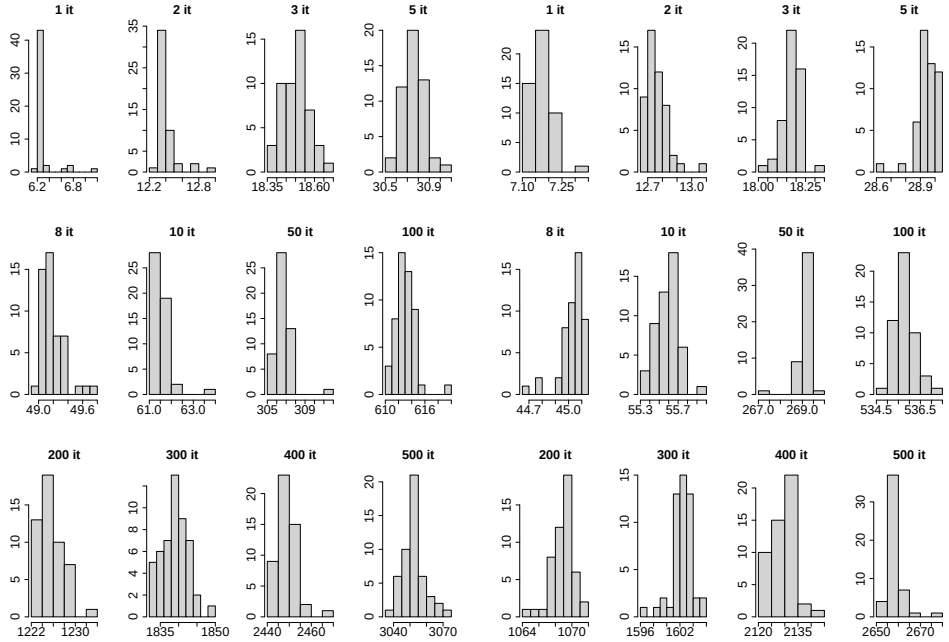
TTCN-3 is also independent from the test environment. The user needs only to define abstract messages exchanged between the test system and test tested entity. Message encoding (serialization), decoding (de-serialization), handling of connections and transport layers are done by the tools.

TTCN-3 also offers to control the test case execution logic and dynamic test selection from within the the TTCN-3 code itself (see listing 5). Module parameters allow for the user to leave data open in the source code and provide the actual values at execution time (IP addresses, IDs, passwords, etc...)

Listing 5: execution control example

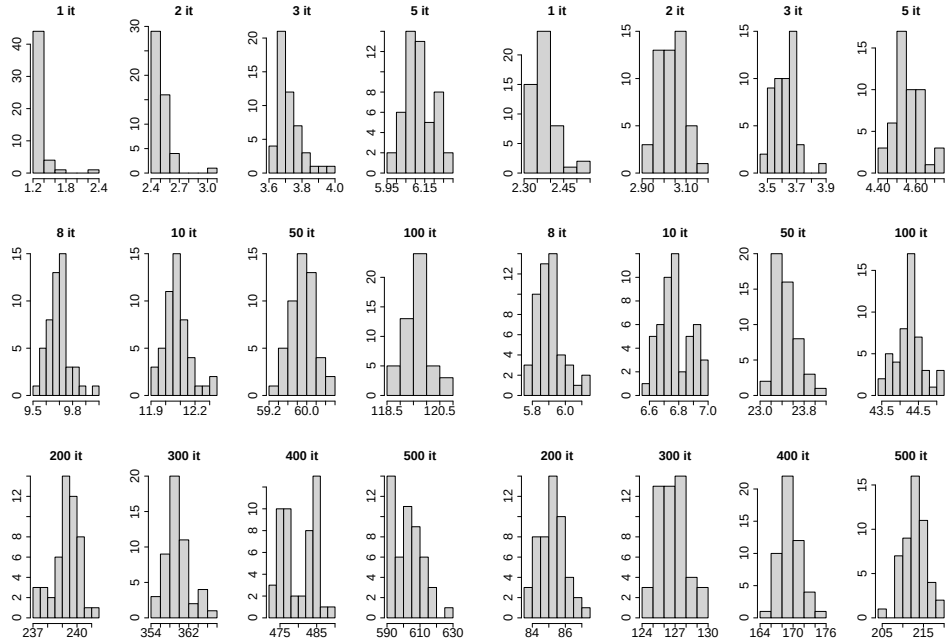
```
control {
  for(var integer i := 0; i < 10; i := i+1)
  {
    execute(parameterised_testcase(i));
  }
  execute(transferTest());
  execute(tc_runsonself());
 }
```

B Measurement details and histograms



(a) C side with timers

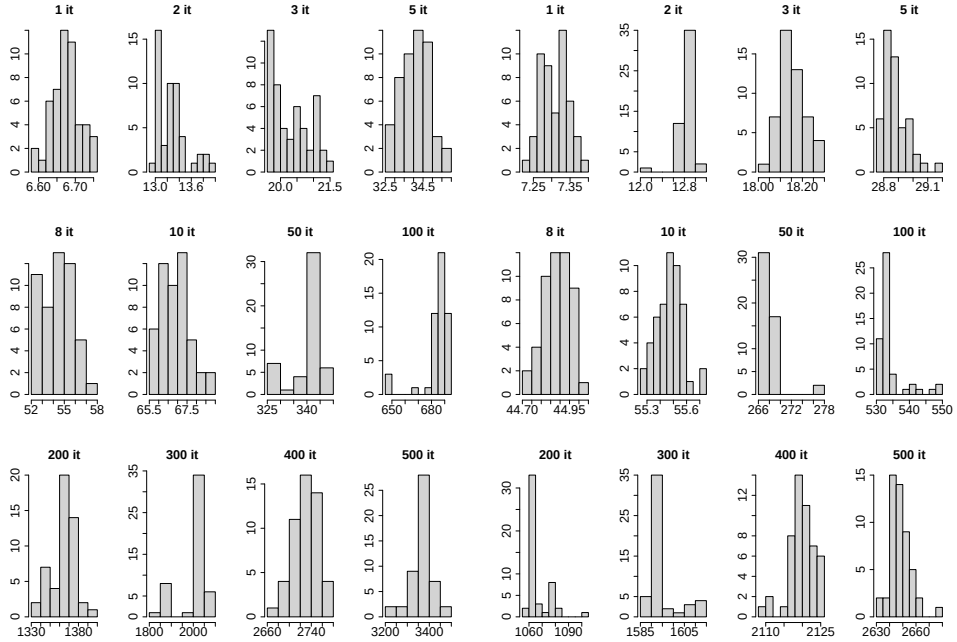
(b) Java side with timers



(c) C side without timers

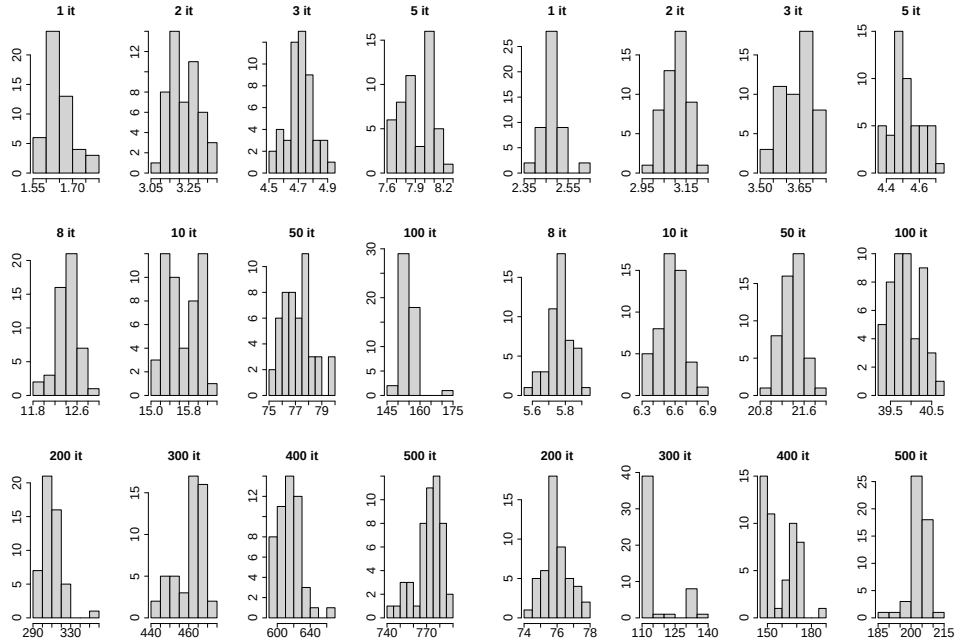
(d) Java side without timers

Figure 5: Histogram of the measured execution times on Laptop 1. C (5(a)) and Java (5(b)) side with timers; C (5(c)) and Java (5(d)) side without timers. On each histogram the horizontal axis shows the execution times, the vertical axis the number of their occurrences.



(a) C side with timers

(b) Java side with timers



(c) C side without timers

(d) Java side without timers

Figure 6: Histogram of the measured execution times on Laptop 2. C (6(a)) and Java (6(b)) side with timers; C (6(c)) and Java (6(d)) side without timers. On each histogram the horizontal axis shows the different execution times, the vertical axis the number of their occurrences.

it. nr	C + full	Java + full	C + inc	Java + IDE m.
1	159.69	18.13	40.80	7.29
2	165.81	23.71	46.92	12.88
3	171.90	29.13	53.02	18.30
5	184.16	39.90	65.27	29.07
8	202.58	55.99	83.69	45.15
10	214.95	66.55	96.06	55.71
50	460.13	280.08	342.24	269.24
100	766.49	546.76	647.60	535.93
200	1378.82	1079.79	1259.93	1068.95
300	1992.45	1613.31	1873.56	1602.47
400	2602.08	2140.54	2483.19	2129.70
500	3205.31	2669.37	3086.42	2658.53

(a) Laptop 1 with timers

it. nr	C + full	Java + full	C + inc	Java + IDE m.
1	154.75	13.33	35.86	2.49
2	155.92	13.99	37.03	3.15
3	157.12	14.57	38.23	3.74
5	159.52	15.51	40.63	4.67
8	163.09	16.86	44.20	6.02
10	165.47	17.73	46.58	6.90
50	213.30	34.43	94.41	23.60
100	273.09	55.22	154.21	44.38
200	392.69	96.13	273.80	85.29
300	513.03	137.67	394.14	126.83
400	633.35	180.51	514.46	169.67
500	756.39	223.90	637.50	213.06

(b) Laptop 1 without timers

it. nr	C + full	Java + full	C + inc	Java + IDE m.
1	231.20	35.44	32.40	7.61
2	237.80	40.97	39.00	13.14
3	244.84	46.28	46.04	18.46
5	258.65	57.01	59.85	29.19
8	278.94	73.01	80.14	45.19
10	291.41	83.60	92.61	55.78
50	565.01	296.07	366.21	268.24
100	684.41	562.35	710.14	534.53
200	1364.92	1094.89	1390.65	1067.07
300	2228.21	1623.23	2029.41	1595.41
400	2953.54	2147.90	2754.74	2120.07
500	3590.27	2676.44	3391.47	2648.62

(c) Laptop 2 with timers

it. nr	C + full	Java + full	C + inc	Java + IDE m.
1	226.18	30.66	27.38	2.78
2	227.75	31.29	28.95	3.41
3	229.24	31.83	30.44	3.94
5	232.45	32.70	33.65	4.82
8	236.94	33.95	38.14	6.07
10	240.20	34.75	41.40	6.87
50	301.68	49.57	102.88	21.69
100	379.07	68.06	180.27	40.18
200	534.88	104.07	336.08	76.19
300	685.23	144.76	486.43	116.87
400	838.43	186.96	639.63	159.08
500	996.47	231.90	797.67	204.02

(d) Laptop 2 without timers

Table 5: The sums of the average build times of the build kinds (already presented in Table 1) and the averages of execution times for different iteration numbers on Laptop 1 with 5(a) and without timers 5(b), on Laptop 2 with 5(c) and without timers 5(d) in seconds (also already presented on Figures 3 and 4). The full postfix of the table means a full build, the inc postfix means the incremental build and the inc means the incremental build in IDE mode.

Received: April 18, 2021 • Revised: June 1, 2021