OpenSUT Requirements

VERSION:	v0.1
CLASSIFICATION:	Internal

Table of contents

This document contains all OpenSUT requirements, and was created with the help of StrictDocs. For text formatting, the reStructuredText markup syntax is supported, see the RST cheatsheet.

If you are new to requirement writing, we recommend reading first the *21 Top Engineering Tips for writing an exceptionally clear requirements document* from QRA (available as PDF here), then refer to *NASA's checklist for writing good requirements* here.

The OpenSUT requirements are split into different sections and subsections. Each requirement has its section number (e.g. *4.1.1.2 Actuation Logic*) and its Unique Identifier (UID). The section number is used only in this document, the UID guaranteed to be globally unique. On top of UID, we also use StrictDocs' MID. For requirement tracing and coverage measurement, we use primarily the UID.

Requirements that use the word *shall* are binding and must be satisfied, while requirements using the word *should* are non-binding, and can be considered optional or nice-to-have. A *rationale* is provided when appropriate for a given requirement.

1. SoW Requirements

UID: SECTION-OR-SoW-Requirements

Derived from the Statement of Work for the purpose of tracing the individual tasks and issues back to the SoW.

1.1. OpenSUT Platform

UID: SECTION-OR-OpenSUT-Platform

Task **TA2.1.1**

1.1.1. Task TA2.1.1.A

UID:

TA2-REQ-42

STATUS:

Completed

STATEMENT:

Develop the Open SUT primarily using existing components and specifications, including:

- Flight Controller
- AutoPilot
- Secure boot
- Mission Key Management
- Mission Protection System components.

Port a subset of OpenSUT components to run in a pKVM guest VM.

1.1.2. Task TA2.1.1.B

UID:

TA2-REQ-44

STATUS:

Completed

STATEMENT:

Specify entire OpenSUT architecture with SysML, and AADL.

1.1.3. Task TA2.1.1.C

UID:

TA2-REQ-45

STATUS:

Completed

STATEMENT:

Develop VERSE Toolchain specifications for components with rich code-level specifications.

1.2. Apply VERSE to Open SUT

UID: SECTION-OR-Apply-VERSE-to-Open-SUT

Task **TA2.1.2**

1.2.1. Task TA2.1.2.A

UID:

TA2-REQ-46

STATUS:

Completed

STATEMENT:

Build assurance case for the Open SUT.

1.2.2. Task TA2.1.2.B

UID:

TA2-REQ-47

STATUS:

Completed

STATEMENT:

Apply Verse Development Environment (VDE) to provide qualitative and quantitative feedback.

1.2.3. Task TA2.1.2.C

UID:

TA2-REQ-48

TAGS:

Completed

STATEMENT:

Define system deltas for program evolution and evaluation.

1.2.4. Task TA2.1.2.D

UID:

TA2-REQ-49

TAGS:

Completed

STATEMENT:

Support two Phase 1 continuous integration events.

2. OpenSUT Code Requirements

UID: SECTION-OR-Code-requirements

In this section we list requirements about the overall OpenSUT code, its structure, coverage and format.

2.1. No undefined behavior

UID:

TA2-REQ-16

STATUS:

Completed

CHILDREN:

→ **TA2-REQ-51** Check for undefined behavior

STATEMENT:

OpenSUT shall not contain any C code with undefined behavior, as defined by Cerberus semantics.

RATIONALE:

An example of undefined behavior include division by zero, out of bounds array access, integer overflow and null pointer dereference.

COMMENT:

This is only valid for the verified application code.

2.2. MISRA-C compliant code

UID:

TA2-REQ-17

STATEMENT:

OpenSUT application code should be MISRA-C compliant.

RATIONALE:

It is acceptable to choose only a subset of MISRA-C, such that it is supported by open-source tools, or regular IDEs (such as CLion).

3. OpenSUT Scenario Requirements

UID: SECTION-OR-OpenSUT-Scenario-Requirements

Requirements related to each OpenSUT scenarios.

3.1. Boot OpenSUT to a known initial state

UID: SECTION-OR-Boot-OpenSUT-to-a-known-initial-state

In this scenario, one or more components of OpenSUT boot using SHAVE Trusted Boot. It means that the application code is measured, hashed, and compared against an expected measure. Only if these values match, the application code is started and the measure is stored in the memory. If they don't match, an error is thrown, the boot is aborted and an error message is possibly sent and logged. If the attestation of each securely booted component passes, the system will be in a known initial state, fully provisioned. Measured boot ensures that only the expected code is running on OpenSUT.

The code is measured either with SHA256 or with quantum safe eXtended Merkle Signature Scheme (XMSS).

For the purpose of this scenario, we assume that each host computer contains a root of trust, a trusted boot that can bring up the hypervisor. In other words, we assume the host OS to be trusted (see the Threat model). Because hardware root of trust, trusted boot and attestation are all complex topics, only the application code will be attested in this scenario.

3.1.1. Signature of application code image

UID:

TA2-REQ-20

STATUS:

Completed

PARENTS:

← **TA2-REQ-19** Secure booting only the application code

CHILDREN:

→ **TA2-REQ-59** Binary code measurement

STATEMENT:

Each application disk image shall contain a digital signature that can be verified by the secure boot.

3.1.2. Secure booting only the application code

UID:

TA2-REQ-19

STATUS:

Completed

PARENTS:

← **TA2-REQ-18** Explicit assumptions

CHILDREN:

→ **TA2-REQ-20** Signature of application code image
→ **TA2-REQ-54** Known initial state

STATEMENT:

Secure boot shall be used to boot only the application code, and only on a subset of OpenSUT components.

RATIONALE:

This simplification is consistent with out threat model. Demonstrating Secure Boot only on a subset of components is sufficient.

3.1.3. Explicit assumptions

UID:

TA2-REQ-18

STATUS:

Completed

CHILDREN:

→ **TA2-REQ-19** Secure booting only the application code

STATEMENT:

In the provided documentation, explicitly list the assumptions and limitations of OpenSUT, such as:

- this is a contrived example
- true secure boot is not possible unless a *chain of trust* going all the way down to *Hardware Root of Trust* is maintained
- in real system a true *Hardware Security Module* (HSM) - such as the one developed on SEASHIP needs to be deployed on each Host computer, and shared with the guests

4. Component Requirements

UID: SECTION-OR-Component-Requirements

Component specific requirements are located in this section

4.1. Mission Protection System (MPS) Requirements

UID: SECTION-OR-Mission-Protection-System-Requirements

An engine protection system, that is redundant, measures engine temperature, and fuel pressure, and shuts down the engine if unsafe values are detected.

The system is connected to two temperature sensors and two fuel pressure sensors. The system has a control interface that allows the user to enter the maintenance mode, and adjust setpoints and trip channels. This control interface is available via a serial console (UART), and as such can be accessed only when the platform is not in operation (imagine the UART port being hidden behind a body panel).

4.1.1. MPS Architectural Requirements

UID: SECTION-OR-MPS-Architectural-Requirements

4.1.1.1. Four instrumentation channels

UID:

TA2-REQ-40

STATUS:

Completed

STATEMENT:

MPS shall have four redundant divisions of instrumentation, each containing identical designs, with two instrumentation channels (Fuel Pressure and Temperature), each channel containing:

A. Sensor
B. Data acquisition and filtering
C. Setpoint comparison for trip generation
D. Trip output signal generation

4.1.1.2. Actuation logic

UID:

TA2-REQ-41

STATUS:

Completed

STATEMENT:

MPS shall have two trains of actuation logic, each containing identical designs:

A. Two-out-of-four coincidence logic of like trip signals
B. Logic to actuate a first device based on an OR of two instrumentation coincidence signals
C. Logic to actuate a second device based on the remaining instrumentation coincidence signal

4.1.2. MPS Functional Requirements

UID: SECTION-OR-MPS-Functional-Requirements

4.1.2.1. High pressure trip

UID:

TA2-REQ-27

STATUS:

Completed

STATEMENT:

MPS shall Trip on high fuel pressure (sensor to actuation)

4.1.2.2. High temperature trip

UID:

TA2-REQ-28

STATUS:

Completed

STATEMENT:

MPS shall Trip on high temperature (sensor to actuation)

4.1.2.3. Voting logic

UID:

TA2-REQ-29

STATUS:

Completed

STATEMENT:

MPS shall Vote on like trips using two-out-of-four coincidence

4.1.2.4. Automatic actuation

UID:

TA2-REQ-30

TAGS:

Completed

STATEMENT:

MPS shall Automatically actuate devices.

4.1.2.5. Manual actuation

UID:

TA2-REQ-31

STATUS:

Completed

STATEMENT:

MPS shall Manually actuate each device upon receiving a user command.

COMMENT:

This command was received over UART, after the second change event the command is received over a socket.

4.1.2.6. Operating modes

UID:

TA2-REQ-32

STATUS:

Completed

STATEMENT:

MPS shall Select mutually exclusive maintenance and normal operating modes on a per division basis.

4.1.2.7. Setpoint adjustment

UID:

TA2-REQ-33

STATUS:

Completed

STATEMENT:

MPS shall Perform setpoint adjustment in maintenance mode.

4.1.2.8. Maintenance mode bypass

UID:

TA2-REQ-34

STATUS:

Completed

STATEMENT:

MPS shall Configure the system in maintenance mode to bypass an instrument channel (prevent it from generating a corresponding active trip output).

4.1.2.9. Maintenance mode forced trip

UID:

TA2-REQ-35

STATUS:

Completed

STATEMENT:

MPS shall Configure the system in maintenance mode to force an instrument channel to an active trip output state.

4.1.2.10. Variables displayed

UID:

TA2-REQ-36

STATUS:

Completed

STATEMENT:

MPS shall Display fuel pressure, and engine temperature.

4.1.2.11. Trip state displayed

UID:

TA2-REQ-37

STATUS:

Completed

STATEMENT:

MPS shall Display each trip output signal state.

4.1.2.12. Bypass indication display

UID:

TA2-REQ-38

STATUS:

Completed

STATEMENT:

MPS shall Display indication of each channel in bypass.

4.1.2.13. Periodic self-test

UID:

TA2-REQ-39

STATUS:

Completed

STATEMENT:

MPS shall run Periodic continual self-test of safety signal path (e.g., overlapping from sensor input to actuation output).

4.1.3. MPS Testing Requirements

UID: SECTION-OR-MPS-Testing-Requirements

Traditionally, this section would be called *Verification Requirements*, but in the context of VERSE *verification* means *providing a formal proof*, thus *testing* is a more appropriate label.

4.1.3.1. Completeness and consistency

UID:

TA2-REQ-21

STATUS:

Completed

STATEMENT:

MPS shall demonstrate the Completeness and consistency of requirements

COMMENT:

Achieved via formalization of the requirements in FRET (see the HARDENS assurance case) and via test cases.

4.1.3.2. Instrumentation independence

UID:

TA2-REQ-22

STATUS:

Completed

STATEMENT:

MPS shall demonstrate Independence among the four divisions of instrumentation (inability for the behavior of one division to interfere or adversely affect the performance of another)

4.1.3.3. Instrumentation independence within a division

UID:

TA2-REQ-23

STATUS:

Completed

STATEMENT:

MPS shall demonstrate Independence among the two instrumentation channels within a division (inability for the behavior of one channel to interfere or adversely affect the performance of another)

4.1.3.4. Actuation logic independence

UID:

TA2-REQ-24

STATUS:

Completed

STATEMENT:

MPS shall demonstrate Independence among the two trains of actuation logic (inability for the behavior of one train to interfere or adversely affect the performance another)

4.1.3.5. Actuation on coincidence vote or manual action

UID:

TA2-REQ-25

STATUS:

Completed

STATEMENT:

MPS shall demonstrate Completion of actuation whenever coincidence logic is satisfied or manual actuation is initiated.

4.1.3.6. Self test and trip independence

UID:

TA2-REQ-26

STATUS:

Completed

STATEMENT:

MPS shall demonstrate Independence between periodic self-test functions and trip functions (inability for the behavior of the self-testing to interfere or adversely affect the trip functions)

4.2. Secure Boot Requirements

UID: SECTION-OR-Secure-Boot-Requirements

A system boot where aspects of the hardware and firmware are measured and compared against known good values to verify their integrity and thus their trustworthiness.

4.2.1. Secure Boot Functional Requirements

UID: SECTION-OR-Secure-Boot-Functional-Requirements

4.2.1.1. Known initial state

UID:

TA2-REQ-54

STATUS:

Completed

PARENTS:

← **TA2-REQ-19** Secure booting only the application code

STATEMENT:

The Secure Boot shall bring a given component to a known initial state.

4.2.1.2. Code attestation

UID:

TA2-REQ-65

STATUS:

Completed

STATEMENT:

Secure boot shall provide attestation for the application code.

4.2.1.3. Boot debug information

UID:

TA2-REQ-55

STATUS:

Completed

STATEMENT:

The Secure Boot shall print information to the console/serial port for debugging purposes.

4.2.1.4. Secure boot termination

UID:

TA2-REQ-56

STATUS:

Deferred

STATEMENT:

The Secure Boot shall always terminate.

COMMENT:

CN cannot currently prove termination properties

4.2.1.5. Launch application or terminate

UID:

TA2-REQ-57

STATUS:

Completed

STATEMENT:

The Secure boot shall either launch the application, or if an error occurs, log the error and terminate.

4.2.1.6. Clear memory

UID:

TA2-REQ-58

STATUS:

Deferred

STATEMENT:

The Secure boot shall erase all RAM containing the secure boot data before a handoff to the application code.

RATIONALE:

This prevents accidental leakage of private information to the potentially compromised application, such as private keys or attestation information.

COMMENT:

Memory erasing is difficult to achieve for a linux process. This requirement will be relevant for embedded scenarios.

4.2.2. Secure Boot Security Requirements

UID: SECTION-OR-Secure-Boot-Security-Requirements

4.2.2.1. Measurement algorithm

UID:

TA2-REQ-60

STATUS:

Completed

STATEMENT:

The Secure Boot shall use high assurance implementation of cryptographic algorithms.

RATIONALE:

For example an implementation that has been formally verified against a "golden" specification, or an implementation automatically generated from such "golden specification".

4.2.2.2. Binary code measurement

UID:

TA2-REQ-59

STATUS:

Completed

PARENTS:

← **TA2-REQ-20** Signature of application code image

STATEMENT:

The Secure Boot shall measure the application binary and compare it against a stored good known value.

4.2.2.3. Secure store of the hash measurement

UID:

TA2-REQ-61

STATUS:

Deferred

STATEMENT:

The Secure Boot should store the measured values in a Trusted Platform Module or other secure memory storage.

RATIONALE:

Normally this requirement would be binding ("shall"), but given the scope of our threat model, this requirement is optional ("should").

COMMENT:

No TPM available.

4.2.2.4. Abort on mismatched measurements

UID:

TA2-REQ-62

STATUS:

Completed

STATEMENT:

The Secure Boot shall abort the boot process and throw an error, if the expected and measured values do not match.

4.2.2.5. Secure Boot stored in read-only memory

UID:

TA2-REQ-63

STATUS:

Deferred

STATEMENT:

The Secure Boot executable shall be stored in a read only memory, or with read-only permissions.

RATIONALE:

This avoids possible modifications to the secure boot executable.

COMMENT:

Not implemented, as for simpler execution we run the secure boot and the application code in the same userspace.

4.2.2.6. Do not compare measurements if expected value is not provided

UID:

TA2-REQ-64

STATUS:

Completed

STATEMENT:

If no expected value of the application binary is provided, the secure boot shall only perform a measurement, save it, and launch the application.

RATIONALE:

If an application is not signed, the secure boot measurement comparison is disabled.

4.3. Mission Key Management Requirements

UID: SECTION-OR-MKM-Requireme ts

Mission Key Management (MKM) processes key requests and distributes keys to other components. Any component can connect to the MKM, request a key, and attest to the code that it's running; the MKM will then send the key if allowed by the MKM's built-in policy.

We require the MKM to implement the following protocol:

1. The client connects to the MKM over TCP.
2. The client component sends a key ID (1 byte), indicating which key it is requesting.
3. The MKM sends a random nonce (16 bytes).
4. The client obtains an attestation matching the challenge (by communicating with its trusted boot daemon) and sends the attestation (64 bytes).

5. If the attestation is valid and MKM policy allows the component to receive the requested key, the MKM sends the key (32 bytes).

If an error occurs, such as an invalid attestation or a policy violation, the MKM simply closes the connection without sending the key.

Since all messages have a fixed size and occur in a fixed order, the protocol does not use any headers or delimiters for messages.

The MKM server listens on localhost (127.0.0.1) port 6000 by default. To change this, set the *MKM_BIND_ADDR* and/or *MKM_PORT* environment variables. For example, *MKM_BIND_ADDR=0.0.0.0 MKM_PORT=6001 ./mkm config.bin* will cause it to listen on port 6001 on all network interfaces.

4.3.1. Close connection on error

UID:

TA2-REQ-66

STATEMENT:

If an error occurs at any time during the key exchange protocol, such as an invalid attestation or a policy violation, the MKM shall close the connection without sending the key.

4.3.2. No headers or delimiters for messages

UID:

TA2-REQ-67

STATEMENT:

All MKM messages shall have a fixed size and occur in a fixed order, and the protocol shall not use any headers or delimiters for messages.

4.3.3. TCP connection

UID:

TA2-REQ-68

STATEMENT:

The client shall connect to the MKM over TCP via a socket.

4.3.4. Wait for key ID

UID:

TA2-REQ-69

STATEMENT:

While the MKM is ready to receive connections, a client component shall send a key ID (1 byte), indicating which key it is requesting.

4.3.5. Send challenge

UID:

TA2-REQ-70

STATEMENT:

When a key ID is received from a client, the MKM shall send a random nonce (16 bytes) in return.

4.3.6. Valid key ID

UID:

TA2-REQ-71

STATEMENT:

The MKM shall process only a valid key ID.

4.3.7. Calculate attestation

UID:

TA2-REQ-72

STATEMENT:

Once the client receives an attestation challenge (nonce) from the MKM, the client shall compute the response by communicating with its trusted boot daemon and send the response back to the MKM.

4.3.8. Challenge response format

UID:

TA2-REQ-73

STATEMENT:

The challenge response shall be computed by concatenating the current measured value (matching the expected hash of the binary) with the received nonce, and then computing HMAC of the concatenated value using a secret key. The resulting response is 64 bytes long.

4.3.9. Secure boot secret key

UID:

TA2-REQ-74

STATEMENT:

The secret key may be identical across different components, so as to simplify the key management. This key is known at build time to the MKM.

RATIONALE:

In real world, secure boot would store unique and shared keys in a Hardware Root of Trust (HROT) and the decision whether to use unique or shared keys would be based on the actual threat model. In either way, the MKM must know the key to validate the attestation response.

4.3.10. Receive response

UID:

TA2-REQ-75

STATEMENT:

Once the MKM receives the attestation response, it shall check its validity. A valid attestation is calculated as described in TA2-REQ-73.

4.3.11. Send key

UID:

TA2-REQ-76

STATEMENT:

If the received response is valid, the MKM shall send back to the client the associated mission key and terminate the connection.

4.3.12. Key format

UID:

TA2-REQ-77

STATEMENT:

The mission key is 32-bytes long symmetric AES key.

4.4. Logging Component Requirements

UID: SECTION-OR-Logging-Component-Requirements

4.4.1. Save telemetry to disk

UID:

TA2-REQ-78

STATEMENT:

The logging component shall connect to the secondary autopilot telemetry port and record some or all telemetry values to disk.

4.4.2. Read from a socket

UID:

TA2-REQ-79

STATEMENT:

The logging component shall read MAVlink messages from a socket

4.4.3. Log file format

UID:

TA2-REQ-80

STATEMENT:

Logs shall be saved in text format, with a timestamp on each line.

4.4.4. Debug print

UID:

TA2-REQ-81

STATEMENT:

Logs may be printed to stdout for debugging purposes.

4.4.5. Encrypted storage

UID:

TA2-REQ-82

STATEMENT:

Logs shall be encrypted by storing them on an encrypted filesystem.

4.4.6. Encryption keys

UID:

TA2-REQ-83

STATEMENT:

The key for the encrypted filesystem shall be obtained from the Mission Key Management component.

4.4.7. Filesystem initialization

UID:

TA2-REQ-84

STATEMENT:

The filesystem shall be initialized on first use.

5. VERSE Toolchain Requirements

UID: SECTION-OR-VERSE-Toolchain-Requirements

VERSE Toolchain specific requirements, driven by the TA2 needs.

5.1. VERSE Toolchain Usability Requirements

UID: SECTION-OR-Robustness-requirements

Requirements related to the user experience with VERSE Toolchain in general.

5.1.1. No crashing

UID:

TA2-REQ-1

STATUS:

Deferred

STATEMENT:

VERSE Toolchain shall not crash on arbitrary input. Instead, an error message shall be produced.

RATIONALE:

Even if a specification is incorrect, or the input file is not a valid C code, VERSE Toolchain should gracefully exit.

COMMENT:

Guaranteeing this requirement for all possible inputs was beyond the scope of the TA1 team's effort.

5.1.2. Special delimiters

UID:

TA2-REQ-2

STATUS:

Completed

PARENTS:

← **TA2-REQ-7** Multiple specification languages

STATEMENT:

VERSE Toolchain should support multiple special delimiters, such as //@ or /*@ or /**@. Which special delimiter should be used can be either configurable, or VERSE Toolchain should support all of them at the same time (see TA2-REQ-15).

RATIONALE:

In some codebases, VERSE Toolchain specs need to co-exist with existing specifications (such as Frama-C ACSL), such that adding VERSE Toolchain specs does not break the existing proofs.

5.1.3. Multiple specification languages

UID:

TA2-REQ-7

STATUS:

Completed

CHILDREN:

→ **TA2-REQ-2** Special delimiters
→ **TA2-REQ-8** Continuity of existing proofs

STATEMENT:

VERSE Toolchain shall run on codebases with multiple specification languages, such as Frama-C, SAW, and Cryptol.

RATIONALE:

High assurance code might contain multiple different spec languages. For VERSE program, we expect that only Frama-C ACSL specifications will exist directly in the C source code. Other specs, such as SAW and Cryptol, do not change the C code directly.

5.1.4. Continuity of existing proofs

UID:

TA2-REQ-8

STATUS:

Completed

PARENTS:

← **TA2-REQ-7** Multiple specification languages

STATEMENT:

Adding VERSE Toolchain specs to a codebase shall not break existing proofs about such codebase.

RATIONALE:

For example, adding VERSE Toolchain specs into an existing high assurance codebase shall not break the existing Frama-C proofs

COMMENT:

We run both Frama-C and CN proofs in the CI

5.1.5. Project specific VERSE Toolchain configuration

UID:

TA2-REQ-15

STATUS:

Deferred

CHILDREN:

→ **TA2-REQ-12** Explicit assertion checking
→ **TA2-REQ-13** Well defined default behavior
→ **TA2-REQ-14** Annotation of pure functions

STATEMENT:

VERSE Toolchain shall support project specific configuration, in the form of a configuration file that will adjust how VERSE Toolchain behaves.

RATIONALE:

This is a top level requirement, further specified in the child requirements.

COMMENT:

Out of scope for P1

5.1.6. Code similarity

UID:

TA2-REQ-52

STATUS:

Deferred

STATEMENT:

The code checked by VERSE Toolchain and the code compiled and deployed on the OpenSUT shall be identical.

RATIONALE:

If the code that can be checked by VERSE Toolchain is substantially different from the code that is compiled and deployed, errors in the production code might not be captured, leading to presence of bugs and vulnerabilities.

COMMENT:

While the code is fairly similar, there are still some workarounds needed for the verification to pass.

5.2. VERSE Toolchain Functional Requirements

`UID:` SECTION-OR-Functional-Requirements

This section lists requirements on the functionality of VERSE Toolchain, and the features it provides.

5.2.1. Versioned releases

`UID:`

TA2-REQ-3

`STATUS:`

Completed

`STATEMENT:`

VERSE Toolchain shall provide versioned releases, such that running VERSE Toolchain with *-V* flag shall print out the version of the tool.

`COMMENT:`

CN prints the commit hash as a version.

5.2.2. Variadic functions

`UID:`

TA2-REQ-5

`STATUS:`

Deferred

`STATEMENT:`

VERSE Toolchain shall support reasoning about variadic functions, such as *printf()*.

5.2.3. Packaged releases

UID:

TA2-REQ-4

STATUS:

Completed

STATEMENT:

VERSE Toolchain shall provide packaged releases using industry standard mechanisms, such as docker, or debian packages.

COMMENT:

CN provides both Ubuntu and RedHat based docker images.

5.2.4. C Unions

UID:

TA2-REQ-6

STATUS:

Deferred

STATEMENT:

VERSE Toolchain shall support reasoning about C union types.

RATIONALE:

For example the MPS code relies heavily on unions, and such code needs to be supported.

COMMENT:

Planned for Phase 2

5.2.5. Nested types

UID:

TA2-REQ-9

STATUS:

Deferred

STATEMENT:

VERSE Toolchain shall support reasoning about structs composed of other structs.

RATIONALE:

For example VERSE Toolchain shall be able to reason about the following struct and prove that there is no undefined behavior and that any user defined specification holds for such a struct:

```
struct S {
    T1 S1;
    T2 *S2;
    T3 S3[];
}
```

5.2.6. User defined type invariants

UID:

TA2-REQ-53

STATUS:

Completed

STATEMENT:

VERSE Toolchain should support checking user defined type and data structure invariants. VERSE Toolchain should allow users to annotate types and data structures with invariants, such that the invariant is preserved at every instance of that type.

RATIONALE:

For example, the user wishes to prove that a pointer of particular type is never NULL. While NULL pointers are allowed under Cerberus C semantics, *dereferencing* a NULL pointer is an undefined behavior. Thus, a user defined invariant that a pointer shall never be NULL should be checkable by VERSE Toolchain.

Or given an array *T3 S3[];* the user wishes to prove that invariants about type T3 are valid for each element of array S3, and this is true for min and max size of S3, with min=0 and max some sensible default value (uint32_MAX?).

5.2.7. Specs in header of source file allowed

UID:

TA2-REQ-10

STATUS:

Completed

STATEMENT:

VERSE Toolchain shall allow the user to write VERSE Toolchain specifications in either header (function declaration) or source file (function definition). If VERSE Toolchain specs are provided at both function declaration and function definition, VERSE Toolchain shall raise an error.

RATIONALE:

In some cases, writing specs in the header files is more ergonomic. In other cases, there might be no header files. The user shall have a choice that is the most suitable for a particular codebase. If accidentally the user writes multiple VERSE Toolchain specs for the same function (in the header and in the source file), VERSE Toolchain needs to throw an error an notify the user, as resolving which specs are valid is a complex problem.

5.2.8. Explicit assertion checking

UID:

TA2-REQ-12

STATUS:

Deferred

PARENTS:

← **TA2-REQ-15** Project specific VERSE Toolchain configuration

STATEMENT:

VERSE Toolchain shall have a configurable option to either ignore inline *assert()* statements, or to statically check them.

RATIONALE:

In some codebases, assertions are used only for selective runtime testing, so static checking might produce findings that are not interesting for the developers. The assertions are removed in the production code. Hence having the configurable option for VERSE Toolchain is important.

COMMENT:

Planned for Phase 2

5.2.9. Well defined default behavior

UID:

TA2-REQ-13

STATUS:

Deferred

PARENTS:

← **TA2-REQ-15** Project specific VERSE Toolchain configuration

STATEMENT:

If a function is not annotated with any VERSE Toolchain specifications, VERSE Toolchain shall explicitly state what are the default (implicit) *require*, *ensure* and *modify* clauses.

RATIONALE:

It needs to be stated whether by default each function requires and ensures nothing, or if there are some implicit assumptions, important for compositional reasoning. Same for modification - a sensible default behavior could be that a function without specs is assumed to modify everything. However, in that case compositional reasoning is not really possible, so having a configurable option here might be preferred.

The implicit *requires* might encompass e.g. a valid stack frame for the function.

COMMENT:

Planned in Phase 2, as a part of the documentation improvement.

5.2.10. Annotation of pure functions

UID:

TA2-REQ-14

STATUS:

Deferred

PARENTS:

← **TA2-REQ-15** Project specific VERSE Toolchain configuration

STATEMENT:

VERSE Toolchain shall have a configurable option to either assume that all functions are *pure* by default, or to require an explicit *pure* annotation.

RATIONALE:

Pure functions are side-effects free, and don't have any persistent static variables (see https://en.wikipedia.org/wiki/Pure_function). In some cases, explicitly stating which functions should be *pure* is easier, while in other codebases, it is reasonable to assume that the functions are *pure* by default. This should be configurable.

5.2.11. Check for undefined behavior

UID:

TA2-REQ-51

STATUS:

Completed

PARENTS:

← **TA2-REQ-16** No undefined behavior

STATEMENT:

VERSE Toolchain shall check C code for undefined behavior as defined in Cerberus semantics, and raise an error when undefined behavior is found.

RATIONALE:

This is a base level functionality of VERSE Toolchain, as code with undefined behavior often leads to errors and unintended results.

5.3. VERSE Toolchain Documentation Requirements

UID: SECTION-OR-Documentation-requirements

Documentation of VERSE Toolchain, including manuals, tutorials, quick-start guides, code and document generation, and hints and error messages.

5.3.1. Generating code documentation with specs

UID:

TA2-REQ-11

STATUS:

Deferred

STATEMENT:

TA1 tools shall generate source code documentation that includes VERSE Toolchain specification with VERSE Toolchain syntax highlighted.

RATIONALE:

Doxygen-like documentation with VERSE Toolchain specs included is ideal. It is important that the specs are not treated like comments, but are lifted and highlighted in the generated documents.

5.3.2. Code coverage measurement

UID:

TA2-REQ-50

STATUS:

Deferred

STATEMENT:

VERSE Toolchain should provide means of measuring code coverage, and specifically reporting:

1. percentage or LOC of code/functions that have *any* specs
2. *any* code excluded from VERSE Toolchain checking (maybe hiding behind *#ifdef* or some other directive, excluding the code from being examined)
3. coverage based on types/classes of specifications (see the different classes we mentioned in the proposal)

RATIONALE:

See https://github.com/GaloisInc/VERSE-Toolchain/issues/93 for more details.