CRA-PRINCESS

Challenge Problem 2: New Computing Platform

# Overview

In this challenge problem, the processor board (system) on which the UUV mission payload software runs is changed. This requires a port of the mission software to an operating system that runs on the new system. Where tools exist to target the new instruction set architecture (ISA), this could be as simple as setting some compiler flags or as difficult as rebuilding the entire system, from the ground up – including producing a new compiler toolchain (i.e., assembler, linker, loader), operating system, and mission software. Where the processor has no or poor existing tool support, an adaptation may be impossible until adequate tools and system libraries are available. PRINCESS facilitates adaptation by synthesizing the tool chain and operating system from descriptions of the new processor and processor board to enable the mission payload software to run on the new UUV hardware platform.

This challenge problem runs offline – while the UUV is docked. There is no automated detection of perturbations; a change of hardware platform is a planned-for event, and the synthesis of the new software takes place outside the UUV. The newly synthesized software would be then downloaded to the UUV.

In Phase 1, we synthesized the toolchain (assembler and linker) for randomized instruction encodings of a MIPS processor. In Phase 2, we will tackle two aspects of new computing platforms:

* Synthesizing the toolchain for two different architectures, ARM and SPARC64, using QEMU emulation to run test programs.
* Synthesizing source code for a subset of the machine dependent parts of the operating system.

This challenge problem has one intent element: “Execute Mission Software.” This intent represents the successful execution of the mission payload software on a new processor. This intent is threatened by changes to the processor that render the compiled binary executable obsolete, especially when tools or system libraries are not available to recompile the mission software.

This challenge problem requires the development of several domain specific languages (DSLs). To date, a machine description consists of ten input files that specify:

1. Base types, including register enumerations and classifications for operands.
2. The assembly language syntax, e.g., comment characters, character sets, token descriptions.
3. Assembler modes.
4. Assembly language “modifiers” – operators used in the construction of relocation records for object files.
5. Operand matching: the correspondence between assembly-language-level opcode strings and operand syntax, and machine-level opcode numbers and operand encodings
6. Instruction and operand encodings, e.g., instruction layouts, field descriptions, opcode values.
7. Pipeline hazards: patterns of instruction sequences that are not legal or not safe to use.
8. Linker relocations: specifying the meaning of the magic numbers used in object files to allow the linker to resolve external references.
9. Miscellaneous other information, e.g., endianness, instruction fetch size.
10. A semantic machine description, written in a domain specific language called Cassiopeia, used for OS synthesis.

With regard to machine perturbations during toolchain evaluation, we provide exactly two machine descriptions, one corresponding to ARM, and another corresponding to SPARC64.

With regard to machine perturbations during OS synthesis evaluation, we will provide fixed machine perturbations, each stated as a separate Cassiopeia program, with differing numbers and allowed usage of registers by machine instructions. We can only provide for fixed perturbations because Cassiopeia is necessarily a very precise description of a machine, and arbitrary, e.g. randomized, perturbations are almost certain to be Turing incomplete and hence uninteresting to evaluate.

**Evaluating the toolchain:**

For this part of the evaluation, the perturbation tool will select one of the supported platforms, on which an off-the-shelf operating system runs. The PRINCESS software will then synthesize a toolchain for the designated platform and run any requested (user-level) C program on the target platform, comparing output to that of a baseline system. The user-level program will be one created using the csmith program, which outputs a checksum. Thus, comparison of outputs requires a simple verification that the checksums produced by two programs are identical.

Per discussion in April 2018, because of concerns about csmith producing test programs that are trivial and/or invalid and/or that run effectively forever, it has been agreed that a deterministic, reproducible test corpus generator will be produced and agreed upon beforehand. It will run csmith repeatedly, giving csmith a series of unpredictable but deterministic seeds, and reject output test programs that fail any of a series of agreed-upon automated checks.

The checks are:

* Reject programs that do not compile with the production compiler at any of the usual optimization levels (-O0 through -O3)
* Reject programs that run for longer than five seconds on the baseline platform
* Reject programs whose checksum output is identically zero
* Reject programs whose code is less than an agreed-upon minimum number of lines
* Reject programs whose output checksum differs when compiled at different optimization levels
* Reject programs whose output checksum differs when compiled as 32-bit vs. 64-bit programs

The evaluation package ships with a corpus of 2000 test programs that meet these criteria.

Lincoln Labs may use the generator to generate additional test corpora. A script to do so is provided, documented in our overview.md.

**Evaluating the synthesized OS code:**

For this part of the evaluation, we have identified approximately six independent pieces of machine dependent kernel code. Each code segment is written in Alewife, described below, with explicit pre- and post-conditions that the code segment must satisfy.

We further provide multiple (3-4) fixed perturbations of the two machine platforms described: one resembling ARMv7-A and one resembling x86-64. The fixed perturbations change the number and kind of registers available to the machine instructions.

Documentation for the machine platforms, perturbed platforms, and OS module descriptions are provided with the evaluation container.

**Synthesizing OS code:**

For OS Synthesis, we have also defined an OS module description language named Alewife. Alewife programs are comprised of *terms* and *blocks*. Informally, an Alewife term refers to an abstract machine resource, that is, an uninterpreted name, or a function computed using machine resources, that is, an uninterpreted function. A block, then, is comprised of pre-and-postconditions stated as logical formulas. In these logical formulas, every atom is an Alewife term. Then, each Alewife block, combined with the terms it relies on, represents a machine-independent OS module description. In the following, we will describe how Cassiopeia machine descriptions provide a means of synthesizing machine-dependent implementations of an OS module in assembly code for that machine.

In the remainder of this discussion, we will equivalently refer to the tuple of pre- and post-condition as a *specification*. Further, when a code segment satisfies its post-condition given that its pre-condition is true on entry to the code segment, we say that the *code segment satisfies its specification*.

We now describe how Alewife and Cassiopeia are used to automatically synthesize OS module implementations in assembly code. From the Alewife DSL specifications, the PRINCESS code will generate, for each block, a Cassiopeia expression of the specification that is a direct translation of the Alewife pre- and post-conditions . The Cassiopeia specification is then used to synthesize assembly code corresponding to the machine-dependent implementation of the OS module described by the Alewife program.

Thus, the goal of the adaptation is to produce, for each combination of Alewife OS module description and Cassiopeia machine description, a snippet of assembly code including a pre-condition and a post-condition such that if the pre-condition holds, the generated code satisfies the post-condition. Next, we will describe how we use a *static* code verifier to ensure that our synthesized assembly code satisfies its specification.

**Evaluating Synthesized OS code using static verification:**

In order to evaluate the quality of our synthesized code, we have developed a static code verification tool within the Rosette system (provided by Univ. Wash.). Further, our static verifier uses an off-the-shelf SMT solver, Boolector, as the underlying automated theorem prover. We have implemented this static verifier as part of the Alewife and Cassiopeia DSL toolchain.

Then, given a Cassiopeia machine description, an Alewife module description, and the synthesized implementation in a Cassiopeia-based intermediate representation (.prog in the container) that corresponds to exactly one assembly snippet, the verifier behaves as follows:

The verifier ensures that the implementation is a valid snippet for the given machine platform.

The verifier automatically *lowers* the Alewife specification into a Cassiopeia specification using a simple syntax-directed procedure.

The verifier uses our Rosette and Boolector-based static verifier to determine if the implementation meets its lowered specifications. We note that, as with all static verification tools, the verifier may present the user with false negatives.

Finally, we formally guarantee that if the verifier determines that the implementation meets its lowered, machine dependent specification, then it also meets the higher-level machine independent specification.

# Test Data

We provide complete machine specifications for the two target platforms for which the toolchain will synthesize code; these are resident on our test VM. Both of these system descriptions will be automatically perturbed for the OS code synthesis portion of the challenge.

# InitialParams (success response to a READY)

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| Baseline | String value: One of “A”, “A\*”, “B”, or “Challenge” |  |
| Target | Int: [0 – (264 – 1)] | Seeds random number generator and selects target platform and/or perturbation.  Note that we have chosen a large integer space for the Target value such that we can partition the range of targets using specific bitfields in the value, to seed the various bits of the perturbation requested.  Further, note that the behavior of the system is intended to be deterministic w/rt a given target value. |
| TargetType | String value: one of “Cprog”, “DSLprog” | Identifies the type of perturbation, one that is testing the toolchain (Cprog) or the OS synthesis (DSLprog). |
| Contents | String value: JSON | integer specifying a file from either the corpus of C programs generated by csmith and then filtered (when TargetType = “Cprog”) or an integer specifying a file from the corpus of DSL OS code snippets (when TargetType = “DSLprog”)  The integer will be supplied as a string value containing at most 10 decimal digits and no other characters (and no leading zeros, unless the value is zero) which will be interpreted as a 32-bit nonnegative integer. This allows the SUT code to interoperate with the prior interpretation of this field where it contained program text.  The integer may be between 0 and one less than the size of the pertinent corpus. |
|  |  |  |

# Test Procedure

In all tests, TA will be implemented using a standard Ubuntu x86\_64 LTS VM. When testing the toolchain, CP2 requires access to the possible target platforms agreed upon by Harvard and Lincoln. These platforms, provided via emulation in QEMU will be used for Baseline A\* and the Challenge Stages as described below.

**Toolchain evaluation test procedure:**

Notes: As per email discussion on or around 4/19-21/18, the csmith runtime will be part of the CP2 evaluation container and may change as the toolchain implementation reaches maturity. This is because the csmith runtime uses a number of shared library routines to make its programs portable across architectures. As of csmith 2.4, 32-bit ARMv7 and SPARC64 are not explicitly supported by the csmith runtime, so Harvard will add support to csmith as necessary.

Finally, note that generating the compiler for a given platform remains out of scope for this project. Hence, we will be using the gcc-cross platform packages provided by an ubuntu LTS (currently 16.04 LTS with GCC 5.4.0)

*Baseline A* – Native execution of the generated csmith C program in the test Ubuntu VM (x86\_64).

*Baseline A\** – As discussed with Lincoln Labs 9/19/17, we define a second Baseline A-**like** stage, which we call Baseline A\*. In the A\* setting, we seek to ensure that the given C program from the Cprog Corpus field is portable; that is, the given C program compiles and executes on the target platform using the native toolchain and yields a program with the same output behavior as the Ubuntu VM. Then, we compile and execute the given program on the target platform using the native toolchain and execute it on that platform. Baseline A\* is passed if and only if the output of A\* matches the output of A.

Baseline B – As discussed with Lincoln Labs 9/19/17, we do not have a test procedure that fits the Baseline B definition.

*Challenge Stage –* On the same target platform as specified in Baseline A\*, build the specified program with the synthesized tool chain and then execute the program on the target platform. The results should match that of both Baseline A and Baseline A\*.

**OS code evaluation test procedure:**

*Baseline A* – Given the specified OS code snippet and reference-architecture pre- and post-conditions expressed in C, verify our own hand-written code for that snippet based on the machine dependent code that appears in the Barrelfish source tree for that module and machine.

*Baseline B* –Verify the specified code snippet with valid pre- and post-conditions and an empty code body. We note here that, first, this ensures that the pre- and post-conditions are non-trivial. That is, the postcondition is not satisfied by the precondition (e.g., POST = True).

*Challenge Stage –* Synthesize an implementation for the OS module over the specified machine. Then, verify that the synthesized code meets its pre- and post-conditions.

Note that our need to distinguish between the identity perturbation in Baselines A, B and the Challenge stage are the reason our scenario schema includes explicit specification of the stage.

# Interface to the Test Harness (API)

Test scenarios (including perturbations to the architecture) are specified in the InitialParams structure of the success response to the READY. The InitialParams is expressed in the JSON encoded structure described above.

Based on the TargetType and file contents, CP2 will first validate that the file contents are well-formed with respect to the TargetType. If the file contents are not well-formed, CP2 will respond with an ERROR message and terminate. The ErrorMsg will contain a JSON encoded structure as described below:

ErrorMsg

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| TargetType | String value: one of Cprog or DSLprog | As defined in InitialParams |
| ErrorType | String value: One of: “BADFILE”, “TOOL BUILD FAILED”, “INSTALL FAILED”, “PROGRAM BUILD FAILED”, “VERIFICATION FAILED”,”CPROG FAILED”, “SYNTHESIS FAILED”,”DISALLOWED OPERATION” | Indicates type of error |
| LogMsg | String value: JSON encoded message generated by the contents validator. |  |
|  |  |  |

With “ErrorType” set to “BADFILE.” Throughout the rest of the test sequence, the ErrorMsg will be returned in response to any error condition. In particular, as CP2 does not support online adaptation, it will respond to a /*perturb* message from TH with an ErrorMsg with “ErrorType” set to “DISALLOWED OPERATION.” Similarly, as CP2 implements the SUT operating system, it will not support the /*disable* command either, and will return ErrorMsg with “ErrorType” set to “DISALLOWED OPERATION.”

If the contents pass validation, CP2 will commence with the perturbation. In doing do, it will generate a sequence of one or more STATUS messages using the following JSON encoded SutStatus:

SutStatus

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| Type | String value: one of eight values: “PERTURBING SYSTEM”, “SYSTEM PERTURBED”, “BUILDING SYSTEM”, “SYSTEM BUILT”, “COMPILING PROGRAM”, “RUNNING SYSTEM”, “RUNNING KEEPALIVE”, “VERIFYING SYSTEM”, “VERIFICATION KEEPALIVE” | Indicates where CP2 is in the test process. |
| LogMsg | String value: output produced by the system. | We do not expect interpretation of the log message, but request access to the LogMsg for debugging purposes. |
|  |  |  |

CP2 will also respond with a JSON-encoded SutStatus response to the */alive* and */query* messages from the TH.

If CP2 experiences any fatal error, it will generate an ERROR message as described above. When CP2 successfully completes, it will send a DONE message as described below:

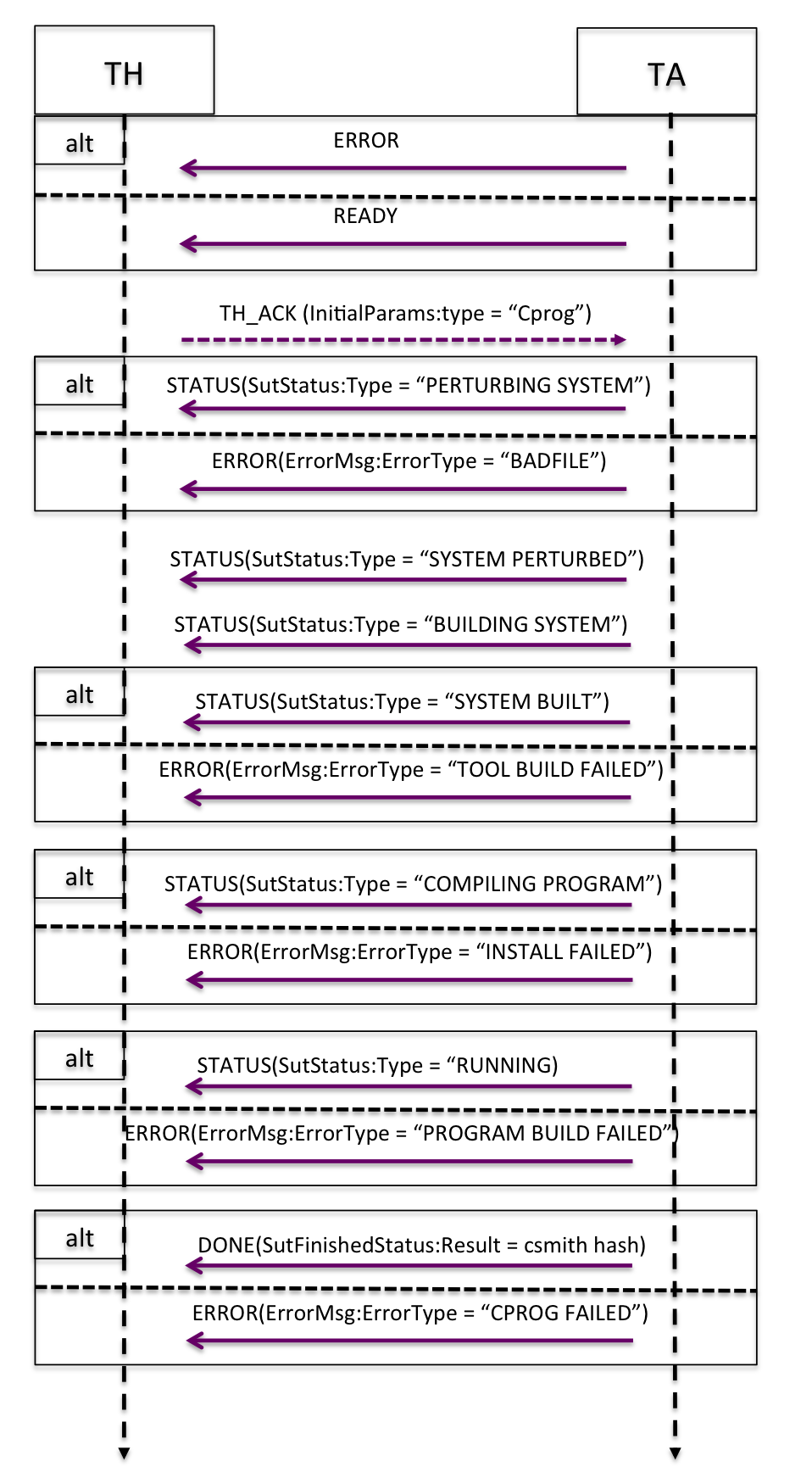
SutFinishedStatus

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| TargetType | String value: one of “Cprog” or “DSLprog” |  |
| Result | String value: JSON encoded output. | Either the hash value produced by the csmith-generated test program or the verification verdict on the code synthesized. |
| LogMsg | String value: output produced by the system. | We do not expect interpretation of the log message, but request access to the LogMsg for debugging purposes. |
|  |  |  |

**Toolchain Adaptation**

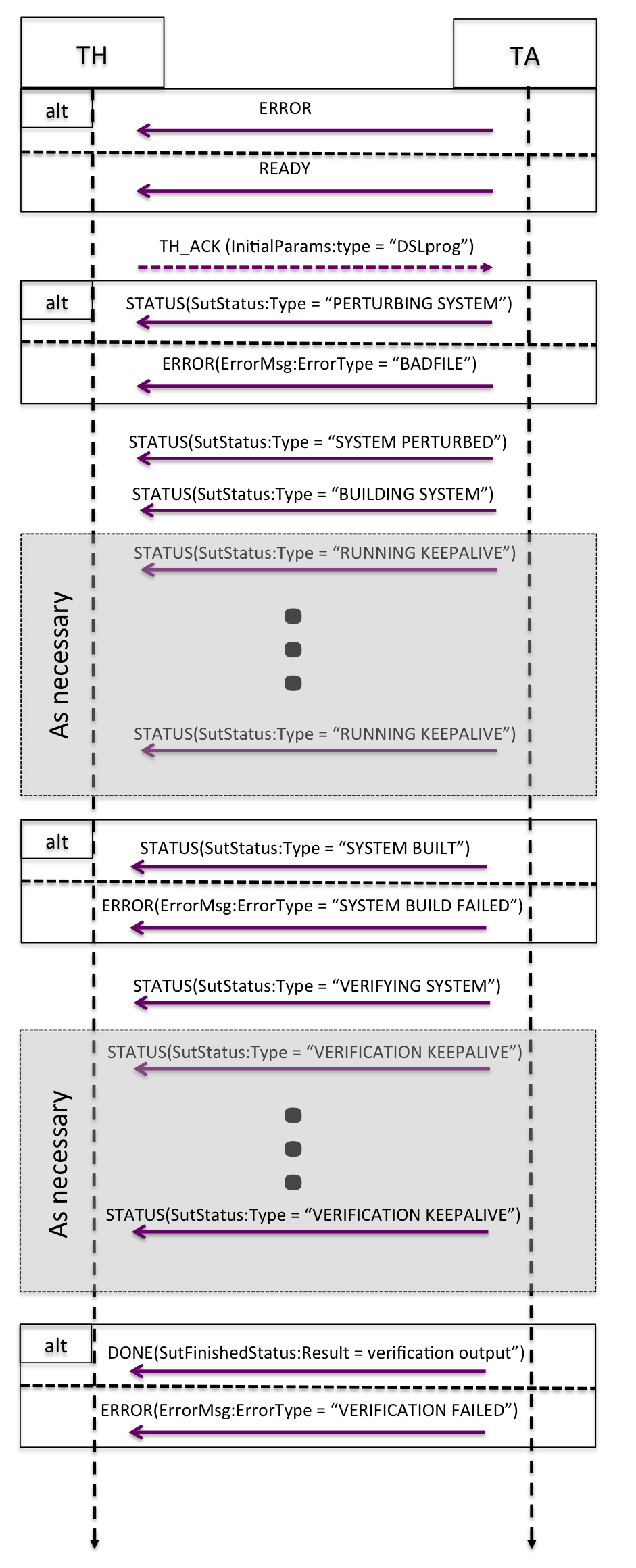
If the “TargetType” of the InitialParams is “Cprog”, CP2 will proceed with toolchain adaptation. It will perform the following steps:

1. CP2 will send a STATUS message with “type “PERTURBING SYSTEM”.
2. Based on the target value, CP2 will select one of the supported architectures, and send a STATUS message with “type” “SYSTEM PERTURBED”.
3. CP2 will send a STATUS message with “type” “BUILDING SYSTEM”.
4. CP2 will build a toolchain for the selected architecture. If this build fails, CP2 will send an ERROR, with “ErrorType” of “TOOL BUILD FAILED” and terminate.
5. Upon successful completion of the build, CP2 will send a STATUS message with “type” “SYSTEM BUILT”
6. CP2 will then install the newly built toolchain on the target VM. On error, CP2 will send an ERROR message with ErrorType “INSTALL FAILED” and terminate.
7. CP2 will write “Contents” to the target VM file system and then send a STATUS message with “type” of “COMPILING PROGRAM”.
8. CP2 will then compile, assemble, link, and run the program specified in “Contents.” On error, CP2 will send an ERROR message with “ErrorType” of “PROGRAM BUILD FAILED” and terminate.
9. CP2 will send a STATUS message with “type” of “RUNNING PROGRAM.” On error, CP2 will send ERROR message with ErrorType “CPROG FAILED” and terminate.
10. CP2 will send a DONE message with “Result” set to the output of the program specified in “Contents”.



**OS Code Adaptation**

If the “TargetType” of the InitialParams is “DSLprog”, CP2 will proceed with OS code adaptation. It will perform the following steps.

1. CP2 will send a STATUS message with “type “PERTURBING SYSTEM”.
2. Based on the Target value, CP2 will perturb the architecture, and send a STATUS message with “type” “SYSTEM PERTURBED”.
3. CP2 will send a STATUS message with “type” “BUILDING SYSTEM”.
4. Using the “Contents,” CP2 will generate an OS snippet, including pre-condition and post-condition code in C. Upon error, CP2 will send an ERROR message with “ErrorType” “SYSTEM BUILD FAILED” and terminate.
5. Should code synthesis proceed more slowly than the agreed upon timeout interval, CP2 will issue STATUS messages of “type” “RUNNING KEEPALIVE” as many times as necessary.
6. When code synthesis is complete, CP2 will send a STATUS message with “type” “SYSTEM BUILT”; “LogMsg” will contain the synthesized C code.
7. CP2 will then enter a “verification” state. It will send a STATUS message with “type” “VERIFYING SYSTEM”.
8. It will attempt to verify that the code snippet satisfies its post-condition, given that its pre-condition is true on entry.
9. Should code verification proceed more slowly than the agreed upon timeout interval, CP2 will issue STATUS messages of “type” “VERIFICATION KEEPALIVE” as many times as necessary.
10. On error, it will send an ERROR message with “ErrorType” “VERIFICATION FAILED.”
11. Upon completion, CP2 will send a DONE message with “Result” containing the output of the verification procedure.

**Known Issues**

* Performance: Note that the time between receipt of the InitialParams and the DONE message is not a subject of evaluation by the TH.
* Runtime of the csmith C file has been problematic in the past, and we look forward to working with Lincoln to continue tuning csmith and our toolchain to avoid unnecessary timeouts.
* Runtime of OS Code Adaptation and verification is unpredictable even with state-of-the-art tools. The “KEEPALIVE” messages are designed to alert TH that forward progress is being made.
* We note again that both the static verifier and the synthesizer can present false negatives that may correspond to intractably long runtimes as well as false negatives that correspond to the verifier or synthesizer returning with a failure.

Intent Specification and Evaluation Metrics

As changes in hardware platform are planned and manual activities, this challenge problem does not include any automatic detection of change in intent.

**Interaction Diagram Discussion:**

The test result model we have implemented is as follows: if the system fails to complete the test run, it will send an ERROR (ErrorMsg) message and exit; otherwise it will send a DONE (SutFinishedStatus) message with a result in it.

Error states can be divided loosely into “a bad test run was requested”, which is indicated by the BADFILE error type (meaning the test parameters were invalid or out of range) and the DISALLOWED\_OPERATION error type (meaning an invalid request was made at the communications level) – and “something broke during the test run”, which is indicated by all the other error types.

For the toolchain evaluation, the cases are:

* TOOL\_BUILD\_FAILED and INSTALL\_FAILED: something broke preparing the new toolchain. You shouldn't ever see these; they mean "something catastrophic has happened".
* PROG\_BUILD\_FAILED: the test program failed to build. If this occurs in Baseline A or Baseline A\* it means the test program is invalid; if the same test program succeeds in those but fails on Challenge it means the new toolchain broke and that counts as a test failure.
* CPROG\_FAILED: the test program crashed when run. The same criteria apply.

The checksum from the test program appears in the Result field of the SutFinishedStatus message. Note that the SUT does not know what the checksum is supposed to be. If the Baseline A\* and Baseline A checksums are different, the test program is unportable and should be rejected; if the baseline A\* and Challenge checksums are different, presumptively the toolchain failed. (Although as we have discussed with LL other reasons are possible, such as the test not being deterministic or exercising undefined behavior. One of the purposes of the corpus generator is to weed out as many unsuitable test programs as possible.)

For the OS synthesis evaluation, the error cases are:

* SYSTEM\_BUILD\_FAILED means that the code synthesis run itself failed (timed out, crashed, etc.) -- this cannot happen for Baseline A or B but only Challenge.
* VERIFICATION\_FAILED means that the verification step broke (timed out, crashed, etc.), not that the verification step rejected the program.

The verdict from the verification step appears in the Result field of the SutFinishedStatus message. The exact string was never specified; currently it either contains “Succeeded” or begins with “Failed:”.

It should be possible to trigger most of the error paths intentionally using the additional machine architecture codes we defined as invalid for testing purposes. These are documented in the overview file that ships with the SUT tree (doc/overview.md).

**Evaluation Metrics:**

**Toolchain**

If C program corpus output checksums match, adaptation is successful.

**OS Snippets**

If verification succeeds, adaptation is successful.

Last Review

7-JULY-2018