What CallContext means?

One test suite may communicate with a number of Agents at the same time. Each agent is an active Agent thread. “CallContext” is a structure representing active process+thread to which this call is directed (i.e. a pair {pid , pthread_t}).

CallContext is not an absolutely required thing. Neither SeC language no CTesk tools require us to have CallContext. It is a way we decided to design OLVER test suite for LSB Core. Many functions have something to do with threading/processes/signals. Many of them are blocking. Even simple functions like ‘strtok_r’ require dealing with threading (because standard directly states that it is a “Thread-Safe Function”) and that is a subject to be checked if we pretend that our test suite is going to be highest quality at some moment of time. Having CallContext scenario for checking thread safety is straight forward:

```
bool simple_strtok_r_scen()
{
      for(…)
      {
            strtok_r_spec(context1, param1);
            strtok_r_spec(context2, param1);
      }
      return true;
}
```

This scenario calls *_spec functions in two contexts (i.e. in two threads). The threads are actually existing on agent’s side, so that it is a multithreaded parallel program. But our scenario is looking like linear which we think is more convenient for testing situations like that.

Finally we decided to have additional parameter CallContext to all specification functions that we test in OLVER for LSB-Core. This decision is for LSB-Core test suite only. Test suites for other components may be not using CallContext if that is decided to be not reasonable. In that case specifications will have number of parameters equal to that of function under test. For example function like:

```
int strcmp(const char *s1, const char *s2);
```

may have specification signature:

```
specification IntT strcmp_spec(StringTPtr s1, StringTPtr s2);
```

which is straight forward mapping.

Details on Agent protocol

TCP/IP protocol “OLVER <=> agent” has primitive textual nature. I think the best way to explain it is via simple example.

Specification for abs(int) function (src/model/math/integer/integer_model.sec) has following signature:

```
specification IntT abs_spec( CallContext context, IntT i ).
```

When it is called from the scenario:

```
abs_spec(context, -10);
```

---- Test Suite side------

To see visualization of this example please refer to the diagram in the end of the chapter.

Step 1: The precondition is checked. (Details on precondition please find in the the next chapter)

The generated .c file (src/model/math/integer/integer_model.c) contains “bool pre_abs_spec(struct ThreadId context, int i)” that is being called.

Step 2: If precondition returns “true” execution proceeds to the following steps.

Step 3: “Mediator” is called. Mediator is an utility function that packs input parameters, sends them to the agent, reads/unpacks the return value and passes it to the following steps (postcondition). Mediator for abs_spec (defined here: src/model/math/integer/integer_media.sec) is:

```
mediator abs_media for specification
IntT abs_spec( CallContext context, IntT i )
{
    call
    {
        TSCommand command = create_TSCommand();
        IntT res = 0;

      // (1) Prepare the string with input parameters
        format_TSCommand( &command, "abs:$(int)", create_IntTObj(i) );
      // (2) send to the agent and wait for the result
        executeCommandInContext( context, &command );
        if (!isBadVerdict())
        {
            timestamp = command.meta.timestamp;
      // (3) unpack the result
            res = readInt_TSStream(&command.response);
        }

        destroy_TSCommand(&command);

        return res;
    }
}
```

In (1) the command is translated to the string form: “abs:int:-10”
In (2) the command is sent to the agent and executed there.
In (3) the return is decoded (it has form “int:10”) and readInt_TSStream decodes it into IntT value.

Step 4: Postcondition is called. (Details on postcondition please find in the the next chapter)
Step 5: Control is returned to the scenario.

---- Agent side------

On agent’s side command is implemented as follows (see src/agent/math/integer/integer_agent.c):

During agent initialization the command handler is registered:

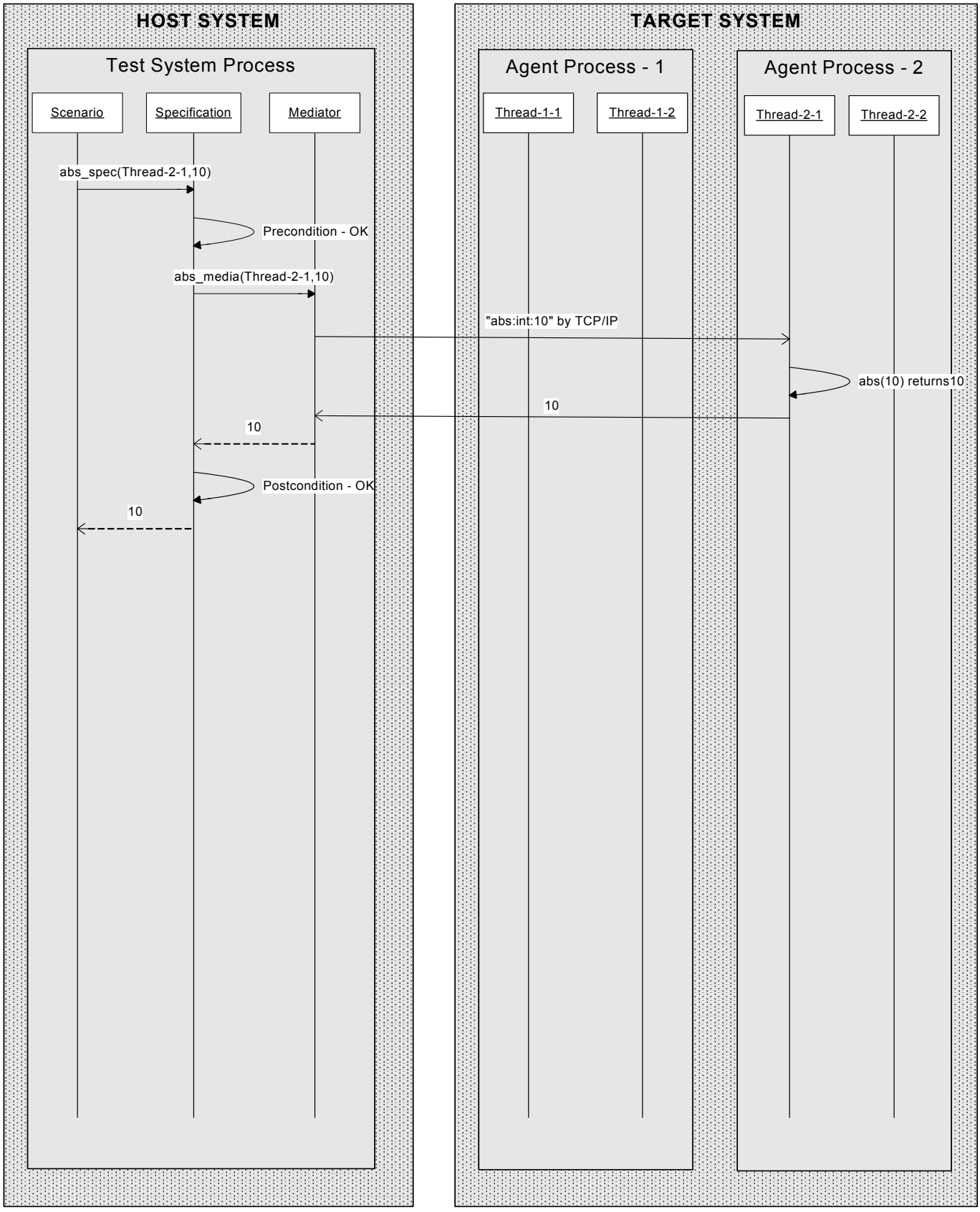
```
ta_register_command("abs",abs_cmd);
```

So that upon call to “abs” the “abs_cmd” function is called:

```
static TACommandVerdict abs_cmd(TAThread thread,TAInputStream stream)
{
    int i;
    int res;

    /* (1) Prepare */
    i = readInt(&stream);

    START_TARGET_OPERATION(thread);

    /* (2) Execute */
    res = abs(i);

    END_TARGET_OPERATION(thread);

    /* (3) Response */
    writeInt(thread, res);
    sendResponse(thread);

    return taDefaultVerdict;
}
```

In (1) it reads integer from the stream.
In (2) the function is called
In (3) the result is sent to the communication stream.

“Thread-2-1” is used as a value of the CallContext parameter, which indeed equal to (pid, pthread_id) as metioned before.



What pre/post/coverage means and how it translates to C

Let me split answer in two parts. First is formal: will try to provide information about pre/post/coverage blocks. And second is an example, which I hope demonstrates how that works for from the developer side.

Formal answer

The main purpose of a **precondition** is self-checking of the test suite. Precondition checks is it possible to call the function under test with the given parameter values or not. If it is possible then the test continues. Otherwise it aborts with the corresponded error message.

To obtain addition information please have a look at the getting-started-math-integer. If contains detailed line-by-line description of precondition for the abs function.

The main purpose of a **postcondition** is checking of correctness of the function under test during the given call to it. Postcondition is so-called universal assertion on the function under test. If it is failed (that could be only if one of REQ macro has a third expression equal to false) we

1. trace error message into trace file,
2. trace an identifier of the requirement violated, and
3. continue testing.

If postcondition returns true we trace all identifires of requirements, which were checked in the postcondition, and continue testing.

The main purpose of **coverage blocks** is partitioning of the domain of the specification functions parameters into the equivalence classes. Coverage blocks are used for additional (advanced) measurement of quality of testing and they could be omitted at all. OLVER provides other technics to measure quality of testing: based on atomic requirements of the standards and based on source code coverage.

The CTesK test system supports 2 modes. The first one is synchronous testing and the second one is asynchronous testing.

The **synchronous** testing is the main mode, which is used in the most of the tests.

The **asynchronous** testing is needed, when we test interfaces with callback functions (see the previous message) or interfaces with blocking functions (such as pthread_mutex_lock() or sem_wait()). It allows to automatically find the correct order of asynchronous events, which correspond to call to funtion under tests or call to callback functions.

Informal example

Here I'll try to answer to the question what "pre/post/coverage" actually does. I supplied the code generated from the abs_spec by a SeC compiler with comments and additional notes to illustrate the way SeC processes .sec files. Hope that helps understanding base concept of our specification language.

// (1) abs_spec is translated to this

```
int oracle_abs_spec( MEDIATOR_FUNC_abs_spec __media, OracleCallProperties* __oracle_properties, struct
ThreadId context, int i )
{
    bool pre = 1;
    int branch[1];
    CallProperties __properties;
    int __res;
```

```
    assertion( __media != (void*)0,  "Mediator of abs_spec is not set up");
    assertion( __oracle_properties != (void*)0, "Oracle properties cannot be NULL");

    __oracle_properties->verdict = 1;
    trace_oracle_start_abs_spec( __oracle_properties->refId );

    assertion(pre, precondition_failed_message);
    {
        assertion(pre, precondition_failed_message);
        {

            int __up = 0, __cmd = -3;
            {
                {
                    {
```

// (2) precondition

```
        /* If the result cannot be represented, the behavior is undefined. */
        /* [INFORMATIVE SECTION: APPLICATION USAGE]
         * [In two's-complement representation, the absolute value of the negative
         * integer with largest magnitude {INT_MIN} might not be representable.]
         */
        /* [Checking is implicit to avoid overflow] */
//        REQ("app.abs.02","Absolute value shall be representable", (i >= 0) || (i + max_IntT) >=0 );
//        return true;
                        if( !strstr("(i >= 0) || (i + max_IntT) >=0"
                        , "TODO_REQ") && (trace_all_requirements || (!strstr("(i >= 0) || (i + max_IntT)
>=0"
                        , "GENERAL_REQ") && !strstr("(i >= 0) || (i + max_IntT) >=0"
                        , "STRUCT_REQ"))) )
                        {
                            traceRequirements("app.abs.02");

                        }

                        if( !((i >= 0) || (i + max_IntT) >= 0) )
                        {
                            traceExceptionValue("Requirement failed"
                            , "{""app.abs.02""} ""Absolute value shall be representable");
                            traceFormattedUserInfo("Requirements failed: {""app.abs.02""} ""Absolute value
shall be representable");
                            { pre = (0); goto __after_pre; }


                        }


                    }

                    ;
                    { pre = (1); goto __after_pre; }


                }
            __after_pre:
                assertion(pre, precondition_failed_message);
                tracePreconditionEnd();
```

// (3) Coverage block

```
                {

/*
        if( i==0 )
            return { Zero, "Zero parameter" };
        else if( i>0 )
        {
            if( i==max_IntT )
                return { IntMax, "INT_MAX" };

            return { Positive, "Positive value" };
        }
        else /* [i<0] */
            return { Negative, "Negative value" };
*/

                    if( i == 0 )
                        { branch[0/*C*/] = 0/*Zero*/; traceCoverageElement("C", 0);  goto
__after_coverage_C; }
                    else
```

```
                        if( i > 0 )
                        {
                            if( i == max_IntT )
                                { branch[0/*C*/] = 1/*IntMax*/; traceCoverageElement("C", 1);  goto
__after_coverage_C; }

                            { branch[0/*C*/] = 2/*Positive*/; traceCoverageElement("C", 2);  goto
__after_coverage_C; }


                        }
                        else
                        { branch[0/*C*/] = 3/*Negative*/; traceCoverageElement("C", 3);  goto
__after_coverage_C; }


                }
                assertion(0, "Not all control paths of coverage 'C' lead to a branch");
            __after_coverage_C:

// (4) Mediator call
                __res = __media( &__properties
                         ,  context
                         ,i
                       );
                __oracle_properties->verdict = __properties.verdict;
                if (__oracle_properties->verdict)
                {
                    {

// (5) Postcondition
                        {

/*
        /* The abs() function shall compute the absolute value of its integer operand, i. */
        /* [Function shall return non-negative result] */
        REQ("abs.01", "Function shall return non-negative result", abs_spec >=0 );
*/

                            if( !strstr("abs_spec >=0"
                            , "TODO_REQ") && (trace_all_requirements || (!strstr("abs_spec >=0"
                            , "GENERAL_REQ") && !strstr("abs_spec >=0"
                            , "STRUCT_REQ"))) )
                            {
                                traceRequirements("abs.01");

                            }

                            if( !(__res >= 0) )
                            {
                                traceExceptionValue("Requirement failed"
                                , "{""abs.01""} ""Function shall return non-negative result");
                                traceFormattedUserInfo("Requirements failed: {""abs.01""} ""Function shall
return non-negative result");
                                { __oracle_properties->verdict = (0); goto __after_post; }


                            }


                        }

                        ;
                        if( i >= 0 )
                        {
                            {
/*
            /* [For non-negative i function abs() shall return i] */
            REQ( "abs.01","For non-negative i function abs() shall return i", i==abs_spec );
*/
                                if( !strstr("i==abs_spec"
                                , "TODO_REQ") && (trace_all_requirements || (!strstr("i==abs_spec"
                                , "GENERAL_REQ") && !strstr("i==abs_spec"
                                , "STRUCT_REQ"))) )
                                {
                                    traceRequirements("abs.01");

                                }

                                if( !(i == __res) )
```

```
                                {
                                    traceExceptionValue("Requirement failed"
                                    , "{""abs.01""} ""For non-negative i function abs() shall return i");
                                    traceFormattedUserInfo("Requirements failed: {""abs.01""} ""For non-
negative i function abs() shall return i");
                                    { __oracle_properties->verdict = (0); goto __after_post; }


                                }


                            }

                            ;

                        }
                        else
                        {
                            {
/*
            /* [For negative i expression i+abs(i) shall equal to zero] */
            REQ( "abs.01", "For negative i expression i+abs(i) shall equal to zero", i+abs_spec == 0 );
*/
                                if( !strstr("i+abs_spec == 0"
                                , "TODO_REQ") && (trace_all_requirements || (!strstr("i+abs_spec == 0"
                                , "GENERAL_REQ") && !strstr("i+abs_spec == 0"
                                , "STRUCT_REQ"))) )
                                {
                                    traceRequirements("abs.01");

                                }

                                if( !(i + __res == 0) )
                                {
                                    traceExceptionValue("Requirement failed"
                                    , "{""abs.01""} ""For negative i expression i+abs(i) shall equal to
zero");
                                    traceFormattedUserInfo("Requirements failed: {""abs.01""} ""For
negative i expression i+abs(i) shall equal to zero");
                                    { __oracle_properties->verdict = (0); goto __after_post; }


                                }

                            }
                        }
/*
        return true;
*/
                        { __oracle_properties->verdict = (1); goto __after_post; }

// (6) Finalization
                    }
                __after_post:
                    if (!__oracle_properties->verdict)
                      ts_trace_bad_oracle_verdict();
                }


            }

        }
    }
    traceOracleEnd();
    return __res;
}
```

(1) oracle_abs_spec is a function that is actually called when abs_spec(context, -1) is used in the scenario. Its structure is equal to the structure of the “specification”.
(2) Precondition
(3) Coverage block. Coverage detects the branch of the functionality that is covered by a particular call.
(4) Mediator call: send command to the agent and decode the return
(5) Post block
(6) Some final actions (release memory when needed, put verdict to the trace)