A Scenario Based On-board Software and Testing Environment for Satellites

Michael Bar-Sinai* Achiya Elyasaf† Aviran Sadon‡ Gera Weiss§

Ben-Gurion University of the Negev, Beer-Sheva, 8410501, Israel

We propose a novel approach to satellites-software development that allows for modularity and formal verification towards safer and more robust satellite software. Specifically, we propose to use *scenario-based programming* where software components (modules) represent different aspects of mission scenarios and anti-scenarios (things that must not happen). We present examples of how specifications can be translated into code artifacts that represent them in a direct and intuitive way. We support this approach with a set of tools that we are designing for the development of *on-board mission software*. The proposed development environment includes an automatic model-checking tool for verifying the produced software and for bug detection. We show that verification can focus on specific parts of the model, on specific logical layers of the applications, and on the entire model at a specific abstraction level. This allows for modular design process, where modules and aspects of the behaviour of the software are tested and verified as soon as their code is ready. Additionally, we describe a *"hybrid laboratory"* for advanced testing of the mission software. On top of the usual components that one may expect to see in such a laboratory, we show how a behavioral-programming component can be added and allow the specification of complex test scenarios that significantly improve the ability to challenge the software under test with complex environment behaviours. Together, we propose a set of tools and show how it can be used to reduce development efforts and to improve the quality, maintainability, and testability of space-missions software.

I Introduction

The most common way for increasing the efficiency and productivity of satellites and other space missions is to make them more autonomic: having better on-board control and troubleshooting capabilities. As a result, these machines are expected to become more software intensive. Since the 1990s, NASA and others have began examining artificial intelligence approaches and algorithms for ground control systems and spacecrafts. The goal of this examinations was to automate some aspects of space missions in order to reduce the number of experts needed to control and to coordinate the missions. This was followed by a research of agent-based methods and control theory concepts, directed towards improving future self-management and survivability in the harsh environments

*barsinam@cs.bgu.ac.il
†achiya@bgu.ac.il
‡sadonav@post.bgu.ac.il
§geraw@cs.bgu.ac.il

This work is supported by the Israeli Innovation Authority, MEIMAD grant number No. 87638011

in which space missions operate. While software-based systems that make complex decisions are common in other fields, bringing advanced features to space is a challenge, as systems must maintain an extremely high reliability standards. In a sense, software engineering is moving in a direction opposite to the culture used in space-mission software development. While techniques such as agile programming, adaptive planning, evolutionary development, early delivery, and continuous integration have many benefits, they cannot be directly implemented in the context of space mission development because of the strict safety and robustness requirements expected from space software.

In this paper we propose a novel approach to satellites-software development that allows for modularity and for formal verification that allow for safer and more robust satellite software. Specifically, we propose to use *scenario-based programming* where software components (modules) represent different aspects of mission and housekeeping scenarios and anti-scenarios (things that must not happen). We present examples of how specifications for flight management can be translated into code artifacts that represent them in a direct and intuitive way. We show how this approach is supported by a development environment that we are designing for creating *on-board mission software*. Our environment includes an automatic model-checking tool for verifying the produced software and for bug detection. Unlike traditional testing, this tool allows for an exhaustive analysis of the code that produces formal guaranties of quality. In Section IV.D, we show that verification can be done on specific parts of the model, on specific logical layers of the applications, and on the entire model at a specific abstraction level. This allows for modular design process, where modules, layers, and aspects of the behaviour can be tested and verified in isolation as soon as their code is ready. Additionally, we describe a *"hybrid laboratory"* for advanced testing of the mission software. In addition to standard components that one finds in such laboratories, our laboratory uses a novel approach that allows for automatic generation of test scenarios, using scenario-based programming. While the examples we provide here are simplified for the purpose of this paper, our group in the Ben-Gurion University is developing on a complete on-board mission software of a satellite. The experience that we are collecting in this project shows that the development environment along with the hybrid laboratory, provides a viable tools for developing reliable satellite software.

II Related Work

Scenario-based programming (SBP) [1,2] is a paradigm for creating executable models for reactive systems that are comprised of independent software modules, or scenarios. Each scenario describes a different aspect of the overall desired and undesired system behaviors, and the scenarios are interwoven at run time by an execution engine. The paradigm was introduced through the language of live sequence charts (LSC) and its Play-Engine implementation [1, 2]. The approach has been generalized to the *behavioral programming* (BP) paradigm [3], and has been extended to other languages and tools, including Java [4], C [5], Erlang [6], JavaScript [7], Blockly [8], SBT [9], Polymorphic Scenarios [10], and ScenarioTools [11]. Research results cover, among others, run-time lookahead (smart playout) [12], synthesis [9], model-checking [13], compositional verification [14], analysis of unrealizable specification [15], abstraction-refinement mechanism [16], automatic correction tools [17], and synchronization relaxation tools [18]. In this paper we apply the BPjs tool [7] and extend it to support the development of space missions towards providing an extendable development environment for programming satellites behaviourally. The tool and the extensions apply concepts , such as logical

execution time and other ideas that allow for embedding SBP in real-time environments, form the papers listed above.

III Using Behavioral Programming for Satellite-Software Development

In this section we elaborate on the execution semantics of Behavioral Programming. We demonstrate these concepts using BPjs, an open-source tool suite for running and analyzing behavioral programs written in JavaScript [7]. BPjs is developed by the authors and others, and was also used to create the satellite on-board software presented in this paper. The reader is referred to `https://bpjs.readthedocs.io` for further information about BPjs, including tutorials and reference.

III.A Behavioral Programming

The Behavioral Programming (BP) [3] paradigm focuses on modular behaviors: how a system behaves within its environment, and how it behaves internally. When creating a system using BP, developers specify many atomic behaviors, that may, must, or must not happen. Each behavior is a simple sequential thread of execution, and is thus called a *b-thread*. B-threads are normally aligned with system requirements, such as "send telemetry when over a ground station" or "don't use the camera while battery is low".

A behavioral program (*b-program*) consists of multiple b-threads. During runtime, all b-threads participating in a b-program are combined, yielding a complex behavior that is consistent with all said b-threads. Unlike other paradigms, BP does not force the developers to pick a single behavior for the system to use. Rather, the system is allowed to choose any compliant behavior. This allows the runtime to optimize program execution at any given moment, e.g., based on available resources. The fact that all possible system behaviors comply with the b-threads (and thus with system requirements), ensures that whichever behavior is chosen, the system as a whole will perform as specified.

B-Threads are combined using a simple protocol, based on events and synchronization points. When a b-thread reaches a point where it needs to synchronize with its peers — for example, in order to use a shared resource — it submits a synchronization statement to the b-program it participates in. This statement details which events said b-thread requests, which events it waits for (but does not request), and which events it blocks (forbids from happening). After submitting the statement, the b-thread is paused. When all b-threads have submitted their statements, we say that the b-program has reached a *synchronization point*. Then, a central event arbiter selects a single event that was requested, and was not blocked. Having selected an event, the arbiter resumes all b-threads that requested or waited for that event. The rest of the b-threads remain paused, and their statements are used in the next synchronization point. The process is repeated throughout the execution of the program. This cycle is presented in frame *BP Cycle* of Figure 1.

III.B A Tutorial Example to Behavioral Programming

To make these concepts more concrete, we now turn to a tutorial example of a simple b-program, written using BPjs. The example presented in this section is an adaptation of one of the first sample programs presented at [4] (the HOT/COLD example). Satellite-related examples will be given in Section IV.

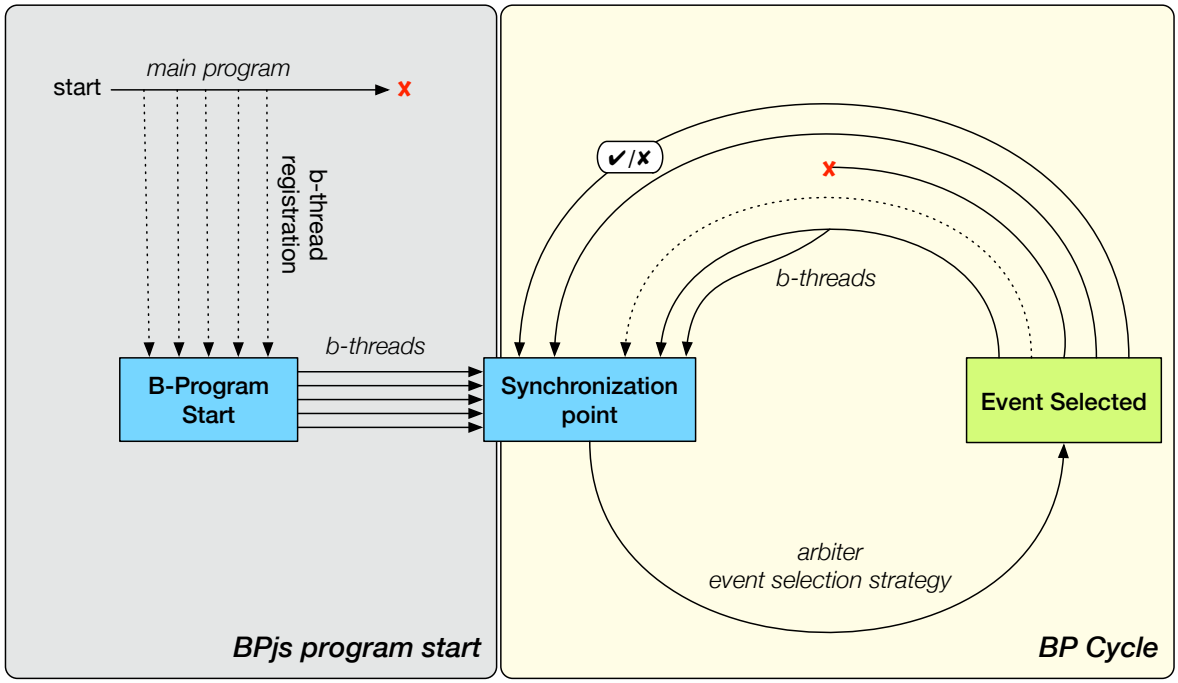

Figure 1. Life cycle of a b-program using BPjs. The execution starts with a regular JavaScript program, which registers b-threads. Once that program terminates, all b-threads are started concurrently. B-threads repeatedly execute internal logic and then synchronize with each other, by submitting a synchronization statement to a central event arbiter. Once all b-threads have submitted their statements, the central event arbiter selects an event that was requested and was not blocked. B-threads that either requested or waited for this event are resumed, while the rest of the b-threads remain paused for the next cycle. During their execution phase, b-threads can terminate, register new b-threads, and perform assertions.

Consider a system with the following requirements:

1. When the system loads, do 'A' three times.
2. When the system loads, do 'B' three times.

Listing 1 shows a b-program (a set of b-threads) that fulfills these requirements. It consists of two b-threads, added at the program start-up. One b-thread, namely `Do-A`, is responsible for fulfilling requirement #1, and the second b-thread, namely `Do-B`, fulfills requirement #2.

Listing 1. A b-program that do 'A' and 'B' three times each. The order between 'A' and 'B' events is arbitrary.

```
bp.registerBThread("Do-A", function(){
  bp.sync({request: bp.event("A")});
  bp.sync({request: bp.event("A")});
  bp.sync({request: bp.event("A")});
});
bp.registerBThread("Do-B", function(){
  bp.sync({request: bp.event("B")});
  bp.sync({request: bp.event("B")});
  bp.sync({request: bp.event("B")});
});
```

The structure of the program is aligned with the system's requirements. It has a single b-thread for each requirement, and it does not dictate the order in which actions are performed (e.g., the following runs are possible: AABBAB, or ABABAB, etc.). This is in contrast to, say, a single-threaded JavaScript program that would have to dictate

Listing 2. A b-thread that ensures that two actions of the same type cannot be executed consecutively, by blocking and additional request of 'A' until the 'B' is performed, and vice-versa.

```
bp.registerBThread("interleave", function() {
  while ( true ) {
    bp.sync({waitFor: bp.event("B"), block: bp.event("A")});
    bp.sync({waitFor: bp.event("A"), block: bp.event("B")});
  }
});
```

exactly when each action should be performed. Thus, traditional programming paradigms are prone to over specification, while behavioral programming avoids it.

While a specific order of the actions was not required originally, in some cases, this behavior may represent a problem. I.e., we now have an additional requirement that the user detected after running the initial version of the system:

3. Two actions of the same type cannot be executed consecutively.

While we may add a condition before requesting 'A' and 'B', the BP paradigm encourages us to add a new b-thread for each new requirement. Thus we add a b-thread, called `Interleave`, presented in Listing 2.

The `interleave` b-thread ensures that are no repetitions by forcing an interleaved execution of the performed actions — first 'A' is blocked until 'B' is executed, first 'B' is blocked until 'A' is executed. This is done by using the `waitFor` and `block` mechanisms. It is interesting to note that this b-thread can be added and removed without affecting other b-threads. This is an example of a *purely additive* change, where system behavior is altered to match a new requirement without affecting the existing behaviors. While not all changes to a b-program are purely additive, many useful changes are, as demonstrated here.

IV Managing the EPS and the ADCS Using BPjs

In this section we demonstrate how BPjs can be used for developing software that manage two satellite sub-systems. The first sub-system is *EPS* (Electric Power System). For this example, we assume that our EPS can switch between three operational modes: "good", "low", and "critical", that are functions of the battery voltage. The second subsystem is *ADCS* (Attitude Determination and Control System), again we assume it can switch between three attitude control modes: 1) *detumbling:* used for reducing angular rates, usually after deployment; 2) *sun pointing:* used for pointing the satellite's solar panels towards the sun in order to charge its batteries; and 3) *payload pointing:* used for pointing the satellite's payload to a desired set point. The ADCS is activated when a payload pass is scheduled. For this example, we assume that both "sun pointing" and "payload pointing" require low angular rates to be active. We also take into account that both the EPS and ADCS has a dedicated hardware that takes care of "low-level requirements" (e.g., activation of actuators and sensors) and that the on-board software sends only "high-level commands" (e.g., switch to 'X' operational mode). Assuming this system architecture, the OBC has the all the cross-system knowledge, thus expected to make the cross-system decisions.

Our on-board b-program software has a dedicated b-thread for each sub-system, implementing its logic. The logic takes inspiration from other real satellite mission, as [19, 20]. The program, as well as additional documentation, are given in [21].

IV.A The EPS

The EPS logic is specified by the b-thread presented in Listing 3. We initially wait for an EPS telemetry event, containing current telemetry received from the EPS board. In this example, we assume this includes the battery voltage and the current EPS mode. According to the battery voltage field in the EPS telemetry, an initial EPS mode is requested using the `setEPSMode` event. After this initialization is done, every time an EPS telemetry is received (i.e., an `EPSTelemetry` event is selected), this b-thread requests an EPS-mode change based on the current battery voltage.

The code in Listing 3 avoids stale requests that rely on old telemetries using a notable technique. Whenever it requests an EPS-mode change, it also waits for EPS-telemetry events. If a new telemetry arrives before the set-mode request has been granted (that is, an `EPSTelemetry` event was selected by the b-program before the requested `setEPSMode` event was), the set-mode request is cancelled and the b-thread has a chance to re-evaluate it.

Listing 3. EPS b-thread.

```
var EPSTelem = bp.EventSet("EPSTelem", function(e){
    return e instanceof EPSTelemetry;
});

var LOW_MAX = 70;
var GOOD_MIN = 60;
var CRITICAL_MAX = 50;
var LOW_MIN = 40;

bp.registerBThread("EPS - Turn ON/OFF logic", function () {

    /* Init */
    var ePSTelem = bp.sync({waitFor: EPSTelem});
    if (ePSTelem.vBatt >= GOOD_MIN) {
        bp.sync({waitFor: EPSTelem,
                 request: StaticEvents.SetEPSModeGood});
    } else if (ePSTelem.vBatt >= LOW_MIN) {
        bp.sync({waitFor: EPSTelem,
                 request: StaticEvents.SetEPSModeLow});
    } else {
        bp.sync({waitFor: EPSTelem,
                 request: StaticEvents.SetEPSModeCritical});
    }
    delete ePSTelem;

    // ongoing control loop
    while (true) {
        var ePSTelem = bp.sync({waitFor: EPSTelem});
        switch ( ePSTelem.mode ) {
            case EPSTelemetry.EPSMode.Good:
                if (ePSTelem.vBatt < GOOD_MIN) {
                    bp.sync({waitFor: EPSTelem,
                             request: StaticEvents.SetEPSModeLow});
                }
                break;

            case EPSTelemetry.EPSMode.Low:
                if (ePSTelem.vBatt > LOW_MAX) {
                    bp.sync({waitFor: EPSTelem,
                             request: StaticEvents.SetEPSModeGood});
                }
                if (ePSTelem.vBatt < LOW_MIN) {
                    bp.sync({waitFor: EPSTelem,
                             request: StaticEvents.SetEPSModeCritical});
                }
                break;

            case EPSTelemetry.EPSMode.Critical:
                if (ePSTelem.vBatt > CRITICAL_MAX) {
                    bp.sync({request: StaticEvents.SetEPSModeLow});
```

```
                }
                break;
        }
    }
});
```

IV.B The ADCS

The ADCS logic is specified by the b-thread presented in Listing 4. It is initialized by requesting a `SetADCSModeDetumbling` event. Each time an `ADCSTelemetry` event is selected, the appropriate `SetADCSMode` event, based on the following: 1) the current ADCS mode; 2) the current angular rate; and 3) whether the satellite is during an active pass or not. This b-thread, too, avoids stale set requests by waiting for telemetry events while requesting ADCS changes.

Listing 4. ADCS b-thread.

```
var ADCSTelem = bp.EventSet("ADCSTelem", function (e) {
    return e instanceof ADCSTelemetry;
});

bp.registerBThread("ADCS Mode Switch logic", function () {

    /* Init Deployment*/
    bp.sync({request: StaticEvents.SetADCSModeDetumbling});

    /* ongoing */
    while (true) {
        var aDCSEvent = bp.sync({waitFor: ADCSTelem});

        switch ( aDCSEvent.mode ) {
            case ADCSTelemetry.ADCSMode.Detumbling:
                if (aDCSEvent.angularRate == "Low" && aDCSEvent.isActivePass) {
                    bp.sync({waitFor: ADCSTelem,
                             request: StaticEvents.SetADCSModePayloadPointing});
                } else if (aDCSEvent.angularRate == "Low") {
                    bp.sync({waitFor: ADCSTelem,
                             request: StaticEvents.SetADCSModeSunPointing});
                }
                break;
            case ADCSTelemetry.ADCSMode.SunPointing:
                if (aDCSEvent.angularRate == "Low" && aDCSEvent.isActivePass) {
                    bp.sync({waitFor: ADCSTelem,
                             request: StaticEvents.SetADCSModePayloadPointing});
                } else if (aDCSEvent.angularRate == "High") {
                    bp.sync({waitFor: ADCSTelem,
                             request: StaticEvents.SetADCSModeDetumbling});
                }
                break;
            case ADCSTelemetry.ADCSMode.PayloadPointing:
                if (aDCSEvent.angularRate == "Low" && !aDCSEvent.isActivePass) {
                    bp.sync({waitFor: ADCSTelem,
                             request: StaticEvents.SetADCSModeSunPointing});
                } else if (aDCSEvent.angularRate == "High") {
                    bp.sync({waitFor: ADCSTelem,
                             request: StaticEvents.SetADCSModeDetumbling});
                }
                break;
        }
    }
});
```

IV.C A Cross-System Requirement

On top of the the logic of each subsystem, an additional logic is required for handling behaviors between different subsystems. Consider for example the following cross-system requirement:

The ADCS should not remain in, or switch to, payload-pointing mode while the EPS mode is low or critical.

Whether this requirement was originally given, or after the first two were written and tested, the BP paradigm encourages us to add a new b-thread for blocking this newly undesired behavior. The additional b-thread is given in Listing 5.

Listing 5. Integrator b-thread.

```
bp.registerBThread("EPS & ADCS Integrator", function () {
    while (true) {
        var ePSTelem2 = bp.sync({waitFor: EPSTelem});
        while ( ePSTelem2.currentEPSMode == "Low" ||
                ePSTelem2.currentEPSMode == "Critical" ) {
            if ( ePSTelem2.isActivePass ) {
                bp.sync({waitFor: ADCSTelem,
                         request: StaticEvents.PassDone,
                         block: StaticEvents.SetADCSModePayloadPointing
                });
            }
            var aDCSEvent2 = bp.sync({waitFor: ADCSTelem,
                                      block: bp.Event("SetADCSModePayloadPointing")});
            if (aDCSEvent2.currentADCSMode == "PayloadPointing") {
                bp.sync({waitFor: ADCSTelem,
                         request: StaticEvents.SetADCSModeSunPointing,
                         block: StaticEvents.SetADCSModePayloadPointing
                });
            }
            var ePSTelem2 = bp.sync({waitFor: EPSTelem,
                                     block: StaticEvents.SetADCSModePayloadPointing
            });
        }
    }
});
```

IV.D Formal Verification of B-Programs

The BPjs tool suite can be used to *verify* system correctness (i.e., that it behaves correctly). This verification can be performed on specific parts of a model, on specific logical layers of an application, or on an entire model at a specific abstraction level. This allows for a modular design process, where modules are tested and verified as soon as their code is ready.

A b-program is verified by looking at all of its possible runs, and ensuring they all comply with a set of formal requirements. This is a computationally-intensive process, and may also be infinite. To this end, BPjs enables model developers to limit the explored runs, for example by limiting the length of each explored run, or by including explicit assumptions, such as the keeping battery voltage in a limited range. Additional approaches for reducing the computational load are described in [13, 14].

Consider for example that we wish to verify that the ADCS does not switch to the payload-pointing mode while the EPS mode is low or critical. Towards this end we use the `bp.ASSERT()` method for marking such possibility as *invalid*. During the verification processes we search for possible runs that reach such states. The assertion is given in a new b-thread, presented in Listing 6.

Listing 6. A safety requirement b-thread, generating a false assertion when the there is a request to switch to the payload-pointing mode while the EPS mode is low or critical.

```
bp.registerBThread("NeverPointingOnLow", function () {
    var relevantEvents = [EPSTelem, USE_PAYLOAD];
    /* Init */
    var canPoint;
    var evt = bp.sync({waitFor: relevantEvents});
    if (EPSTelem.contains(evt)) {
        canPoint = evt.mode.equals(EPSTelemetry.EPSMode.Good);
    }

    /* ongoing verification */
    while (true) {
        var pointingRequested = false;
        var evt = bp.sync({waitFor: relevantEvents});
        if (EPSTelem.contains(evt)) {
            canPoint = evt.mode.equals(EPSTelemetry.EPSMode.Good);
        } else if (USE_PAYLOAD.equals(evt)) {
            pointingRequested = true;
        }

        bp.ASSERT(!(pointingRequested && !canPoint));
    }
});
```

During execution, a false assertion causes the b-program to terminate, and the host Java program is informed of the failure. From the b-program side, assertions are an "emergency shutdown" mechanism. From the host Java program side, false assertions allow restarting the b-program with different parameters, or with a modified set of b-threads, in order to avoid the error because of which the false assertion was generated. One example for this is automated patching of b-programs [16, 17].

The main usage of assertions, though, is during verification process, where BPjs traverse the possible b-program runs, looking for invalid states. Once such state is found, BPjs returns the sequence of the events that lead it to the discovered invalid state. This type of verification has an advantage over tests, since it can specify what went wrong, rather than "just" detecting when something is wrong.

Simulating the Environment For the Verification

As we previously said, the b-program is verified by looking at all of its possible runs. The verification is done on the b-program only, excluding the actual system interactions with the environment. Therefore, the environment events (e.g., the telemetry events) have to be simulated during the verification process.

We start with a b-thread that randomly generates all of the possible environment events (shown in Listing 7). It requests EPSTelemetry and ADCSTelemetry events in order, but each event is arbitrarily chosen from all possible events of each type. For example, and EPSTelemtry event can be any of the 606 possible combinations of active pass (yes/no), battery level (0 to 100 inclusive) and EPS mode (good/low/critical).

Listing 7. A b-thread that randomly generates all of the possible EPSTelemetry and ADCSTelemetry event.

```
var ACTIVE_PASSES = [false,true];

// Generate all possible EPS Telemetries
var EPS_MODES = EPSTelemetry.EPSMode.values();
var possibleEPSes = [];
for ( var vBatt=0; vBatt <= 100; vBatt++ ) {
    for ( var modeIdx in EPS_MODES ) {
        for ( var activePassIdx in ACTIVE_PASSES ) {
            possibleEPSes.push( EPSTelemetry(vBatt,
                                             EPS_MODES[modeIdx],
                                             ACTIVE_PASSES[activePassIdx]) );
        }
    }
```

```
}

// Generate all possible ADCS Telemetries
var ADCS_MODES = ADCSTelemetry.ADCSMode.values();
var ANGULAR_RATES = ADCSTelemetry.AngularRate.values();
var possibleADCSes = [];
for ( var adcsModeIdx in ADCS_MODES ) {
    for ( var angularRateIdx in ANGULAR_RATES ) {
        for ( var activePassIdx in ACTIVE_PASSES ) {
            possibleADCSes.push( ADCSTelemetry(ADCS_MODES[adcsModeIdx],
                                               ANGULAR_RATES[angularRateIdx],
                                               ACTIVE_PASSES[activePassIdx]) );
        }
    }
}

bp.registerBThread("Environment", function(){
    while ( true ) {
        bp.sync({request:possibleEPSes}); // Chooses a random EPSTelemery event
        bp.sync({request:possibleADCSes}); // Chooses a random ADCSTelemery event
    }
});
```

We used these b-threads to verify our system. First, as a baseline, we removed the cross-system b-thread (Listing 5), and successfully verified that the system indeed violates the cross-system requirement. Next, we tried to verify the system including, the cross-system requirement. The verification process scanned approximately 650K possible states using 1.3M iterations without finding any violations. Since we choose the events randomly, it is not promised that we will cover all possible runs.

A more realistic environment simulation generates the telemetry events according to previous selected events. For example, whenever a `SetEPSModeGood` event is selected, the following EPSTelemetry event will represent this mode switch. The complete code for this simulation can be found in [21]. We verified the system using this environment, again in two steps, the first without the cross-system b-thread, and the second with. As before, the first run resulted with a trace of the events that led to the violation, presented in Listing 8. For all possible runs of up to 200 consecutive selected events, the second approach was able verify that there are no violations—in less than 2.5 minutes. Runs of a greater depth (i.e., longer) are also possible to be verified using this approach.

Listing 8. A trace of the events that led to a violation of the system that does not include the the cross-system b-thread.

```
[BEvent name:SetADCSModeDetumbling]
[BEvent name:EnvSetAngularRateLow]
[BEvent name:PassDone]
[EPSTelemetry vBatt:1 currentEPSMode:Good activePass:false]
[BEvent name:SetEPSModeCritical]
[ADCSTelemetry currentADCSMode:Detumbling angularRate:Low activePass:false]
[BEvent name:SetADCSModeSunPointing]
[EPSTelemetry vBatt:2 currentEPSMode:Critical activePass:false]
[ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
[EPSTelemetry vBatt:3 currentEPSMode:Critical activePass:false]
[ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
[EPSTelemetry vBatt:4 currentEPSMode:Critical activePass:false]
[ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
[EPSTelemetry vBatt:5 currentEPSMode:Critical activePass:false]
[ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
[EPSTelemetry vBatt:6 currentEPSMode:Critical activePass:false]
[ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
[EPSTelemetry vBatt:7 currentEPSMode:Critical activePass:false]
[ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
[EPSTelemetry vBatt:8 currentEPSMode:Critical activePass:false]
[ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
[EPSTelemetry vBatt:9 currentEPSMode:Critical activePass:false]
[ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
[EPSTelemetry vBatt:10 currentEPSMode:Critical activePass:false]
[ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
[BEvent name:ActivePass]
```

```
27  [EPSTelemetry vBatt:11 currentEPSMode:Critical activePass:true]
28  [ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:true]
29  [BEvent name:SetADCSModePayloadPointing]
```

V Hybrid Lab Testing Environment

In this section we describe our on-going work on a novel hybrid laboratory for advanced simulation and testing of a satellite-mission software, where scenario-based programming is used to automatically generate complex test scenarios. This lab is already being used for integrating and testing a full-scale satellite on-board software that by our group at Ben-Gurion University is working on.

The hybrid lab consists of a simulation computer (desktop PC), and an *on-board computer* (OBC), on which the on-board software runs. The simulation computer runs MATLAB, STK, and custom software for simulating satellite sensors (e.g., temperature, sun, magnetometers), actuators (e.g., magneto-torquers, wheels, heaters), and all the other satellite subsystems that the OBC interacts with (e.g., EPS, communications, and payload). MATLAB is used for evaluating and running the satellite's dynamics model. STK is used for evaluating and running the environment model (e.g., gravity and magnetic fields, and earth coverage).

The hybrid lab's layout is depicted in Figure 2. The simulation computer interacts with the OBC using the native electrical on-board connectors used by the real hardware that the PC simulates. Thus, from the OBC's standpoint, the hybrid lab setup is identical to the setup used in orbit.

The presented hybrid lab allows developers to examine how the on-board software performs in various scenarios. Such scenarios may be hardware failures, specific orbits and environmental conditions, written as a scenario-based test software. Additionally, the lab's layout makes it possible to gradually replace simulated devices with actual ones. In turn, this gradual replacement may be used to test the integration of hardware components with the simulator, the OBC, and the on-board software.

We note that this hybrid laboratory design allows for testing on-board satellite software that is not necessarily written using scenario-based programming.

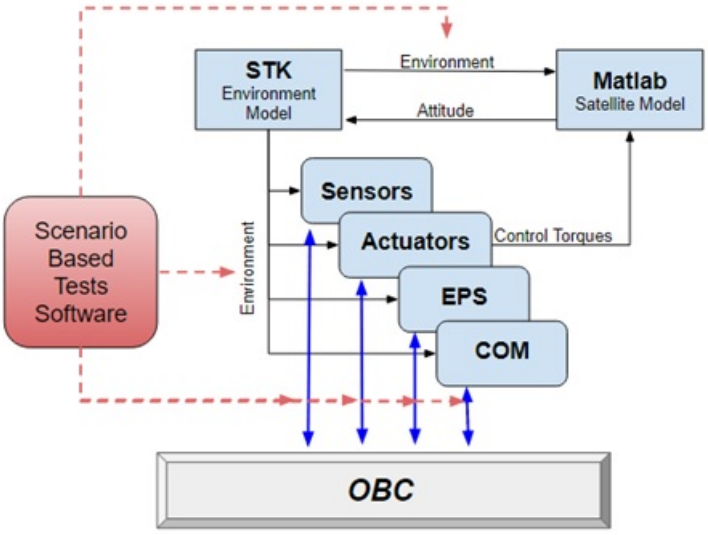

Figure 2. The hybrid lab layout. The simulation computer simulates the satellite's dynamics, environment, and various hardware components. The OBC and the simulation computer interacts using the native electrical on-board connectors used by the real hardware that the PC simulates.

VI Conclusion

We present three contributions to the field of on-board satellite software programming and testing:

Software. We presented a novel approach for programming satellites software, that allows for incremental development and an alignment with the requirements. The approach is based on the behavioural programming methodology and on the BPjs tool. We use *scenario-based programming* where software components (modules) represent different aspects of mission scenarios and anti-scenarios (sequences of events that must not happen).

Verification. A tool and methodology for verifying the correctness of the proposed on-board satellite software, by looking at all of its possible runs, and ensuring they all comply with a set of formal requirements. This verification can be performed on specific parts of a model, on specific logical layers of an application, or on an entire model at a specific abstraction level.

Testing. A novel hybrid laboratory for advanced simulation and testing of a satellite-mission software, where scenario-based programming is used to automatically generate complex test scenarios.

While the provided examples were simplified for the purpose of this paper, we strongly believe that the suggested approaches can be applied to full scale on-board satellite software. Our group in Ben-Gurion University is currently working on such a case study that hope that will validate the scaleability of the proposed approach.

References

[1] Damm, W. and Harel, D., "LSCs: Breathing life into message sequence charts," *Form. Methods Syst. Des.*, Vol. 19, No. 1, 2001, pp. 45–80, doi:10.1023/A:1011227529550.

[2] Harel, D. and Marelly, R., *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer Science & Business Media, 2003.

[3] Harel, D., Marron, A., and Weiss, G., "Behavioral Programming," *Communications of the ACM*, Vol. 55, No. 7.

[4] Harel, D., Marron, A., and Weiss, G., "Programming coordinated behavior in Java," in D'Hondt, T., ed., "ECOOP – Object-Oriented Programming," Springer Berlin Heidelberg, 2010.

[5] Shimony, B. and Nikolaidis, I., "On coordination tools in the PicOS tuples system," in "ICSE," , 2011, pp. 19–24.

[6] Wiener, G., Weiss, G., and Marron, A., "Coordinating and Visualizing Independent Behaviors in Erlang," in "Proceeding 9th ACM SIGPLAN Work. Erlang," ACM, 2010, pp. 13–22.

[7] Bar-Sinai, M., Weiss, G., and Shmuel, R., "BPjs: An Extensible, Open Infrastructure for Behavioral Programming Research," in "Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings," ACM, New York, NY, USA, MODELS '18, 2018, pp. 59–60, doi:10.1145/3270112.3270126.

[8] Marron, A., Weiss, G., and Wiener, G., "A decentralized approach for programming interactive applications with javascript and blockly," in "Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions," ACM, 2012, pp. 59–70.

[9] Kugler, H., Plock, C., and Roberts, A., "Synthesizing biological theories," in "CAV," , 2011, pp. 579–584.

[10] Maoz, S., "Polymorphic scenario-based specification models: Semantics and applications," in "International Conference on Model Driven Engineering Languages and Systems," Springer, 2009, pp. 499–513.

[11] Greenyer, J., Gritzner, D., Gutjahr, T., König, F., Glade, N., Marron, A., and Katz, G., "ScenarioTools–A tool suite for the scenario-based modeling and analysis of reactive systems," *Science of Computer Programming*, Vol. 149, 2017, pp. 15–27.

[12] Harel, D., Kugler, H., Marelly, R., and Pnueli, A., "Smart play-out of behavioral requirements," in "FMCAD," Springer, Vol. 2, 2002, pp. 378–398.

[13] Harel, D., Lampert, R., Marron, A., and Weiss, G., "Model-checking Behavioral Programs," in "Proceedings of the Ninth ACM International Conference on Embedded Software," ACM, New York, NY, USA, EMSOFT '11, 2011, pp. 279–288, doi:10.1145/2038642.2038686.

[14] Harel, D., Kantor, A., Katz, G., Marron, A., Mizrahi, L., and Weiss, G., "On composing and proving the correctness of reactive behavior," in "EMSOFT," , 2013, pp. 1–10.

[15] Maoz, S. and Sa'ar, Y., "Counter play-out: executing unrealizable scenario-based specifications," in "ICSE," , 2013, pp. 242–251.

[16] Katz, G., "On module-based abstraction and repair of behavioral programs," in "Log. Program. Artif. Intell. Reason.", ACM, pp. 518–535, 2013.

[17] Harel, D., Katz, G., Marron, A., and Weiss, G., "Non-intrusive Repair of Safety and Liveness Violations in Reactive Programs," *Transactions on Computational Collective Intelligence XVI*, pp. 1–33.

[18] Harel, D., Kantor, A., and Katz, G., "Relaxing Synchronization Constraints in Behavioral Programs," in "Logic for Programming Artificial Intelligence and Reasoning," , 2013, pp. 355–372.

[19] Ran, D., Sheng, T., Cao, L., Chen, X., and Zhao, Y., "Attitude control system design and on-orbit performance analysis of nano-satelliteTian Tuo 1," *Chinese Journal of Aeronautics*, Vol. 27, No. 3, 2014, pp. 593–601.

[20] Deng, S., Meng, T., Wang, H., Du, C., and Jin, Z., "Flexible attitude control design and on-orbit performance of the ZDPS-2 satellite," *Acta Astronautica*, Vol. 130, 2017, pp. 147–161.

[21] *Appendices for A Scenario Based On-board Software and Testing Environment for Satellites. 59 IACAS*, 2019. `https://github.com/bThink-BGU/59IACAS-Appendix`.