An open simulation kernel for system-level spacecraft digital twinning

Serge Barral [(1)], Adam Chikha [(1)]

[(1)] Asynchronics sp. z o.o., Wilcza 72/14, 00-670, Poland
serge.barral@asynchronics.com, adam.chikha@asynchronics.com

ABSTRACT

System-level simulation is an activity that focuses on providing a full digital replica of the spacecraft and its subsystems at the functional level. System simulators can be typically delivered much ahead of actual hardware and their extreme flexibility lends them unique capabilities in terms of failure injection and simulation of operational conditions. While extensively utilised by Large System Integrators, their integration into the New Space sector and small spacecraft development remains limited. To bridge this gap and democratise access to system simulation at small spacecraft integrators and academic institutions, a high-performance, open-source system simulation framework was developed that relies on a modern software development stack and incorporates asynchronous programming techniques to deliver flexibility and scalability.

1 INTRODUCTION

The use of simulation for mission design dates back to the dawn of the space age. Its application for system-wide modelling of spacecraft platforms at the avionics level (electrical and data buses, state machines, etc.) is, however, a more recent development. This activity, commonly referred to as system-level simulation, has become a key part of spacecraft development at large system integrators (LSIs) and spans a number of applications ranging from on-board software simulation facilities (SVF) to ground system tests simulators and training, operations and maintenance simulators (TOMS).

European LSIs and space agencies have heavily invested in the development of dedicated simulation frameworks, which today include Simulus/SimSat (ESA), Basiles (CNES), SimTG (Airbus D&S), EuroSim (Airbus D&S), Gram (TAS), K2 (TAS) and Rufos (OHB). The consolidation of these various efforts is an ongoing process, supported to a large extent by the ESA RatioSim initiative and the ECSS SMP standardisation effort [1][2].

In contrast, the rate of adoption of system-level simulation remains marginal in small satellite missions, including in the nascent New Space sector. Considering that an estimated 43% of small satellite missions fail [3] and that 81% of these failures could be prevented with a more thorough validation and verification process [4], the introduction of system-level simulation in V&V activities appears as an obvious low-hanging fruit.

We posit that the main obstacles to wider adoption in smaller missions extends beyond lack of awareness and largely stem from an absence of affordable, easy-to-use software frameworks. With the exception of the end-of-life BASILES simulator [5], no space industry simulators are available

on non-discriminatory terms, and none of them were meant for use by third parties, expose the idiosyncrasies of their origin institution due to their tight integration with internal tools and workflows.

We report here on the development of an open source, asynchronous discrete-event simulation kernel [6][7] meant to serve as a foundation for an open ecosystem for spacecraft system simulation and digital twinning. Although it intentionally eschews ECSS SMP compatibility for the sake of ergonomy and performance, its development was guided and informed by experience with LSI simulators. Beyond its liberal open-source licensing, the simulator kernel differs from other simulators in several key aspects, including:

- implementation in the Rust programming language rather than C++,
- fully asynchronous inter-model communication,
- transparent, concurrency-safe parallelization,
- ergonomic API and lack of dependency on external code pre-processor or Integrated Development Environment.

This contribution provides an overview of the simulator architecture and implementation. It is followed by an introduction to the main modelling concepts, illustrated by an application to a simple cold gas propulsion system. It concludes with an outlook on its current applications for digital twinning and spacecraft-wide simulation and on prospective application.

2 DESIGN OVERVIEW AND RATIONALE

2.1 Asynchronous Is Better Than Synchronous

From a high-level perspective, asynchronous data exchange is a simple concept. In the context of discrete-event simulators (DES), it is usually understood as the ability for a model to send data to another model in a fire-and-forget fashion, allowing such data to be consumed at a later time by the recipient.

The various SMP specifications, most notably SMP2 and ECSS SMP, provide support for asynchronous inter-model communication through events and dataflow patterns, where a signal or the modification of an output by a sender may be processed by a connected model in a deferred manner. Other SMP idioms such as interfaces and immediate events are, however, synchronous by nature: the caller causes the callee to be invoked immediately and its execution is resumed as soon as the call returns. Such synchronous patterns enable models to perform a request to another model and wait until a response is received before proceeding, a pattern which in C++ could only be performed efficiently with synchronous calls until recently[1].

Allowing synchronous communication between models comes, however, with two significant downsides:

1. *single-threaded execution*: the execution of the various models that compose a simulation cannot be parallelized without requiring the programmer to explicitly handle synchronisation, a notoriously difficult and error-prone task,
2. *unexpected model mutation*: because the designer of a model has no control on how models are connected during simulation bench assembly, if models are connected in a way that allows model A to synchronously calls model B which in turn synchronously calls model A, then data owned by model A might be modified while waiting for the call to model B to

[1] This is now possible with *coroutines*, which were introduced in C++20.

return, a situation which may not be expected and frequently leads to software defects that are very difficult to detect.

The first shortcoming has only become more acute with the plateauing of single-core processor performance and the ever-growing number of processor cores available on commodity hardware. To the knowledge of the authors, as of today, no production-ready SMP simulator supports transparent parallel execution, and it is unlikely that one ever will.

2.2 Asynchronous Programming And The Case For Rust

The request-reply pattern enabled by synchronous SMP interfaces and events is nevertheless extremely useful, and often essential in the practical implementation of simulation models.

Because this pattern is also useful in the implementation of network services which routinely deal with thousands of concurrent network requests, many mainstream programming languages, starting with C#, have introduced language-level support for what is referred to as *asynchronous programming*. The so-called *async* paradigm enables programmers to write programs as if they were to be executed in a sequential, uninterrupted manner, yet their execution may be suspended at specific locations marked by the `await` keyword, in a manner that is largely transparent to the programmer.

```rust
// A regular, synchronous function.
fn read_bus_voltage() {
    // A synchronous call to a model performing the acquisition of the bus
    // voltage: the `send` request is executed immediately and returns as soon
    // as it completes.
    let bus_voltage = remote_acq_port.send(BUS_VOLTAGE_REQ);
}

// An asynchronous function.
async fn read_bus_voltage() {
    // An asynchronous call: execution may be suspended to let the simulation
    // runtime run other tasks; execution will resume at a later time once the
    // request has been processed.
    let bus_voltage = remote_acq_port.send(BUS_VOLTAGE_REQ).await;
}
```

In the context of a simulation, this means that when a model sends a request, the runtime suspends the model's execution and allocates the current operating system process for the execution of another model's tasks, which may or may not be the one servicing the request; once the request is eventually executed and the response generated, the runtime pushes the task that issued the request to the run queue so it can be resumed whenever processor resources become available. While asynchronous programming can, and often is, combined with parallel execution on several processor cores, it does not need to be. Javascript, notably, supports the concurrent execution of *async* routines but interleaves their execution on a single system thread only.

In most programming languages that support it, the management of concurrent asynchronous routines and their possible multiplexing on several processor cores is handled by the runtime and cannot be tuned to specific use-cases. Rust and modern C++ implementations are an exception to this rule and allow (and even require) custom executors. This is an opportunity for application

domains outside network services where concurrent workloads may benefit from different optimisations.

Due to its overall greater support for modern programming facilities and excellent support for concurrent programming, the Rust programming language was preferred to C++ for the development of a greenfield parallel executor dedicated to simulation. Rust's strong focus on memory safety and its fast adoption in safety- and mission-critical industries were other determining factors in this choice.

2.3 Efficient Parallel Execution

Parallel discrete event simulators have been a field of active research for many decades. Most use-cases focus, however, on distributed systems and on appropriate strategies to limit network overhead by letting simulation network nodes to process by letting the nodes run out of . These strategies present various trade-offs and, almost invariably, demand specific expertise from the part of the simulation user.

In contrast, synchronisation within today's multi-core processors can be done with very low latency, offering a potential for parallelization that remains largely untapped in the context of discrete event simulation.

Historically, parallelizing a simulation on such a processor would involve running each simulation model on a different operating system thread (a logical unit of execution) and let the operating system multiplex these threads on the available processor cores (physical units of execution). This is, however, a costly approach due to *context switch*, an operation that requires making a copy of the complete state of the process associated with a model, and reloading this copy when execution resumes.

With the advent of asynchronous programming, cheaper strategies for parallelization on multi-core processors have emerged where the multiplexing of tasks over physical cores is performed without assistance from the operating system and, crucially, without need for costly context switches. While such so-called *cooperative multithreading* is strictly less powerful than operating system threads, its limitations are largely irrelevant in the context of simulation.

State of the art asynchronous runtimes such as the Go language scheduler and the *tokio* Rust library [8] are very successful examples of the application of one such strategy, referred to as *work-stealing*. In a work-stealing simulation runtime, each model is initially assigned to a particular processor core. Wherever a model in *idle* state receives a message on one of its inputs, its state changes to *ready-to-run* and it is queued for execution on the same processor as the model that triggered the output. If, however, the runqueue of a processor becomes empty, one or several models in *ready-to-run* state are migrated ("stolen") from other models to maximise the use of available processing resources.

2.4 Transparent Parallel Execution

Because simulation model designers are not necessarily versed into the subtleties of multi-threaded programming, a key goal in the development of the simulator was to make parallel execution fully transparent.

This goal was achieved by combining the work-stealing strategy outlined earlier with a programming paradigm known as the *actor model*. In the actor model, the various messages sent to the inputs of a simulation model are buffered in a single thread-safe queue associated with this specific model, and dispatched to the model's inputs according to their respective order of arrival. This ensures that a model only ever processes one message at a time, thus avoiding the possibility

of data races, i.e. the concurrent modification of the same data by different processes. From a model designer viewpoint, therefore, the implementation of a model looks nearly exactly as if the model was running on a conventional, single-threaded executor.

The actor model also allows the simulator to uphold specific message ordering guarantees that are unaffected by parallel execution, which can be stated as follows:

- *one-to-one message ordering guarantee*: if model *A* sends two events or requests *M1* and then *M2* to model *B*, then *B* will always process *M1* before *M2*,
- *transitivity guarantee*: if *A* sends *M1* to *B* and then *M2* to *C* which in turn sends *M3* to *B,* even though *M1* and *M2* may be processed in any order by *B* and *C*, it is guaranteed that *B* will process *M1* before *M3*.

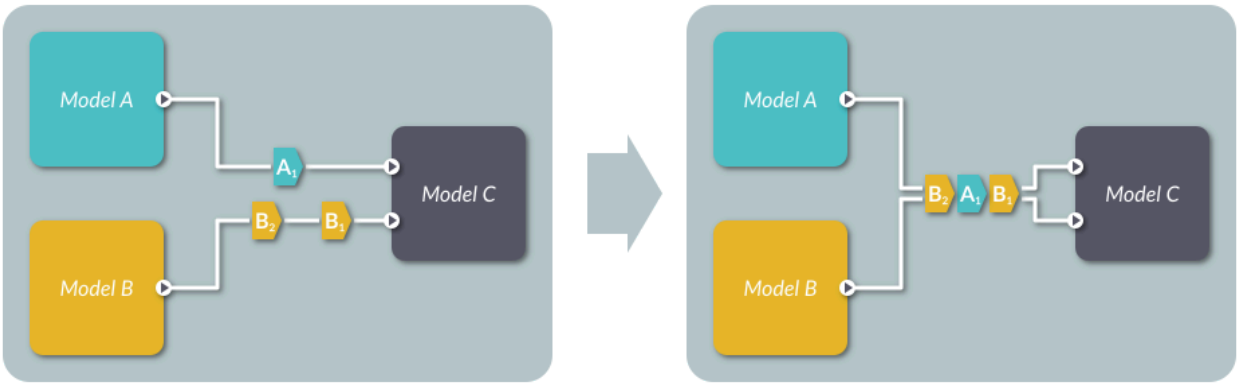

Figure 1: Implementation of asynchronous message-passing between simulation models based on the actor model: input messages are dispatched from a single ordered queue.

3 MODEL IMPLEMENTATION

3.1 A simple example

A simplified cold gas propulsion system including a thruster and its tank are used to illustrate the general usage of the simulation model API.

The system interfaces and connections to be modelled are shown in Fig. 2.

3.2 Tank model

The propellant mass is assigned to the tank upon model construction, and is thereafter updated by the tank model whenever a model (in this case the thruster) sends to the `consumed_mass` input the amount of propellant that has been extracted from the tank.

A model may also inquire the tank for the current reading from the pressure sensor through a `pressure` telemetry request port.

Finally, whenever propellant mass is updated, the new value of the propellant mass is broadcast to any model connected to the `propellant_mass` output port (e.g. a spacecraft dynamics model). The only subtlety in the implementation of this model is the necessity to broadcast the initial propellant mass when the simulation starts, which can be done by implementing the `init` method of the `Model` object interface (or *trait* as interfaces are called in Rust).

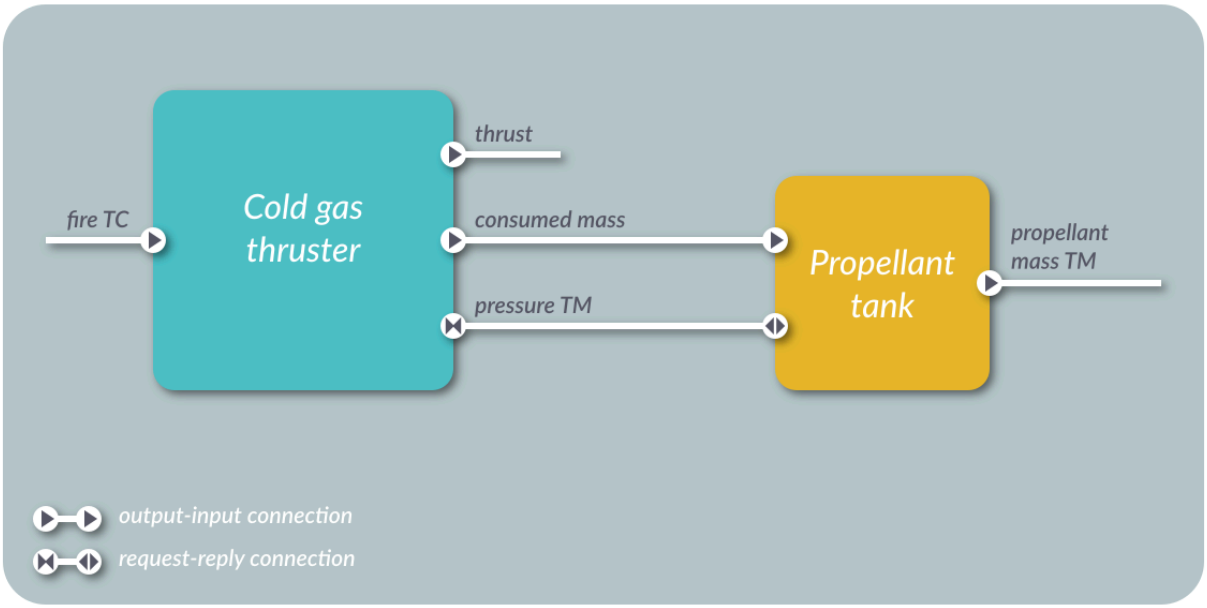

Figure 2: External interfaces and inter-model connections of a simplified cold gas propulsion system.

In the simulator model API, input ports are simply asynchronous object methods that take a single (optional) argument corresponding to the input type and return nothing. Replier ports are in turn asynchronous object methods that take a single (optional) argument and return a value. With these premises, the propellant tank model can be implemented in Rust as follows:

```rust
/// A pressurised propellant tank.
pub struct PropellantTank {
    /// Propellant mass telemetry -- output port [kg].
    pub propellant_mass: Output<f64>,

    /// Remaining propellant mass -- internal state [kg].
    mass: f64,
}

impl PropellantTank {
    /// A tank- and propellant-specific constant relating propellant mass to
    /// tank pressure (the influence of temperature is neglected) [Pa/kg].
    const TANK_CONSTANT: f64 = 1e5;

    /// Creates a tank with the specified propellant mass.
    pub fn new(mass: f64) -> Self {
        assert!(mass > 0.0);

        Self {
            propellant_mass: Output::new(),
            mass,
        }
    }

    /// Mass consumed by a connected model [kg] -- input port.
    pub async fn consumed_mass(&mut self, mass: f64) {
```

```
        self.mass -= mass;
        assert!(self.mass >= 0.0);

        // Propagate the change in mass to any model that may be connected.
        self.propellant_mass.send(self.mass).await;
    }

    /// Tank pressure telemetry [][Pa] -- requestor port.
    ///
    /// (The `async` signature is omitted here since there is no `await`)
    pub fn pressure(&mut self) -> f64 {
        self.mass * Self::TANK_CONSTANT
    }
}

impl Model for PropellantTank {
    /// Broadcasts the propellant pressure to any model connected to the
    /// relevant output when the simulation starts.
    async fn init(mut self, _: &Scheduler<Self>) -> InitializedModel<Self> {
        self.propellant_mass.send(self.mass).await;

        self.into()
    }
}
```

3.3 Thruster model

The implementation of the cold gas thruster model demonstrates a few other features, such as `Requestor` ports and scheduling. Any input port (that is, a model method taking an input value as argument) and any replier port (a method that also returns a value) can take an additional `Scheduler` argument that allows it to schedule one of the model's methods at an arbitrary future time. In the case of the thruster model, this capability is used to schedule the closing of the thruster valve.

```
/// A simple cold gas thruster.
pub struct ColdGasThruster {
    /// Instantaneous thrust -- output port [N].
    pub thrust: Output<f64>,
    /// Propellant mass consumed by a single pulse -- output port [kg].
    pub consumed_mass: Output<f64>,
    /// Pressure telemetry from the tank -- requestor port [][Pa].
    pub pressure: Requestor<(), f64>,

    /// State of the valve -- internal state.
    valve_is_open: bool,
}

impl ColdGasThruster {
    /// A constant relating tank pressure to thrust [m^2].
    const THRUST_CONSTANT: f64 = 1.2345e-6;
    /// Exhaust velocity [m/s].
    const V_SP: f64 = 300.0;

    /// Creates a cold gas thruster with the valve initially closed.
    pub fn new() -> Self {
        Self {
            thrust: Output::new(),
```

```
            consumed_mass: Output::new(),
            pressure: Requestor::new(),
            valve_is_open: false,
        }
    }

    /// Generates a thrust pulse based on the requested impulse bit [N.s] -- input port.
    ///
    /// This port receives the impulse bit to be generated and opens the valve
    /// for the appropriate duration based on the current tank pressure.
    pub async fn fire_cmd(&mut self, impulse_bit: f64, scheduler: &Scheduler<Self>) {
        // Ignore commands sent while the valve is already open.
        if self.valve_is_open {
            return;
        }
        self.valve_is_open = true;

        // Retrieve the tank pressure through an asynchronous request. In theory
        // an arbitrary number of models might be connected to the pressure TM
        // port so the request returns an iterator rather than a single value.
        let pressure_tm: Vec<_> = self.pressure.send(()).await.collect();
        let p = match &pressure_tm[..] {
            [p] => p,
            [] => panic!("the thruster must be connected to a tank"),
            _ => panic!("the thruster cannot be connected to several tank"),
        };

        // Compute the pulse parameters.
        let thrust = p * Self::THRUST_CONSTANT;
        let pulse_duration = Duration::from_secs_f64(impulse_bit / thrust);

        // Propagate the change of thrust to any model that may be connected.
        self.thrust.send(thrust).await;

        // Schedule the valve-closing method, passing the expected propellant
        // mass bit at pulse completion.
        let expected_mass_bit = impulse_bit / Self::V_SP;
        scheduler
            .schedule_event(pulse_duration, Self::close_valve, expected_mass_bit)
            .unwrap();
    }

    /// Closes the valve -- private input port [kg].
    ///
    /// The argument is the mass of propellant consumed by the pulse.
    async fn close_valve(&mut self, mass_bit: f64) {
        self.valve_is_open = true;

        // Update the thrust.
        self.thrust.send(0.0).await;

        // Update the propellant mass in the tank.
        self.consumed_mass.send(mass_bit).await;
    }
}

impl Model for ColdGasThruster {
    // Use the default implementation for `init()`, which does nothing.
}
```

4 CONCLUSION AND OUTLOOK

The rapid development of the New Space sector and the transition of mission-critical industries from C++ to Rust have created an opportunity to reassess not only the technical design of current system simulators, but also their ability to address the needs of the growing number of small system integrators.

Although it is at the moment much less featureful than mature SMP simulators, the new simulator already demonstrates very significant gains in terms of model development API ergonomy and lays the technical foundations for fully transparent parallel execution.

Its open-source licensing is also expected to make the simulator a de-facto standard for missions led by small institutions and universities. It has been in particular selected as the baseline system simulator for the upcoming 50kg ROMEO mission led by the Institute of Space Systems (IRS) at the University of Stuttgart.

The open-source licence is also expected to foster the development of an ecosystem of digital twins that would allow avionics manufacturers to provide their customer with a reliable, unambiguous digital specification of their device. The simulator is already being used for this purpose within the frame of project INVICTUS funded by the European Union, which foresees the development of a system digital twin for a high power Hall thruster propulsion system currently under qualification.

5 REFERENCES

[1] Blommestijn R., *Rationalisation of Simulators in Europe*, Workshop on Simulation and EGSE for Space Programmes, Noordwijk, 2017.

[2] *Simulation modelling platform*, ECSS-E-ST-40-07C standard, 2020.

[3] Jackli S. A., *Small-Satellite Mission Failure Rates*, NASA/TM–2018–220034, 2018.

[4] Venturini C., Braun B., Hinkley D., *Improving Mission Success of CubeSats*, 32nd Annual AIAA/USU Conference on Small Satellites, Utah, USA, 2018.

[5] *BASILES Numerical Simulation Framework*, http://basiles.fr/

[6] Barral S., Chikha A., *Asynchronix: an auto-parallelizing, high-performance Rust framework for system simulation*, Workshop on Simulation and EGSE for Space Programmes, Noordwijk, 2023.

[7] *Asynchronix—High-performance asynchronous computation framework for system simulation*, https://github.com/asynchronics/asynchronix

[8] *Tokio—an asynchronous Rust runtime*, https://tokio.rs/