Asynchronix: an auto-parallelizing, high-performance Rust framework for system simulation

Workshop on Simulation and EGSE for Space Programmes (SESP)
28-30 March 2023

ESA-ESTEC, Noordwijk, The Netherlands

Serge Barral[1], Adam Chikha[2]

[1]Asynchronics
Wilcza Street 72/14, 00-670 Warsaw, Poland
Email: serge.barral@asynchronics.com
[2]Asynchronics
Wilcza Street 72/14, 00-670 Warsaw, Poland
Email: adam.chikha@asynchronics.com

INTRODUCTION

Having long relied on complex networks of digital sub-systems, spacecraft can be rightfully considered forerunners of modern cyber-physical systems. It is therefore only natural that the space industry was one of the first to invest in simulation tools that could efficiently model not only physical, but also digital devices. Today, numerical simulation has become an integral and indispensable part of spacecraft Functional Verification (FV) activities.

There is an increasing awareness, however, about the need to modernize the technological stack on which these tools were initially built [1]. At the same time and despite successful standardization efforts such as the ESA-led SMP [2], spacecraft numerical simulation continues to largely rely on proprietary tools developed in-house by Large Space Integrators and national agencies, leaving few options to smaller integrators and to the New Space sector.

These observations were the main impetus for the development Asynchronix, an innovative open-source discrete event simulator for cyber-physical systems. While it does heavily draw from lessons learned with simulators used in Software Validation Facility or Assembly, Integration and Verification, this new tool builds upon a very different technological foundation, starting with the use of Rust in place of C++ as its implementation language.

As its name hints, Asynchronix embraces asynchronous programming, a form of cooperative multi-tasking with very low overhead which has in recent years led to unprecedented advances in the domain of highly concurrent network services. Combined with the unique memory safety warranties of Rust and a paradigm known as the Actor Model, this approach promises to finally achieve a long-sought goal of FV simulators: the fully automatized, safe parallelization of simulation execution. Remarkably, and unlike other approaches [3], this parallelization can be achieved in a manner that is entirely transparent to the user, without need for explicit synchronization and without risk of data races.

This contribution provides an overview of the general design and implementation of Asynchronix followed by an evaluation of its key performance metrics, concluding with a discussion on further development perspectives.

SIMULATOR OVERVIEW

Background and heritage

System-level simulators in the European space industry have historically shared a large set of concepts and features, a de-facto convergence that has ultimately given shape to what is today the ECSS SMP standard [2]. From a high vantage point and simplifying somewhat, it can be said that simulators such as SimSat, SimTG, BASILES, K2/Gram, EuroSim are all Discrete Events Simulators (DES) and promote a component-oriented architecture that closely resembles flow-based programming: a model is essentially an isolated entity with a fixed set of typed inputs and outputs, communicating with other models via connections defined during bench assembly.

This combination of DES with component-oriented architecture has proven hugely successful for spacecraft simulation. It arguably maps very well to cyber-physical systems other than spacecraft, too: a CPS is essentially a fixed network of subsystems (the components), and most subsystems are themselves state machines for which a discrete-event approach is better suited than a continuous-time approach. Asynchronix is, for this reason, designed around very similar concepts.

Aiming for early SMP compatibility was deemed too constraining, however, not least because SMP assumes a C++ implementation and does not support asynchronous programming (which does exists in C++ since C++20 in the form of coroutines). While Rust provides seamless ABI compatibility with the C language, interfacing with C++ is admittedly less straightforward. Nevertheless, partial support for SMP may be contemplated at some point in the future.

Inter-model communication

An explicit goal of Asynchronix was to reduce the set of concepts pertaining to communication between models. Whereas SMP distinguishes between *interfaces*, *dataflow communication* and *events*, Asynchronix only exposes two closely related concepts:

- *events*, which are payloads sent between *output* and *input* ports and are conceptually similar to their counterpart in SMP but with the ability to transport a payload with an arbitrary, statically known type,
- *request/replies*, which are exchanged between *requestor* and *replier* ports and constitute an extension of *events* that allows a payload to be returned asynchronously from the replier to the requestor port.

Output and requestor ports are respectively instances of the `Input<T>` and `Requestor<T, R>` types, which are parametrized by the payload type `T` and the reply type `R`. Input and replier ports are in turn asynchronous methods that take an argument of type `T` and, in the case of replier ports, return a value of type `R`.

This departure from SMP hinges on several observations:

1. In many practical situations, inter-model communication can be alternatively implemented with interfaces or with a combination of dataflow fields and events [4], making these concepts largely redundant. Because they can carry a payload, Asynchronix events are simpler than the combination of dataflow fields and events, while introducing less coupling between models than interfaces.
2. Practice has shown that dataflow behavior is confusing and responsible for a disproportionately large amount of software defects. Most such issues relate to the fact that SMP simulators can trigger an action on a model which input has been updated (though ECSS SMP does not specify how). Because dataflow communication assumes idempotence, however, implementations do not necessarily trigger an action if an input was updated with a value equal to the previous value. This behavior, while logical in the context of dataflow communication, is not only inconvenient when implementing state machines (transition conditions rarely being idempotent) but is also frequently overlooked even by experienced implementers.
3. In many instances, the impossibility to obtain a return value asynchronously pushes complexity to the assembly step or requires special support from software development tools. One such example is given in [4] where an ISIS Power Line (a line that responds to a voltage with a current) must be split into 2 interfaces of opposite directionality, or into several data and event lines. The request/reply pair, which exploits Rust's asynchronous programming facilities, makes it possible to implement such pattern in a clean manner with a single, bi-directional connection.

A practical comparison between Asynchronix and SMP for the ISIS-type system interfaces [4] is provided in Fig. 1 and Fig. 2, illustrating the considerable conceptual simplification enabled by the messaging concepts introduced in Asynchronix.

Inter-model connection rules are very flexible: as long as the event payload is of the same type, output and input ports can be connected in fan-in (many-to-one) and fan-out (broadcast) patterns, or a mixture thereof (many-to-many). The same goes for requestor and replier ports with compatible request and reply payloads. Note that a request sent on a requestor port returns an iterator over a collection containing 0 or more replies, depending on the number of connected replier ports.

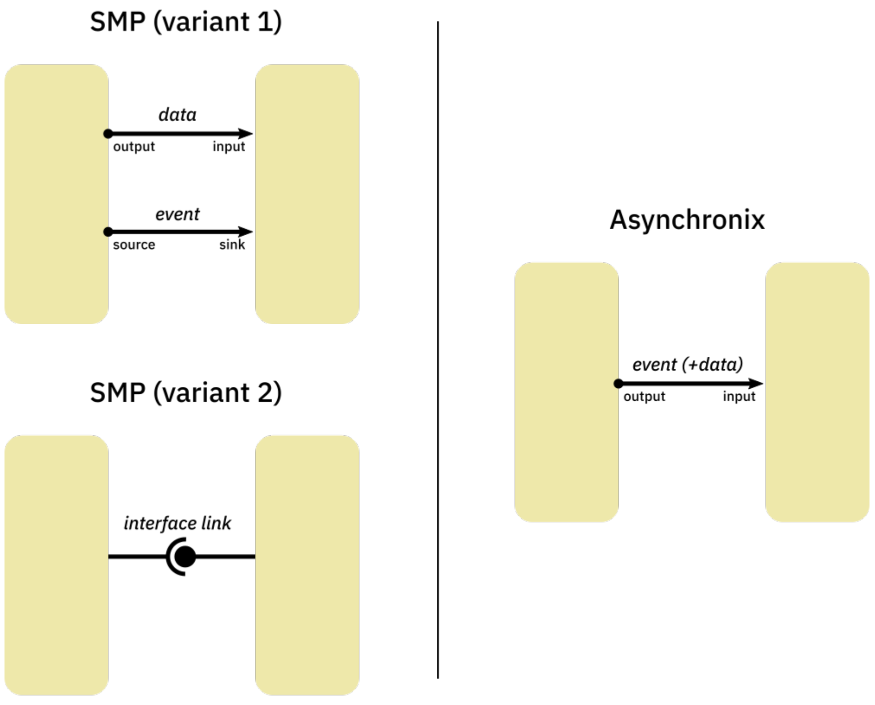

Fig. 1. Representation of a uni-directional ISIS system interface in SMP [4] and Asynchronix.

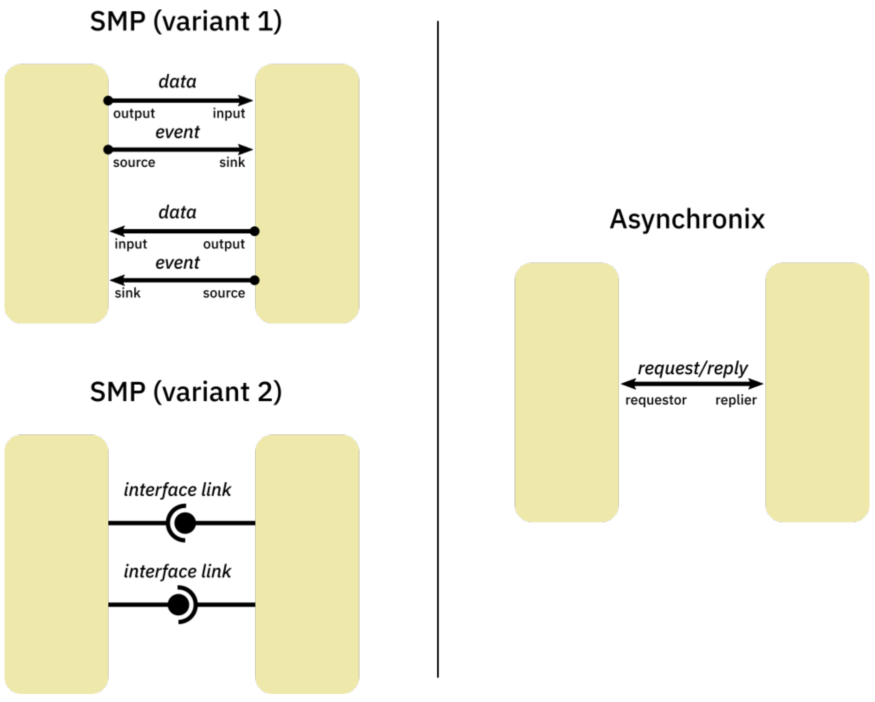

Fig. 2. Representation of a bi-directional ISIS system interface in SMP [4] and Asynchronix.

Model handles

Another important difference with SMP simulators is that connecting model outputs and inputs does not involve pointers, as this could lead to use-after-free once a model is destructed and would thereby fail to uphold the fundamental guarantees provided by Rust.

Instead, and in keeping with the traditional nomenclature of the actor model [5], a model instance is associated to a unique *mailbox* and by possibly many *addresses,* where the latter can be conceptually seen as references to a mailbox.

Under the hood, a mailbox is the consumer handle of a thread-safe multi-producer queue associated to a specific model instance. Addresses are in turn queue producer handles. The use of addresses is always safe, in the sense that sending a payload to a defunct mailbox never leads to undefined behavior (in the current implementation, the payload is simply ignored).

Message delivery ordering

SMP is mostly based on asynchronous messaging and gives very few guarantees regarding the relative ordering of messages. Dataflow propagation order is not specified, nor is the order of associated actions. SMP does specify that event sinks be notified in the order in which they have been added to a source, but relying on this arguably makes it a leaky abstraction since proper behavior then ultimately depends on a subtle detail of the bench assembly procedure.

Even though it pushes asynchronicity much further by allowing parallel execution, Asynchronix paradoxically offers stricter ordering guarantees. Borrowing from the Pony programming language [6], we refer to this contract as *causal messaging*, a property that can be summarized by these two rules:

1. *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`,
2. *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`.

However, this must be nuanced by the fact that despite weaker ordering guarantees, a (single-threaded) SMP simulator can typically ensure deterministic execution whereas Asynchronix, operating in multi-threaded mode cannot. Determinism is theoretically irrelevant for a properly designed system simulation (*i.e.* one that only relies on message ordering guarantees), but it can be useful in debugging a misbehaving simulation bench, which is why Asynchronix also offers a single-threaded execution mode.

SIMULATOR DESIGN

Architecture

Under the hood, Asynchronix is based on the so-called actor model [5]: each simulation model is an actor, *i.e.* an isolated entity that communicates with other models solely via message-passing. Importantly, a simulation model is associated to a unique mailbox that collects all messages (events and requests). This mailbox is the consumer side of a thread-safe, multi-producer, single-consumer queue. This is a key property of the actor model which ensures that an actor (simulation model) only ever processes one message at a time, thus preventing data races even when multiple messages are being sent concurrently.

Simplifying somewhat, messages exchanged between models are closures, *i.e.* method pointers indicating the targeted model method (in other words: the targeted input or replier port), bundled with the payload sent by an output or requestor port.

It should be noted that input and replier ports are asynchronous methods, meaning that they do not return an immediately usable value but a *future*, where a future is essentially a compiler-generated object representing a state-machine that returns a value in its final state. There are two main reasons for using asynchronous methods. Firstly, this makes it possible for model methods to suspend execution when attempting to send an event to a model which mailbox is full. Secondly, it makes it possible to suspend execution when sending a request until a reply is available, which is necessary since models targeted by a request cannot be expected to immediately process such request and send a reply.

Unlike say Go or C#, Rust does not include an executor that can schedule the futures generated by asynchronous methods. Similarly to C++, Rust expects executors to be user-provided, which has the significant advantage of allowing domain-specific optimizations and behavior. In the case of a multi-threaded DES, the executor should optimize for performance in a context where messages are frequently exchanged between models that can run on different threads. Since this is similar to typical network server workloads, the native Go runtime and the Rust Tokio runtime [7] were considered the most relevant references, leading to the selection of an approach based on work-stealing multiplexed over a thread pool.

Key implementation choices

Since developing a custom asynchronous executor is a major endeavor, the use of Tokio was investigated first. It was eventually deemed unworkable, however, due to the fact that a DES considers that computations for a given time step have completed once all models are blocked on an empty mailbox, which is technically a deadlock. Tokio does expose some deadlock-detection features for forensic purposes, but a dedicated executor needs to consider such deadlocks as a normal occurrence.

Adapting Tokio was rejected due to the monolithic character and complexity of the Tokio codebase, which would have made adaptation and further development very challenging. Since many features of Tokio were not needed, the development of a prototype based on the more lightweight Smol executor [8] was undertaken instead. Performance was found to be significantly worse than Tokio, however, mainly due to differences in the work-stealing strategy.

Ultimately, the decision was made to develop a fully custom executor with a work-stealing strategy similar to that of Tokio and the Go scheduler. Despite the consequent development effort, this decision paid off and allowed novel optimizations to be introduced that resulted in even better performance than Tokio (widely considered state of the art) on workloads dominated by message-passing. A library with thread-safe, bounded FIFO and LIFO work-stealing queue was also spun off from this effort [9].

Another critical component of the simulator kernel is the thread-safe channel used for message-passing. A channel is a combination of a thread-safe FIFO queue with primitives that handle task suspension and notifications in situations where a producer is blocked on a full queue, or the consumer is blocked on an empty queue. In a classical channel, these primitives rely on operating system facilities such as condition variables. In an asynchronous channel, however, the goal is to avoid operating system calls whenever possible and let instead the asynchronous executor suspend and resume tasks using lightweight cooperative multitasking.

Asynchronous channels are still an area of relative immaturity, and it was felt that existing Rust channel libraries had various performance shortcomings for the case at hand, either because of their broader scope (*e.g.* support for multiple consumers) or because they optimized for different priorities (*e.g.* fairness among blocked producers). This prompted the development of a custom channel based on a high-performance bounded queue devised by D. Vyukov [10]. This queue was specialized for the single-consumer case, further decreasing the synchronization cost of dequeue operations. Custom asynchronous signaling primitives were developed, one of which was published as an independent library [11] after it demonstrated exceptional performance in benchmarks with starved consumers (*i.e.* consumer frequently blocked on an empty queue, which is a very frequent occurrence in DES applications). A general-purpose version of the channel was also released [12], which was shown to perform better than existing Rust channels in nearly all multiple-producer, single-consumer benchmarks.

Development and testing policy

Asynchronix and its dependencies use so-called *unsafe* Rust, an opt-in superset of the Rust language which makes it possible to partially escape compiler analysis and allows program authors to introduce constructs they believe to be safe (free of undefined behavior) but which the compiler cannot prove as such.

Unsafe Rust is the only potential way to introduce undefined behavior in Rust and is therefore rightfully considered a very sharp tool. Nevertheless, developing a reasonably fast asynchronous runtime is widely acknowledged to be impossible without opting into unsafe. As for asynchronous channels, the *Flume* library is probably the only one that offers an implementation based on the safe Rust subset, but despite its claim it was shown to perform notably worse than Tachyonix [12], particularly in consumer-starved benchmarks.

Even though simulators are not considered mission-critical software, software quality is a very high priority for Asynchronix and much effort has been accordingly devoted to minimize the risk of software defects, in particular in relation to unsafe Rust. Apart from regular unit and integration tests, Asynchronix heavily relies on the MIRI interpreter [13] and on the Loom concurrency testing library [14] to assess proper behavior of all custom concurrency constructs.

License and availability

The Asynchronix framework is released under the MIT and APACHE license (v2) open source licenses [15].

PERFORMANCE CHARACTERIZATION

Methodology

A direct comparison against existing SMP simulators on a representative spacecraft model would obviously be most desirable, but its high cost makes such effort unlikely to be carried out in the near future. We are, in fact, not aware of the existence of any such benchmark between current SMP simulators.

However, reckoning that the performance of Asynchronix largely derives from that of its asynchronous executor and asynchronous channels, benchmarking was performed via a comparison against the best-known and best-performing Rust executors and channels.

This report focuses on the so-called “pinball” benchmark, which is believed to be most representative of typical conditions in which a DES operates. Other benchmarks [16] not reported here also characterize performance when the simulator is highly saturated with messages (mailboxes frequently full) and show that Asynchronix performs similarly well in such degraded conditions.

The pinball benchmark is a generalization of the classical ping-pong benchmark where messages are sent back-and-forth between two channels. Its main goal is to measure the combined executor + channel performance in situations where many consumers are often starved but producers are never blocked. This depicts the common situation where models are most of the time waiting on an empty mailbox until an event or request is received.

Each test rig consists of a fully connected graph which edges are channels transporting messages (“balls”) between nodes (“pins”). The nodes, which can be thought of as the simulation models, forward any message they receive to another randomly chosen node. The results reported here correspond to graphs containing 13 nodes, each node containing in turn 1 consumer and 12 producers (1 for each node connected to it). Importantly, each channel has enough capacity to never block on sending. The benchmark concurrently runs 61 such rigs of 13 nodes.

The test is performed for various numbers of messages, which are initially fairly distributed across the graph. The messages then perform a random walk between the nodes until they have visited a predefined amount of nodes.

The source code and additional details concerning this benchmark are available in a public repository [16].

Executor performance

The Asynchronix executor can be used as a standalone generic executor through the activation of a compile-time option, which makes it possible to compare it directly to other executors.

A comparison of Asynchronix with other executors on the pinball benchmark is shown in Fig. 3. All tests are performed with the Tachyonix channel [12], which is based on the internal Asynchronix channel implementation.

Asynchronix comes on top of all benchmarks on both Intel and ARM platform, even slightly ahead of Tokio which is considered state-of-the-art and powers all the best-performing Rust networking libraries. This superiority was confirmed as well in other benchmarks and other processors [16].

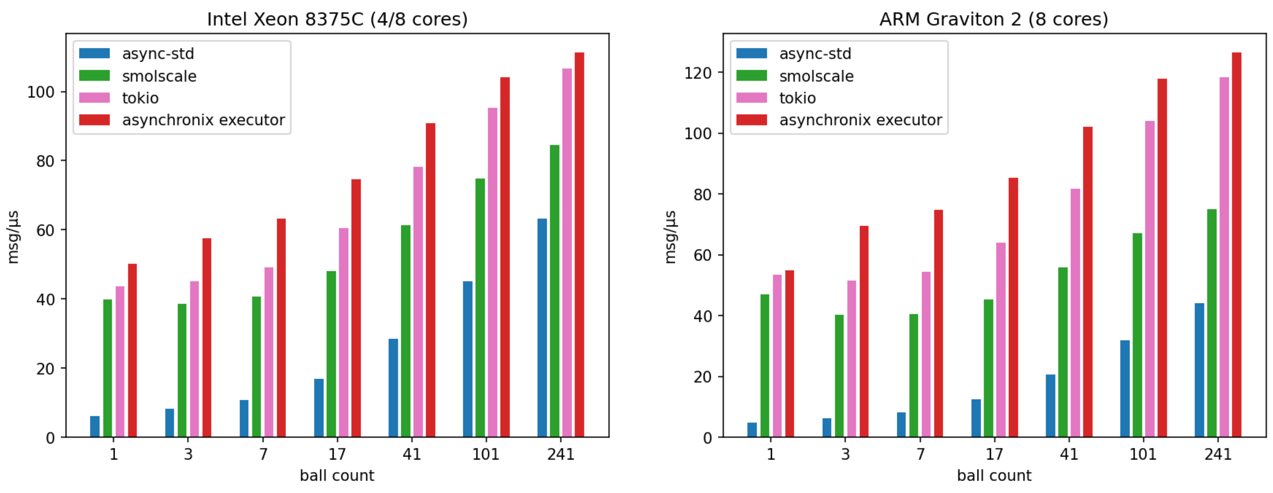

Fig. 3. Multi-core throughput (higher is better) of the Asynchronix executor (red) against other Rust executors for the pinball benchmark. Left: Intel processor, 4 physical/8logical cores. Right: ARM processor, 8 cores.

Channels performance

Although the Asynchronix channels cannot be directly compared to other general-purpose channels due to a number of idiosyncrasies in their implementation, the spin-off Tachyonix library [12] offers general-purpose channels that use the exact same underlying queue and notification primitives.

A comparison of Tachyonix with other Rust channels on the pinball benchmark is shown in Fig. 4. All tests are performed with Tokio, the best-known and—with the exclusion of Asynchronix—best-performing Rust executor for message-passing applications.

Tachyonix leads these benchmarks by a very clear margin for all parameter, on both Intel and ARM platforms. Similar levels of performance are as well observed in other benchmarks and other processors [16].

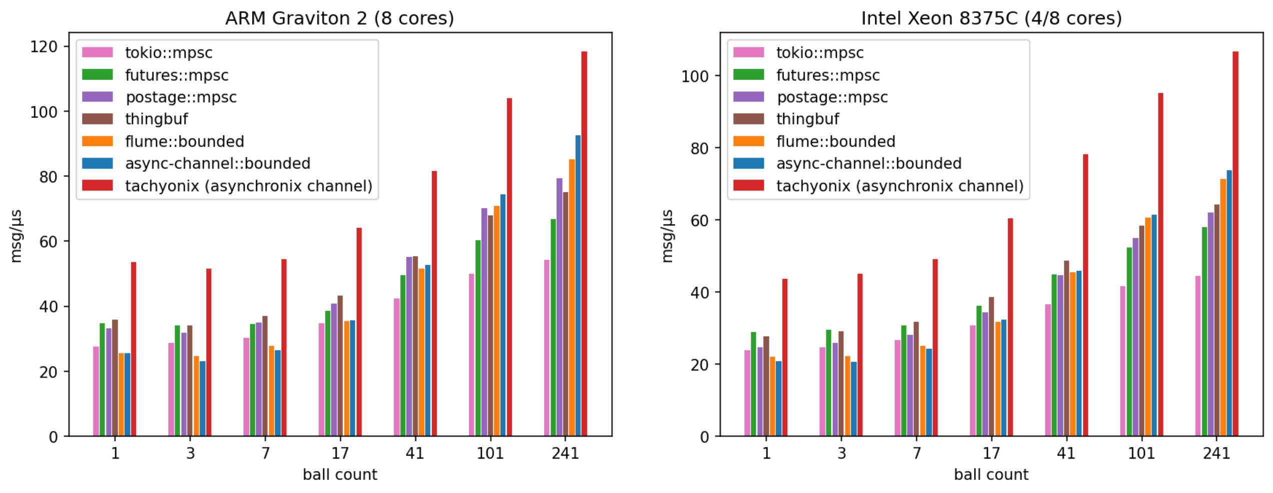

Fig. 4. Multi-core throughput (higher is better) of the Asynchronix channel implementation (red) against other Rust channels for the pinball benchmark. Left: Intel processor, 4 physical/8logical cores. Right: ARM processor, 8 cores.

CONCLUSION AND PERSPECTIVES

Although it shares many of the same underpinning concepts as SMP simulators such as SimSat, SimTG, BASILES, K2/Gram, or EuroSim, Asynchronix constitutes a very radical revision of the design of space industry simulators. These differences pertain to a number aspects, including programming language, inter-model messaging concepts and multi-threading support.

Asynchronix also pushes the state of the art in the domain of Rust asynchronous programming, with respect to both asynchronous executor technology and thread-safe channel technology.

It is necessary to underline, however, that Asynchronix is currently considered to be at a Minimum Viable Product stage. It still lacks many features that would be considered essential for commercial missions, such as support for serialization, remote network operation, SRDB etc. Nevertheless, the quality of its implementation and performance are believed to be already fully appropriate for initial demonstration on nano/microsat-class missions, or for the development of cyber-physical systems of intermediate complexity.

REFERENCES

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

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

[3] G. Bouteille, B. Rocca, “New simulation framework, GRAM”, Workshop on Simulation and EGSE for Space Programmes, March 2021.

[4] H.T. Pham, R Atori, F. Quartier, S. Salas Solano, A. Strzepek, N. Rousse, W. Arrouy, J.M. Fournié, S. Montigaud, J.P. Gest, G. Bouteille, J. Whitty, “Models Exchange through ISIS and SMP2: From Prototype to Reality”, Workshop on Simulation and EGSE for Space Programmes, March 2017.

[5] “Actor Model”, https://en.wikipedia.org/wiki/Actor_model.

[6] “The Pony Programming Language”, https://www.ponylang.io.

[7] C. Lerche *et al.*, “Tokio—An Asynchronous Runtime for the Rust Programming Language”, https://tokio.rs.

[8] S. Glavina *et al.*, “Smol—A Small and Fast Async Runtime”, https://github.com/smol-rs/smol.

[9] S. Barral, “St^3—The Stealing Static Stack”, https://github.com/asynchronics/st3.

[10] D. Vyukov, “Bounded MPMC queue”, https://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue.

[11] S. Barral, “Diatomic Waker—Async, Lock-Free Synchronization Primitives for Task Wakeup”, https://github.com/asynchronics/diatomic-waker.

[12] S. Barral, “Tachyonix—An Asynchronous, MPSC bounded channel that operates at tachyonic speeds”, https://github.com/asynchronics/tachyonix.

[13] R. Jung *et al.*, “MIRI—An Interpreter for Rust's Mid-Level Intermediate Representation”, https://github.com/rust-lang/miri.

[14] C. Lerche *et al.*, “Loom—Concurrency Permutation Testing Tool for Rust”, https://github.com/tokio-rs/loom.

[15] S. Barral, “Asynchronix—High-Performance Asynchronous Computation Framework for System Simulation”, https://github.com/asynchronics/asynchronix.

[16] S. Barral, “Tachyobench—Async Benchmark for MPSC Channels with Support for Multiple Executors”, https://github.com/asynchronics/tachyobench.