Designing a Simulation Environment fit for the next ten years

Alberto Ingenito[(1)], Peter Fritzen[(1)], Paul Steele[(2)], James Eggleston[(2)]

[(1)] Telespazio VEGA Deutschland GmbH
Europaplatz 5
D-64293 Darmstadt
Email: Alberto.Ingenito@telespazio-vega.de, Peter.Fritzen@telespazio-vega.de

[(2)] European Space Operations Centre
Robert-Bosch Strasse 5
D-64293 Darmstadt
Email: Paul.Steele@esa.int, James.Eggleston@esa.int

A CHANCE TO RETHINK SIMSAT

For over 20 years, SIMSAT has been the simulation engine used at ESOC for its operational simulators. SIMSAT, in its current form, has been designed more than ten years ago. While many of the core concepts are still valid, technologies and standards have changed significantly since then.

Obsolescence of technologies

Many of the key underlying technologies within SIMSAT are becoming obsolete. The most significant example in this respect is CORBA. CORBA has stagnated for several years, and has reached the point where Oracle has first deprecated and then removed it from the Java SE Platform [1]. In the current design of SIMSAT, removing CORBA would be a complex operation that would affect almost all classes and source files. CORBA is not only used for inter-process communication, but also as the basis of the SIMSAT component model and as the interface between all parts of the simulation engine.

Abstraction from standards

SIMSAT, by design, is independent from a specific simulation standard. It uses internally its own CORBA based interfaces, and supports the Simulation Models Portability 1 (**SMI**) and 2 (**SMP2**) through adapters. In February 2019, an experimental **ECSS-SMP** adapter was also released.
This powerful feature of SIMSAT is also a weakness when it comes to performances of the system, both in time and in memory.
The time overhead of SIMSAT comes from the multiple layers that have to be traversed to reach the necessary information. In the SIMPERF study [2], we measured the overhead involved in retrieving published field values and in executing scheduled operations; both areas were significantly affected by abstraction and CORBA usage.
The memory overhead of SIMSAT comes from memory duplication. Most of the information in the system is duplicated, one instantiation within the particular standards adapter, and one instantiation within the internal representation.

A new SIMSAT

To prepare for the next ten years, SIMSAT will evolve into a new product: SIMSAT Next Generation (SIMSAT-NG).
SIMSAT-NG has to rely on technologies that are still strong today. This implies that CORBA cannot remain a central technology in the new SIMSAT design.
The upcoming ECSS-SMP standard will become applicable to every future simulator; it was therefore decided for SIMSAT-NG to focus on the new standard. SIMSAT-NG will support natively ECSS-SMP, without the need of adapters. Support for SMP2 will be dropped; support for other standards will be provided through adapters if necessary, for example the Functional Mock-up Interface (**FMI**) used in the Automotive Industry. An FMI adapter for SIMSAT was prototyped by us in the SIMULUS Next Generation study [3].
No support for SMP2 is possible, not even through adapters. The ECSS-SMP working group made a conscious decision to reuse in ECSS-SMP the fully qualified names used in SMP2. This prevents the two standards from working together.
The implementation of SIMSAT-NG has been based on the public review draft of ECSS-SMP and, while not fully featured yet, can already be used to run ECSS-SMP simulations, providing support to assembly, scripted operations, and remote graphical interface.

DESIGN PRINCIPLES

The design principles of SIMSAT-NG have been derived from the findings of the SIMULUS Next Generation study [3]. The following are the key principles that have been applied since the start:

Lightweightness, the system has to be light enough to be an effective infrastructure, both in terms of production and testing. This is not only related to runtime performance, but also to execution setup and preparation.
Composability, the system must be easily composed and it must be possible to swap in and out all parts without affecting the neighbouring components.
Extensibility, the system must allow to add new features to the system without modifying the existing components.
Embeddability, the system must support to include SIMSAT-NG as a library in other software. This includes graphical user interfaces and design tools that want to support an internal runtime environment.
ECSS-SMP, the system must natively support ECSS-SMP, leaving the possibility of adapters to support other standards.

System architecture
As shown in Fig. 1 SIMSAT-NG design composes a simulation in three main layers.

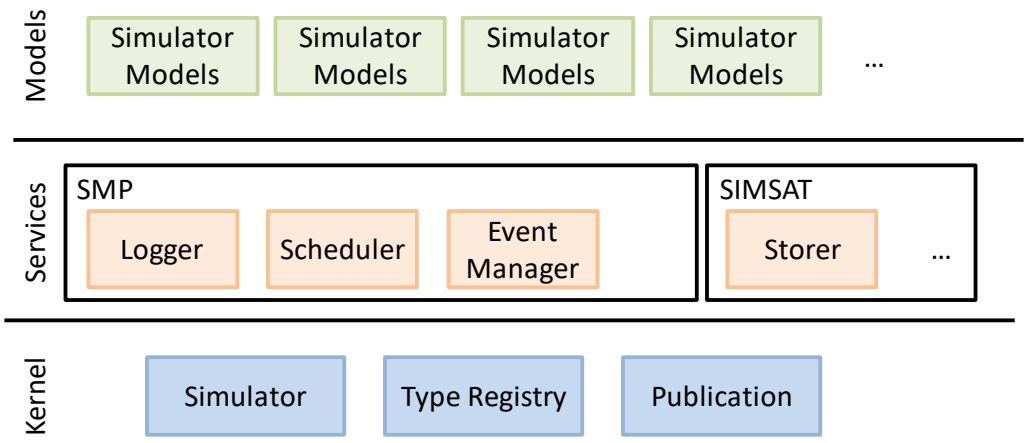

Fig. 1. High level SIMSAT-NG architecture

At the bottom layer is the simulation kernel. The kernel is a monolithic block that cannot be separated. It provides a minimal set of SMP features that co-operate too tightly to be independent.
The services layer provides the required ECSS SMP services, plus additional services that provide SIMSAT-NG specific features. Each service is independent, and can be replaced with a different implementation or even removed completely.
The simulation models are in the top layer. The models access the underlying layers through the SMP interface. When models need to access SIMSAT-NG specific features, they can use a clearly separated set of interfaces, clearly defining portable and non-portable implementation.

SMP as a component model
ECSS-SMP defines a component model, including a standardised interface to load components in a simulation both statically and dynamically. This allowed us to drop the old CORBA model, and to rely solely on ECSS-SMP mechanisms. SIMSAT-NG supports the loading of SMP packages, which can contain one, or more, of the following:

- A set of model factories, registered at package initialisation,
- A service, added to the simulator at package initialisation,
- Some C++ assembly code, instantiating and connecting simulation models at package initialisation.

This approach has a significant positive impact on the overall design of the new services, which are also ECSS-SMP components and in many cases can be loaded in any ECSS-SMP compliant environment.

Minimal Kernel
The “Kernel” of the simulation is, out of necessity, a monolithic block. For this reason, it has been kept as small as possible. It contains only the implementation of SMP ISimulator, ITypeRegistry, and the classes required to implement IPublication. Everything else is provided as a service.
This allows, when needed, to provide different implementations of services that fit with different scenarios or functionality. One example is the logger service, where in some cases a simple console logger is sufficient, and in some other cases a full database is necessary. Another example is state storage; different scenarios may require different storage formats.
This “composition” approach has the significant advantage of allowing simulation developers to tailor SIMSAT-NG to their specific needs, selecting the best fitting services set, or possibly creating their own services as SMP components.
It is also possible to reuse the services on a different simulation environment, not necessarily based on SIMSAT-NG.

SIMSAT-NG as test environment
As already mentioned above, the SIMSAT-NG kernel has been designed to be used as C++ library. This, coupled with the ability to select a subset of the available services, allows SIMSAT-NG to be used also as a unit and integration test environment. The tests may not need all services, and for those a *dummy* service version can be used.
The tests do not require daemons to be running or external processes to be started, and this enables a simple and effective integration with integrated development environments, e.g. allowing the tests to be executed with a debugger attached.

FEATURES NOT COVERED BY THE SMP STANDARD

The ECSS-SMP standard does not try to cover all possible features of simulations. Such an attempt would not only add complexity to the standard, but would also increase the cost of, and possibly limit, the adoption of the standard. Taking this into account, the SIMSAT-NG design since its inception had to provide support for *extensions*, i.e. features not defined in the ECSS-SMP standard but anyway provided by SIMSAT-NG.
In addition to this, at the time of writing this paper, the ECSS-SMP standard is still in a draft version. The SIMSAT-NG design and implementation was taken as an opportunity to test, and in some cases propose, comments to the standard.

Extensions as Interfaces
From experience, it is clear that SIMSAT-NG has to provide more features than the set covered by the ECSS-SMP standard. We decided to apply a fully interface based approach for the SIMSAT-NG kernel and services; enabling not only to extend services, but also to replace them entirely.
Fig. 2 shows the type of interactions that are allowed in the SIMSAT-NG design.

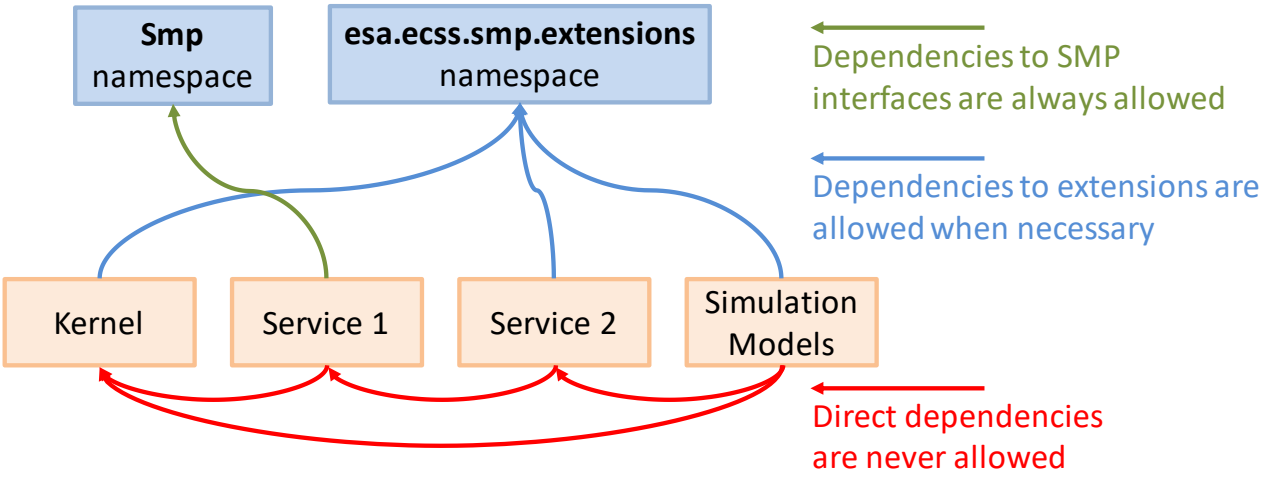

Fig. 2. Usage of SIMSAT-NG interfaces

Neither models nor services are allowed to directly interface with the implementation of the SIMSAT-NG kernel and services. All dependencies have to go through the interfaces define in either SMP or the extensions.
The usage of SMP interfaces has no restrictions; it is always valid to use SMP interfaces.
The usage of the “extensions” namespaces has some, limited, restrictions.

The Kernel delegates to services, but not to specific implementations. The SIMSAT-NG Kernel can delegate some functionalities to services. If the service is not available, the functionality is disabled. The Kernel cannot assume a specific implementation of the service, as it is must always be possible to replace services.

Models and Services can, but should limit the usage of extensions for Kernel elements. The absence of extensions in the Kernel elements should not fully compromise the functionality of components. In a pure ECSS-SMP environment, components should still load and operate, possibly disabling functionalities that require SIMSAT-NG specific interfaces. I.e. the ECSS-SMP standard offers the ability to retrieve published fields, the SIMSAT-NG ISimulator offers in addition the ability obtain published properties and operations. A service collecting data on components can always support fields, and properties only if the extended interface is available.

Models and Services can, but should limit the usage of extensions for Service elements. The same point as above, with an important distinction. SIMSAT-NG services are as much as possible portable. This implies that, in several cases, the dependency on extended services does not prejudice the portability of models.

Kernel and Services cannot assume that models implement the extensions. The extensions contain some interfaces that allow model packages to enable extra features. These interfaces cannot be taken for granted, as models origin cannot

always be controlled. Kernel and Services can provide additional features and optimisations when the models offers the extension interfaces, but have to support at least the SMP mandatory features for models that do not provide the extensions.

Extension can be shared between different simulation environments. This has already been validated by extending the old SIMSAT with some extensions interfaces to support the loading and execution of tests models. In the future, this could be used by the working group to test and agree on new functionalities before the next revision of the ECSS-SMP standard.

Storer as a Service

The SMP standard defines the *"state vector storage*" as part of the ISimulator interface. In SIMSAT-NG, the "*state vector storage*" is provided by a service. This decision was driven by multiple factors:

- The monolithic kernel has to contain only the features that cannot be separated, and the storage is not part of this set,
- The storage can be implemented in multiple ways with different benefits,
- The storage as service can be reused in any ECSS-SMP simulation environment, not only in SIMSAT-NG.

In the past SIMSAT offered a single "Storer" component, attempting to fulfil multiple roles and eventually providing the storage feature with limited performances.

In Fig. 3 is shown how, in SIMSAT-NG, the Kernel has an association to the interface IStorer. The first registered service to implement the IStorer interface is bound to the association, and is then used in all storage operations.

Each Storer service has only to implement a single storage mechanism (e.g. to database, to XML, to memory). It is also possible to store a state vector in SIMSAT-NG, and restore it in a different simulation environment by loading the SIMSAT-NG Storer as a normal SMP service.

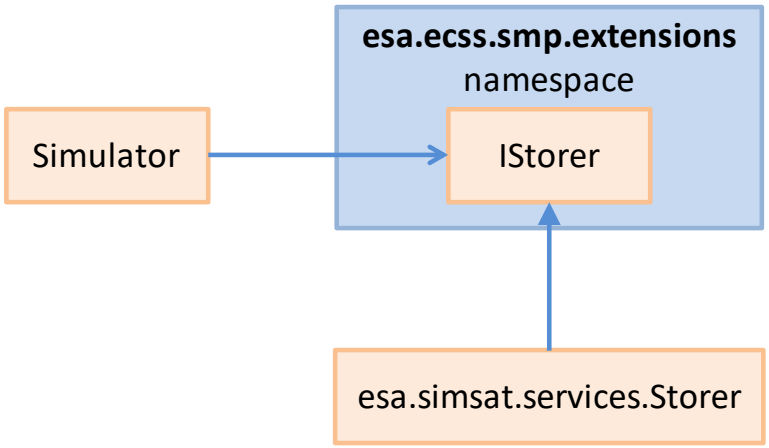

Fig. 3. Kernel interface to the Storer

Java Support

For a number of features in SIMSAT-NG, the integration of the Java Virtual Machine (**JVM**) provides significant benefits, in particular the scripted control of the simulator, but also the implementation of some services and the integration of existing Java libraries.

Through services, SIMSAT-NG integrates with the Java Virtual Machine. The "Jvm" service can provide other systems with an in-process instance of the JVM. The access to the JVM is via the Java Native Interface (**JNI**).

SIMSAT-NG also provides a full mapping of the simulation tree. Starting from the Simulator instance, using a reflection interface, Java models can navigate the tree, manipulate fields and properties, and invoke published operations. The mapping is already implemented and tested.

At the time of writing this paper, the ability to publish Java classes as SMP2 components has also been prototyped. The prototype offers the ability to create Java components that can be added to the simulation tree by means of a C++ proxy, the proxy supports the publication of properties and operations. Planned features are: publication of fields, supporting via code generation the implementation of C++ interface by Java models, and better declaration of Java components.

Remote connections

One of the historical requirements of SIMSAT is the ability to control the simulation remotely, either via GUI or from external applications. In the past, CORBA provided the infrastructure for remoting.

For SIMSAT-NG, we wanted not only to limit the usage of aging technologies, but also to provide better encapsulation and to support more use cases.

The remote control of SIMSAT-NG is now provided by a single service. This allows the easy replacement of the remote control system, or even removing it completely, e.g. in unit test scenarios.

The remote connector is implemented in Java, and relies on the Java mapping to access the simulation. The communication is based on message passing (Protocol Buffer), over a simple TCP connection. This simple, yet powerful, communication approach is borrowed from modern network and web application. This approach will enable us in the future to setup complex simulation scenarios, e.g. running simulations in a container cloud.
In Fig. 4 is shown how the chain of communication is all based on defined interfaces. The remote connector service is the gateway for the MMI. The remote connection uses the IJvm interface to access the JVM and the Java mapping to SMP. The Java mapping uses the IReflection interface to access operations and properties stored in the simulation publication tree.

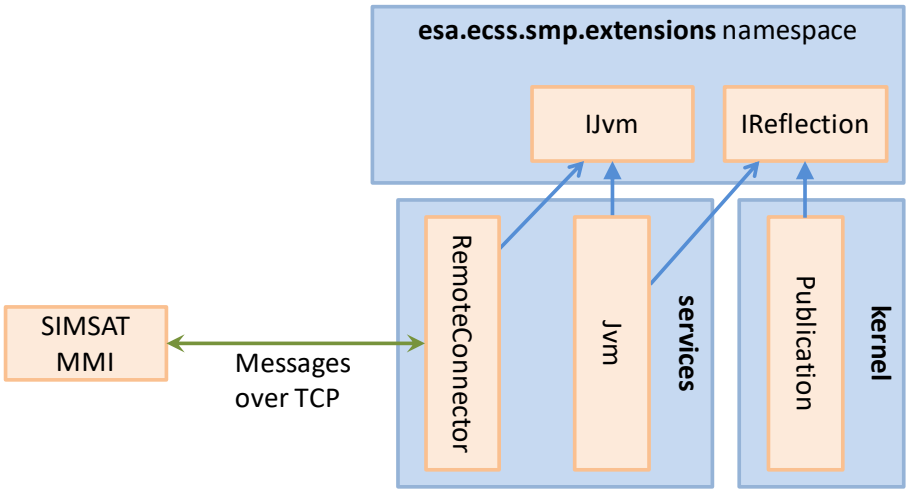

Fig. 4. Kernel interface to the Storer

Scripting
Another historical requirement of SIMSAT is to control the simulation via scripting, both remotely (GUI) and directly (system testing).
For SIMSAT-NG scripting has been implemented on top of the Java mapping, using the Java Scripting interfaces, and adding late binding to access intuitively fields and operations.
The scripting service allows creating several independent script engines, loading any script engine available at that moment to Java (e.g. JavaScript, Groovy, Ruby…). Late binding has been implemented for Groovy, and is planned for JavaScript.
The implemented scripting layer is able to control all aspects of the simulation, from loading SMP packages, to the termination of the simulation.
Unlike the previous SIMSAT implementation, the scripting engine is able to run synchronously with the simulation, allowing for very accurate timing of executions in testing scenarios.
We have also prototyped a scripted SMP component. This feature, based on the Java SMP components, will allow the creation of models with dynamic behaviors fully integrated with the classical C++ components.
A good example of use case for this dynamic behavior is the Generic Payload Model (**GEPLOAD**) provided by the SIMULUS Generic Models (**GENM**). GEPLOAD already provides the ability to configure the model behavior trough XML. Each configurable behavior was designed and implemented in C++. New behaviors require changing both the C++ implementation and the XML schema of the GEPLOAD configuration.
The ability to add scripted behaviors to GEPLOAD would allow defining new behaviors and interactions without having to extend GEPLOAD.

Assembly
In addition to the standardised C++ assembly mechanisms, we wanted to support a more dynamic assembly mechanism. The features we wanted are:

- Ability to change to simulation without recompilation of binary artefacts,
- Ability to load and configure services,
- Ability to compose, connect and configure models,
- Ability to define sub-assemblies that can be reused as is, without requiring modifications.

Scripting proved immediately able to provide all the desired features.
To load services, it is enough to load the relative package. The configuration of the service can be done through access of published fields, properties and operations.
To compose models are required more functionalities. We defined a mechanism to navigate through model factories based on the fully qualified C++ class name of the created model. The Java mapping of the simulator offers a method to access model factories and namespaces in the global namespace. E.g. the factories for the C++ models

- *::Example1* and
- *::ExampleNs::Example2*

are accessed respectively as

- *simulator.factories.Example1* and
- *simulator.factories.ExampleNs.Example2*.

It is possible to store in the scripts factories for later usage (similar to the C++ using keyword).
We defined a mechanism to access containers and references from components. As these SMP objects are not part of the resolvable tree, are not required to have unique names in the parent object. The approach used for late binding uses the character underscore (_), prepended to the object name to request a container or reference. The Java mapping then provides a system to add components to the retrieved collections.
This approach allows a very intuitive setup of the simulation trough scripting.
In the first phase, the simulation packages are loaded. This automatically instantiate services and load factories.
In the second phase, the models are composed and connected. Configuration can happen immediately after model creation.
In the third phase, the simulator is transitioned to standby.

Standardising the scripted assembly could prove beneficial. The assembly language standardisation does not necessarily need to mandate the language to use. It could, however, provide consistent names and semantic for the late binding and scripted operations. This would ease migration of assemblies between different simulation platforms.

Future work
SIMSAT-NG is being actively developed at the time of writing, and it is already able to provide a unit and integration framework for the ESOC Generic Models library.
The work will continue, fully replacing SIMSAT in the migration of existing simulator to ECSS-SMP. Further work include the support of execution in distributed containers, performance analysis and optimisation, and add simulation services to increase the support provided by SIMSAT-NG to simulations.

[1] Oracle Corporation, “JDK 11 Release Notes,” in press.
[2] J. Eggleston, “SIMPERF 2 Final Report”, unpublished.
[3] P. Steele, P. Fritzen, and J. Whitty, “SIMULUS Next Generation (SIM NG)”, unpublished.