Model Checking Driven Simulation of Sat Procedures

Giovanni Verzino,[*] Federico Cavaliere,[*] Federico Mari,[†] Igor Melatti,[†]
Giovanni Minei,[*] Ivano Salvo,[†] Yuri Yushtein,[‡] and Enrico Tronci[†]

A Satellite Operational Procedure (OP) consists of a set of instructions reading information from the satellite (telemetries, TM) and sending commands to it (telecommands, TC). An OP can be executed by a human or by a computer (on-board procedures). Typically OPs are mission critical systems since their failure may entail hardware damages, degradation of satellite services or costly human based recovery actions. For this reason OPs are typically thoroughly tested in order to have reasonable assurances about their correctness. Unfortunately, traditional simulation based verification of OPs is highly expensive, since it requires a high amount of time from highly skilled personnel and does not provide formal assurance about the correctness of the OP under verification.

We show how a model checker (CMurphi) can be used to drive a satellite simulator (namely, SIMSAT). The proposed approach has the following benefits. First, it improves OP quality assurance by automatic exhaustive exploration of all possible simulation scenarios whereas a manually driven simulation campaign cannot offer any formal assurance on the coverage achieved by the simulation campaign. Second, it decreases OP verification costs by using a model checker to automatically drive (via fault injections) the simulator. The model checker will record the considered simulation scenarios and automatically generate fresh (i.e., not previously considered) scenarios automatically stopping when all meaningful scenarios have been considered. Third, our approach allows humans to focus on the design of disturbance models (e.g., how many faults it makes sense to consider, when such faults may occur, etc.) which are highly reusable across verification of similar OPs.

We implemented a prototype system by interfacing the CMurphi model checker to the SIMSAT simulator. Our experimental results show the feasibility of the proposed approach.

I. Introduction

MOTIVATIONS Building a satellite, getting it into orbit and then maintaining it from the ground control facility is a big financial endeavor. When orbiting, satellites are controlled from the ground by means of satellite *Operational Procedures* (OPs), executed by *human operators*. OPs consist of a set of instructions reading information from the satellite (telemetries, TM) and sending commands to it (telecommands, TC).

OPs are *mission critical*. In fact, OPs failure may entail hardware damages, degradation of satellite services as well as costly human based recovery actions. Verification of OPs is thus needed in order to avoid failures. However, traditional simulation based verification of OPs is highly expensive, since it requires a huge amount of time of highly skilled personnel. The previous considerations motivate research on methods and tools that allow automatic verification of OPs. This is the focus of the present paper.

CONTRIBUTION In this paper we present a model checking based approach for the automatic verification of OPs. Our approach is aimed at *improving OP quality assurance* by automatic exhaustive exploration of all possible simulation scenarios. Moreover, our solution aims at *decreasing OP verification time* (and thus cost) by using a model checker to automatically drive (via fault injections) the simulator. Finally, our approach allows humans to focus on the *design of disturbance models*–e.g. how many faults are allowed, etc.–which are highly reusable across similar OPs.

Since we use model checking for OPs verification, we need a model for the satellite. Unfortunately, modeling the satellite from scratch using a model checker input language is prohibitively expensive. We overcome this obstruction by exploiting availability of satellite models inside a satellite simulator, namely the SIMSAT simulator[1].

[*]Telespazio S.p.A., PSC Napoli, Via Gianturco 31, 80146 Napoli, Italy, Emails: {federico.cavaliere,giovanni.minei, giovanni.verzino}@telespazio.com. *Contact point*: Giovanni Verzino.

[†]Computer Science Department, Sapienza University of Rome, Via Salaria 113, 00198 Roma, Italy, Emails: {mari,melatti,salvo, tronci}@di.uniroma1.it. *Contact point*: Enrico Tronci.

[‡]Systems, Software & Technology Department, ESA/ESTEC, Keplerlaan 1, PO Box 299, 2200AG Noordwijk, The Netherlands, Email: Yuri.Yushtein@esa.int

We choose CMurphi[2, 3] as the model checker suitable for the context of this paper. The model checker role is twofold. First, it acts as a *driver* for the simulator. To this end, the model checker reads the simulator state on one side and, on the other side, it feeds the simulator with *disturbances*. Second, the model checker models the OP and the human operator behavior. For example, the time needed by the operator to send telecommands to the satellite could affect the procedure result itself. Thus it must be taken into consideration by the model checker.

RELATED WORK In this paper we extend our previous results [4] by formulating a more general formal framework including a model for disturbances. Moreover we apply our approach to a fault-tolerant on-board OP.

PAPER OVERVIEW In Sect. II we give an overview on the model checking problem. In Sect. III we describe how to model the system under verification, together with the initial settings and the properties we want to verify. Sect. IV describes our main contribution, namely how the model checker and the simulator interacts in order to verify operational procedures. Then, Sect. V instantiates the general approach described in Sect. IV. In particular, we use the model checker CMurphi to drive the SIMSAT simulator. Finally, in Sect. VII we show experimental results on using the described method to validate a significant operational procedure, described in Sect. VI.

II. Safety Verification via Model Checking

A model checker is as a software tool that takes as input the definition of a dynamical system, the definition of a property the system should satisfy and checks if during its evolution the system can reach a state where the given property does not hold (error state). Here we only focus on safety properties.

Thus a Model Checking Problem (MCP), that is the input to a model checker, is a tuple $(\text{Init, Next, Adm, Safe})$ such that: Init is a finite set of (initial) system states; Next is a function defining the system dynamics, that is, $x' = \text{Next}(x, d)$ defines the system next state x', from the system present state x and the system uncontrollable input d; $\text{Adm}(x, d)$ is a boolean function returning True (1) if in state x input d is admissible and returning False (0) otherwise; $\text{Safe}(x)$ is a boolean function returning 1 if x is a safe state, 0 otherwise.

A system run (or trace) is a (finite or infinite) sequence $x(0), u(0), x(1), u(1), \ldots x(t), u(t), x(t+1), \ldots$ of state-input such that $x(0)$ is an initial state (that is $x(0)$ is in Init) and for each $t \geq 0$ we have that $x(t+1) = \text{Next}(x(t), u(t))$ and $\text{Adm}(x(t), u(t))$. A counterexample is a finite system run which last state $x(t)$ is unsafe, that is $\text{Safe}(x(t)) = 0$.

Given an MCP $(\text{Init, Next, Adm, Safe})$ a model checker will return PASS if no system run contains an unsafe state (that is one where Safe is 0), FAIL with a counterexample otherwise. Model checking is the problem of computing an answer to an MCP.

Clearly, to enable using model checking in our setting we need to define the tuple $(\text{Init, Next, Adm, Safe})$. This is typically done using the modelling language provided by the model checker. This means that if we want to verify safety of an (on-board) Operational Procedure (OP) we need to model all the environment it is interacting with. This may be prohibitively expensive.

In this paper we will show how safety of an OP can be verified via model checking by exploiting the availability of a satellite simulator to model the OP environment. This enables us to easily define the next state function Next of the system under verification thus enabling cost effective formal verification of OPs.

III. Modeling the System Under Verification

In this section we explain how to model an operational procedure and its environment so as to allow verification via model checking. Namely, Sect. III.A describes how to model the operational procedure environment with the simulator. A model for operational procedures is given in Sect. III.B. Disturbances are modelled in Sect. III.C. Sect. III.D describes the model for the whole system under verification. Then, Sects. III.E and III.F describe the models for initial conditions and the safety properties to verify. Finally, Sect. III.G describes the model checking problem for OP safety verification.

III.A. Modeling the Operational Procedure Environment with the Simulator

We view the simulator as a black box. This is motivated by the fact that indeed not all details of the models inside a simulator may be available in a general setting. This is definitely the case in our specific setting where some of the models inside the simulator have been developed by a third party and are not fully visible to the simulator users. A simulator MS is thus defined by a pair of functions (F, G) computing, respectively, the simulator internal state and the simulator observable output (telemetries in our case).

We take a discrete time approach with sampling time T. As usual in Computer Science, we write z for the present value (i.e., at time kT for some k) of variable z, z' for the next value (i.e., at time $(k+1)T$) of z and we drop indication of T. Thus we have: $x' = F(x, u, d)$ and $y = G(x)$ where: x is the simulator state; x' is the next simulator state; u is the simulator input (telecommands in our case); d is the disturbance input (modelling external events such as faults); y is the simulator output (telemetries in our case).

Control input u does not change during the interval $[kT, (k+1)T]$, that is, it is always u. This stems from the fact that the speed of variation of the control input is finite since in our setting it is provided by a human operator following the control policy defined by an Operational Procedure (OP). The same holds when u is provided by a computer. In any case there is a (positive) time T between changes in the control input. That is, if u changes we must call F again and recompute x accordingly.

The observable output y is just a function of the present state x, present input u and present disturbance d. Disturbance models will be introduced in Sect. III.C.

III.B. Operational Procedures

An Operational Procedure (OP) can be seen as a program observing the simulator output y and sending a command (TC) u to the simulator. Resting on the above considerations, we model an OP as a pair of functions (A, B) computing, respectively, OP internal state and OP output towards the simulator (i.e., telecommands in our case). We have $w' = A(w, y, d)$ and $u = B(w, y)$ where: w is the OP internal state (i.e., program counter, local variables, etc); w' is OP next state.

The above model seems to entail that any T seconds a telecommand is sent to the simulator. Of course, in general this is not the case. This can simply be handled by adding NOP codes to u, meaning that nothing is sent to the simulator.

III.C. Disturbances

Disturbance input d models uncontrollable events such as faults, parameter variations, etc. In general, any event that may influence the system operations and is not under the operator control is modeled as an uncontrollable input, that is a disturbance. If there were not disturbances the system evolution would be perfectly known, which is unrealistic. A disturbance model is thus essential in order to verify correctness of control policies (i.e., Operational Procedures) under realistic conditions. For example, a too conservative disturbance model (very few disturbances) may lead to consider adequate a control policy that instead is not able to cope with real world (unforeseeable) situations. On the other hand, a too liberal disturbance model may rule out adequate control policies, forcing us to use a complex (and expensive) control policy, or even preventing us from finding an adequate policy.

We model disturbances with a pair of functions (Q, D) such that $q' = Q(q, w, y, d)$ and $D(q, w, d) = 1$, where q is the disturbance model present state (e.g., number of faults injected so far, time elapsed since last fault injected, etc.).

When considering OP executed by a human operator the behaviour of the latter should also be considered. Essentially a human operator may introduce delays in the sending of a telecommand to the simulator. Thus the human operator can be modelled as a disturbance that delays the OP execution (for one or more time time steps) and just sends NOP codes to the simulator.

III.D. System Under Verification

The system to be verified is described by the OP, the simulator as well as by the disturbance model. Thus we have:

$$\begin{aligned} x' &= F(x, u, d) && (1)\\ y &= G(x) && (2)\\ w' &= A(w, y, d) && (3)\\ u &= B(w, y) && (4)\\ q' &= Q(q, w, y, d) && (5)\\ & \text{s.t. } D(q, w, d) = 1. \end{aligned}$$

Using the notation introduced in previous sections we have that the system state is $X = [x, w, q]$ whereas the disturbance input is d as defined in Sect. III.C.

Replacing u and y with their definitions (equations 4 and 2 resp.) we get:

$$\begin{aligned}\mathrm{Next}(x, w, q, d) &= [F(x, B(w, G(x)), d), \quad A(w, G(x), d), \quad Q(q, w, G(x), d)] && (6)\\ \mathrm{Adm}(x, w, q, d) &= D(q, w, d) && (7)\end{aligned}$$

Equations 6 and 7 define the functions Next and Adm of the tuple (Init, Next, Adm, Safe) representing the model checking problem (Sect. II) on the system under verification.

Note that our system modeling is a discrete time one (with sampling time T). Since in our framework we do not have a definition of F and G to work with, but can only use the simulator as a black box to compute F and G, it does not appear that a continuous time modeling and verification approach can be pursued in our context. On the other hand, since the human operator reaction time is a (possibly small) finite number, no relevant system behaviour appears to be lost using a discrete time approach.

III.E. Initial States

In general we will be interested in showing a property of our system when it starts from a reasonable initial state. This models the fact the OPs are started from reasonable initial conditions. Accordingly, we assume that we are given a finite set

$$\mathrm{Init} = \{(x_1, w_1, q_1), \ldots, (x_k, w_k, q_k)\} \qquad (8)$$

of initial states. Of course, in general we may wish to consider infinite sets of initial states, since many state components may take up continuous values. However, an explicit model checker can only handle a finite number of initial states. Thus we only consider finite sets of initial states.

Restricting to finite sets of initial states appears reasonable since in our context continuous state variables mainly represent positions (of the satellite, of the moon of the sun, etc). Variations in such values below a certain threshold are not relevant in our context.

We also note that current manual OP testing of course only addresses a finite number of initial states, and indeed a number of initial states that is much smaller than the one a model checker will be able to handle. Thus, even restricting to finite sets of initial states we will still improve OP quality assurance.

III.F. System Properties to be Verified

We are interested in verifying safety properties defined on telemetries, OP internal state and disturbance model state. That is, invariants Inv(y, w, q) where Inv is, as usual for safety properties, a function mapping tuples of telemetries, OP states and disturbance state into boolean values. We ask that for all reachable states Inv must be true. If a reachable state is found where Inv is false (unsafe state) the model checker will stop and return a counterexample, that is a sequence of events (i.e., values for d) leading to the just found unsafe state. In Sect. IV we will illustrate the model checking approach we will consider in our setting.

Thus we have:

$$\mathrm{Safe}(x, w, q) = \mathrm{Inv}(y, w, q) = \mathrm{Inv}(G(x), w, q). \qquad (9)$$

Note that the above formula is in agreement with the fact that we can only save and restore the simulator state and can only observe TMs (y).

III.G. Model Checking Problem for OP Safety Verification

Resting on the above discussion we have that the model checking problem we are interest in is (Init, Next, Adm, Safe), where Init, Next, Adm, Safe are as defined in equations 6–9.

IV. Model Checking Driven Simulation

In this section we present how formal verification of operational procedures can be carried out by using a model checker together with a simulator. In Sect. IV.A we describe how our model checking driven simulation works. In Sect. IV.B we show how the simulator is seen from the model checker. In Sect. IV.C we discuss modeling issues for the system components. Finally, in Sect. IV.D we choose a model checker.

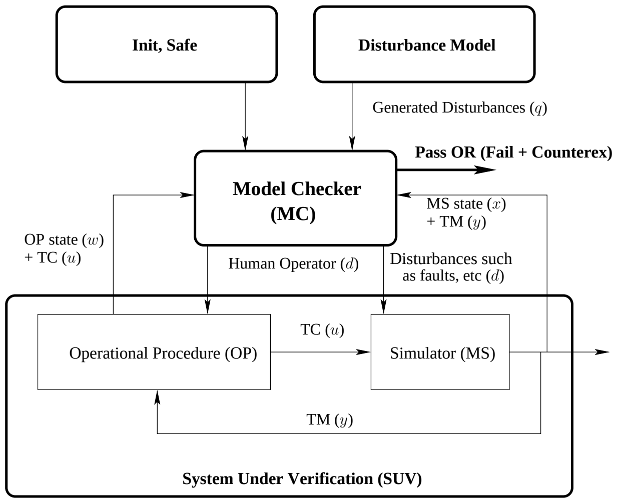

Init, Safe
Disturbance Model
Generated Disturbances (q)
Pass OR (Fail + Counterex)
Model Checker (MC)
MS state (x) + TM (y)
OP state (w) + TC (u)
Human Operator (d)
Disturbances such as faults, etc (d)
TC (u)
Operational Procedure (OP)
Simulator (MS)
TM (y)
System Under Verification (SUV)

Figure 1. Model Checking Driven Simulation.

IV.A. General Description

We assume that at each time instant either we get a disturbance form the environment (e.g., a fault) or the human operator decides to execute the next step of the OP. That is we serialize all events. In our context, this is not a restriction, as long as T is small enough. Note that all possible interleaving of faults and human actions are still considered. Simply we rule out simultaneous events.

In our context we assume that *Telemetries* (TM) y, *Telecommands* (TC) u, OP state w, disturbance model state q, as well as all state x are observable by the model checker.

These considerations lead to the schema in Fig. 1 where the model checker acts as a malicious controller for the SUV. That is, the model checker will try to choose sequences for d and u so as to drive the SUV to an unsafe state. This realizes a model checking driven simulation.

The SUV, formally defined by Eqs. 6 and 7, is composed by the simulator, the OP, the human operator executing the OP and the disturbances reaching the simulator. The OP reads TMs from the simulator while sending TCs to it. The model checker drives the simulation by substituting the human operator (executes the next OP instruction or stays idle) and by injecting disturbances to the simulator. In order to explore all possible SUV evolutions, the model checker will suitably set simulator states.

Note that in our setting we cannot check for equality between two states. In fact, this entails a deep understanding of the simulator domain which is what we want to avoid here. Conservatively we assume that any event leads to a new–not previously visited–state. Consequently, we identify a state with the sequence of events needed to reach it. Then two states are equals if they are reached with the same sequence of events from the same initial state. In our context we cannot have infinite sequences of events since OPs always terminate. Thus the state space explored by the model checker in our setting is finite.

Our formal verification approach is based on using a model checker as a driver for a given system simulator (a satellite simulator in our context). Along the same lines, our approach can be applied to each system whose description is rather complicated but for which a simulator exists, e.g. automotive or avionic systems.

IV.B. The Simulator As Seen From The Model Checker

The model checking approach we are using in our context is *explicit*. Namely, the model checker performs a simulator state space exploration via *Depth First Search* (DFS). In order to allow the model checker to properly interact with the simulator, we need to model the simulator itself inside the model checker. Thus we have to define the following functions:

- A function reading an initial *simulator state*, say `read_initial_state()`;
- A function *reading a given TM* inside the current simulator state, say `read_TM()`.
- A function setting the simulator state, say `set_simulator_state()`.

- A function giving the *next simulator state obtained by sending a TC*, say `simulator_TC_step()`.
- A function giving the *next simulator state obtained by injecting a disturbance*, say `simulator_disturbance_step()`.

Implementation of the above set of functions depends on the system at hand. In Sect. V.C we describe how they are defined in our context.

IV.C. Modelling Of System Components

In order for the model checker to properly work as a driver for the simulation, we have to model the behavior of OPs, the human operator, disturbances and safety properties. In our approach this is done by feeding the model checker with an input file describing such models. We assume that OPs are deterministic and human operators correctly execute OP instructions. Thus, if we ignore disturbances (e.g. faults), there is only one source of non-determinism in OPs: the *human operator idle time*, that is, the time elapsing between the execution by the operator of two consecutive instructions. In fact, if there are no disturbances, two executions of a given OP only differ in the timing, that is the time intervals elapsing between the execution of two OP instructions. Such a non-determinism allows us to check, for example, if in the OP constraints about the time allowed between two operations (for example, small enough, or large enough) are missing. Furthermore, when disturbances are present non-deterministic delay between OP operations allows us to verify correctness of the interaction between disturbances and time delay in execution of OP instructions.

IV.D. Selecting a Model Checker

First of all we note that we do not have available a system description in our setting. In fact, while we can compute the system next state using the simulator as a black box, we do not have a description of the function implemented by the simulator. This rules out symbolic approaches as, for example, those used in symbolic model checkers for hybrid systems such as HyTech [5], UPPAAL [6], PhaVer [7]. Indeed, we note that all symbolic model checkers for hybrid systems target linear hybrid systems. If we had a description of the function implemented by the simulator it would certainly be nonlinear. Thus, even in that case a symbolic approach may not be directly usable.

As a matter of fact, one may claim that indeed a description of the function implemented by the simulator is available as the source code of the program implementing the simulator itself. For small systems this approach can indeed be pursued using software model checkers like CBMC [8]. However we note that our system is all but small. Furthermore, it will involve complex arithmetical computations, which typically make verification intractable for SAT based or OBDD based model checkers. See [9] for a survey on software model checking.

The above considerations have led us to focus on explicit model checkers. Examples are SPIN [10] and CMurphi [2, 3]. Since CMurphi has already the capability of handling finite precision (i.e., C-like) real numbers, as well as interfaces toward external functions (like the one implemented by the simulator) we decided to base our work on CMurphi.

V. Driving SIMSAT Simulator with CMurphi Model Checker

In this section we instantiate the general approach described in Sect. IV. In particular, we use the model checker CMurphi to drive the SIMSAT[1] simulator. SIMSAT (Simulation Infrastructure for the Modeling of SATellites) is the simulation infrastucture, able to host a spacecraft and ground segment simulation, developed by EGOS[11]. Here we use SIMSAT as an oracle to predict the next state of the satellite system. This removes the need to explicitly implement a satellite model in our framework. In the remaining part of this section we describe the realization of the system as shown in Fig. 2.

V.A. System Overview

From an architectural point of view, the model checker and the simulator will run in parallel as different processes. Thus, interactions (TMs and TCs) between them are exchanged via inter-process communications. Note that such processes may also be executed on different hosts, thus communication takes place through a LAN. This yields the *client-server* architecture shown in Fig. 2. The system actors are the simulator SIMSAT, the model checker CMurphi and a client-server interface between the simulator and the model checker. We name such interface *Model Checking for Operational Procedures* (MC4OP) *Interface*.

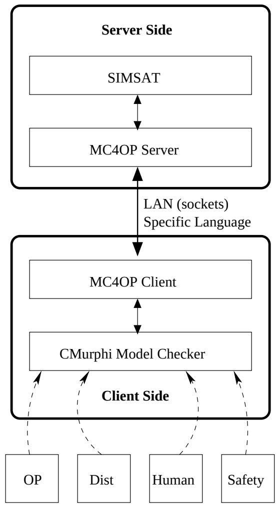


Figure 2. Driving SIMSAT Simulator with CMurphi model checker.

The MC4OP Interface acts as a protocol converter between CMurphi and SIMSAT. On the *server side*, detailed in Sect. V.B, MC4OP receives commands from the client side and forwards them to the simulator. MC4OP then sends the simulator answers back to the client. On the *client side*, detailed in Sect. V.C, MC4OP drives the simulation interfacing with CMurphi. Inputs to the client are the OP model, the disturbance model, the safety properties specification and the human operator model, coded in the CMurphi input language.

Communications between MC4OP client and server use a specific language described in Sect. V.D.

The formal verification process is carried out using a DFS on the SIMSAT simulator state space, as detailed in Sect. V.E.

V.B. Server side

We supply the MC4OP interface with a set of functionalities allowing to control the SIMSAT simulator. Namely we can: *start* and *halt* a simulation; *save* and *load* a breakpoint; *set* and *get* a SIMSAT item, i.e. TCs, parameters or TMs. The ability to set an item is needed in order to inject failures by changing values in the simulation model, as well as to be able to send TCs and receive TMs.

V.C. Client side

The MC4OP client supplies the model checker with the functionalities needed to interface with the simulator, explained in Sect. IV.B. Moreover, the client side of the architecture contains the models for OPs, disturbances, human operator and safety properties to verify. The set of functions are implemented as follows.

The simulator state consists of *Telemetries* (TMs) values (which may be retrieved by the OP, and thus from the model checker) as well as SIMSAT state files (called *breakpoints*). Since each breakpoint is a huge file (order of MBs), which prevents whole breakpoints sending from SIMSAT to the model checker (especially when they run on different hosts), and since the internal structure of each breakpoint is not known (some of the models inside the simulator have been developed by a third party), only breakpoints *names* can be seen by the model checker. Breakpoint names univocally identify states on SIMSAT, thus in the following we will consider them equivalent.

Function `read_initial_state()` takes as input an index i and returns the name of the i-th initial simulator state file name. The rationale is that a number of meaningful initial scenario is prepared in files on the simulator machine. Function `read_initial_state()` returns the name of the i-th of such files. In this way we can easily handle the case in which many initial states are possible.

Function `read_TM()` takes as input the current simulator state file name s and a TM name j. It then returns the value of TM j in state s.

Function `set_simulator_state()` takes as input a simulator state file name s and sets the current simulator state to s.

Function `simulator_TC_step()` takes as input the current simulator state name s, a *Telecommand* (TC) c given by the OP, and a time T, and returns as output the simulator state name s' after T time units, as a result of executing TC c. Note that the actual SIMSAT state file for s' is saved on the host where SIMSAT is running, while the model checker only gets the state file name for s'.

Function `simulator_disturbance_step()` takes as input the current simulator state name s, a disturbance d, and a time T, and returns as output the simulator state name s' after T time units, as a result of injecting disturbance d on s.

In our setting not all system states are observable, thus `simulator_TC_step()` and `simulator_disturbance_step()` return a fresh name each time that they are called. Of course it may very well be the case that two different sequences of events lead to the same state. Considering different such states returns correct results albeit it duplicates the work since the computation goes through states that have already been considered. Methods to correctly and efficiently detect duplicate states may be an interesting further development for the present study.

V.D. Client-Server Communication Language

MC4OP client and server communicate by using a specific language. It consists of six commands, detailed in the following.

- `RUN_TC T_Slice Cmd [Param]`

  Function: Executes a simulation time slice with a command. Namely: i) sends the command `Cmd`, ii) starts the simulation, iii) waits `T_Slice` milliseconds and iv) stops the simulation

  Example: Start simulation switch on Heater 032 and stops after 10 seconds: `RUN_TC 10000 Z44AD`

  Returns: `RUN_TC DONE`

- `RUN_NOP T_Slice`

  Function: Executes a simulation time slice. Namely: i) starts the simulation, ii) waits `T_Slice` milliseconds and iv) stops the simulation

  Example: Start simulation and stops after 10 seconds: `RUN_NOP 10000`

  Returns: `RUN_NOP DONE`

- `SET_P Name Value`

  Function: Sets parameter `Name` to value `Value`

  Example: Set parameter S.TTC.SBT1.Loop to 100: `SET_P S.TTC.SBT1.Loop 100`

  Returns: `SET_P DONE`

- `GET_P TM_Pkt TM_Name`

  Function: Gets value of TM parameter name

  Example: Get value of parameter T057: `GET_P STCU1 T057`

  Returns: `GET_P DONE Value`

- `SAVE_BRK Id`

  Function: Saves a breakpoint, which name is built from `Id`.

  Example: `SAVE_BRK 2`

  Returns: `SAVE_BRK DONE`

- `RESTORE_BRK Id`

  Function: Restores a breakpoint, which name is built from `Id`.

  Example: `RESTORE_BRK 98`

  Returns: `RESTORE_BRK DONE`

V.E. Verification Process

At the beginning of the verification process, the OP to be checked, the disturbance model (both faults and human operator) and the invariants to check are loaded (see Fig. 1). Then the model checker performs a depth first search on the finite simulator state space, using the simulator as a model. Finally, CMurphi checks whether each read simulator state is safe against the input safety properties or not, raising an error flag if this is not the case. In this latter case, a counterexample is returned.

This process ends when all reachable SIMSAT states are visited by CMurphi. Since in our context simulator states are finite, the described procedure will always end.

VI. A Case Study

MODEL OF OP To validate the above approach we have applied it to the fault-tolerant operational procedure `TempCtr()` in Lst. 1, aiming at driving the temperature of a satellite heater inside a certain region. We describe `TempCtr()` using a PASCAL like pseudo-programming language (similar to the CMurphi input language). The purpose of such a procedure is to drive TM `TM_Heater_Temp` to a value between 23 and 25. This is done by properly sending TCs `TC_Heater_On` or `TC_Heater_Off`.

We want our procedure to be robust with respect to failures in the thermistors. In fact, if failures in sensors are not adequately handled we may reach an unsafe state. For example, a broken thermistor returns -40°C. If we keep heating until we reach the target temperature without considering the fact that a thermistor may be broken we actually reach unsafe temperatures since, being the resistor stuck at -40°C, we would be heating indefinitely. To avoid this kind of problems sensor faults should be always considered in the OP. Of course a safe approach could be to turn everything off as soon as a fault is detected. However this is often too conservative. For example, in our case we have two thermistors: `THR_057_GT_01` (index 1), and `THR_058_GT_02` (index 2). Thus the best approach, to avoid unnecessarily turning off the heating process, is to design an OP that exploits both of them. This is done in OP `TempCtr()` shown in Lst. 1.

OP `TempCtr()` makes 5 attempts (counted by variable `tentative` to drive the temperature within the desired interval. The temperature itself is measured by telemetry (TM) `TM_Heater_Temp`. If after 5 attempts the temperature is still outside the desired range then the procedure reports a failure. If the measured temperature is below -40°C then we should try to use the other thermistor if available. If both thermistors are broken `TempCtr()` reports a failure.

Once a reliable temperature measure has been acquired the temperature is checked. That is, if the measured temperature is below 23°C then telecommand (TC) `TC_Heater_On` turning the heater on is sent else TC `TC_Heater_Off` turning the heater off is sent. After having sent a TC, procedure `TempCtr()` waits 6 minutes in order to give enough time to the temperature to increase (heater on) or decrease (heater off). After that, a new measure of the temperature is taken and a check is made to see if the temperature is in the desired range of values. If this is the case the procedure terminates successfully, else it loops for another attempt.

```
procedure TempCtr()
begin
  tentative := 0; TM_Heater_Id = 1;
begin_loop:
  tentative := tentative + 1;
  if (tentative > 5)
    then return (FAILURE); endif;
  TM_Heater_Temp := read(TM_Heater_Id);
  -- check for failures in thermistors
  while (TM_Heater_Temp <= -40) do
    switch (TM_Heater_Id)
      case 1:
        TM_Heater_id := TM_Heater_id + 1;
        TM_Heater_Temp := read(TM_Heater_Id);
      case 2:
        send(TC_Heater_Off); return (FAILURE);
    enswitch;
  endwhile
  -- switch heater on or off
  if (TM_Heater_Temp <= 23)
    then send(TC_Heater_On);
    else send(TC_Heater_Off);
  endif;
  -- wait and read again
  wait 6 minutes;
  TM_Heater_Temp := read(TM_Heater_Id);
  -- check for failures in thermistors
  while (TM_Heater_Temp <= -40) do
    switch (TM_Heater_Id)
      case 1:  TM_Heater_id := TM_Heater_id + 1;
               TM_Heater_Temp := read(TM_Heater_Id);
      case 2:  send(TC_Heater_Off); return (FAILURE);
    enswitch;
  endwhile
  -- if temperature is in range then SUCCESS
  if ((TM_Heater_Temp >= 23) and (TM_Heater_Temp <= 25))
    then return (SUCCESS);
    else goto begin_loop;
  endif;
end;
```

Listing 1. A fault-tolerant OP: TempCtr()

MODEL OF DISTURBANCES In our setting a disturbance is the breaking of a thermistor. Namely, as shown in Lst. 2, the model checker can set the parameter `SPACECRAFT.THC.Thermistors.THR_057_GT_01.IsFailed` or `SPACECRAFT.THC.Thermistors.THR_057_GT_02.IsFailed` to **True**. The disturbance model we used in our verification experiments asks that in each system run there are no more than two disturbances and that the occurrences of such disturbances are not too close in time.

```
-- precondition:
--   1. send at most twice per execution;
--   2. disturbances not too close in time;

  SPACECRAFT.THC.Thermistors.THR_057_GT_01.IsFailed := true;
or
  SPACECRAFT.THC.Thermistors.THR_057_GT_02.IsFailed := true;
```

Listing 2. A small model for disturbances

MODEL OF HUMAN OPERATOR We have defined a small model for human operator, shown in Lst. 3. Namely, the model checker can execute the next OP step with a TC or can stay idle. For the purposes of this paper, we assume that there are no delays from the human operator, namely the parameter `idle_time` is set to 0. This means that we are considering `TempCtr()` as if it were an *on-board procedure*.

```
procedure execute_next_OP_step();
procedure stay_idle(idle_time); -- for idle_time seconds
```

Listing 3. A small model for human operator

SAFETY PROPERTIES We have defined a significant set of safety properties for the above OP, listed in Lst. 4. Namely, the safety property to be verified for procedure `TempCtr()` is that the temperature never falls outside a given safety interval ($[-50°\text{C}, 25°\text{C}]$ in our case) and that within the given time horizon (200 minutes in our case) the OP terminates (with success or with failure). The latter property ensures that there cannot be never-ending loops in the OP.

```
TM_Heater_Temp >= -50;
TM_Heater_Temp <= 25;
Within a given maximum time (200 mins) we have success or failure.
On success we have tentative <= 5;
On failure we have tentative > 5;
```

Listing 4. A small set of safety properties

VII. Experimental Results

In order to assess feasibility of our approach, we have applied it to the case study of Sect. VI.

Fig. 3 shows the state space (as a graph) explored by the model checker during the verification activity. Each node in the graph represents a simulator state, edges represent transitions. Each transition (edge) is labelled with the event causing it. Namely, `RUN_NOP` denotes the case in which no TC is sent to the simulator (NOP action), `Z44AD` denotes the event where the TC turning on the heater is sent to the simulator, `GET_P` denotes the reading of TMs from the simulator. Codes `P0`, `P1` denote, respectively telemetries for the two thermistors we used (`THR_057_GT_01` and `THR_058_GT_02`).

The verification performed by CMurphi ends with no error (that is, all reachable states satisfy the given safety property) and the number of reachable states is very small (i.e., 268) for model checking standards.

In the remaining part of this section we analyse performances of our approach. In Sect. VII.A we evaluate the time saving achieved with our proposed approach with respect to the time needed to attain the same coverage when the simulation campaign is manually driven (rather than model checking driven as in our case). In Sect. VII.B we evaluate the computational effort from the model checker as well as from the simulator in order to identify possible bottlenecks.

VII.A. Effectiveness of the Proposed Approach

Simulation has the goal of verifying that a given OP meets the given specifications. Such a verification activity is typically carried out by replacing the satellite with a simulator (SIMSAT in our case) and by relying on human experts for execution of the OP and for insertion of disturbances (such as faults). Note that at design time only the set of possible faults (disturbance model) is known. The faults that will actually be inserted during a specific simulation campaign are only known to the personnel in charge of injecting such faults. That is the personnel executing the OP does not know in advance the fault sequences that will be injected.

Of course, in order to cover as many scenarios as possible, many runs of the system are considered. Each starting from the given initial state and with a different sequence of fault injections. To speed up the simulation it is possible to save and later restore a system state (breakpoint). The simulation can thus be started from any of such previously saved states.

In the proposed approach the model checker replaces both the personnel executing the OP as well as the one inserting the faults. Note that the model checker will exercise the system with all possible (according to the disturbance

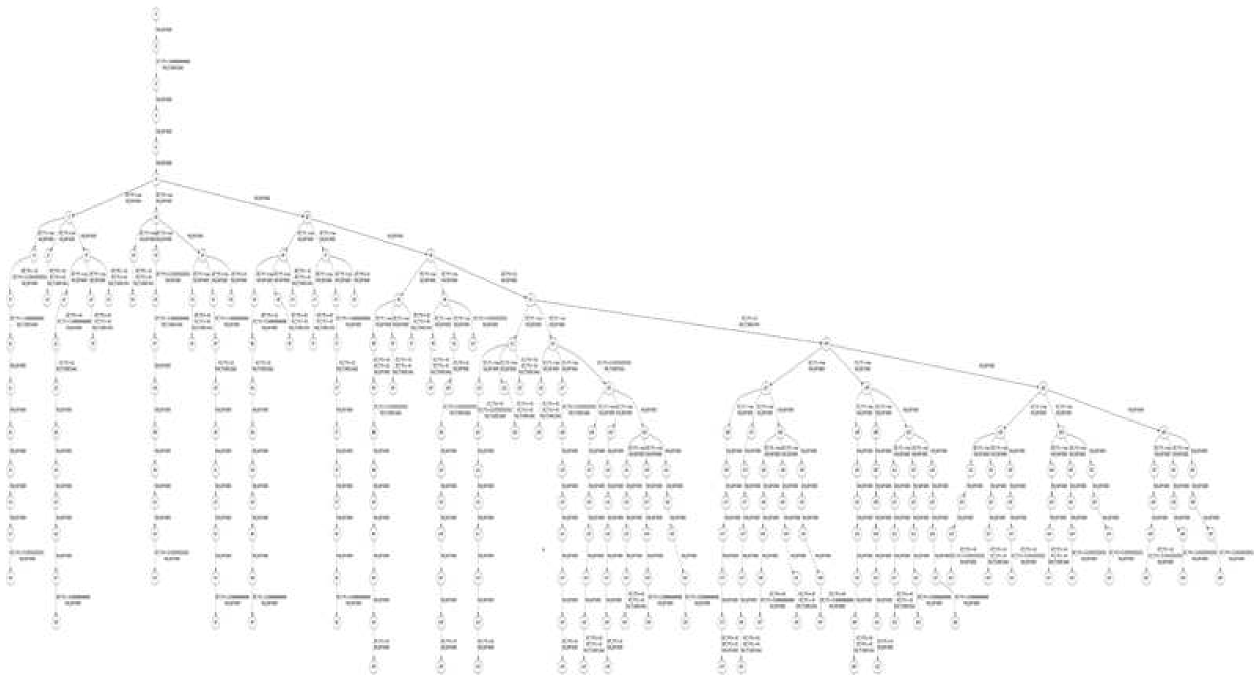
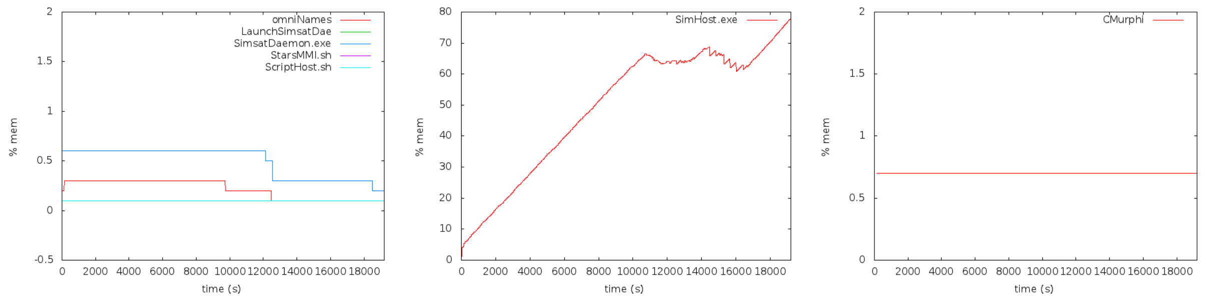


Figure 3. State space for verification of `TempCtr()`.



(a) RAM usage of SIMSAT processes (b) RAM usage of only `SimHost.exe` (c) RAM usage for CMurphi model checker

Figure 4. RAM usage.

model) sequences of faults. Thus the first benefits of the proposed approach is that of automating the execution of a simulation campaign as well as its bookkeeping (making sure that all possible sequences of faults are considered). This will allow the personnel to focus on development of disturbance models (which are highly reusable).

Furthermore, using the proposed approach the time taken by the simulation campaign decreases. This is due to the ability of the model to save and restore all states reached during the verification activity. This can be hundreds or thousands of states. Handling such an amount of saved states is infeasible for a human being, but it is easy for a computer. In the following we will clarify this point using as an example the verification activity summarized in the state space graph shown in Fig. 3.

Each *full path* (that is a path from the root to a leaf) in the tree of Fig. 3 defines a simulation run. The time taken by a simulation run is roughly proportional to the number of edges in the full path defining it. In the following we will compute the time (as the number of edges traversed) taken by the manually driven simulation campaign and the time taken by the model checker driven simulation campaign for the case study in Sect VI. This will illustrate how the model checking driven simulation approach saves time.

First of all, the length of all full paths is computed. For example the leftmost path of Fig. 3 counts 15 edges, whereas the second path counts 8 edges. Then the time taken by a manual simulation campaign is proportional to the sum of the edges of the full paths. For example, the first two full paths sum to $15+8 = 23$. For the whole state space in Fig. 3 such a sum is 694. This computation assumes that after a full path (simulation run) is completed the simulation is always restarted from the initial state (tree root). Of course we may save some states (breakpoints) to avoid always restarting from the root. Using breakpoints will decrease the number of edges traversed and thus the simulation time. Note however that using a manual approach we cannot handle too many breakpoints just because their bookkeeping becomes too complex (and boring) for a human being.

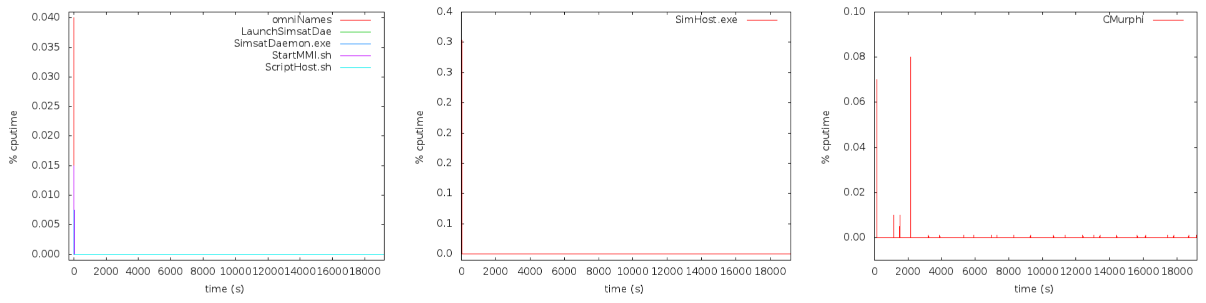
Using the model checking driven approach proposed in this paper the time of the simulation campaign is (roughly) proportional to the number of edges in the tree of Fig. 3, that is 268. This stems from the fact that, by exploiting the save and restore mechanism, the model checker never goes twice through a graph edge. With respect to the manual approach this yields a time saving of $(694 - 268)/694 = 426/694 = 0.61$. That is, the proposed model checking driven simulation approach saves about 60% of the time needed to complete the simulation campaign.

VII.B. Performance Evaluation

In order to evaluate the performances of the proposed approach for the OP previously described we measured CPU and memory usage for both the SIMSAT simulator and the model checker driving it. In the following we present the results obtained.

All experiments have been carried out on a PC with 3GB RAM and an AMD Athlon 64 x2 Dual Core Processor 4600+. The operating systems was a Linux, distribution Ubuntu 10.04LTS. We refer to this PC in the following as the host machine. Our target scenario is one in which the simulator and the model checker run on different machines. We reproduced such a scenario by creating on the host machine a virtual machine on which the SIMSAT simulator will run. The virtual machine is configured to have 1 GB RAM, and an AMD Athlon 64 x2 Dual Core Processor 4600+. The operating system running on the virtual machine is a Suse Linux Enterprise SDK 9 (SLES SDK 9). Summing up, the model checker runs on the host machine (Linux Ubuntu PC) whereas the simulator runs on the above virtual machine (Linux Suse PC).

The total (system and CPU) time needed to complete verification in the above setting was about 6 hours. In the following we will examine how this time is spent. For the CMurphi model checker we monitored the CPU and



(a) CPU usage of SIMSAT processes (b) CPU usage of only `SimHost.exe` (c) CPU usage for CMurphi model checker

Figure 5. CPU usage.

memory usage during the verification activity. The SIMSAT simulator consists of many processes. We monitored those relevant for our purposes. Namely: `omniNames`, `LaunchSimsatDae`, `SimsatDaemon.exe`, `StartMMI.sh`, `SimHost.exe` e `ScriptHost.sh`.

MEMORY USAGE Fig. 4 shows the memory usage. In particular, Figs. 4(a) and 4(b) show the memory usage for the SIMSAT simulator processes. Note that since the simulator is running on a (virtual) machine with 1GB of RAM we have that a RAM usage of 2% corresponds to about 20 MB of RAM. Note that all simulator processes but `SimHost.exe` use a small amount of RAM (no more than 20 MB). Note moreover that, unlike the other simulator processes, `SimHost.exe` has a memory usage that is about linearly increasing with time. The reduction in memory usage at time 10000 is likely due to the Java garbage collection. Fig. 4(c) shows the memory usage for the CMurphi model checker. Note that since the model checker is running on a machine with 3GB of RAM we have that a RAM usage of 1% corresponds to about 30MB of RAM. Thus the amount of RAM taken by the model checker is constantly small (about 30 MB). This may look strange model checker being notoriously memory eager. However we note that in this case the number of visited states (268) is very small (for model checking standards) and, although each state (breakpoint) has a size of about 3MB, the model checker only needs to save state names (a few bytes each). For this reason the RAM that CMurphi allocates upon starting suffices to complete the verification task.

Summing up, Fig. 4 shows that all processes but `SimHost.exe` only use a small amount of RAM. On the other hand, given enough running time, process `SimHost.exe` will take all the available memory. Indeed, presently, the amount of memory taken by this process is the main limiting factor for the available prototype system. We think this is due to the fact the SIMSAT has not been designed to efficiently handle a single simulation run in which thousands of breakpoints are saved and restored (our situation here). It would be very useful if future versions of SIMSAT could smoothly handle the above described scenario.

CPU USAGE Fig. 5 shows the CPU usage. In order to compute CPU usage of a process we proceed as follows. Each T sample seconds (5 seconds in our setting) we measure how many CPU seconds the process has used since the beginning of its execution. Let CPU(t) be such a number at sample t. At sample 0 (starting of the process) we have CPU$(0) = 0$. The CPU usage CPU-usage(t) at sample $t > 0$ is the fraction of the sampling time used by the process. That is at $t > 0$ we have CPU-usage(t) = (CPU(t) $-$ CPU$(t-1)$)$/T$.

In particular, Figs. 5(a) shows for the simulator processes a peak during the initialization phases and then a very small CPU usage. The same basically holds for the model checker, as shown in Fig. 5(c). This is due the the large amount of time spent in saving (restoring) breakpoints to (from) the disk. In fact, to complete the verification activity takes about 6 hours whereas the CPU time taken by the model checker is just 0.3 seconds.

VIII. Conclusions

We have presented a model checking approach for the automatic verification of satellite operational procedures (OPs). In order to apply our approach we have to model the satellite, the OP, the human behavior and disturbances within the model checker. The main obstruction to overcome is to model the satellite. In this paper we have shown how to overcome this obstruction by using a suitable simulator (SIMSAT) for the satellite. With our approach, the model checker (CMurphi) has a twofold role: 1) to act as a driver for the SIMSAT simulator generating disturbances (such as faults), and 2) to model the OP human operator.

Our approach is aimed at improving reliability by supporting automatic exhaustive verification of OPs. In fact, all possible simulation scenarios, that is sequences of events (paths on our state space), are considered by the model checker driving the simulation.

In order to assess feasibility of our approach we presented experimental results on a fault-tolerant on-board OP. Our results show that we can save up to 60% of the verification time w.r.t. the *simple human driven simulation*.

Note that our model checking driven simulation approach can be adopted for all safety-critical systems in which a simulator exists, e.g. automotive, avionics, etc.

The activity carried out suggests many interesting future developments. First, easy the verification of on-board procedures by taking as input the program (e.g., Javascript) defining such a procedure. Second, improve SIMSAT handling of save/restore activities in our scenario. This will avoid that SIMSAT process `SimHost.exe` fills up the available RAM thus preventing automatic verification of large OPs. Third, use the system to automate the simulation campaign of some of the control software running on the satellite.

ACKNOWLEDGEMENTS This research has been partially supported by ESA ITI AO6067/B00010362 "Model Checker Validator for Satellite Operational Procedure".

References

[1] "Introduction to SIMSAT Web Page: http://www.egos.esa.int/portal/egos-web/products/Simulators/SIMSAT/," 2011.

[2] Della Penna, G., Intrigila, B., Melatti, I., Tronci, E., and Venturini Zilli, M., "Exploiting transition locality in automatic verification of finite-state concurrent systems," *International Journal on Software Tools for Technology Transfer (STTT)*, Vol. 6, No. 4, 08 2004, pp. 320–341.

[3] "CMurphi Web Page: http://mclab.di.uniroma1.it/software_cmurphi.html," .

[4] Cavaliere, F., Mari, F., Melatti, I., Minei, G., Salvo, I., Tronci, E., Verzino, G., and Yushtein, Y., "Model Checking Satellite Operational Procedures," *DAta Systems In Aerospace (DASIA), Org. EuroSpace, Canadian Space Agency, CNES, ESA, EUMETSAT. San Anton, Malta, EuroSpace.*, May 2011.

[5] Henzinger, T. A., Ho, P.-H., and Wong-Toi, H., "HyTech: A Model Checker for Hybrid Systems," *Software Tools for Technology Transfer*, Vol. 1, No. 1, dec 1997, pp. 110–122.

[6] Larsen, K. G., Pettersson, P., and Yi, W., "UPPAAL: Status and Developments," *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, edited by O. Grumberg, Vol. 1254 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 456–459.

[7] Frehse, G., "PHAVer: Algorithmic Verification of Hybrid Systems past HyTech," *International Journal on Software Tools for Technology Transfer*, Vol. 10, No. 3, jun 2008.

[8] Clarke, E., Kroening, D., and Yorav, K., "Behavioral consistency of C and verilog programs using bounded model checking," *Proceedings of the 40th annual Design Automation Conference*, DAC '03, ACM, New York, NY, USA, 2003, pp. 368–371.

[9] Schlich, B. and Kowalewski, S., "Model checking C source code for embedded systems," *Int. J. Softw. Tools Technol. Transf.*, Vol. 11, June 2009, pp. 187–202.

[10] Holzmann, G. J., *The SPIN model checker: Primer and reference manual*, Addison Wesley, 2004.

[11] "EGOS ESA Ground Operating System Web Page: http://www.egos.esa.int/portal/egos-web/," .