MODEL CHECKING SATELLITE OPERATIONAL PROCEDURES

Federico Cavaliere[1], Federico Mari[2], Igor Melatti[2], Giovanni Minei[1], Ivano Salvo[2], Enrico Tronci[2], Giovanni Verzino[1], and Yuri Yushtein[3]

[1] *Telespazio S.p.A., PSC Napoli, Via Gianturco 31, 80146 Napoli, Italy, Emails:* {federico.cavaliere,giovanni.minei,giovanni.verzino}@telespazio.com
[2] *Computer Science Department, Sapienza University of Rome, Via Salaria 113, 00198 Roma, Italy, Emails:* {mari,melatti,salvo,tronci}@di.uniroma1.it
[3] *Systems, Software & Technology Department, ESA/ESTEC, Keplerlaan 1, PO Box 299, 2200AG Noordwijk, The Netherlands, Email:* Yuri.Yushtein@esa.int

ABSTRACT

Satellite Operational Procedures (OPs) are mission critical systems which verification typically requires months of simulation. Automating OP verification by using model checking techniques to explore all possible scenarios will decrease OP verification cost and increase OP reliability. The main obstruction is modeling the satellite inside a model checker. We show how, by using an explicit model checker (CMurphi), it is possible to exploit a satellite simulator (SIMSAT) to automate OP verification. We model OPs, disturbances as well as the behavior of the human operator by using the model checker modeling language and instead use the satellite simulator as a model of the satellite itself. We achieve this by using the model checker as a driver for the simulation activity. In order to assess feasibility of our approach we present experimental results on a simple yet meaningful OP. Our results show that we can save up to 90% of verification time.

Key words: Automatic verification of satellite operational procedures; Model checking; SIMSAT.

1. INTRODUCTION

Motivations 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[8].

We choose CMurphi[3, 2] 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.

Paper Overview In Sect. 2 we describe how to model the system under verification, together with the initial settings and the properties we want to verify. Sect. 3 describes our main contribution, namely how the model checker and the simulator interacts in order to verify operational procedures. Then, Sect. 4 instantiate the general approach described in Sect. 3. In particular, we use the model checker CMurphi to drive the SIMSAT simulator.

Finally, in Sect. 6 we show experimental results on using the described method to validate a small operational procedure, described in Sect. 5.

2. 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. 2.1 describes how to model the operational procedure environment with the simulator. A model for operational procedures is given in Sect. 2.2. Sect. 2.3 describes the model for the whole system under verification. Finally, Sects. 2.4 and 2.5 describe the models for initial conditions and the safety properties to verify.

2.1. 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 into account that the human operator takes at least time T to react. As usual in Computer Science, we write z for the present value (i.e., at time kT) 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: $x' = F(x, u, d, t)$ and $y = G(x, u, d)$ where: x is the simulator state at time t; x' is the next simulator state at time $t+T$; u is the simulator input (telecommands in our case) at time t; d is the disturbance input (modelling external events such as faults) at time t; y is the simulator output (telemetries in our case) at time t.

The rationale behind such a modeling is the following. In general the simulator model may be non-stationary, that is function F may also have the present time t as an argument. This is definitely our case since depending on time t the satellite and the celestial bodies (e.g., sun, moon, etc) will have different positions. Thus, even starting from the same state x, the system evolution may be different. However, given the kinematics of satellite and celestial bodies, the simulator computes their trajectory from the actual position. Thus, as a matter of fact, the present time is just an easy way of defining the present position of satellite and celestial bodies. If such information is in state x (as it is the case for us) we may use the above stationary model for MS.

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.

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.

As for control inputs, we assume that the disturbance input stays constant at least for time T. Note that while such an assumption always holds for controllable inputs u, since they are human generated in our setting, this may be not the case for disturbances which, in general, are not human generated. For example, we may have two consecutive faults (for example, generated by very slow wearing processes) arbitrarily close in time. By making T small enough we can always assume, for all practical purposes, that faults are never simultaneous. We also note that the actual testing approach to OPs injects faults manually, so as a matter of fact the same considerations for u apply also there. Thus, assuming that disturbances are time-separated by at least T seconds we do not lose coverage with respect to the actual manual testing approach for OP.

The observable output y is just a function of the present state x, present input u and present disturbance d.

2.2. Operational Procedures

An *Operational Procedure* (OP) can be seen as a program observing the simulator output y and sending commands u to the simulator. However, unlike computer programs, an OP is executed by a human operator. Thus the time elapsing between two steps of an OP may be arbitrary. For example, it cannot be too small (since human operators are not infinitely fast), and moreover we do not know it a priori. In other words, the human operator is an uncontrollable input (much as a disturbance) to the OP deciding when OP should move to the next step. This means that at each time instant the human operator can decide (uncontrollable input) if OP should move to the next step. We use a synchronous modeling where any T seconds the human operator decides if moving to the next step of OP or just stay in the present OP state (may be waiting for some other external event). By making T

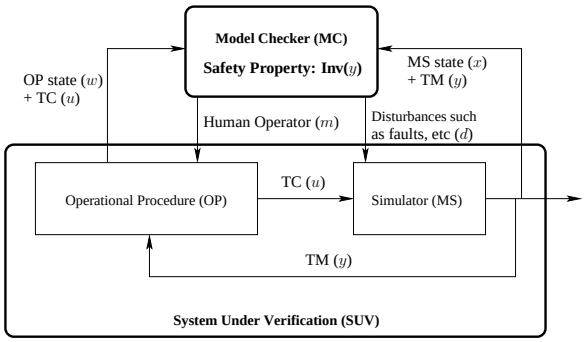

Figure 1. Model Checking Driven Simulation

small enough we can make our model as precise as we like.

Resting on the above considerations, we model an OP as a pair of functions (A, C) computing, respectively, OP internal state and OP output towards the simulator (i.e., telecommands in our case). We have $w' = A(w, y, m)$ and $u = C(w)$ where: w is the OP internal state (i.e., program counter, local variables, etc); w' is OP next state; y is OP input (i.e., telemetries) from the simulator; m is the operator decision (uncontrollable event) about executing the next step of OP or stay idle; u is the telecommand OP sends to the simulator.

Note that OP only observes the simulator output at times $0, T, 2T, 3T, \ldots$. This is in agreement with the fact that the human operator takes T seconds to react. As a result, OP implements a sample-and-hold control schema for the simulator.

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.

2.3. System Under Verification

The system to be verified is described by the simulator together with the given OP. Thus we have:

$$x' = F(x, u, d, t) \quad (1) \qquad w' = A(w, y, m) \quad (3)$$
$$y = G(x, u, d) \quad (2) \qquad u = C(w) \quad (4)$$

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

$$x' = F(x, C(w), d, t) \quad (5)$$
$$w' = A(w, G(x, C(w), d), m) \quad (6)$$

Equations 5 and 6 define the system state, that is (x, w), as a function of the uncontrollable inputs (d and m), that is, the inputs that the model checker will set in order to exercise 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.

2.4. 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 $I = \{(x_1, w_1), \ldots, (x_k, w_k)\}$ 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.

2.5. System Properties to be Verified

We are interested in verifying safety properties defined on telemetries and OP internal state. That is, invariants $\text{Inv}(y, w)$ where Inv is, as usual for safety properties, a function mapping pairs of telemetries and OP states 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 and m) leading to the just found unsafe state. In Sect. 3 we will illustrate the model checking approach we will consider in our setting.

3. 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. 3.1 we describe how our model checking driven simulation works. In Sect. 3.2 we show how the simulator is seen from the model checker. In Sect. 3.3 we discuss modeling issues for the system components. Finally, in Sect. 3.4 we choose a model checker.

3.1. General Description

We assume that at each time instant at most one of d or m is active. That is, 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, 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. 5 and 6, 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.

3.2. 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. 4.3 we describe how they are defined in our context.

3.3. 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.

3.4. 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 [6], UPPAAL [9], PhaVer [5]. 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 [1]. 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 [10] for a survey on software model checking.

The above considerations have led us to focus on explicit model checkers. Examples are SPIN [7] and CMurphi

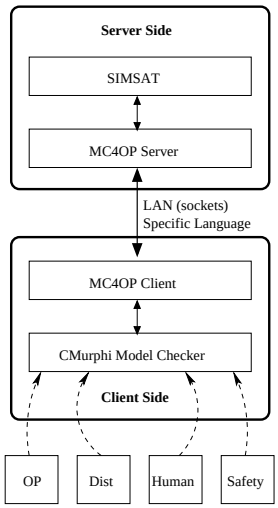

Figure 2. Driving SIMSAT Simulator with CMurphi model checker

[3, 2]. 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.

4. DRIVING SIMSAT SIMULATOR WITH CMURPHI MODEL CHECKER

In this section we instantiate the general approach described in Sect. 3. In particular, we use the model checker CMurphi to drive the SIMSAT[8] 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[4]. 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.

4.1. 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 novel client-server interface between the simulator and the model checker. We name such interface *Model Checking for Operational Procedures* (MC4OP) *Interface*.

The MC4OP Interface acts as a protocol converter between CMurphi and SIMSAT. On the *server side*, detailed in Sect. 4.2, 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. 4.3, 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. 4.4.

The formal verification process is carried out using a DFS on the SIMSAT simulator state space, as detailed in Sect. 4.5.

4.2. 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.

4.3. Client side

The MC4OP client supplies the model checker with the functionalities needed to interface with the simulator, explained in Sect. 3.2. 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*). Being each breakpoint 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 being the internal structure of each breakpoint 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.

4.4. 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`

4.5. 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. 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.

5. A CASE STUDY

Model of OP To validate the above approach we have applied it to the small yet meaningful operational procedure `HeaterRegulation()` in Lst. 1, aiming at driving the temperature of a satellite heater inside a certain region. We describe `HeaterRegulation()` 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 25 and 35. This is done by properly sending TCs `TC_Heater_On` or `TC_Heater_Off`.

Model of disturbances We have defined a small model for disturbances, shown in Lst. 2. Namely, the model checker can set the parameter SPACECRAFT.THC.Thermistors.THR_057_GT_01.IsFailed to true. This setting can be done at most once along any OP execution. Moreover, this disturbance cannot be sent at the very beginning of the OP execution.

Table 1. Results on the case study in Lsts. 1– 4

T_{op}	Time	Reach	SimStates	Runs	Gain
1800	4.51e+03	69	491	14	85.9%
1620	1.53e+04	229	2225	65	89.7%
1560	3.45e+04	390	3865	112	89.9%

```
procedure HeaterRegulation()
begin
  tentative := 0;
begin_loop:
  tentative := tentative + 1;
  if (tentative > 3)
    then return (FAILURE); endif;
  read(TM_Heater_Temp);
  if (TM_Heater_Temp <= 25)
    then send(TC_Heater_On); endif;
  if (TM_Heater_Temp >= 35)
    then send(TC_Heater_Off); endif;
  wait 30 seconds;
  read(TM_Heater_Temp);
  if ((TM_Heater_Temp >= 25) and
      (TM_Heater_Temp <= 35))
    then return (SUCCESS);
    else goto begin_loop;
  endif;
end;
```

Listing 1. A small OP: HeaterRegulation()

```
-- precondition:
--   1. send at most once per execution;
--   2. not at the very beginning;

  SPACECRAFT.THC.Thermistors.THR_057_GT_01.
      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. The latter situation cannot happen more than one time along any OP execution and cannot happen at the very beginning of the OP execution.

```
-- precondition:
--   1. send at most once per execution;
--   2. not at the very beginning;

procedure execute_next_OP_step();
procedure stay_idle();
```

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.

```
TM_Heater_Temp >= -100;
TM_Heater_Temp <= 100;
On success we have tentative <= 3;
On failure we have tentative > 3;
Within a given maximum time we have success
    or failure.
```

Listing 4. A small set of safety properties

6. EXPERIMENTAL RESULTS

In order to assess feasibility of our approach, we have applied it to the case study of Sect. 5.

To this aim, we run CMurphi (and SIMSAT) on three different disturbance and human operator models for the OP shown in Lst. 1. Namely, let T_{op} be a model parameter defined as the number of seconds we wait before injecting a disturbance or allowing the human operator to stay idle. Note that decreasing T_{op} will inject earlier disturbances, thus increasing the number of reachable states. Then, we run CMurphi by setting T_{op} to three decreasing values. Our experimental results are in Tab. 1. Tab. 1 columns meaning is as follows. Column T_{op} is the model parameter discussed above. Column **Time** is the total verification time in seconds (CMurphi + SIMSAT). Column **Reach** is the number of *reachable* states, that is the states visited by the model checker CMurphi. We can compare our model checking driven simulation with a *simple human driven simulation* that always starts simulation runs from the initial state. Of course in principle a human may save and restore breakpoints as a model checker does, however for more than a few dozens of breakpoints this is in practice unfeasible for a human. Accordingly, column **SimStates** shows the number of states a *simple human driven simulation* would have to visit. Column **Runs** is the number of different simulations we would have to run without using our approach (i.e. traditional manual simulation) to achieve the same coverage. Column **Gain** is the gain we obtain with our approach, i.e. $1 - \text{Reach}/\text{SimStates}$. Figure 3 graphically depicts the resulting simulation tree for the case $T_{op} = 1800$.

CMurphi memory usage is negligible (8 MB) since the number of states is small by model checking standards. On the server side, each SIMSAT state file has a size of 3MB, thus at most 1.1GB of disk space is needed for the longest verification (i.e. for $T_{op} = 1560$).

From Tab. 1 we see that we are able to save nearly 90% w.r.t. the traditional simulation approach.

7. 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

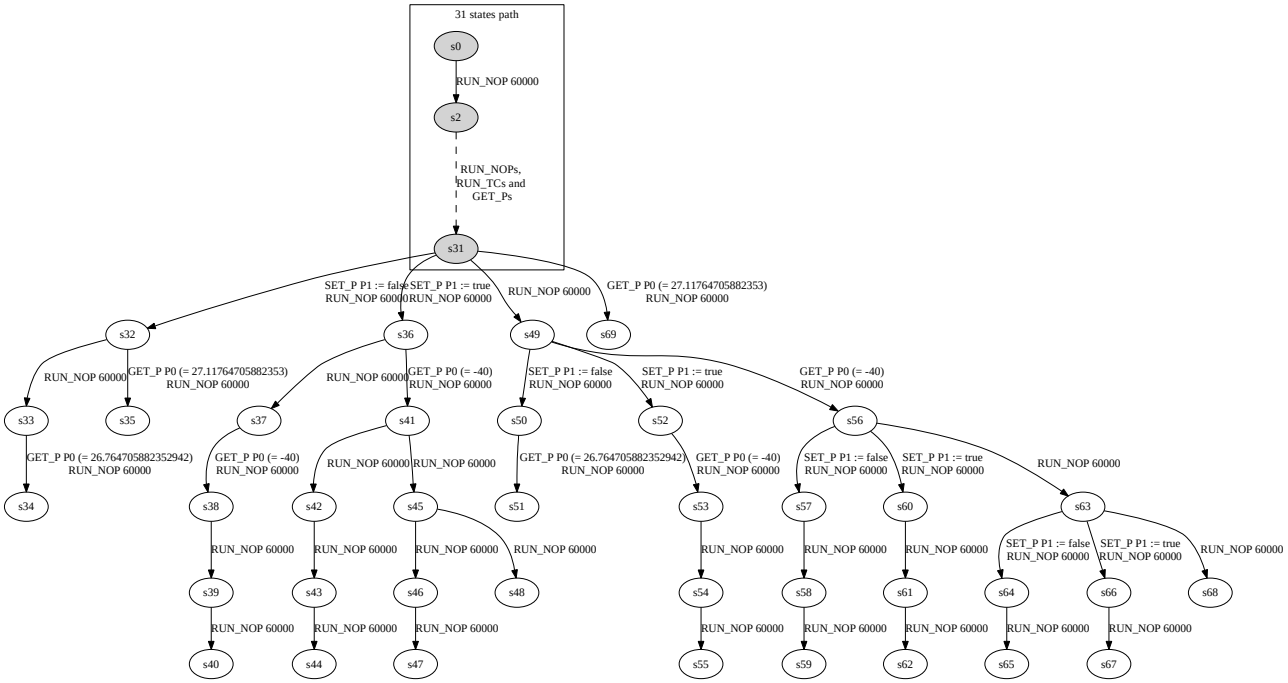

Figure 3. State space for verification of `HeaterRegulation()` *with* $T_{op} = 1800$

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 preliminary experimental results on a simple meaningful OP. Our results show that we can save up to 90% of the verification time w.r.t. the *simple human driven simulation*.

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

As future work, we plan to apply our approach to the verification of on-board control procedures.

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

REFERENCES

[1] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *DAC*, pages 368–371, 2003. ACM.

[2] Cmurphi web page: http://mclab.di.uniroma1.it/software_cmurphi.html.

[3] G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. Venturini Zilli. Exploiting transition locality in automatic verification of finite-state concurrent systems. *STTT*, 6(4):320–341, 08 2004.

[4] EGOS esa ground operating system web page: http://www.egos.esa.int/portal/egos-web/.

[5] G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. *STTT*, 10(3), jun 2008.

[6] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. *STTT*, 1(1):110–122, dec 1997.

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

[8] “Introduction to SIMSAT” web page: http://www.egos.esa.int/portal/egos-web/products/Simulators/SIMSAT/, 2011.

[9] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL: Status and developments. In *CAV*, LNCS 1254, pages 456–459. Springer, 1997.

[10] B. Schlich and S. Kowalewski. Model checking C source code for embedded systems. *STTT*, 11:187–202, June 2009.