A Statistical Approach for Timed Reachability in AADL Models

Harold Bruintjes, Joost-Pieter Katoen
Software Modeling and Verification Group
RWTH Aachen University, Germany
Email: {h.bruintjes,katoen}@cs.rwth-aachen.de

David Lesens
Airbus Defence and Space
Route de Verneuil
78133 France
Email: david.lesens@astrium.eads.net

***Abstract*—We introduce a simulator (`slimsim`) for a subset of AADL extended with formalized behavioral semantics for nominal and error models. The simulator allows to perform probabilistic analysis using the Monte Carlo method, on linear-hybrid, stochastic models, which describe a combination of nominal and error behaviors of hard- and software components. The tool supports the use of different *strategies*, which control the behavior of the simulator when dealing with various forms of non-determinism. The simulator is tested using benchmarks of the COMPASS toolset, as well as a case study by Airbus Defense and Space.**

I. Introduction

The design of safety critical systems requires a thorough analysis to ensure that all the safety requirements are met. In practice, this means that engineers must ensure that the probability of failure for a system is below a given threshold. Various methods are applied, such as Fault Tree Analysis, with varying degrees of automation. In particular, the interest in (safety) analysis of real-time systems is still growing, resulting in the creation of new methods and tools.

As the main contribution of this paper, we present the `slimsim` tool, which is a Monte Carlo simulator for specifications written in an extended subset of the Architecture Analysis & Design Language (AADL) [1]. It has been integrated into the COMPASS toolset [2], featuring non-deterministic, real-time and continuous stochastic semantics, which previously could not be treated at the same time as the expressivity of all these features goes beyond the capabilities of the numerical analysis engines so far. Statistical model checking in these areas is not new, and various tools such as UPPAAL-SMC [3], Modes [4] and PLASMA-lab [5] exist, but either do not support part of the semantics, or interpret it in a different way. In particular, the `slimsim` tool can handle specifications with transitions guarded by real-time constraints or triggered by exponential distributions, and processes that may synchronize on a shared alphabet of events. Non-determinism is resolved by different *strategies*, allowing the simulation process to be tailored to the specific needs of the analysis.

Additionally, by means of a small, synthetic case study we show the effect of applying different strategies to resolve non-determinism of the input model. Resolving non-determinism is a well-known problem and various approaches are known, including those mentioned in this paper as well as related work.

The paper is structured as follows: In Section II some preliminary details are given concerning Monte Carlo analysis and an overview of the COMPASS toolset and its SLIM language. The implementation of the simulator is explained in Section III, detailing its architecture, the use of strategies, and parallelization. Section IV shows the results of a benchmark of the simulator, followed by a case study performed on a larger example in Section V.

II. Preliminaries

A. Statistical Model Checking

Statistical model checking techniques [6] make use of the Monte Carlo method to check the satisfiability of a temporal logic formula on a given model. Discrete event simulation is used to randomly generate finite paths in the given model, verifying whether or not some time-bounded property holds for that path. As more paths are generated, better statistical information about the property can be derived from these results.

This way the Monte Carlo approach approximates the probability of the property holding based on the simulated paths. As the outcome of each generated path can be seen as some binary result (which is true if and only if the property holds true), the outcomes of all paths can be seen as identically distributed Bernoulli random variables. This allows various statistical conclusions to be made using only a finite number of paths.

Statistical model checking can be used both for qualitative and quantitative purposes. Qualitative analysis is generally based on hypothesis testing, determining whether a certain property holds or not. The tool described in this paper focuses on the analysis of *quantitative properties*, specifically timed reachability properties, determining the probability that a given property holds true.

B. Quantitative Statistical Analysis

In order to perform quantitative statistical analysis, we use the Chernoff-Hoeffding (CH) bound, which is described in [7]. This bound is based on two parameters, δ and ϵ, which control the statistical confidence and error bound of the statistical outcome respectively. These parameters drive the formula $P[|\bar{X} - \pi_0| \leq \epsilon] = 1 - \delta$, stating that the probability of the difference between the true probability π_0 of the property and the estimator $\bar{X}$ being bounded by ϵ lies within the confidence $1 - \delta$. The estimator $\bar{X}$ is determined by A/N, where the CH-bound N is determined by the formula $N = 4\ln(\frac{2}{\delta})/\epsilon^2$, and A is the number of randomly generated paths satisfying the property of interest.

C. COMPASS Toolset

The COMPASS toolset [2] was developed as part of several research projects funded by the ESA in order to improve tool capabilities for formal analysis during the early design stages of for instance spacecraft [8]. As such, it is geared towards, but not limited to, the space and avionics industry. The toolset has successfully been used for various case studies by various industrial partners and the ESA [9], [10], and is accessible from http://compass.informatik.rwth-aachen.de.

The main modeling capabilities are expressed in the SLIM language, a carefully designed dialect of AADL. It is used to specify the system being analyzed (explained further in Section II-D). This model can be analyzed in various ways:

Correctness analysis allows qualitative properties to be checked using either BDD or SAT based model checking [11], [12] (for explicit respectively bounded model checking), abstracting from stochastic semantics.

Performability is its quantitative counterpart, translating the input model to a Continuous Time Markov Chain (CTMC) or Interactive Markov Chain (IMC) [13]. Here, real-time semantics are abstracted away.

Properties for both analyses are expressed using user friendly specification patterns, which are translated into LTL, CTL or CSL equivalents, based on the tool used to perform that analysis.

Safety analysis allows the generation of Fault Trees and FMEA (Failure Mode and Effects Analysis) tables for input models containing failure modes. Furthermore, the Fault Trees can be further evaluated to determine the probabilities of the various events.

FDIR analysis can check whether certain fault conditions in the model can be detected, isolated and recovered from. These fault conditions are based on the notion of *alarms* and *observables*, which are Boolean elements in the model that may be triggered by certain conditions.

Finally, *diagnosability* [14] is supported, which verifies that the specification is able to correctly diagnose faults, meaning that a property expressing the diagnosis must either always or never hold in any two states with the same set of observations.

D. The SLIM Language

The SLIM language [15] (short for System-Level Integrated Modeling language) is strongly based on AADL, a language developed in the avionics and automotive industry [1]. It supports the specification of timed and hybrid systems (supporting linear dynamics). Furthermore, it includes fault specifications, which describe various possible causes of faults that can optionally be triggered by events occurring with an exponential rate.

Nominal models are specified as sets of components, representing various parts of the complete system, such as processors, buses and threads. These components may contain other (sub)components, which can either be another system component, or a data component. Data components can be defined as integers (or ranges thereof), Booleans, real numbers, and clock and continuous variables. The latter allow for the specification of *timed* or *linear hybrid* dynamics.

The behavior of components is specified by means of modes and mode transitions. Transitions are triggered by a discrete event, possibly internal, with an optional Boolean guard over data components and optional side effects modifying data components. Components can be connected by means of ports, which can be data ports (where output values are expressions over input values) or event ports. Data connections are limited to the discrete and real types. Event port connections enable transitions of various components to synchronize.

Based on the state of a component, its sub-components may be enabled or disabled. This is referred to as *dynamic reconfiguration*, and allows modeling of e.g. integration or removal of components, or change in their electrical state. This can be used to model spare components for example, where one component can be exchanged with another.

An example SLIM model is shown in Listing 1, which represents a simplified GPS unit with two operational modes, `acquisition` and `active`. When activated, the GPS attempts to acquire a signal, which is specified to succeed within two minutes (but no faster than ten seconds). It then switches to an active mode, upon which a variable indicating a fix, `measurement`, is set to true.

Error models are specified separately from nominal models. They describe fault behavior by means of error states and transitions, similar to the modes and transitions of nominal components. Transitions however are triggered either by error events or error propagations. Error propagations are similar to events in nominal specifications, and can synchronize with other error components. Error events cannot synchronize, but they are typically associated with exponential distributions, controlling the rate at which they occur.

An error component can be associated with a nominal component by means of fault injections, a process referred to as model extension [15]. Fault injections specify the effect of a fault occurring in the error model on the nominal model, by means of modifying its data. Model extension automatically adds error propagation connections between sibling components, or components with a parent-child relationship, allowing them to propagate.

An example error model is depicted in Listing 2. In this model, a system can switch from a nominal state, `ok` in this example, to any error state (`transient`, `hot`, or `permanent`) by means of an error event (with a different rate for each type of fault), governed by an exponential distribution. Recovery from a transient fault is possible by a non-deterministic time delay in the interval [200,300] msec, and from a hot fault by restarting (the `@activation` event).

E. Model Semantics

We present here a simplified version of the formal model underlying the SLIM language, with the full semantics available in [16]. It is similar to Priced Timed Automata [4], Stochastic Timed Automata (STA) [17] and the Probabilistic timed processes described in [18], but with small differences, most notably pertaining to exponential rates for individual transitions, support for event based synchronization and/or forms of non-determinism.

Listing 1. Example SLIM nominal model of a simple GPS unit

```
device gpsDevice features
    measurement : out data port bool default false;
end gpsDevice;

device implementation gpsDevice.i
  flows
    measurement := true in modes (active);
  modes
    acquisition : activation mode
                        urgent in 2 min;
    active      : mode;
  transitions
    acquisition -[ within 10 sec
                   to 2 min ]-> active;
end gpsDevice.i;
```

Listing 2. Example SLIM error model of a simple GPS unit

```
error model gpsError features
    repair : out error propagation;
end gpsError;

error model implementation gpsError.i
  events
    e_trans     : error event occurrence
                          poisson 0.1  per hour;
    e_hot       : error event occurrence
                          poisson 0.1  per day;
    e_permanent : error event occurrence
                          poisson 0.01 per day;
  states
    ok          : initial state;
    transient   : error state urgent in
                          300 msec;
    hot         : error state;
    permanent   : error state;
  transitions
    ok          -[ e_trans     ]-> transient;
    transient   -[ repair within 200 msec
                   to 300 msec ]-> ok;
    ok          -[ e_hot       ]-> hot;
    hot         -[ @activation ]-> ok;
    ok          -[ e_permanent ]-> permanent;
end gpsError.i;
```

A specification is defined by one or more *processes* $P = \langle L, l_0, \mathrm{I}, \mathrm{Tr}, \mathit{Var}, A, T \rangle$, where L is the finite set of locations, $l_0 \in L$ the initial location, Var the set of variables, A the set of actions including internal action τ, and

- $\nu : \mathit{Var} \to V$ the valuation function with $\nu(v)$ being a value from the domain of variable $v \in \mathit{Var}$;
- $\mathrm{I} : L \to \mathit{Expr}$ the function that assigns to each location an invariant expression in Expr, which is a Boolean expression over the continuous variables in Var. They restrict the residence time in that location.
- $\mathrm{Tr} : L \times \mathit{Var} \to \mathbb{R}$ the function that assigns the (constant) derivative of all *continuous* variables in all locations.
- $T \subseteq L \times A \times \mathit{Expr} \cup \mathbb{R} \times \mathrm{E} \times L$ the set of discrete transitions, with Expr being the set of Boolean guard expressions over the variables Var; $\mathbb{R}$ being a real-valued parameter which if set indicates an exponential delay – this value may only be set for transitions with the internal action τ; and $\mathrm{E} : (\mathit{Var} \to V) \to (\mathit{Var} \to V)$ being the function that updates the valuation upon taking the transition.

In order to prevent ill-defined semantics for probabilistic transitions, a single location may contain only transitions with a Boolean guard or exit rate, it cannot combine both. Furthermore, for locations with transitions that have an associated exit rate, the invariant must be true.

The state of a process is defined as a tuple $L \times (\mathit{Var} \to V)$, assigning a valuation of the variables to the current location. A *discrete transition* $\langle l_s, \alpha, g, e, l_t \rangle \in T$ allows the system to move from location l_s to l_t, executing action α. The transition is enabled if g is an expression that evaluates to true in the current state, or describes an exponential delay. Upon execution of the transition, the effect e is applied upon the valuation ν of the variables.

A *timed transition* updates the valuation of the continuous variables, based on the invariants of the current location. For a continuous variable $v \in \mathit{Var}$ and a delay $d \in \mathbb{R}$, $\nu'(v) = \nu(v) + \mathrm{I}(v) * d$. For a discrete variable v, $\nu'(v) = \nu(v)$ remains the same.

A complete specification is defined by a *network* of communicating processes. The state space of such a specification is the cross-product of the state spaces of each process. Discrete transitions can occur individually or in parallel, synchronizing on the shared part of the combined alphabet of actions. Note, a transition with an exponential delay cannot synchronize with any other transition as the internal action τ does not synchronize with other processes.

Various sources of non-determinism exist. Multiple transitions T may be enabled at the same time, between different processes or within a single process. Second, multiple delays may be possible depending on the invariant of the current location. Before discrete event simulation is possible, these forms of non-determinism have to be resolved, see Section III-B.

Path generation of such processes is then possible by evaluating for a given state the possible discrete and timed transitions. By employing a given strategy to resolve non-determinism for both the discrete and continuous dynamics, the next state can be generated by means of a probability distribution over either the discrete or timed transitions. Generally, a path is then generated by alternating between timed and discrete transitions, see also [18].

Figure 2 shows a simplification of the model specified in Listing 2. Here, three locations are represented as nodes, with the label shown above the invariant of that location. Edges represent the discrete transitions, with the action shown above the guard or exit rate. The clock representing the delays in the model is an implicit variable, reset to zero at every discrete transition.

F. COMPASS Toolset Architecture

The COMPASS toolset makes use of various tool components to provide the supported functionality. A frontend parses input models and properties, which can then be translated by backends into formats usable by the various tools. As mentioned before, the properties can be represented by various logics such as LTL, CTL and CSL, converted into the format

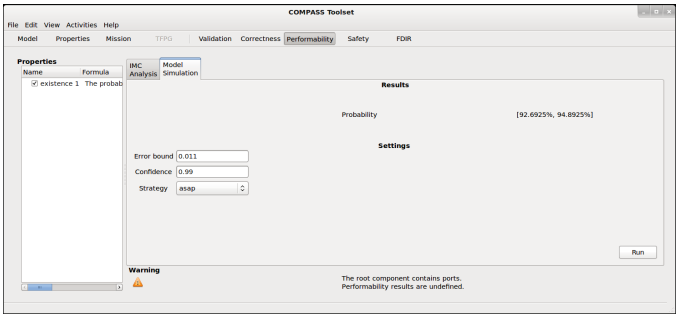

Fig. 1. Example of `slimsim`. After opening a model file, the user can enter the required confidence and error bound, and specify the strategy to use. Then, the run button will start the simulation.

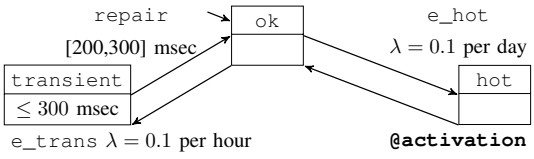

Fig. 2. Simplified STA of GPS error model.

supported by the various tools. Two backends are available for the translation of the model. The primary, preexisting backend translates the model into SMV [19] for use with the NuSMV model checker [11]. NuSMV can then be used for analysis, or its outputs can be chained to other tools (such as MRMC, see Section IV).

As part of the work presented in this paper the simulator format has been added as a secondary backend. This allows for a single, consistent interface for validating the input model before performing the analysis (such as checking for recursively defined components).

The toolset provides the translated input model to the tool(s) corresponding to a particular analysis. After analysis, it processes the results and presents it to the user. Two interfaces are available for this: A GUI (see Figure 1) provides user friendly access, and a CLI (Command Line Interface) provides more direct control and automation.

III. Implementation

A. Simulator Architecture

The architecture of the simulator can roughly be divided in three parts: One part represents the static structure of the input model; One part implements the behavior of the model and finally a third part deals with the actual simulation.

The specification is loaded from a file that is generated by the toolset, which in turn is translated into the corresponding data structures. From this data, the actual model instance is constructed which contains the concrete connections between the various components that may have been defined. Furthermore, the various data of the model are allocated and initialized.

The model is then used to construct the event-data network, which builds the connection topology of event and data ports and keep tracks of the current state of the model [15]. This network, the *Network of Event Data Automata* can then be used to analyze the behavior of the model, by allowing the state to change by progressing time or selecting discrete transitions between modes. The event connections and data flows are used by the network to determine which transitions can synchronize, and the global effect of data assignments.

The simulation logic consists of three sub-parts: The strategy determines how non-determinism is resolved, and is explained in Section III-B. The generator part deals with statistical analysis of the current results and determines whether or not further simulation is required to attain the desired accuracy and precision. Currently the generator implements the Chernoff-Hoeffding bound, but future extensions may allow other approaches such as Chow-Robbins or Gauss [20] (this may require further considerations, see also Section III-C). Finally, there is a part responsible for path generation, integrating the other two parts into the actual simulation engine.

The complete simulator has been implemented in the C++ language, consisting of approximately 14,000 lines of code. Additionally about 200 lines of Python code were necessary to integrate it into the COMPASS toolset.

B. Strategies

The simulator supports the definition of *strategies*. This mechanism allows the user to control the behavior of the simulator where the input specification does not precisely dictate what the next step should be (due to non-determinism). Since such behaviors can alter the outcome of the simulation, it is left to the user to decide what approach suits the analysis best. In [18], it is shown that various approaches are possible, each possibly leading to different outcomes of the statistical analysis. Before the analysis is started, the user has to specify what strategy to use (along with the confidence and error bound).

The simulator implements four automated strategies, and one manual input strategy:

- ASAP – The ASAP strategy implements the resolution of time delays by determining the first possible time point at which a discrete step becomes enabled. This defines an ‘urgent’ semantics, where the model moves as fast as possible. This strategy is similar to the one employed by the MODES tool [4].
- Progressive – The progressive strategy determines the exact intervals in which a discrete transition is active, and randomly selects a time point from these intervals by a uniform distribution, similar to UPPAAL-SMC [3].
- Local – The local strategy only considers the invariant of the current location, selecting the widest possible range of delays.
- MaxTime – The MaxTime strategy will delay as much as is allowed by the invariant of the current location. This strategy can in particular helpful to find actionlocks [18].
- Input – The input strategy asks the user what the next step should be for each step in the simulation. It presents the possible alternatives, both as discrete transitions and time delays, as well as the current state of the model.

As explain before, Figure 2 shows a simplified automaton of the GPS example from Listing 2. The effects of the various strategies can be exemplified by the transition from location `ok` to `transient`. It is guarded by a non-deterministic time interval between 200 and 300 msec. Here, the ASAP strategy will schedule a delay of 200 msec, whereas MaxTime will schedule 300 msec. The Progressive strategy uniformly selects from the interval [200, 300] msec, determined by the guard. The Local strategy ignores the guard, and selects from the interval [0, 300] msec, based on the invariant.

For all strategies, underspecification of choice is always resolved using a uniform distribution using the notion of equiprobability, where the Progressive, Local, and MaxTime strategies select the delay before the transition, and the ASAP strategy select the (first executable) transition first, with only one possible delay. Underspecification of time is, in so far a strategy considers, an interval resolved by a (continuous) uniform distribution as well.

C. Parallelization

The algorithm for Monte Carlo simulations lets itself be parallelized rather trivially, as the outcome of each simulation does not depend on any other. Thus, it makes sense to distribute the workload in order to improve performance. However, care should be taken that the use of multiple parallel processes does not introduce any bias. The work in [21] shows that taking a sample from a process into account as soon as it arrives alters the outcome based on the number of processes. A solution is to balance the workload between processors, ensuring each processor performs the same amount of simulations. In the case of the CH-bound, the number of samples required is known a-priori and so a trivial solution is to have each processor calculate N/k samples, for k processors. However, a more general procedure is described in [22] where N does not have to be known in advance. Here, the results of the processors are buffered, until at least one sample is available from all processors. Then, these samples are taken from the buffer. This approach has been implemented in our simulator, to support the use of other generators such as Chow-Robbins or Gauss [20] in the future.

D. Deadlocks

One particular problem Monte Carlo simulators face is dealing with deadlocks. In such cases, it is not possible to produce further events, and the path generation has to stop. Depending on the semantics of the underlying model, this may be valid or invalid behavior. SLIM admits the specification of models containing deadlocks and depending on the intention of the user, this may not be desirable. In such cases, `slimsim` can be configured to generate an error upon the detection of a deadlock. In other cases, a path leading to a time- or deadlock is considered to falsify the property being checked, as reaching a goal state from such a state is not possible.

IV. Benchmarks

A comparison was made between the simulator and the original analysis flow using CTMCs. In order to generate the CTMCs from the input model, the toolset takes several steps. First, the input model is translated into a NuSMV model.

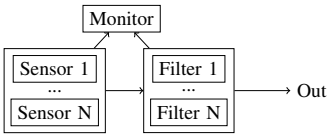

Fig. 3. The layout of the sensor-filter model.

TABLE I. Benchmarking results for the sensor-filter model. The model size indicates the number of both sensors and filters. The values for the simulator are maxima.

Model Size	CTMC Time (s)	Simulation ($\epsilon = 10^{-3}$) Time (s)	Simulation ($\epsilon = 10^{-4}$) Time (s)	CTMC Memory (MB)	Simulation Memory (MB)
2	5.33	47.50	4453.09	23.19	19.91
4	29.93	51.61	4887.09	76.29	21.99
6	59.75	50.58	4725.42	89.14	24.21
8	289.97	52.65	4801.40	180.22	26.63
10	725.67	52.65	5120.87	178.35	29.13
12	1360.11	57.80	5398.62	196.43	32.75
14	3187.02	59.88	5668.64	2469.20	35.00

Using the NuSMV model checker, the reachable state space is generated as a BDD, which is then exported to a data format used by the Sigref library [23]. In the next step, the Sigref library is used to reduce this state space by means of weak probabilistic bisimulation (preserving the reachability properties), and generate a CTMC. Finally, MRMC [24] is used to analyze the model based on the property specification.

As this part of the tool-chain is limited to *discrete* models, an un-timed model (a model without clocks) was used to perform the comparison. This model describes a system consisting of a sensor and a filter component, both with various degrees of redundancy. A monitor can detect a fault in either component, and switch to a redundant version. When either all sensors or all filters have failed, the entire system fails. By increasing the number of redundant components, the complexity of the model is increased. See also Figure 3.

The sensor provides a discrete output in a limited range (1..5). This output is then multiplied by a constant factor in the filter. The sensor has a failure mode in which the output becomes too high (>5); the filter has a failure mode which sets the output to zero. These values are interpreted by a monitor, which distinguishes between these two failures and switches the corresponding component to a redundant version. When a component fails and there are no more redundant components, the entire system has failed. The benchmark defines a time bounded property that determines the probability of this event.

The main results of the benchmark are shown in Table I. The benchmark was executed on a HP 685c G7 blade system, having four AMD 6172 Opteron processors (with 48 cores total) and 192GB of RAM. Both the simulator and MRMC used the same input model, with the input parameter δ set to 0.98 and ϵ to $1 \cdot 10^{-3}$ and $1 \cdot 10^{-4}$ (see Section II-B). The simulator was run with 20 threads in parallel.

It can be seen that the time and resource usage of the CTMC approach increase greatly with model size. This is to be expected as the entire state space needs to be generated, which drastically increases in size with larger models. For the simulator, these values increase slightly, as the number of paths generated (and thus results stored) remains constant. However, the simulation time increases quadratically as the error bound

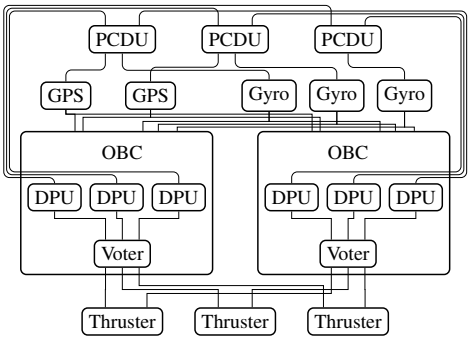

Fig. 4. The architecture of the industrial case study. The connections between the GPS, Gyro and DPU units have been hidden for clarity. Rounded connections are for power, the others for signals.

is reduced in size (which is directly related to the use of the CH-bound).

Some remarks are in order: Although the CTMC approach is sensitive to the size of the model, it is less so in the time bound of the property being analyzed. The opposite holds for the simulator. Here, an increase in the time bound may lead to the path generator requiring more steps to reach this bound, thus increasing simulation time. It should be noted that the tool chain spends relatively little time in the MRMC model checker, and requires significantly more to generate and minimize the state space [25], thus the size of the complete state space being of great influence. Furthermore, the analysis performed by MRMC is a great deal more accurate, which is important for properties for which the probability is very close to either zero or one (see also Section VI for a discussion of rare-event simulation). Nevertheless, the results show that especially for model with large state spaces, simulation is a viable alternative if the loss of precision is acceptable.

V. Industrial Case Study

A case study was performed together with Airbus Defense and Space. The goal of the case study was to evaluate the analysis capabilities of the entire toolset. As such, it is not intended to reflect an actual design, but rather serves as an abstract example of a realistic design. In this paper, we present part of the case study involving the design of a launcher, an overview of which is shown in Figure 4.

a) Launchers: A launcher is generally short-lived, less than a few hours, with the purpose of delivering a payload such as a satellite into earth's orbit. It requires systems with high availability, as loss of control for a few milliseconds can mean loss of the entire launcher itself, as well as its payload.

As such, many systems run in warm or hot redundancy. Upon detection of a fault, the system immediately switches to a different component and/or disables the faulty component. This requires some mechanism that is capable of detecting a fault and performing the recovery operation. In the case study, the output signals of all the components are abstracted and encoded as Booleans, indicating whether or not a correct signal is available. Thus, by simply observing the value of the signal, the system can decide whether or not the output of a component is still correct.

Three types of faults can occur in the system: Transient, hot and permanent. Transient faults are those that correct themselves within a certain amount of time. Hot faults require intervention and can be corrected by resetting the system, e.g. by turning it off and back on again. Finally, permanent faults cannot be recovered from.

b) Launcher Components: The components of the launcher in the case study can roughly be divided into four groups: One group responsible for providing power to the other components; one group providing information about the location and trajectory for navigation (the sensors); one group processing the navigational data; and finally the thrusters (actuators) which are controlled by the processors.

The systems are connected via buses. These buses are used both to distribute power and transfer data signals. The case study models these buses as opaque objects which have no associated failure model. Rather, failures are modeled for the other components.

Power is distributed by means of multiple PCDUs (Power Conditioning and Distribution Units). Each PCDU is modeled as a battery and a number of power outputs, which can be connected to the various devices. The battery is modeled using continuous, linear dynamics to represent the amount of energy left. A battery is associated with a single, permanent failure mode. Upon failure of the battery, no power is delivered and the PCDU fails, including all its outputs.

Navigation is based on inputs from GPS and gyroscope (gyro) devices. A GPS is modeled as a simple device with two states: acquisition and active (see also Listing 1). A signal is only available in the active state. A GPS is modeled with three failure modes. Two of which, a transient and a hot failure, can be recovered from. The third is a permanent failure mode. Each failure mode is associated with a separate failure probability (as described in Listing 2).

A gyro is a device that can measure the orientation on two axes. Both axes are modeled as independent devices, each of which can independently fail. Furthermore, a dependent failure can cause the entire gyro to fail. In both cases, only a single, permanent failure mode is modeled.

The inputs from the GPS and gyro devices are processed by a Data Processing Unit (DPU), which in turn sends command signals to the rest of the equipment, including the thrusters. The DPUs are contained within two redundant On Board Computers (OBCs), which consists of a voting triplex of three DPUs running in hot redundancy. The output of the triplex is based on a majority vote of the DPUs. If any DPU fails, it will switch to a duplex mode. If a second DPU fails, the triplex itself fails, causing the OBC to fail.

c) Error Models: Two error models for the DPUs have been modeled, with two different behaviors for the OBCs. In the first case, the DPUs are associated with a permanent fault, and the OBCs simply vote on the output of the DPUs. In the second case, the DPUs are associated with a hot fault with a higher failure rate, which can be recovered from after a non-deterministic delay. The OBCs will switch off and on a DPU once after detecting a fault to attempt to clear the fault. However, due to the limited range of the interval, the recovery might fail if it is performed too early. The difference in failure

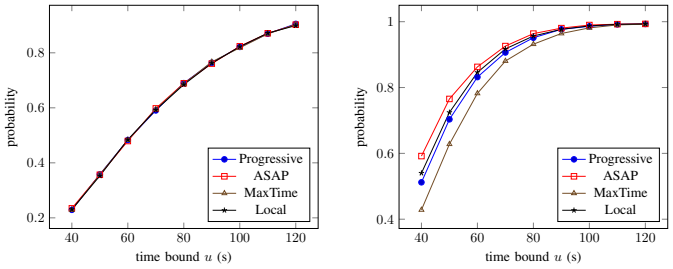

Fig. 5. Probabilities of system failure containing DPUs without (left) and with (right) repair.

rates has been applied to make the (lack of) effect of the strategies more clear.

To connect the error specifications with the components, fault injections have been specified which affect the power and control signals. In case of a fault (be it transient, hot or permanent), either power is disabled, or the signal set to false, indicating a failure.

The case study itself consists of approximately 800 lines of SLIM code, featuring 20 nominal and error component definitions and 37 component instances. Furthermore, 20 fault injections were specified. The case study and tool are accessible from [16].

d) Experimental Results: In order to evaluate the reliability of the system, probabilistic reachability properties were defined, evaluating the probability of a system failure occurring. The system is considered failed if control of the thrusters is lost, which happens if neither triplex can send a command to the thrusters because it has failed. This property has been specified using the probabilistic existence pattern, which can be translated into the CSL formula $\Pr(\diamond^{[0,u]} \mathit{failure})$, where $\mathit{failure}$ is an atomic proposition indicating failure, in this case both triplex commands being false and the system being in flight. Here, u controls the upper time-bound of the property.

Figure 5 shows the evaluation results for the two versions for the case study with permanent (left graph) and recoverable (right graph) faults for the DPUs (experiments were run with parameters $\delta = 0.9$ and $\epsilon = 0.005$). In the left graph, the results are the same for all possible strategies. This is due to the fact that the behavior of the model only contains probabilistic or deterministic transitions, thus time scheduling has no effect. In the right graph, the different strategies result in different behavior, due to the non-deterministic delay required for recovery. The ASAP strategy always schedules the repair too early, whereas the MaxTime strategy never does so. The Local and Progressive strategies are somewhere in between, randomly selecting delays before restarting a DPU. The Progressive strategy performs slightly better, as it makes it more likely for the DPU error model to preempt a too early recovery attempt.

VI. Related Work

Statistical model checking is an active field, and there is a fair amount of past and current research. On the practical side, various tools have been developed that make use of this technique as well. Tools such as YMER [21], (P)Vesta [26], [27], MRMC [24], and APMC [7] allow various Markov models to be analyzed by statistical means. More recently, and more closely related to our tool, the MODES [4] and UPPAAL-SMC [3] tools have been developed, which support real-time stochastic models, introducing support for underspecification of time. Aside from the input language, these tools differ from ours mainly on the semantics of the input language, as well as the possible scheduling approaches. A major difference between UPPAAL-SMC and our tool is that UPPAAL-SMC supports broadcast events only. For MODES, the main difference is that only the ASAP strategy is supported. The PLASMA-lab [5] and Prism [28] tools find their way in the middle, supporting non-deterministic, but not real-timed, models specified in the Prism Reactive Modules language.

An important aspect of statistical model checking lies in the scope of rare-events: occurrences of behavior that only occur with a very low probability. Whereas model checking using the full state space guarantees such events are detected, they are inherently unlikely to be found by regular statistical analysis. Various methods are introduced to support rare-event simulation for discrete systems [29], [30], [31], [32], which introduce a bias in the model to make such events more likely to occur, adjusting the final probability to take this into account.

One particular downside of using statistical model checking for non-deterministic models such as ours is that it is currently not possible to accurately determine the possible lower and upper bounds of the properties' probability. Various approaches are suggested to address this, such as reinforcement learning [33] or defining history-dependent schedulers [34].

Finally, an important aspect of the efficiency of the simulator is the number of paths that it requires to determine the final outcome. Various methods exist to find such bounds. The Chernoff-Hoeffding bound of our implementation is well known, but other approaches exist, both for quantitative and qualitative analysis. The work of [20] introduces various tests, and classifies them according to correctness, power and efficiency.

VII. Conclusion

This paper introduced our statistical model checker for the COMPASS toolset, supporting an AADL dialect input formalism (SLIM) with linear-hybrid and stochastic aspects. The main aim is to support performability analysis (that is, probabilistic dependability) for which no tools existed that supported the SLIM semantics.

The Monte Carlo method was chosen as it provides a tractable, yet powerful approach to support the combination of real-time and stochastic semantics. By allowing the implementation of various strategies, we address the issue of different possible interpretation of non-determinism which is inherently required for the path generation.

The tool has been integrated into the existing COMPASS toolset, allowing it to be used alongside the various other analyses that it supports, and integrating it into both a GUI and CLI based system.

A benchmark and an industrial case study of a launcher show that the use of Monte Carlo simulation is a viable approach. Although it shows that for smaller discrete model it can be outperformed by the existing tool-chain, for larger models, or timed models, simulation can be a better alternative.

A. Future Work

One point of improvement is more complete support for path generation strategies, giving the user more control to steer the simulation, for instance regarding non-deterministic transitions. This also includes controlling the scheduling order of transitions and the memory policies [18] being used. Automating or guiding the strategy selection would help improve usability, as this requires a less intricate knowledge of the methods.

Another item of interest is support for the full spectrum of CSL specifications [35], beyond the predefined patterns of the COMPASS toolset. This would include nested operators. The work of [21] shows that this has a fairly high complexity, but is manageable by using memoization techniques.

ACKNOWLEDGMENTS

The authors would like to thank Sebastian Junges and Jens Katelaan for their help in designing and building the simulator. This work was partially supported by ESA/ESTEC (contract no. 4000107221 (HASDEL)) and the EU (project reference 318490 (SENSATION)).

REFERENCES

[1] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.

[2] COMPASS Consortium, "COMPASS toolset web-site," http://compass.informatik.rwth-aachen.de, [Online; accessed 8-December-2014].

[3] P. Bulychev, A. David, K. G. Larsen, M. Mikučionis, D. Bøgsted Poulsen, A. Legay, and Z. Wang, "UPPAAL-SMC: Statistical Model Checking for Priced Timed Automata," in *QAPL*, ser. EPTCS, vol. 85. Open Publishing Association, 2012, pp. 1–16.

[4] J. Bogdoll, A. Hartmanns, and H. Hermanns, "Simulation and Statistical Model Checking for Modestly Nondeterministic Models," in *MMB/DFT*, ser. LNCS, vol. 7201. Springer, 2012, pp. 249–252.

[5] B. Boyer, K. Corre, A. Legay, and S. Sedwards, "PLASMA-lab: A Flexible, Distributable Statistical Model Checking Library," in *QEST*, ser. LNCS, vol. 8054. Springer, 2013, pp. 160–164.

[6] A. Legay, B. Delahaye, and S. Bensalem, "Statistical Model Checking: An Overview," in *RV*, ser. LNCS, vol. 6418. Springer, 2010, pp. 122–135.

[7] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet, "Approximate Probabilistic Model Checking," in *VMCAI*, ser. LNCS. Springer, 2004, vol. 2937, pp. 73–84.

[8] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri, "Safety, dependability and performance analysis of extended aadl models," *Comput. J.*, vol. 54, no. 5, pp. 754–775, 2011.

[9] M.-A. Esteve, J.-P. Katoen, V. Y. Nguyen, B. Postma, and Y. Yushtein, "Formal Correctness, Safety, Dependability, and Performance Analysis of a Satellite," in *ICSE*, ser. ICSE '12. IEEE Press, 2012, pp. 1022–1031.

[10] M. Bozzano, A. Cimatti, J.-P. Katoen, P. Katsaros, K. Mokos, V. Y. Nguyen, T. Noll, B. Postma, and M. Roveri, "Spacecraft early design validation using formal methods," *RESS*, vol. 132, no. 0, pp. 20–35, 2014.

[11] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *CAV*, ser. LNCS, vol. 2404. Springer, 2002, pp. 359–364.

[12] G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani, "Bounded model checking for timed systems," in *Formal Techniques for Networked and Distributed Sytems–FORTE 2002*, ser. LNCS, vol. 2529. Springer, 2002, pp. 243–259.

[13] H. Hermanns and J.-P. Katoen, "The how and why of interactive markov chains," in *FMCO*, ser. LNCS, F. S. de Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, Eds. Springer, 2010, vol. 6286, pp. 311–337.

[14] P. Bertoli, M. Bozzano, and A. Cimatti, "A Symbolic Model Checking Framework for Safety Analysis, Diagnosis, and Synthesis," in *Model Checking and Artificial Intelligence*, ser. LNCS, vol. 4428. Springer, 2007, pp. 1–18.

[15] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri, "The COMPASS approach: Correctness, modelling and performability of aerospace systems," in *Computer Safety, Reliability, and Security*, ser. LNCS, vol. 5775. Springer, 2009, pp. 173–186.

[16] H. Bruintjes, "Tool and case study download," http://compass.informatik.rwth-aachen.de/slimsim_dsn, [Online; accessed 8-December-2014].

[17] H. Bohnenkamp, P. R. D'Argenio, H. Hermanns, and J.-P. Katoen, "MODEST: A Compositional Modeling Formalism for Hard and Softly Timed Systems," *IEEE TSE*, vol. 32, no. 10, pp. 812–830, 2006.

[18] D. Bohlender, H. Bruintjes, S. Junges, J. Katelaan, V. Y. Nguyen, and T. Noll, "A Review of Statistical Model Checking Pitfalls on Real-Time Stochastic Models," in *ISoLA*, ser. LNCS, vol. 8803. Springer, 2014, pp. 177–192.

[19] K. L. McMillan, "The SMV language," Technical report, Cadence Berkeley Labs, Tech. Rep., 1999.

[20] D. Reijsbergen, P.-T. de Boer, W. Scheinhardt, and B. Haverkort, "On hypothesis testing for statistical model checking," *STTT*, pp. 1–19, 2014.

[21] H. L. S. Younes, "Ymer: A Statistical Model Checker," in *CAV*, ser. LNCS, vol. 3576. Springer, 2005, pp. 429–433.

[22] P. Bulychev, A. David, K. G. Larsen, A. Legay, and M. Mikučionis and Danny Bøgsted Poulsen, "Checking and distributing statistical model checking," in *NASA Formal Methods*, ser. LNCS, vol. 7226. Springer, 2012, pp. 449–463.

[23] R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, and B. Becker, "Sigref–a symbolic bisimulation tool box," in *ATVA*, ser. LNCS, vol. 4218. Springer, 2006, pp. 477–492.

[24] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen, "The Ins and Outs of The Probabilistic Model Checker MRMC," in *QEST*. IEEE, 2009, pp. 167–176.

[25] Y. Yushtein, M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, X. Olive, and M. Roveri, "System-software co-engineering: Dependability and safety perspective," in *SMC-IT*. IEEE, 2011, pp. 18–25.

[26] K. Sen, M. Viswanathan, and G. A. Agha, "VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems," in *QEST*. IEEE Computer Society, 2005, pp. 251–252.

[27] M. AlTurki and J. Meseguer, "PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool," in *CALCO*, ser. LNCS, vol. 6859. Springer, 2011, pp. 386–392.

[28] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-time Systems," in *CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 585–591.

[29] G. Rubino and B. Tuffin, *Rare Event Simulation Using Monte Carlo Methods*. Wiley Publishing, 2009.

[30] D. Reijsbergen, P.-T. de Boer, W. Scheinhardt, and B. Haverkort, "Rare event simulation for highly dependable systems with fast repairs ," *Performance Evaluation*, vol. 69, no. 78, pp. 336 – 355, 2012.

[31] D. Reijsbergen, P.-T. de Boer, B. Haverkort, and W. Scheinhardt, "Automated Rare Event Simulation for Stochastic Petri Nets," in *QEST*, ser. LNCS, vol. 8054. Springer, 2013, pp. 372–388.

[32] C. Jégourel and Axel Legay and Sean Sedwards, "An Effective Heuristic for Adaptive Importance Splitting in Statistical Model Checking," in *ISoLA*, ser. LNCS, vol. 8803. Springer, 2014, pp. 143–159.

[33] D. Henriques, J. Martins, P. Zuliani, A. Platzer, and E. M. Clarke, "Statistical Model Checking for Markov Decision Processes," in *QEST*. IEEE, 2012, pp. 84–93.

[34] P. D'Argenio, A. Legay, S. Sedwards, and L.-M. Traonouez, "Smart Sampling for Lightweight Verification of Markov Decision Processes," *CoRR*, vol. abs/1409.2116, 2014. [Online]. Available: http://arxiv.org/abs/1409.2116

[35] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, "Model Checking Algorithms for Continuous-Time Markov Chains," *IEEE TSE*, vol. 29, no. 6, pp. 524–541, 2003.