A Model-Based End-to-End Toolchain for the Probabilistic Analysis of Complex Systems

Alessandro Pinto
United Technologies Research Center Inc.
Berkeley, CA
pintoa@utrc.utc.com

Sudha Krishnamurthy
United Technologies Research Center
East Hartford, CT
krishnadots@gmail.com

Suresh Kannan
Dept. of Aerospace Engineering
Georgia Institute of Technology
Atlanta, GA, suresh.kannan@aerospace.gatech.edu

***Abstract*— We present a model-based environment for the probabilistic analysis of systems operating under uncertain conditions. This uncertainty may result from either the environments in which they operate or the platforms on which they execute. Available probabilistic analysis methods require to capture the system specification using languages that are semantically very close to Markov Chains. However, designers use model-based environments working at much higher abstraction levels. We present an integrated tool, called StoNES (Stochastic analysis of Networked Embedded Systems), that automates the model transformation and probabilistic analysis of systems. We apply our translation and analysis methodology to explore the trade-off between sensor accuracy and computational speed for the vision algorithm of an autonomous helicopter system.**

I. Introduction

Model-based design is an important paradigm in the development of safety-critical embedded systems. The main principle of the paradigm is to use models all along the development cycle, from design to implementation. Model-based design enables the use of tools for analysis, simulation, verification, synthesis, and code generation. Our objective is the analysis of embedded systems that operate in uncertain environments. This uncertainty may arise from different factors, such as the environmental conditions and the performance of the platforms on which they execute. Consider, for example, an autonomous helicopter. Its dynamics is affected by wind that is uncertain; the estimate of its position is affected by sensor inaccuracies; the execution times of the image processing algorithms used in autonomous missions are data dependent, and since data values are not known a priori, the execution time is uncertain; and finally, hardware components may fail with a certain probability. It is essential for designers to be able to assess the performance of the entire system early in the design. The interesting question to be answered is the following: what is the probability that a mission can be accomplished autonomously, under such uncertain conditions? For such systems, a design-and-test approach is inadequate, because it is not possible to run tests in any randomly selected scenario.

Ideally, a designer would capture the system in a high level language, by including all sources of uncertainty, and would then rely on a push-button solution to explore the impact of design choices on the mission success probability. Matlab Simulink and Stateflow (MSS) are often the languages of choice, especially in many safety-critical application domains

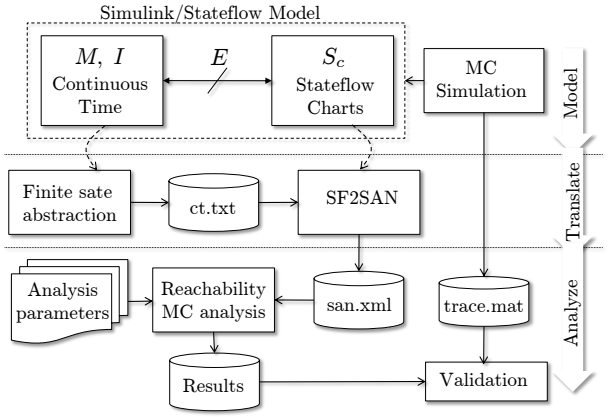

Fig. 1. Overview of the StoNES front-end tool.

such as avionics and automotive. However, while the MSS models are primarily used for simulation, there is little support for formal verification and virtually no support for stochastic analysis. One possibility would be to use Monte Carlo simulations. However, the complexity of this type of analysis depends on the number of uncertain parameters in the system and on their distributions. On the other hand, when the models can be reduced to Markov Chain (MC) based formalisms, such as Stochastic Petri Nets (SPN) [1] or Stochastic Automata Networks (SAN) [2], analytical methods exist to compute the probability distribution over the states of the underlying MC. While the complexity of the analytical methods depends on the number of states of the MC and suffer from the state explosion problem, the advantage is that the result covers all possible behaviors of the system, and is exact within the assumptions of the model. Manual development of probabilistic models requires a considerable amount of time and is also error prone. Thus, there is need for a methodology and a companion tool that can bridge this gap by automatically generating models that are amenable to probabilistic analysis starting from a high-level specification of real systems, and providing a flexible back-end tool for analysis and design space exploration.

We have developed a tool-chain, called StoNES, with the objective of automating the stochastic analysis and design of networked embedded systems, starting from their high-level specification. StoNES has three key modules: a func-

tional translator that generates a probabilistic description from a high-level functional model of the system given in Simulink/Stateflow; an architectural translator that translates the hardware architecture of the system given in AADL [3] into a probabilistic model; and a mapper that generates a probabilistic model of the system by integrating the functional and architectural models for a given mapping of the functional units on architectural components. The probabilistic model generated by the front-end is used for analysis by the back-end. In this paper, we primarily focus on the translation and analysis of the functional model (Figure I). We illustrate the use of StoNES through a case study that consists of the probabilistic analysis of the success of an autonomous helicopter tasked with the mission of searching for and identifying a specific symbolic pattern within a group of buildings. We also validate our tool by showing that the results from analyzing the high-level functional model using Monte-Carlo simulation are consistent with the results obtained from analyzing the translated SAN model using probabilistic techniques.

The remainder of the paper is organized as follows. In Section II, we describe the methodology, including the input and output models of the translation process. In Section III, we describe the translation algorithms. In Section IV, we illustrate the use of our methodology on an autonomous helicopter mission and present results that show the equivalence between the probabilistic model output by the translator and the original functional model. We review related work in Section V and then present our conclusions in Section VI.

II. The StoNES methodology

The functional analysis flow implemented in StoNES is shown in Figure I. The functional specification of a system is input to the StoNES framework in the form of a Simulink diagram that contains a model of the plant to be controlled (e.g. the set of differential equations describing the flight dynamics of a helicopter), sensors and actuators (*Continuous Time* Figure I), and state machines implementing high level control strategies. The state machines are captured using *Stateflow charts*. Thus, the specification is in general, a hybrid system. The translation process generates a SAN model, which is then input to the analysis engine. Let E be the set of Simulink signals, I the set of integrator variables (i.e. the continuous states of the system), and S_c be the combined state of all Stateflow charts. We consider two sources of uncertainty: 1) a set of parameters P of the Simulink blocks can be considered random variables, and 2) some of the signals $M \subseteq E$ can be driven by Markovian processes.

A. Inputs to the SF2SAN translator

The continuous time part of the MSS model may be used, among other purposes, to capture the environment in which the control system operates. In our methodology this part of the model is abstracted into a finite state system that can be analyzed using our analysis engine. The finite abstraction can be built using different methods. One abstraction technique that is also amenable to automation consists in discretizing the variables in I (we assume that the set of signal $E_{cs} \subseteq E$ directed from the Simulink blocks to the Stateflow charts are directly dependent on the variables in I). Each discrete set of values is associated with one state of a finite state system. For each possible value of the set of signals $E_{sc} \subseteq E$ directed from the Stateflow charts to the Simulink blocks, we compute the probability of transitioning to other states using Monte Carlo simulations (by sampling the parameters P). Thus, the values of the signals in E_{sc} are translated into a guard condition for the transition and the probabilities computed using Monte Carlo simulations are used to compute the transition rates. The result of this abstraction are saved in a format that is compatible with the Stateflow syntax. Each discrete state is assigned a name and a set of state actions that assign the values to the variables in E_{cs} corresponding to the discrete values of I in that state. Each transition is associated with a guard condition expressed using the same syntax of a guard condition in a Stateflow chart. The only addition is a rate parameter that is the rate of an exponential distribution representing the time spent in the source state. This file can be easily parsed by the translator that can treat this model as an additional Stateflow chart in the translation process. For this reason, we will only describe the concrete syntax [4], [5] and the translation of Stateflow charts. We assume the reader if familiar with the syntax and semantics of Sataflow. Let $E[C]\alpha_c/\alpha_t$ the the label of a transition where E is a trigger event, C is a guard condition , α_c is the condition action and α_t is the transition action. α_c and α_t. We do not translate function calls or history junctions. The state action $\alpha(s)$ of a state s is a entry action and it is restricted to simple assignments of outputs to constant values. Also, we do not support condition actions in this first implementation of the translator. Finally, we assume that the input stateflow model has a finite number of states[6].

B. Output of the SF2SAN translator

A Stochastic Automata Network (SAN) $N = (\{A^{(i)}\}_n, E, \Rightarrow)$ is a collection of Stochastic Automata, an alphabet of events E, and a relation among events $\Rightarrow \subset E \times E$. A Stochastic Automaton is a tuple $A^{(i)} = (S^{(i)}, T^{(i)}, L^{(i)}, G^{(i)})$ where $S^{(i)}$ is the set of states, $T^{(i)} \subseteq S^{(i)} \times S^{(i)}$ is the set of transitions, $L : T \rightarrow 2^E$ is a labelling function that associates a set of events to each transition, $G : T \rightarrow 2^{S \times \mathbb{R}_+}$ is a relation that associates with each transition a set of guard-rate pairs, and $S = S^{(1)} \times \ldots \times S^{(n)}$.

In the StoNES toolkit, we represent the SAN definition in XML format. In our XML schema, an automaton contains a set of states and a set of transitions, and it is characterized by an initial state and a name. SAN states and SAN transitions have a name and a numeric identifier as attributes. In addition, a transition may have events and one or more guards as its children. A guard is characterized by a name as well as a rate, and has a guard expression as a child node. In the case of a continuous time model, the rate denotes the mean value of an exponential distribution representing the transition time, whereas in case of a discrete time model, it

denotes the probability of making the transition. Associating a rate with the guard allows to capture state dependent rates. We use a Boolean expression to represent the set of states of a guard.

A formal definition of the semantics of the model requires some additional notation, therefore we give an intuitive definition. A stochastic automata network in state $s_i = (s_i^{(1)}, \ldots, s_i^{(n)})$ can transition to state $s_j = (s_j^{(1)}, \ldots, s_j^{(n)})$ if and only if $(s_i^{(k)}, s_i^{(k)}) \in T^{(k)}$ [1], all guard conditions associated with the transitions are satisfied and all event synchronizations are satisfied. There are two types of event synchronizations: if two transitions t_1 and t_2 are labeled with the same event, then they must occur concurrently. Second, if t_i is labeled with event e_i, $i = 1,2$, and e_1 is synchronized with e_2 (denoted as $e_1 \Rightarrow e_2$), then if t_1 is executed and t_2 is enabled, then t_2 must also be executed. A transition $t(s_i^{(k)}, s_j^{(k)})$ is enabled if the automaton $A^{(k)}$ is in state $s_i^{(k)}$ and guard $G^{(k)}(t)$ is true.

A simulation trace of a MSS model is the timed trace generated by the Simulink simulator for one realization of the set of processes driving the signals in M and one value of the set of parameters P. The state space S of the SAN model generated by the SF2SAN translator is partitioned as $S = S_I \cup S_M \cup S_f$, where S_I is a finite abstraction of the continuous states I, S_M is the state space of the Markovian processes and S_f is a representation of the set S_c. Assume the specificaiton only contains a set of Stateflow charts, and that a formal description of its semantics was availalbe. Given the semantics of the SAN model, it would be possible to prove that our translation is correct (meaning that the set of executions of the SAN model contains the set of executions of the Stateflow model). However, there is no formal semantics for MSS models released by the Mathworks. Further, the discretization technique used for the Simulink part is only approximate. Thus, we rely on probabilistic conformance testing, We measure the approximation error $|P(s_c(t) = s) - P(s_f(t) = s)|$ for all states $s \in S_c$. We compute $P(s_c(t) = s)$ by using a Monte Carlo method, while we compute $P(s_f(t) = s)$ using our analysis engine.

C. Analysis of a SAN model

The SAN obtained as output of the translation process is the input to the analysis engine. We now briefly describe the steps involved in analyzing a SAN.

The first step of the analysis is the computation of the set of reachable states of the model. We use symbolic reachability analysis [7] that has been implemented using the CUDD package [8], [9]. We encode the states, the transitions and the guard conditions enabled for each transition as binary variables. Since SAN support complex synchronization relations among events, the transition function is partitioned into the transition set generated by the free transitions (i.e. the ones that are not subject to any synchronization constraints), and the set generated by the synchronization relation. As a result, StoNES generates a reachability graph representing the underlying Markov Chain (MC), where transitions are labeled with symbols. The symbols can be numbers if transition rates are already known at translation time, or parameters that can be substituted with numbers during transient analysis. For example, the transition rates of a Stateflow chart are not known to the translator. Thus, the translator uses a symbol to denote the rate. This rate is either assigned later by the user using a parameter file, or it is inherited during mapping of the chart on computational resources. When a chart is executed on a processor, the execution time of the chart on the processor becomes the transitions rate.

The transient analysis step solves the underlying MC by first substituting the parameter values given as input by the user. After the substitution step, we obtain the classical Kolmogorov equation $\dot{\pi} = Q^T \pi$ where $Q(i,j)$ is the transition rate from state i to state j of the MC. This equation can be solved using numerical integration or the uniformization method [10]. The user can define filters to project the result of the integration along particular *views*. A view is a linear projection of the state space of the MC and it can be defined in terms of the state of the original MSS model in a separate file.

III. SF2SAN Translator

The SF2SAN translator generates a SAN model from the input MSS model of the system. The concrete output of the translator is represented using XML A SAN `system` node is the topmost object in the SAN hierarchy, representing the root of the document tree. Each chart in the MSS model is then appended to the system as a new automaton. Connections between charts are then represented by guard conditions and event synchronization statements.

Algorithm 1: transformParallelStates

Input: StateflowState s (AND state)
1 $S^{(1)}, \ldots, S^{(n)}$ sets of states of the children of s;
2 Define the new set of children of s to be $S = S^{(1)} \times \ldots \times S^{(n)}$;
3 Set the initial state to be $(s_0^{(1)}, \ldots, s_0^{(n)})$ where $s_0^{(i)}$ is the initial state of the i-th child ;
4 **foreach** $s_1 = (s_1^{(1)}, \ldots, s_1^{(n)}) \in S$ **do**
5 $\alpha(s_1) = \alpha(s_1^{(1)}) \cup \ldots \cup \alpha(s_1^{(n)})$;
6 **end**
7 **foreach** $s_1 = (s_1^{(1)}, \ldots, s_1^{(n)}), s_2 = (s_2^{(1)}, \ldots, s_2^{(n)}) \in S$ **do**
8 **if** $\forall\ k = 1 \ldots n$, $\exists$ *a transition between* $s_1^{(k)}$ *and* $s_2^{(k)}$ **then**
9 add transition (s_1, s_2);
10 guard $G(s_1, s_2) = G(s_1^{(1)}, s_2^{(1)}) \wedge \ldots \wedge G(s_1^{(n)}, s_2^{(n)})$;
11 **end**
12 **end**

The SAN syntax does not support hierarchy. If a chart contains AND (parallel) states, the translator uses Algorithm 1

[1] Depending on the model of concurrency, we may have also self transitions that are always enabled, i.e. $s_i^{(k)} = s_i^{(k)}$).

to reduce an AND state to a hierarchical state containing only OR states. The algorithm computes the synchronous product of the state machines that are children of the AND state s. This algorithm is applied recursively until no more AND states exist. The result is a Stateflow chart that contains only OR states, but that is still hierarchical. To remove a hierarchical state s, each incoming transition is connected to the initial state of the contained state diagram, and each outgoing transition is replicated from each child state to the destination state. The result of this second transformation is a flat Stateflow chart consisting only of OR states.

Translation of states. Each state of the flat Stateflow chart is translated into one state of the SAN automaton representing the chart. The state actions are parsed and stored in a data structure, `ActionList` to be used during the translation of transitions. Also, a map m_s is constructed so that for a Stateflow state s, $m_s(s)$ is the corresponding SAN state.

Translation of Transitions. An outgoing transition of a state in a Stateflow chart is translated to a SAN transition and added to the automaton being translated. Events and guard conditions from the Stateflow transition are added to the SAN transition element. Algorithm 2 shows the translation steps. First, the Stateflow transition is mapped back to the corresponding pair of states of the current automaton A. The new transition is added to the set of transitions $A.T$ of automaton A. Then, the transition label is parsed to obtain the event ev, guard expression C, transition action α_t, and rate λ of the transition. The rate λ is a number if it is explicitly specified in the label or, as in the case of Stateflow charts, is a symbolic parameter that will have to be assigned later during the analysis step.

Algorithm 2: transformTransition

Input: Stateflow state s, current automaton A

foreach $t(s,s')$ *outgoing* s **do**
- $t_a \leftarrow (m_s(s), m_s(s'))$;
- $A.T \leftarrow A.T \cup \{t_a\}$;
- $(ev, C, \alpha_t, \lambda) \leftarrow$ parseLabel(t) ;
- $A.L(t_a) \leftarrow A.L(t_a) \cup ev \cup \alpha_t$;
- add ev and α_t to the global list of events ;
- $exp \leftarrow$ Exp(C);
- $A.G(t_a) \leftarrow A.G(t_a) \cup \{(exp, \lambda)\}$

end

An event that triggers a transition in a Stateflow chart is mapped to a SAN event and associated with the transition. The event is also added to a global list of events, which is used to generate the pairs of events that are synchronized in the SAN, as explained later in this section.

The transition action, if specified in the transition label, signifies actions that are triggered after the transition is executed. We currently assume that all transition actions signify event broadcasts. Hence, an event that is triggered as a result of a transition action in a Stateflow chart is also translated to an event in the SAN, associated with the transition and added to the global list of transaction actions, which is used to generate the pairs of events that are synchronized.

Translation of Condition Expressions. The SF2SAN translator maps a Stateflow condition expression C to a SAN `guard` element G using Algorithm 3. If the transition label in the Stateflow model does not have a condition expression, the corresponding guard of the SAN transition is a simple TRUE expression.

Algorithm 3: Exp

Input: Stateflow guard condition C

if C *is a simple expression* **then**
- $S \leftarrow$ findGuardStates(C);
- **return** createSANExprElement(S);

end

if $C = C_1 bop_1 \ldots bop_{n-1} C_n$ **then**
- **return** Exp(C_1) $bop_1 \ldots bop_{n-1}$ Exp(C_n) ;

end

If C is a simple expression, i.e. $C = [x\ op\ a]$, where x is a variable, a is a constant value, and op is a relational operator (i.e. $op \in \{\geq, \leq, ==, >, <, !=\}$), then Algorithm 3 calls `findGuardStates`, which traverses the list of state actions, `Actionlist`, to find the states in which C_1 is true. For example, if $C = [x == 9]$, then `findGuardStates` finds all the states in the which the variable x has a value of 9. This is obtained by traversing the global list of state actions that is populated when each state is created (as explained earlier in this section). Note that variable x may be an input to the current charts coming from a different automaton. Thus, `findGuardStates` first obtains the mapping of variable x to the variables in each automaton, using the input-output connections in the MSS model. Then it determines if there are any states in that automaton in which the condition C is true. If $A^{(1)}.s^{(1)}$, ..., $A^{(k)}.s^{(k)}$ are the states in which C is true ($A^{(i)}$ being the automaton, and $s^{(i)}$ the state of automaton $A^{(i)}$), then the translator translates C to $G = [A^{(1)}.s^{(1)} \parallel \ldots \parallel A^{(k)}.s^{(k)}]$. G is an expression tree in which the states form the leaf nodes and the operators form the internal nodes.

If C is not a simple expression, then it is a compound expression of the form $C = [C_1 bop_1 \ldots bop_{n-1} C_n]$, where each C_i is a sub-expression, and bop_i is a Boolean operator. Algorithm 3 recursively translates each sub-expression, as explained above, and computes the guard using the Boolean operators bop_i.

Event Synchronization. To determine the synchronization relation $\Rightarrow$, the translator traverses the global list of transition actions after all of the charts of the model have been processed. For each event e_1 in this list, it checks if there is a matching event e_2 in the global event list. Both of these global lists are populated when transitions are translated. The matching condition is satisfied if the broadcast of event e_1 triggers event e_2, where e_1 and e_2 may be events in different automata. In such a case, $e_1 \Rightarrow e_2$, and the translator adds the pair (e_1, e_2) to the relation $\Rightarrow$.

IV. Case Study - Autonomous Helicopter Application

Consider an autonomous helicopter which is assigned the mission of finding a building marked with a special symbol

in a urban area. In the early design stages of this system, we need to explore the trade-off between sensor accuracy and computation complexity in order to identify the building with the required probability of success, as specified by the mission. To understand this trade-off, consider the helicopter flying around a building marked with a special symbol. Since the vision algorithm used to match the symbol against a known pattern is sensitive to scaling, the position estimation error (caused by the finite accuracy of the GPS and other sensors) has two effects: 1) the symbol may be missed even if it is present in the current frame (false negative), and 2) image features that are similar to the symbol may result in a good matching (false positive). On the other hand, if several frames are processed per second, the likelihood of discarding similar symbols while retaining the real one is very high (at the expense of carrying more weight on-board).

The model of the helicopter mission is shown in Figure 2 and has two parts. The trajectory followed by the helicopter is computed by a trajectory generation algorithm for given way-points around the building (Figure 2a)). The estimated position is obtained by adding colored noise to the real trajectory. We add colored noise because the white noise from the sensor is filtered before being used by the vision algorithm. Several objects are placed in the scene, but only one of them corresponds to the symbol to be found. A camera model is used to generate a Boolean flag that is equal to TRUE if the object is in the field of view of the camera and FALSE otherwise (`Field of view model`). The vision algorithm is a Stateflow model (Figure 2b)) that maintains a matching score for each of the objects in the scene (the object being parallel states). If an object is in the field of view, then its score is increased or decreased depending on the error in the position estimate. In particular, if the error is low, then the score associated with the symbol increases, while the score associated with the other objects decreases. If the error is high, the opposite occurs. We used three levels for the score: good, average, and bad.

The system is translated into a SAN, where the variance of the colored noise is one of the parameters. Similarly, the transition rate associated with the Stateflow transitions is a parameter (as explained in Section III). These two parameters represent the accuracy of the sensors and the speed of execution (i.e. frames per second) of the vision algorithm, respectively.

Figure 3 shows the results of the analysis. Each row corresponds to a different error level while each column corresponds to a different rate of execution of the vision algorithm (fps). In the first row, we observe that when the processing speed is 10 fps, even if the object is in view for only a short period of time, the probability of distinguishing object 0 (i.e. high difference between probabilities of being in the good rather than bad state) is very high. For a processing speed of 0.5 fps, it is essential to have the object in view for a longer time. In particular, if object 0 would have been in view only for the first 10 seconds, it would have been discarded. The situation is different when the error is large (last row). In this case, increasing the controller speed at which the vision algorithm is computed does not help. From the results, we infer that a matching probability of 0.4 is a limit for our input mission profile.

Object	State	Monte Carlo	StoNES
0	good	0.7908	0.8485
	avg	0.2087	0.1506
	bad	0.0005	0.0009
1	good	0.0005	0.0009
	avg	0.2168	0.1577
	bad	0.7827	0.8414
2	good	0.0004	0.0009
	avg	0.2092	0.1517
	bad	0.7904	0.8474
3	good	0.0005	0.0009
	avg	0.2172	0.1570
	bad	0.7824	0.8420

TABLE I

COMPARISON OF RESULTS OBTAINED THROUGH MONTE CARLO SIMULATIONS AND STONES FOR $\sigma = 0.21$, AND COMPUTATIONAL SPEED EQUAL TO 10 FPS (FOR THE STONES ANALYSIS)

We notice that there is a trade-off between the accuracy of the sensors and the computational complexity needed to execute the vision algorithm for a given probability of success. In fact, if $\sigma = 0.21$ and the object stays in view for more than 40 *seconds*, then a computation rate of 0.5 fps would be sufficient to detect the symbol with high probability. This means that the amount of hardware on board (and the dissipated power) are limited in this case. However, the sensor may be very costly. For $\sigma = 0.84$, the sensor inaccuracy is very high and so the probability of success would be very low. However, if a rate of 10 fps can be achieved, then the results show that it would even be feasible to use sensors with inaccuracy $\sigma = 0.42$, which may be less expensive than highly accurate sensors.

Further, the results suggest that the response from the vision algorithm could be used to regulate the speed at which the helicopter flies, so that the same symbol remains in the field of view for a higher number of frames. For instance, when the difference between good and bad matching probabilities is not high enough (say > 0.3), the helicopter may slow down or even back up to take other frames from the same field of view. However, if the mission must be accomplished in a given time, it may not be possible to always reduce the speed of the helicopter.

Table I shows the validation result obtained by running Monte Carlo simulations with over 25000 samples in Simulink. We report the steady state probabilities obtained by Monte Carlo simulations and by StoNES. In the simulations, the transitions of the Stateflow model are instantaneous, while in the SAN mode they are associated with a rate to take into account architectural information. Thus, to compare the analysis results, we set the frame rate to 10 fps, which is high compared to the discretization step selected for the continuous time model. For higher values of the probabilities, the error is within 8%. However, the error is higher for low probabilities. This is due to two factors: the accuracy of the results obtained by Monte Carlo simulations, and the

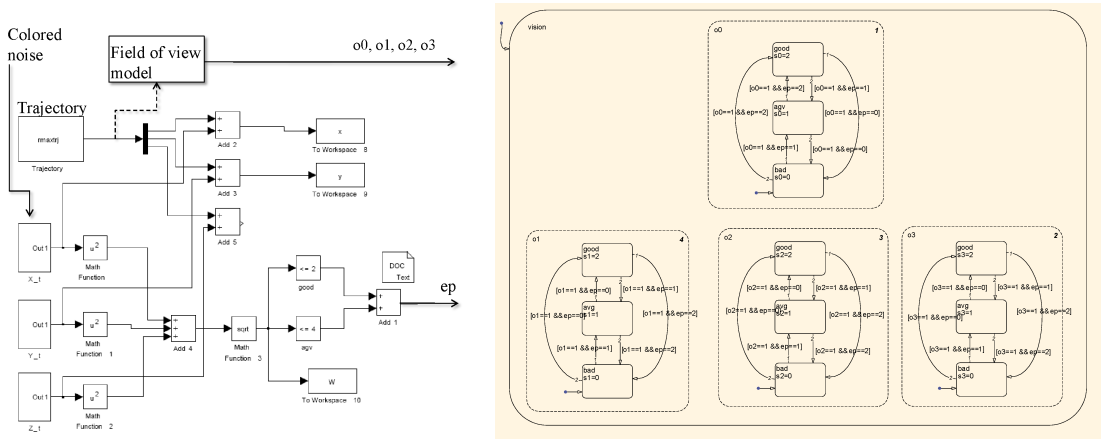

Fig. 2. Simulink/Stateflow model of the autonomous mission.

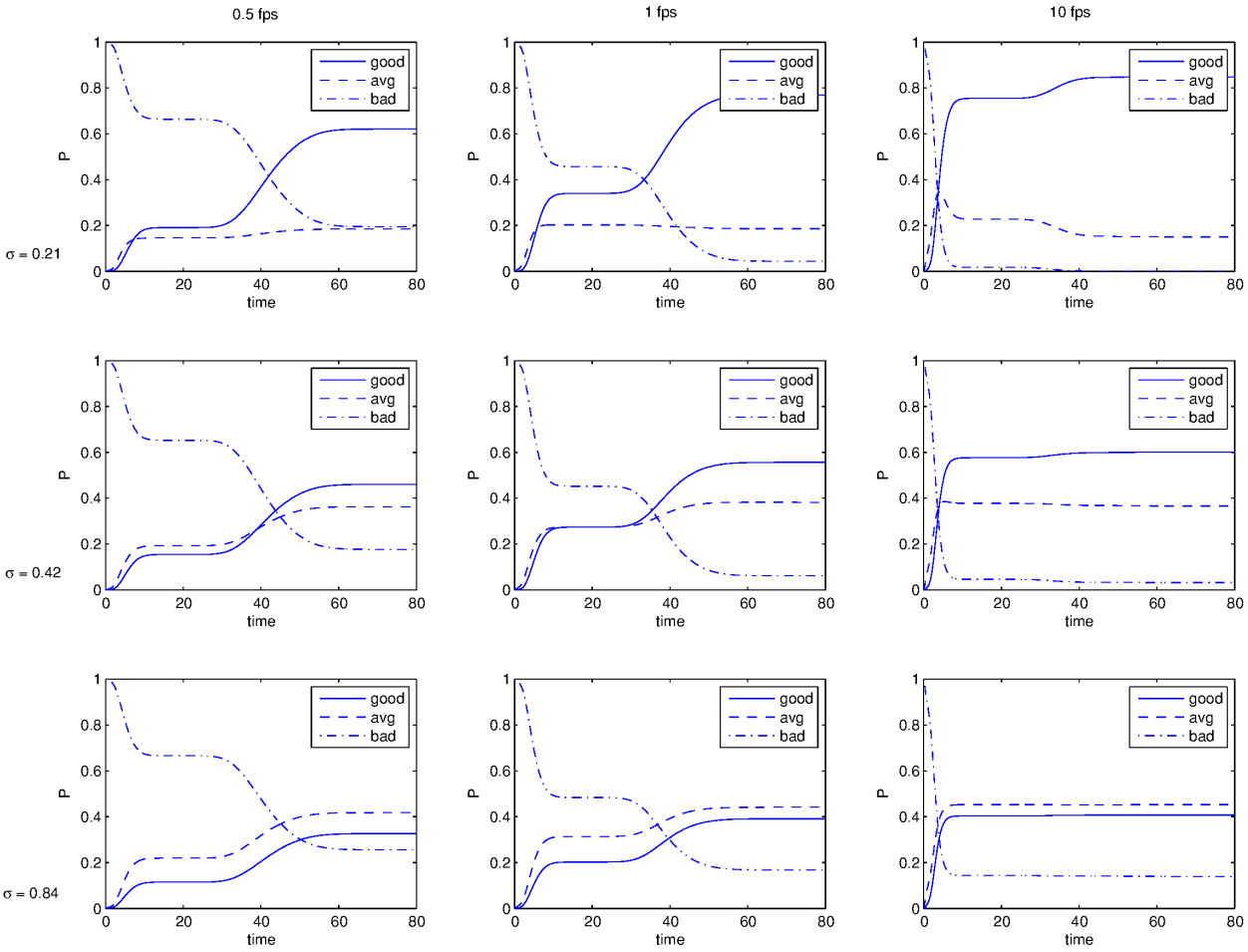

Fig. 3. Probability of a bad, average, and good matching of the symbol for different values of sensor accuracy and computational speed.

approximation of the continuous time part of the MSS model.

V. Related Work

In this section, we present some of the related work in the area of model-to-model translation and probabilistic analysis. Semantic mapping between different design formalisms is a common problem one often encounters in the design of embedded systems and tools have been developed to perform these mappings in an automated manner. In [11], the authors present a semantic translator that makes use of graph transformations to transform models expressed in the Matlab Simulink and Stateflow (MSS) language into equivalent models in Hybrid System Interchange Format (HSIF), which is an XML based standard for representing dynamic networks of hybrid automata. The hybrid automata formalism is then used for model-checking. Instead of mapping to the hybrid automata formalism, the *ss2lus* tool-chain translates Simulink/Stateflow models to the synchronous dataflow programming language Lustre [12], [13]. Compilers automatically generate code for different platforms from the Lustre models and thereby automate the implementation of embedded controllers.

In [14] the authors present a translator that maps an input Simulink model to an equivalent NuSMV model, which is an open source symbolic model checker. Model-checking is also the goal of translation in the HiVy tool-set [15], which maps each input statechart to an equivalent hierarchical sequential automaton (HSA). A HSA consists of a finite set of cooperating sequential automata. HiVy implements a HSA as parallel processes in Promela, which is the input language of the SPIN model checker. Thus, the input statechart, translated into the HSA formalism, is then used as a basis for model-checking and automatic code generation. While the primary goal of a majority of the tool-chains described above is model-checking and automatic code generation, the goal of the translation in the StoNES toolkit is to produce a semantically equivalent probabilistic model that is amenable to stochastic analysis, from the functional and architectural description of a system.

The two main approaches for the probabilistic analysis of systems that are relevant to StoNES are transient analysis [1], which aims mainly at evaluating the performance of a system, and probabilistic model checking [16], which aims at checking that a CTMC, DTMC or MDP model satisfies a formula expressed in the PCTL logic (and also relies on transient analysis). StoNES leverages many of the advancements in these fields to provide a flexible engine for design space exploration.

VI. Conclusions and Future Work

We presented a model-based toolchain for the probabilistic analysis of systems that operate under uncertain conditions. Given an input specification in a high level language such as Simulink/Stateflow, the StoNES toolchain automatically translates the specification into a (parametric) Stochastic Automata Network (SAN). This model is then analyzed in several steps that include reachability analysis, transient analysis, and presentation of data. We showed how the tool can be used for design space exploration and probabilistic analysis by exploring the tradeoff between sensor accuracy and computational needs for the mission of an autonomous helicopter. The validation shows that the results obtained from model-based probabilistic analysis and Monte Carlo simulations are consistent, which in turn establishes the semantic equivalence between the SAN model generated by the StoNES translator and the input Simulink/Stateflow model.

We plan to extend our work in multiple directions. The SAN language has limited expressiveness that makes the translation process complex. We plan to introduce a new model that is more abstract than SAN, so as to balance the effort between the translator and the model encoding into Binary Design Diagrams done by the analysis engine. We plan to extend our analysis capabilities to avoid discretizing the continuous time part of the system, because the discretization algorithm has an exponential complexity in the number of continuous variables.

References

[1] D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and G. Conte, *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., 1994.

[2] B. Plateau and K. Atif, "Stochastic Automata Network of Modeling Parallel Systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 1093–1108, 1991.

[3] S. Aerospace, *Architecture Analysis and Design Language (AADL)*, SAE, January 2009.

[4] Mathworks, "Simulink." [Online]. Available: http://www.mathworks.com/academia/student_center/tutorials/simulink-launchpad.html

[5] ——, "Stateflow." [Online]. Available: http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/

[6] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi, "Defining and translating a "safe" subset of simulink/stateflow into lustre," in *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*. New York, NY, USA: ACM, 2004, pp. 259–268.

[7] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 13, no. 4, pp. 401–424, 1994.

[8] "http://vlsi.colorado.edu/ fabio/cudd/."

[9] F. Somenzi, "Cudd: Cu decision diagram package release 2.2.0," 1998.

[10] G. Bolch, S. Greiner, H. d. Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains*. Wiley-Interscience, 2005.

[11] A. Agrawal, G. Simon, and G. Karsai, "Semantic Translation of Simulink/Stateflow models to Hybrid Automata Using Graph Transformations," *Electronic Notes in Theoretical Computer Science*, vol. 109, pp. 43–56, 2004.

[12] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi, " Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre," Verimag Technical Report, Tech. Rep. TR-2004-16, 2004, this is the full version of the paper accepted by EMSOFT04.

[13] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, "Translating Discrete-Time Simulink to Lustre," *ACM Transactions on Embedded Computing Systems*, vol. 4, no. 4, pp. 779–818, 2005.

[14] B. Meenakshi, A. Bhatnagar, and S. Roy, "Tool for Translating Simulink Models into input Language of a Model Checker," *Lecture Notes in Computer Science*, 2006.

[15] P. Pingree and E. Mikk, "The HiVy Tool Set," *Lecture Notes in Computer Science*, pp. 466–469, 2004.

[16] D. A. Parker, "Implementation of symbolic model checking for probabilistic systems," Tech. Rep., 2002.