Developing design tools for uncertain systems in an industrial setting

Conference Paper · November 2010

DOI: 10.1109/ALLERTON.2010.5707123 · Source: IEEE Xplore

2 authors, including:

Alessandro Pinto
United Technologies Research Center
62 PUBLICATIONS **1,286** CITATIONS

SEE PROFILE

Developing design tools for uncertain systems in an industrial setting

Alessandro Pinto, Sudha Krishnamurthy

Abstract— **We motivate the need for a cyber-physical system analysis and design tool that embraces uncertainty as key characteristic of these type of systems. We outline the features that such tool should provide and we present a prototype implementation. The challenges faced during its development go beyond the sheer complexity of analyzing large Markov Models. We close the paper with some examples of analysis of uncertain systems.**

I. Introduction

There are several reasons why design paradigms for cyber-physical systems should include the notion of uncertainty. The physical side of the system (i.e. the plant that the cyber side controls) is only known to some extent. For example, the weather condition that an aircraft will face during a mission is a random parameter, and it becomes a stochastic process for long distance missions. Even when the dynamic behavior of the environment is not subject to any randomness, the parameters of its model may not be known exactly either because difficult to measure, difficult to control during the manufacturing process, or simply because of an abstraction process required to limit the complexity of an accurate model.

The cyber side of the system comprises the control software, the hardware and the communication network, and is subject to random failures and data dependent performance metrics. Optimal control strategies rely on the solution of optimization problems whose run-time depends on the input data. Moreover, the worst case execution time of software is data dependent because of low level implementation techniques such as cache memories, branch prediction, pipeline execution etc. Notoriously, communication delays are also uncertain, especially when collision-based and wireless protocols are used.

Many techniques have been developed to analyze systems in the presence of uncertainty. For instance, fault tree analysis [1] is used to compute the probability that an event occurs starting from the probabilities (assumed known) that basic events occur. Markov Chain based methods [2] have also been used for reliability and performance analysis. Uncertainty quantification is well studied for dynamical systems (see e.g. [3], [4]). Lately, model checking techniques have been extended to probabilistic systems, so that Markov Chains or Markov Decision Processes can be checked against a specification defined by a probabilistic temporal logic formula [5]. Together with analysis techniques, several languages have been introduced to capture uncertain systems, such as Stochastic Petri Nets [6] and Stochastic Automata Networks [7]. These languages allow to describe the system using a formal model that is more abstract than their underlying Markov processes. Nevertheless, a description captured in such languages is always translated into the underlying Markov model for quantitative analysis.

These tools and languages provide a solid groundwork to enable the analysis and design of uncertain systems. However, the adoption of these tools in industrial design flows is not straightforward. There is a semantic gap between the system specification captured using domain specific languages (e.g. Simulink [8], [9]), and the input to many analysis and design flows for uncertain systems. Deriving a model of a system at the level of abstraction in which it can be analyzed by these tools is often done manually, and becomes a tedious, difficult, and error prone process. Further, control algorithms and hardware/software architectures are developed by different teams within the same organization, or by different organizations. These two aspects should be brought together in a virtual environment to allow exploration of different implementation options [10]. Finally, verification of correctness and assessment of performance are not the only type of questions to be answered. A realistic design environment should also provide some parametric analysis capability to perform design space exploration. However, when one attempts to derive a probabilistic model from a system specification using automatic model extraction tools, the state space of the resulting model may grow rapidly if such tools cannot make any specific assumption on the input model. Thus, these tools should be able to leverage the structure of the model to simplify the subsequent analysis task.

This article presents some of the ideas that have been implemented in a design tool for uncertain systems. We define the type of system specifications of interest (Section II) and the structure of a tool that provides the features outlined in this introduction (Section III). Then, we present the challenges in extracting probabilistic models from specifications (Section IV). We present some of the challenges in implementing the back-end analysis tool (Section V) and we conclude with some examples (Section VI).

II. Anatomy of a system specification

The input to a design process is the specification of the system requirements. From these requirements, a set of models are constructed using model-based design environments. Because systems are often heterogeneous, a typical design

A. Pinto is with the Embedded Systems and Networks group of the Systems Department at the United Technologies Research Center, Inc., Berkeley, CA. S. Krishnamurthy is with the Embedded Systems and Networks group of the Systems Department at the United Technologies Research Center, East Hartford, CT. Email: alessandro.pinto,KrishnS1@utrc.utc.com

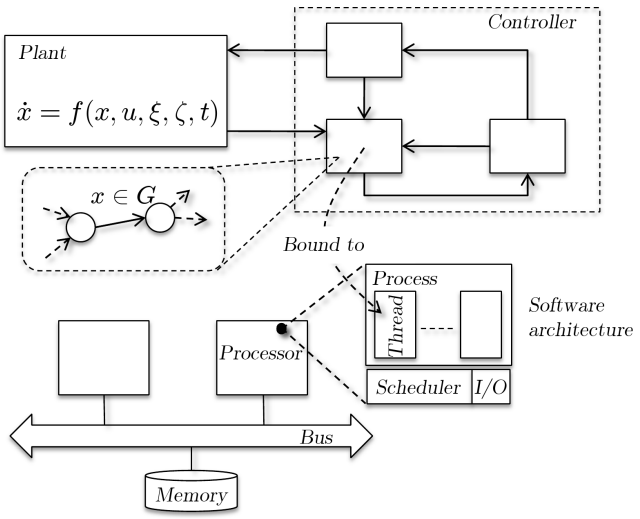

Fig. 1. High level view of the specification of a networked control systems.

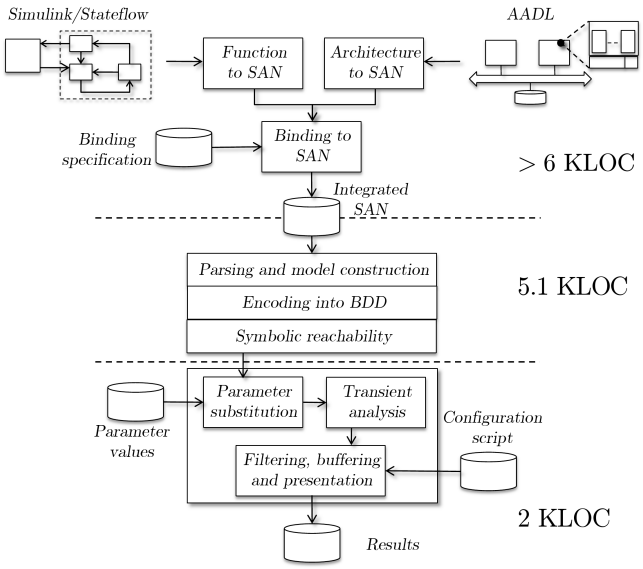

Fig. 2. Description of the design tool for uncertain systems.

flow may involve the use of several languages and tools depending on which aspect of the system the modeler is interested in capturing. For example, control algorithms may be captured using Simulink/Stateflow, while the hardware and software architecture may be modeled using a different language such as AADL [11]. Figure 1 shows a high-level view of a systems specification. The model is divided into two parts: The functional specification (top) and the architectural specification (bottom). The functional specification consists of a model of the dynamic behavior of the physical part of the system (i.e. the plant), and a model of the control logic that can be though of as a set of communicating Extended Finite State Machines (EFSM [12]). The dynamics of the plant is affected by noise (a stochastic process $\zeta(t)$) and by uncertainty in some of its parameters (the vector of random variables ξ). At this abstraction level, the digital controller is considered ideal, meaning that resource constraints are not embedded in the model. This abstraction removes part of the uncertainty coming from performance metrics and failures. However, transitions in the EFSM may be guarded by expressions that make explicit reference to the state of the plant ($x \in G$ in Figure 1) which is a random process, hence transitions are taken with some probability.

The architecture comprises the description of the hardware (which includes processors, storage elements, communication networks and interconnections among them), and the software (which includes processes, threads, schedulers and I/O interfaces). An architectural component may be characterized by a behavioral model as well. For example, the scheduler used by the real-time operating system or by the communication protocol is implemented by a state machine. Moreover, architectural components are annotated with performance metrics such as delay and failure rate. As remarked in the introduction, performance metrics are usually probabilistic and may be available only after the functional model has been bound to the architectural components. For example, the execution time of a thread depends on the content of the thread (i.e. the EFSM bound to it).

III. Anatomy of a design tool

A tool to analyze and design these type of systems must be independent from the input model and should only be based on the assumptions that can be made about the modeling languages used to capture the specification. This tool should accept a functional model described as a stochastic hybrid system [13], an architectural model including performance annotations, and the specification of the binding of the functionality (i.e. the controller) on the architectural resources (i.e. processors, networks and storage elements). Designers should be given the opportunity to define parametric uncertainties in the input model, namely symbolic variables representing transition probabilities, that can be used to sweep over a range of possible values in the performance analysis step. Because of the complexity of the system description, the result of the analysis step is typically difficult to interpret. Thus, designers should also be provided with a practical way of getting insightful information from the result of the analysis.

Figure 2 shows the architecture of the tool we developed. To provide all the aforementioned features, the tool is divided into two parts, a front-end and a back-end, that exchange data over an intermediate modeling language.

The functional and architectural specifications are first translated into an intermediate model. The semantics of the intermediate language is Stochastic Automata Networks and is described in details in Section IV-A. The functional specification is bound to the hardware components at the level of the intermediate language (because function and architecture are defined using different languages). The user provides binding information as input to the tool. The intermediate model is then passed to a back-end tool for analysis.

The first step in the analysis of the model is to compute the set of states that can be reached by the system.

In fact, the intermediate model is in the form of a set of automata that interact using synchronization primitives. This interaction restricts the set of reachable states. The intermediate model is first parsed and then encoded into a Binary Decision Diagram [14][15] to perform symbolic reachability analysis. The result of the reachability analysis is the set of all reachable states. It is possible to store, as a byproduct of the reachability algorithm, the set of transitions between reachable sates. This set can be used to construct the infinitesimal generator of the Markov Chain (MC) underlying the system. The MC is then solved by standard techniques for transient analysis [2], or it can be used for probabilistic model checking [16][17][5]. If the goal is performance analysis, the tool allows the user to provide a configuration file that can be used to filter the data and provide projections of the results along some of the states (e.g. "probability of being in an unsafe state at all time"). The tool also allows to define parameters instead of numeric transition rates. These parameters can be used for quick comparison of different system configuration, or they can be directly used in optimization problems.

Remarkably, our implementation shows that the the front-end development effort (6 thousands lines of code) is comparable with the back-end develop effort, even for the restricted subset of the input languages that we are able to translate at the moment. This result highlights the importance of the model extraction problem, from high-level description to analyzable probabilistic models.

IV. Front-end development and challenges

A. Intermediate model

The choice of the intermediate representation of the system is crucial. It needs to be amenable to analysis and at the same time be expressive enough to enable the representation of a wide range of systems. The model that we selected is a revised version of the Stochastic Automata Network model of Plateau [18].

A Stochastic Automata Network (SAN) $\mathscr{S} = (\mathscr{A}, E, \Rightarrow)$ comprises a set of stochastic automata $\mathscr{A} = \{A^{(1)}, ..., A^{(n)}\}$, a global set of events E, and a relation $\Rightarrow \subseteq E \times E$. A stochastic automaton is a tuple $A^{(i)}(S^{(i)}, T^{(i)}, L^{(i)}, G^{(i)})$ where $S^{(i)}$ is a set of states, $T^{(i)} \subseteq S^{(i)} \times S^{(i)}$ is a set of transitions, $L^{(i)} : T^{(i)} \to 2^E$ is a labelling function that associates a set of events to each transition. The set of possible system states (not necessarily all reachable) is $S = \times_{i=1}^{n} S^{(i)}$. Let $\Pi(S)$ be the set of all partitions of S and let $\Lambda = \cup_{P \in \Pi(S)} [P \to \mathbb{Q}_+ \cup \mathscr{P} \cup \{\top\}]$ be the set of all functions from partitions to the union of the positive rationals, a set of parameters $\mathscr{P}$, and a special symbol $\top$ which denotes *any* rate. The guard function $G^{(i)} : T^{(i)} \to \Lambda$ associates to each transition a state dependent rate.

The relation among events $\Rightarrow$ (reflexive, antisymmetric and intransitive) imposes restrictions on the set of possible behaviors of the SAN. If $(e_1, e_2) \in \Rightarrow$ we also write $e_1 \Rightarrow e_2$ and we say that e_1 *implies* e_2, or e_2 is *implied* by e_1. To define the semantics of a SAN, we first define its language, i.e. the set of possible computation paths. A computation path is a sequence of states and transition times $\pi = (s_0, \tau_0, s_1, \tau_1, ..) \in (S \times \mathbb{R}_+)^{\omega}$. Such path is valid if it satisfies the following set of conditions expressed in terms of the state transitions $t_i = (s_i, s_{i+1})$: 1) $t_i^{(k)} \in T^{(k)}$ where we indicate with $t_i^{(k)}$ the projection of t_i on the states of the k-th automaton, i.e. $t_i^{(k)} = (s_i^{(k)}, s_{i+1}^{(k)})$; 2) $G^{(k)}(t_i^{(k)})([s_i]) \neq 0$, where $[s_i]$ is the partition containing s_i; 3) Either $L^{(i)}(t_i^{(k)}) = \emptyset$ or, $\forall\ e \in L^{(i)}(t_i^{(k)})$, if e is implied by some e' then there must be a transition $t_i^{(j)}$ for some automaton $A^{(j)}$ such that $e' \in L^{(j)}(t_i^{(j)})$. Time t_i is the time spent in state s_i and depends on the transition rate at s_i. The transition rate is not straightforward to define. Consider two events in a relation $e_1 \Rightarrow e_2$ and the two respective transitions $t_1(s_i^{(k)}, s_{i+1}^{(k)}) \in T^{(k)}$ and $t_2(s_i^{(j)}, s_{i+1}^{(j)}) \in T^{(j)}$. The rates of this transitions are $G^{(k)}(t_1)([s_i])$ and $G^{(j)}(t_2)([s_i])$, respectively, which might be different. During reachability analysis, the rate of the implied transition will be constrained to be equal to the rate of t_1 [1]. For illustration purposes and to keep the exposition simple, we will use *binary* guard functions. A binary guard function maps a transition to a subset $\Lambda' \subset \Lambda$, where Λ' only contains mappings whose domains are the binary partitions of the state space. Moreover, one element of the partition must map to zero. We refer to binary guard functions as to *guard conditions*.

The language of a SAN is the set of all valid computation paths. It is easy to see that each computation path is the realization of a Markov Process. In fact, a SAN can be described by its underlying Continuous Time Markov Chain (CTMC). For a characterization of the probability space defined by a CTMC, please refer to [19]. The type of SAN presented in this section provides the basic elements to capture complex synchronization patterns among the transitions of the automata of the SAN. It will appear clear later that this is important for the type of systems that we want to describe.

B. Translation

The translation of the input specification into a SAN involves the translation of the functional model, the translation of the architectural model, and the implementation of the binding constraints.

a) Translation of the functional specification: The main challenge in this task is to represent the continuous time dynamics of the plant into a SAN (for a detailed discussion of the translation of the functional specification please refer to [20]). The first obvious observation is that the behavior of a SAN is a Markov process whereas this is not true in general for the discretized dynamical system. To generate a SAN abstraction of the dynamical system, we use a technique that is well known from the dynamical system community [21]. This techniques consists in discretizing the continuous states and generating a finite partition of the continuous state space.

[1] The reachability analysis algorithm (Section V-A) takes care of other possible situations that may arise, such as an event implied by several input events. Some models, however, might simply be rejected as inconsistent

However, it is only possible to guarantee that the state distribution in the SAN converges to the one of the dynamical system as the partition gets finer. However, little can be said about the speed of convergence and the approximation error. This finite abstraction is unfortunately necessary because analysis techniques for Stochastic Hybrid Systems [13] are still in their infancy.

The translation of the controller (represented by EFSMs) does not present major technical challenges. However, when the controller is described as a set of Stateflow charts in the Simulink/Stateflow tool, then there are many semantic issues that need to be addressed [22]. Most of these issues are solved by simply restricting the type of models that can be translated. This is necessary to keep the size of the model bounded. From a practical implementation standpoint, the main difficulty is in the translation of the guard conditions that are present in the EFSMs. Each guard condition must be translated into a guard condition for the SAN which requires to identify the sub-set of states in which the guard evaluates to true.

Moreover, the transition rates of the EFSMs are not known as they depend on the performance of the underlying hardware platform. Thus, the user has two options. If a functional analysis needs to be carried out, then the rates can be set to 0, if the guard conditions are not satisfied, while transitions are considered "immediate" transitions when the guard conditions are satisfied. If the function is going to be bound to a hardware architecture, then the rates can be set to 0 if the guard conditions are not satisfied, and $\top$ (i.e. to be decided) when the guard conditions are satisfied.

b) Translation of the architectural specification: The translation of the hardware and software architecture also presents some difficulties. The main focus of architecture description languages is to capture components, interconnections, performance metrics and design alternatives. The Architecture Analysis and Design Language (AADL) [11] is one example of such formalisms. However, the behavior of the architectural components is essential to enable the evaluation of the performance of the overall system. Some examples of interesting behavioral aspects are communication protocols as well as scheduling algorithms.

Consider the architectural model in Figure 1. A model of one of the two processing elements is shown in Figure 3. The model comprises two threads, a scheduler, and two I/O buffers (one for transmitting, and one for receiving data). The thread model is an automaton with three states: in the *sleep* state the thread is inactive; in the *ready* state the thread is ready to be executed, but needs to wait to get ownership of the shared computational resource; in the *run* state the thread is executed on the processing element. This model is an abstraction of the thread model defined by the AADL language (one of the few behavioral aspects included in the standard). The scheduler implements a first-come-first-served policy. We have labelled each transition with one event but we have not shown guard conditions.

The thread activation policy determines when the transition from the *sleep* state to the *ready* state occurs. For

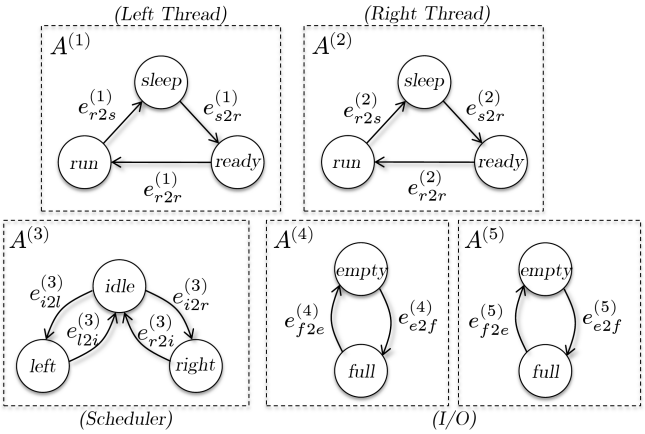

Fig. 3. Example of model of a processing element with two threads a scheduler and I/O buffers.

example, a thread in the AADL language is associated with a dispatch protocol property. A thread can be dispatched periodically, aperiodically, sporadically, or it can be dispatched only once until completion. The transition from the *ready* state to the *run* state is driven by the scheduler. The scheduler decides which thread takes ownership of the processor. The scheduler starts from the *idle* state and it can transition to the $left$ or *right* states to grant the exclusive use of the processing element to the left or right thread, respectively. Consider transition $(idle, right)$. This transition is only taken when the right thread is ready to run, which correspond to the set of system states $S_{rr} = \{s \in S | s^{(2)} = ready\}$. Thus $G^{(3)}(idle, right)(\neg S_{rr}) = 0$ while $G^{(3)}(idle, right)(S_{rr})$ corresponds to the overhead associated with loading the context of the new thread to be executed. When the transition is taken, the thread must move to the *run* state, thus $e^{(3)}_{i2r} \Rightarrow e^{(2)}_{r2r}$. The thread can now be executed. When the execution is over, the thread transitions back to the *sleep* state and releases the resource, i.e. $e^{(2)}_{r22} \Rightarrow e^{(3)}_{r2i}$. Guard conditions and synchronizations are similarly defined for the left thread. The I/O buffers are one place buffers directly used by the application software while in the *run* state. In this example we showed a first-come-first-served scheduler, but other schedulers can be modeled. Also notice that performance metrics are captured by exponentially distributed transitions. This is not realistic in general and there are techniques to deal with this problem such as the use of phase-type distributions [23], or the solution of the underlying Markov Regenerative Process [24].

Figure 4 shows the model of a communication protocol that allows to transfer data between the I/O buffers of two threads. The protocol model is not so different from the scheduler of Figure 3. In fact, it manages access to a communication medium (the shared resource) from multiple sources. The scheduling policy implemented by the protocol is token-ring. The initial state is $left$ meaning that the left I/O TX buffer is checked first. If it is empty, then the protocol passes to check the right buffer. If the left buffer is full, then it is served by moving the message to the right receiving buffer. This is achieved by two synchronizations as follows:

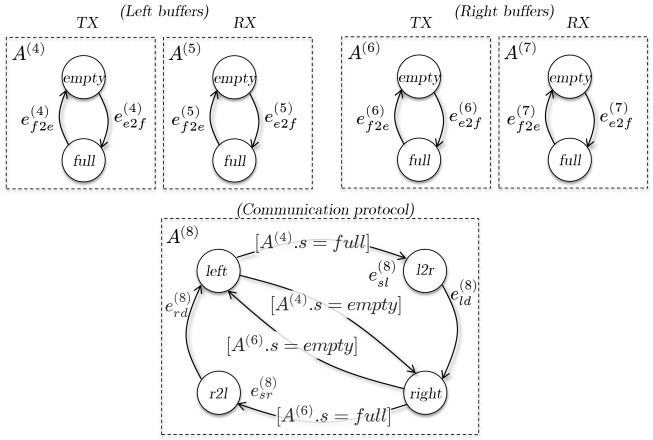

Fig. 4. Example of model of I/O buffers and a communication protocol.

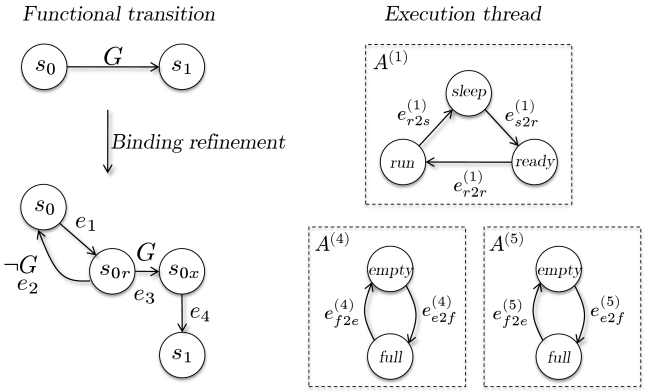

Fig. 5. Example of mapping of a guarded transition on processing element.

$e_{sl}^{(8)} \Rightarrow e_{f2e}^{(4)}$ and $e_{ld}(8) \Rightarrow e_{e2f}^{(7)}$. While the model seems intuitive, the difficulty lies in the potentially large set of implementation options associated with this simple transfer. To mention a couple, we have assumed that the routing of messages is statically defined. This allows to solve the transfer of messages among queues using synchronizations only. As a matter of fact, we are not even defining messages. The definition of message types in the architecture is only used to determine the average transfer delay associated with the transition $(l2r, right)$. Further, we have assumed an overwriting policy for the buffers, i.e. the protocol does not check whether the RX buffer is full before executing transition $(l2r, right)$.

C. Mapped model

Functional controllers are allocated to the available computational resources in the architecture. The user specifies the binding between EFSMs and threads, as well as other architectural features such as I/O buffers. The automatic generation of the mapped design includes a refinement of the functional specification. Figure 5 shows an example of refinement of a transition with guard condition G_1 to be executed on a thread. The single transition is refined into a sequence of states which capture the main steps executed by the code running inside the thread. This is not the only possible refinement, but it is general enough to allow the back-annotation of performance and the definition of different task activation policies. When implementing a finite state machine in software, transitions correspond to change of states that can happen only when the thread implementing the machine runs on a processing element. Thus, the first synchronization to be included in the model is $e_{r2r}^{(1)} \Rightarrow e_1$. When the thread runs, the guard condition is checked. If the guard condition is false, then the transition cannot be executed; the sate machine stays in its current state s_0 and the thread goes back to sleep, i.e. $e_2 \Rightarrow e_{r2s}^{(1)}$. If the guard condition is satisfied, then the transition can be executed. The change of state might have to be communicated to the rest of the system (especially in distributed architectures). In this case, the new state is written in the output buffer using synchronization $e_3 \Rightarrow e_{e2f}^{(4)}$. The thread waits until the transmission is over, i.e. $e_{f2e}^{(4)} \Rightarrow e_4$, and then goes back to sleep, i.e. $e_4 \Rightarrow e_{r2s}^{(1)}$.

In refining the functional specification, the new transitions need to be associated with transition rates (more generally, with transition time distributions). This task, called performance annotation, is complex because it requires taking into account the properties of architectural components (e.g. the processor type) and the content of the functional model (i.e. the complexity of the operation executed by the software). For example, the rate of transition (s_{0r}, s_0) correspond to the time required to evaluate the guard condition, and the rate of transition (s_{0r}, s_{0x}) corresponds to the time required to evaluate the guard condition plus the time required to compute the new state. Thus, these lengths of time depend on the actual running time of the functional code on the selected processor, including the effect of other architectural features such as cache memory, pipeline etc.

This is only one example of the possible patterns that a translator needs to be able to generate. Unfortunately, the number of architectural components and ways of implementing a network of automata on a distributed architecture is large, and consequently the translation task becomes difficult, often requiring the use of other external tools to compute functionality and data dependent performance metrics (such as worst case execution time). For example, we have shown an implementation style where each automaton broadcasts its new state at the end of a transition. This implementation style may work in some cases. In other cases, communication is implemented by unicast messages only to those automata that are affected by the state change. Further, explicit acknowledgements might be required for robustness.

V. Back-end development and challenges

After processing the input specification, the generated model, in the form of an integrated SAN, must be analyzed. Analysis is broken down into two steps: reachability analysis and solution of the underlying Markov Chain.

A. Reachability analysis

The set of states that can be reached by a network of automata can be computed using either explicit or symbolic reachability analysis. Explicit reachability analysis starts from the initial state of the system and uses either a Breath First Search or a Depth First Search algorithm to explore the set of reachable states. Symbolic reachability analysis [25] uses an implicit representation of sets of states and transitions as Boolean functions. These Boolean functions can be efficiently represented using Reduced Ordered Binary Decision Diagrams [14] (ROBDD). The use of this technique requires encoding the transition function of the input model (i.e. the SAN in our case) into a ROBDD that can be then used in a standard fixed point algorithm to compute the set of reachable states [25]. However, the model encoding is challenging because of the type of automata network supporting the SAN model. The two main technical difficulties arise from the state dependent nature of the rates associated with transitions, and the complex synchronizations captured by relation $\Rightarrow$.

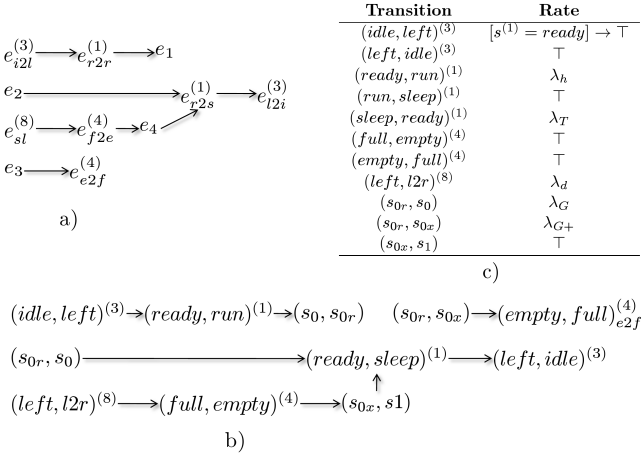

Fig. 6. Synchronization graph summarizing the chain of interactions in the example SAN presented in this paper.

A detailed description of the encoding of a SAN into a BDD is out of the scope of this paper. We will only discuss a few details that capture the complexity of the problem. Figure 6a) shows a graph-based representation of the synchronization relation for the example explained in this paper. For the sake of this discussion, let us restrict the transition labeling functions to the case where events are uniquely associated to transitions[2]. Events in the synchronization graph can then be replaced by the corresponding transitions as in Figure 6b). Figure 6c) shows the guard conditions associated with each transition.

The synchronization graph can be decomposed in a set of disjoint connected components. Some events are input nodes, meaning that they are not implied by any other event, and some events are output nodes, meaning that they do not imply any other event. Further, some event my imply multiple events, and some may be implied by multiple events. For instance, event $e^{(1)}_{r2s}$ is implied by e_2 and e_4. For this type of event we use an OR semantics, meaning that if $e^{(1)}_{r2s}$ occurs, then e_2 or e_4 must have occurred. In some cases, determining the rate of a transition does not present any challenge. For example, transition $(left, idle)^{(3)}$ inherits the rate of transition $(ready, sleep)^{(1)}$. In other cases, determining the rate is more involved. For example, the rate of transition $(ready, sleep)^{(1)}$ depends on the event that triggers it, and it can be either λ_G or λ_d. If $\lambda_G \neq \lambda_d$ and the two transitions are both enabled, then the rate of $(ready, sleep)^{(1)}$ is not defined (unless a specific rule of composition is provided such as the product $\lambda_G \cdot \lambda_d$).

To be able to perform symbolic reachability analysis, the transition function of the combined system needs to be encoded into a ROBDD. We encode each transition as follows. First, for each rate defined in the system, we introduce a new symbol $\gamma_i \in \Gamma$, and we keep a map $\Gamma \to \mathbb{Q}_+ \cup \mathscr{P} \cup \top$. A transition is encoded as a string of binary variables:

$$x^{(1)}_1 y^{(1)}_1 \ldots x^{(1)}_{b_1} y^{(1)}_{b_1} \ldots x^{(n)}_1 y^{(n)}_1 \ldots x^{(n)}_{b_n} y^{(n)}_{b_n} g_1 \ldots g_{b_\gamma}$$

where the string $x^{(i)}_i \ldots x^{(i)}_{b_i}$ encodes the current state of automaton $A^{(i)}$, and b_i is the number of bits required to encode all its states. Also, $y^{(i)}_i \ldots y^{(i)}_{b_i}$ encodes the next state after the transition is executed. Finally, $g_1 \ldots g_{b_\gamma}$ encodes the guard condition associated with the transition. The disjunction of all possible transitions compatible with the semantics of the SAN is the transition function.

B. Computing performance metrics

The result of the symbolic reachability analysis algorithm is a ROBDD containing the set of reachable states. In our implementation we keep also the ROBDD that encodes the set of firable transitions, thereby keeping a symbolic representation of the rate matrix of the underlying Markov Chain. A traversal of the ROBDD generates the rate matrix of the Markov chain where some of the entries are rational numbers and some are elements of $\mathscr{P}$. Before transient analysis, the user must provide a value for these parameters in the form of a table associating each parameter to a rational number. The user can also sweep parameters to generate several results an manually explore the design space. The transient analysis is based on standard techniques [2] and uses sparse matrices.

However, there are two problems that the analysis engine needs to deal with. First, the state space is large and it is in general not possible to keep the entire result in memory. During analysis, partial results must be dumped on disk and the solution must start over by using the last probability distribution computed at time t as the new initial conditions for the system. Most importantly, the state space is now unstructured. The input specification has lot more information due to the fact that the state space is structurally

[2]We do not include bidirectional synchronizations just because they are captured by assignment of the same event to different transitions. However, in the construction of the transition function, bidirectional synchronizations must be treated separately and included in the synchronization graph as special type of arcs.

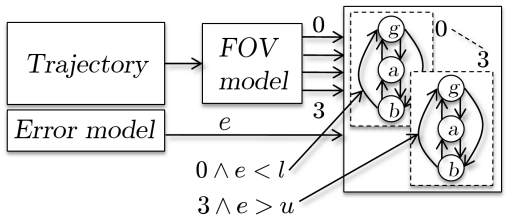

Fig. 7. High level diagram of the model of the autonomous mission.

(λ, σ)	Score	0	1	2	3
$(0.5, 0.21)$	g	0.620735	0.025383	0.010964	0.024086
	a	0.184631	0.182338	0.179239	0.18054
	b	0.194634	0.792279	0.809797	0.795374
$(1, 0.42)$	g	0.556104	0.047224	0.038085	0.045598
	a	0.382108	0.374072	0.380567	0.373811
	b	0.061787	0.578703	0.581347	0.580591
$(10, 0.84)$	g	0.407133	0.142445	0.140132	0.142345
	a	0.453272	0.452727	0.452559	0.452914
	b	0.139595	0.404828	0.407309	0.404741

TABLE I

PROBABILITIES OF GOOD, AVERAGE AND BAD MATCHING FOR DIFFERENT VALUES OF ERROR VARIANCE AND COMPUTATION SPEED.

partitioned among the different automata. This information is hidden once the system is represented by a single Markov Chain. The user can define filters to project the state space over a few meaningful traces. For example, suppose the user is interested in looking at the probability that $A^{(1)}$ and $A^{(2)}$ are running. The analysis engine creates a matrix M that projects the probability vector of the entire system onto those two states. This matrix will have two rows and as many columns as π (i.e. the size of the state space). Element $M(1,i)$ is equal to one if $s_i^{(1)} = run$, and it is zero otherwise. Only the projected space needs to be stored and presented to the user. Filters can be in general complex including the conjunction or disjunction of states. In general, a link between the underlying CTMC and the original system must be kept.

VI. EXAMPLES OF APPLICATIONS

In this section we show a functional analysis and an architectural analysis. The architectural analysis shows some metrics computed starting from the model presented in this section. The purpose of this section is to show the versatility of the tool rather than inferring properties about the systems being analyzed.

A. Example of functional analysis

Consider an autonomous helicopter which is assigned the mission of finding a building marked with a special symbol in a urban area. 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) can cause either a false negative (i.e. the symbol is missed), or a false positive (i.e. an object is recognized as the symbol). Each object with a minimum level of matching is kept in a table with an associated score. At each frame the score is updated depending on the quality of the matching. We discretized the score into three levels: good (g), average (a) and bad (b). We assume that there are four objects randomly placed in the scene and that object 0 is the symbol.

The model is shown in Figure 7[3] and has two parts. The trajectory followed by the helicopter is computed by a trajectory generation algorithm for given way-points around the building. A camera model is used to generate Boolean flags that are equal to TRUE if the corresponding object is in the field of view of the camera and FALSE otherwise (`Field of view model`). The vision algorithm is a finite state model that maintains a matching score for each of the objects in the scene (the object being parallel states). When in the field of view, the score assigned to object 0 increases if the position error is below a lower bound l, and decreases if it is above an upper bound u (transitions are reversed for the other objects). Because the update is done at each frame, higher computation rates improve the ability to distinguish image features. The system is translated into a SAN, where the variance σ of the colored noise e is one of the parameters. The transition rate λ associated with transitions in the vision algorithm model is another parameter. These two parameters represent the accuracy of the sensors and the speed of execution (i.e. frames per second) of the vision algorithm, respectively.

The helicopter follows a circular trajectory around the building. We report the values of the probability of being in each of the three level of the matching score for all the objects in Table I.

B. Example of architectural analysis

We consider a distributed architecture composed of processors running a single thread and communicating over a token ring bus. This architecture is built using the templates presented in Section IV-B. Each thread th_i transitions from the *sleep* state to the *ready* state when the transmission buffer TX_i is empty. When the thread is scheduled to run, it first reads from buffer RX_i, and then writes to TX_i. The token ring bus serves the TX buffers and broadcasts their content to all RX buffers in the system. We consider transition rates of 10^5, 10^4 and 10^3 for transitions $(sleep, ready)$, $(ready, run)$ and $(run, sleep)$ respectively. We also consider a rate of 8000 for the protocol to pass the token among users, while we leave the data transmission rate λ as a parameter (to mimic the effect of different packet sizes). We consider three architectures: $sys2$ with two processors (182 reachable states), $sys4$ with four processors (24708 reachable states) and $sys4f$ with 4 unreliable processors (2118680 reachable states). Unreliable processors can fail with rate 0.0003, and recover from failure with rate 0.3. The results of the analysis are shown in Table II where we report the probability of being in the sleep, ready or run state for thread th_2 at time $t = 1ms$.

[3]A more detailed Simulink/Stateflow model can be found in [20]

λ	System	$P(sleep)$	$P(ready)$	$P(run)$
	$sys2$	0.275	0.058	0.666
8000	$sys4$	0.380	0.038	0.581
	$sys4f$	0.371	0.038	0.591
	$sys2$	0.378	0.040	0.581
4000	$sys4$	0.453	0.025	0.522
	$sys4f$	0.439	0.025	0.535
	$sys2$	0.459	0.026	0.515
2000	$sys4$	0.505	0.016	0.479
	$sys4f$	0.489	0.016	0.495

TABLE II

PROBABILITIES OF BEING IN THE SLEEP, READY OR RUN STATE AT $t = 1ms$ FOR THREAD th_2.

The results show two obvious trends. When the number of processors increases, the token rotation time increases and the time a task spends in the sleep state also increases. If the transmission time increases, the time spent in the sleep state also increases. Interestingly, the time spent in the run state is higher for $sys4f$ than for $sys4$. This is because thread th_2 can leverage the time when other processors are silent because of a failure.

VII. CONCLUSIONS

The design and analysis of cyber-physical systems requires the ability to deal with uncertainty. While many tools exist, the development of an integrated framework for analysis and design of uncertain systems exposes challenges that include model extraction from heterogeneous specification, parametric analysis to enable design space exploration, fast computation over large Markov models, and presentation of meaningful results to designers.

We have presented a tool that is under development at the United Technologies Research Center and that addresses the integration of stochastic analysis techniques into standard design flows. The tool is divided into a front-end that performs the model extraction from high level languages, and a back-end that implements analysis techniques on SAN models. On the one hand, this separation allows to make the analysis independent from the specific application and modeling language. On the other hand, much of the complexity of the analysis can be reduced by an appropriate modeling strategy, which suggests a tight integration of the front-end and the back-end.

Our future work includes the ability to deal with Stochastic Hybrid Systems, and the implementation of automatic design space exploration tools.

VIII. ACKNOWLEDGMENTS

The authors wold like to thank the United Technologies Research Center (UTRC), East Hartford, CT, for the support provided for this work. They would also like to thank their colleagues Andrzej Banaszuk, Tuhin Sahai, Amit Surana and Cong Liu for comments and suggestions.

REFERENCES

[1] M. Stamatelatos and W. Vesely, "Fault tree handbook with aerospace applications," *NASA Office of Safety and Mission Assurance*, 2002.

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

[3] P. E. Kloeden and E. Platen, *Numerical Solution of Stochastic Differential Equations*. Springer-Verlag, 1999.

[4] D. Xiu and G. E. Karniadakis, "The wiener–askey polynomial chaos for stochastic differential equations," *SIAM J. Sci. Comput.*, vol. 24, no. 2, pp. 619–644, 2002.

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

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

[7] B. Plateau and K. Atif, "Stochastic automata network of modeling parallel systems," vol. 17, no. 10, pp. 1093–1108, Oct. 1991.

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

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

[10] A. Sangiovanni Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi, "Benefits and challenges for platform-based design," in *Proceedings of DAC*, June 2004, pp. 409–414.

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

[12] K.-T. Cheng and A. S. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, no. 1, pp. 57–79, 1996.

[13] L. J. Hu, J. and S. Sastry, "Towars a theory of stochastic hybrid systems," *Proceedings of the Third international Workshop on Hybrid Systems: Computation and Control*, vol. N. A. Lynch and B. H. Krogh, Eds. Lecture Notes In Computer Science, vol. 1790. Springer-Verlag, pp. 160–173, March 2000.

[14] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

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

[16] L. D. Alfaro, "Model checking of probabilistic and nondeterministic systems." Springer-Verlag, 1995, pp. 499–513.

[17] C. Baier, "On algorithmic verification methods for probabilistic systems," Ph.D. dissertation, 1998.

[18] 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.

[19] C. Baier, J.-P. Katoen, and H. Hermanns, "Approximate symbolic model checking of continuous-time markov chains (extended abstract)," 1999, p. 781. [Online]. Available: http://www.springerlink.com/content/t22lk0et8c9r40c4

[20] S. K. Sudha Krishnamurthy, Alessandro Pinto, "A model-based end-to-end toolchain for the probabilistic analysis of complex systems," in *Proceedings of the 6th IEEE Conference on Automation Science and Engineering (CASE)*, August 2010.

[21] M. Dellnitz and O. Junge, *Set Oriented Numerical Methods for Dynamical Systems*. Elsevier, 2002, ch. 5.

[22] 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.

[23] M. F. Neuts, *Matrix-geometric solutions in stochastic models : an algorithmic approach / Marcel F. Neuts*. Johns Hopkins University Press, Baltimore :, 1981.

[24] R. German, D. Logothetis, and K. S. Trivedi, "Transient analysis of markov regenerative stochastic petri nets: A comparison of approaches," in *In 6-th International Conference on Petri Nets and Performance Models - PNPM95*. IEEE Computer Society, 1995, pp. 103–112.

[25] 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.