GENERATION OF SAFE OPTIMISED EXECUTION STRATEGIES FOR UML MODELS

Luke Thomas Herbert [1], Zaza Nadja Lee Hansen [2]

[1]*Deloitte, Copenhagen, Denmark* `herbert.luke@gmail.com`
[2]*DTU Management, Lyngby, Denmark* `znlh@dtu.dk`

When designing safety critical systems there is a need for verification of safety properties while ensuring system operations have a specific performance profile. We present a novel application of model checking to derive execution strategies, sequences of decisions at workflow branch points, for a fragment of the Unified Modelling Language (UML) statechart language which is extended to include modelling of workflows which exhibit stochastic behaviour. Strategy generation is made possible by performing model checking on specific permutations of the set of possible actions to generate adversaries which optimise a set of reward variables, while simultaneously observing constraints which encode any required safety properties and accounting for the underlying stochastic nature of the system. By evaluating quantitative properties of the generated adversaries we are able to construct an execution strategy which fully specifies, from any state in the system, the actions needed for an actor to achieve the optimal values of the quantitative goals. We show that our method is computationally feasible and apply it to an illustrative example featuring an industrial robot. Our approach make it possible to readily test and debug a wide range of possible designs, thus creating a more effective development of a safety critical system.

I. INTRODUCTION

Modern safety-critical systems are often characterised by the need to incorporate unreliable or unpredictable components which are frequently composed to build complex concurrent behaviour, often while subject to various non-functional performance requirements.[1] Ensuring that these systems are both dependable and efficient poses a significant challenge. By including quantitative data in system models, determination of bounds on the performance properties of such systems becomes possible. Further, combined with analysis of stochastic behaviour, this approach is typically employed to analyse unreliable or unpredictable systems.[2] While verification is typically performed at each stage of system development; it is well established that early detection of faults and design limitations is the key to limiting their impact and cost[3] and recent years have seen considerable progress in developing methods for the verification of such systems.[4,5] However, safety critical systems frequently combine the need for verification of safety properties while simultaneously requiring a specific performance profile. To achieve these goals, such systems often require sophisticated execution strategies especially when the system involves scholastic elements. Being able to synthesize a strategy for the optimal execution of such systems early in their design phase allows for accurate determination of how the system will be employed, in the form of the sequence of actions performed by elements of the system, and consequently holds the potential for the early identification and exclusion of inefficient designs.

There exists a wide range of system modelling languages of which the *Unified Modelling Language* (UML)[6] is broadly accepted as the de facto standard notation for the analysis and design of object-oriented software systems.[7] The Statechart formalism of UML is focused on organizing the way a device, computer program, or other (often technical) process works such that an entity or each of its sub-entities is always in exactly one of a number of possible states and where there are well-defined conditional transitions between these states. It has found widespread use as a early stage design tool with the predominant approach being to employ UML to develop initial conceptual system models, which are then used as a basis for development of a practical implementation.

I.A. Contribution

The focus of the paper is the generation of execution strategies from UML models, which unfortunately has an imprecise and incomplete semantic definition. For this reason we have chosen to extend our previously developed methods,[8] for the analysis of models based on the *BPMN* modelling language, to accommodate the widely used UML language. Specifically, UML Statecharts which we extend to accommodate both data annotations (rewards) and stochastic behaviour.

Fundamentally, the addition of stochastic behaviour in a system of interest significantly complicates the application of many traditional strategy approaches. In this case a strategy is potentially associated with many different executions, which must be all taken into account in order when determining the effect of a specific sequence of actions on the overall goal. We present an approach based on translation of UML models into *Markov decision processes* (MDPs)[9] described using the PRISM modelling language[10] employed by the model checker PRISM.[11] The choice of PRISM is motivated by the great expressivity of its *Probabilistic Computational Tree Logic* (PCTL) query language, which makes it possible to express probabilistic reachability problems. Within this paper a strategy will be defined as the *sequence of actions to be taken from an initial state in the system to obtain the optimum of one or more quantitative properties associated with a model.* To allow such problems to be solved using model checking we reformulate a static scheduling problem as a reachability problem that can be subjected to model checking using PRISM. Employing the model checker PRISM to perform model checking on specific permutations of a set of constraints to generate *optimal adversaries* which optimise (minimise / maximise) a reward value while observing constraints which encode any required safety properties. By evaluating quantitative properties of the generated adversaries within the full state-space of the UML model we are able to construct an execution strategy with the desired safety and performance properties, if such a strategy exists. Moreover, by defining this translation, our work allows the analysis to be done from a starting point expressed directly in widely used UML.

Fundamentally, our approach (fig. 1 show an overview), allows for systems designers to test a wide range of possible designs, and to readily *debug* them, before committing to a specific practice, which achieves more effective development of the system in question.

Central to the approach taken to strategy generation is to exploit the generation of *adversaries* (see Section V), also known as strategies, inherent in MDP model checking. The generation of *optimal adversaries* in PRISM is determined as part of computing a PCTL query for model and capture the specific choice of actions, resolutions to points of non-determinism, which ensure that a probability of reward query are maximised or minimised. A novel and recently implemented feature in PRISM, is that multi-objective properties can defined for adversary generation.[12,13] This allows a combination of reward and probability queries to be combined and a single adversary generated which optimises all of these values. It should be noted that PCTL queries can also encode required safety properties such that a generated strategy will optimise the strategy goals while excluding strategies which violate a safety requirement.

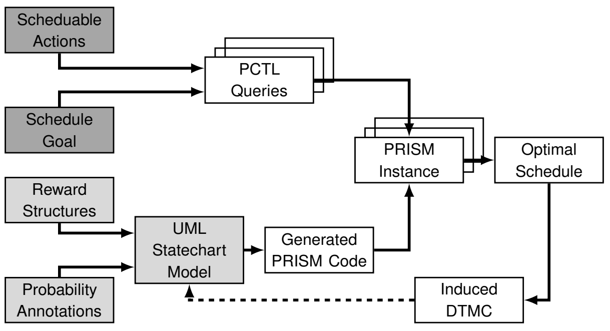

Fig. 1. UML strategy generation via PRISM model checking (grey show inputs needed and dark grey show user choices).

The result of strategy generation is a *Deterministic Time Markov Chain* (DTMC)[9] which encodes the specific sequence of actions required to best approach the strategy goal. This DTMC is labelled so as to record the sequence of actions that a strategy encodes and this can be mapped back to the original UML Statechart model so as highlight the specific choices needed to execute the optimal strategy.

I.B. Related work

A number of proposals for the formalisation of UML Statecharts[14,15] has been made, some of which also incorporate formal methods based analysis of the model's properties. The development which is the most similar to our work is the approach taken by Jansen et al..[16] Their work is also based on UML Statecharts, which are extended with probabilities in a fashion similar to what is developed here. However, our work generalises the data added (in the form of rewards) to a UML model, making it possible to analyse not just temporal properties, but also other resources consumed by a system.

The introduction of rewards allows for the determination of ideal strategies, and whereas multiple strategies may exist, quantitative methods allows for the selection of strategies which optimise rewards of interest. Two similar approaches to solving strategy problems using model checking are given by Wijs et al.[17] and Basu et al..[18] However, in both cases, the construction of the model from which to generate a strategy is a manual process which requires considerable tuning.

While the traditional approach to employing model checking to solve strategy problems is mostly concerned with resolving all points of nondeterminism so as to minimise or maximise one or more values of interest. A classic example of this approach is that of Ruys[19] and can be extended to account for costs, such as in the work of Behrmann et. al..[20] However, the addition of stochastic behaviour in a system of interest significantly complicates the application of many traditional strategy approaches. In this case a strategy is potentially associated with many different executions, which must be all taken into account in order when determining the effect of a specific sequence of actions on the overall goal. Conceptually similar to the approach taken here is the work of Giunchiglia and Traverso,[21] where planning problems are seen as including non-determinism not under the control of a planner, and a similar approach using model checking to efficiently explore the resultant statespace is developed here.

II. UML STATECHARTS

UML Statecharts, also known as a UML state machines, are an enhanced realization of the concept of a finite automaton expressed in UML notation. The description of how a process (e.g. a computer program or a business workflow) works is organized so that an entity, or each of its sub-entities, is always in one of a number of possible states, and where there are well-defined transitions between all states.

II.A. UML Statecharts Syntax

UML Statecharts consist of six basic elements :

● **Initial:** These elements i, denote the start of a process and are the points at which execution of a UML Statecharts begins. This is a so-called *pseudo state*, where the state has no variables describing it further and no associated activities. For each UML Statechart there is one unique initial element.

× **Terminate:** These elements $\mathbf{E^T}$, denote a final state of the system at which execution of a process halts. This ia a pseudo state from which the state machine does not exit and nor does it perform any actions other than those associated with the transition leading to the terminate state. A UML Statechart may have zero, one, or more termination states.

▭ **State:** These elements $\mathbf{S}$, denote states of a process. Each state models a situation during which some, usually implicit, invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some behaviour (i.e., the model element under consideration enters the state when the behaviour commences and leaves it as soon as the behaviour is completed).

◇ **Choice:** These elements $\mathbf{G^C}$, denote a decision point at which flow of execution of a process will proceed along one of the outgoing flows from the element. Control flow is determined by the evaluation of the guards of its outgoing transitions.

| **Fork/Join:** These elements denote both forking and merging of execution of a process. Fork elements $\mathbf{G^F}$, serve to split an incoming transition into two or more transitions. Join elements $\mathbf{G^J}$, serve to merge several transitions. The transitions from/to a fork/join vertex may not guards. Note this element may be drawn rotated, if desired, in order to improve the readibility of a Statechart.

$\xrightarrow{l[g]}$ **Transition:** These elements $\mathcal{F}$, denote a transition from one state to another, where l is an optional label for the transition. A guard $[g]$ can be added to a transition where the body g of the guard is a boolean expression which when true indicates that the given transition *may* be performed.

It should be noted that the full specification of UML Statecharts[6] also includes a number of other elements which provide various syntactically useful constructs allowing for grouping elements in various ways, but these do not add further significant features to the language in terms of the analysis developed. We will formally define a complete UML Statechart model as:

Definition 1 (UML Statechart Model) *A UML Statechart is a tuple* $(\mathbf{N}, \mathcal{F}, \mathbf{L}, \mathsf{lab})$ *where* $\mathbf{N} \subseteq \mathbf{S} \cup \mathbf{E} \cup \mathbf{G}$*, is a set of nodes composed of the following disjoint sets:*

- *States* $\mathbf{S}$*, are the basic states of a given process.*
- *Events* $\mathbf{E} \subseteq i \cup \mathbf{E^T}$*, where i is the unique initial state and* $\mathbf{E^T}$ *represents termination states.*
- *Gateways* $\mathbf{G} \subseteq \mathbf{G^C} \cup \mathbf{G^F} \cup \mathbf{G^J}$*, where the disjoint sets* $\mathbf{G^C}$*,* $\mathbf{G^F}$ *and* $\mathbf{G^J}$ *respectively represent choice, fork and join elements.*

$\mathcal{F} \subseteq \mathbf{N} \times \mathbf{N}$ *is a set of flow relations representing transitions, where* $a\mathcal{F}b$ *denotes a directed transition from a to b.* $\mathbf{L}$ *is a set of unique labels and* $\mathsf{lab} : \mathcal{F} \to \mathbf{L}$ *is a labelling function which assigns labels to flows.*

The definition of a UML Statechart given in Definition 1 models processes by using elements of $\mathcal{F}$ to define a directed graph with nodes which are elements of $\mathbf{N}$. This definition allows for graphs which are unconnected, do not have start or end elements or various other properties which place them outside what is implied to be permitted in standard UML models. The UML specification alludes to well-formedness requirements for UML models[6] and these take the form of *constraints* which are textual rules which restrict usage of various elements, however these are ambiguously expressed. Likewise, to ensure that a UML Statechart describes a meaningful process we will impose a number of *well-formedness* conditions. These conditions are defined using the following functions:

- The input nodes of $n \in \mathbf{N}$ are given by the function $\mathsf{in}(n) = \{x \in \mathbf{N} | x\mathcal{F}n\}$.
- The output nodes of $n \in \mathbf{N}$ are given by the function $\mathsf{out}(n) = \{y \in \mathbf{N} | n\mathcal{F}y\}$.

Definition 2 (Well-formed UML Statechart) *A UML Statechart is well-formed if the following conditions hold:*
E1 $i : \mathsf{in}(i) = \emptyset \wedge |\mathsf{out}(i)| = 1$
E2 $\forall e \in \mathbf{E^T} : |\mathsf{in}(e)| = 1 \wedge \mathsf{out}(e) = \emptyset$
T1 $\forall s \in \mathbf{S} : |\mathsf{out}(s)| = 1$
G1 $\forall g \in \mathbf{G^C} : |\mathsf{in}(g)| = 1 \wedge |\mathsf{out}(g)| \geq 2$
G2 $\forall g \in \mathbf{G^C} : \forall s \in \mathsf{out}(g) : \mathsf{lab}((g,s)) \neq \perp$
G3 $\forall g \in \mathbf{G^F} : |\mathsf{in}(g)| = 1 \wedge |\mathsf{out}(g)| \geq 2$
G4 $\forall g \in \mathbf{G^J} : |\mathsf{in}(g)| \geq 2 \wedge |\mathsf{out}(g)| = 1$
F1 $\forall n \in \mathbf{N} : \exists (i,e) \in i \times \mathbf{E^T} : i\mathcal{F}^* n \wedge n\mathcal{F}^* e$ *where* $\mathcal{F}^*$ *is the reflexive transitive closure of* $\mathcal{F}$.

Definition 2, is chosen such that the conditions defined impose the minimum semantic interpretation necessary to determine the control flow of a model. A well-formed UML Statechart will thus have no more semantic interpretation than given in the UML standard.[6]

II.B. Stochastic UML Statecharts Semantics

Since the UML standard only contains an informal description of how to execute UML Statecharts and a mathematically precise semantic framework is required for automated analysis, we will employ the structured operational semantics for UML Statecharts developed by Michael von der Beeck.[22] These semantics define an execution of UML where each state in a sequence of transitions is executed in turn. Forking constructs split the flow of execution so as to follow all paths of transitions emanating from a forking gateway, with these separate flows of execution running concurrently and independently until a merging gateway is encountered. A merging gateway functions as a point of synchronization where all incoming transitions must reach the merge point before execution of the outgoing transition may take place. Progress between states is made under a maximal progress assumption where a transition to an enabled state is performed as soon as it becomes possible. Note that UML Statecharts have no notion of time, but are simply an ordering of events, where events contained in different parallel flows may in fact take place concurrently.

To deal with the complexity of modelling large systems, we divide them into smaller sub-systems which interact by message passing, in a fashion similar to the work of Jan Jürjens.[23] The concept here is that a number of UML Statecharts are made to interact by means of common labels on transitions. When a transition is encountered, in UML Statechart USC_a, with a label that is also used for a transition in USC_b, execution of USC_a halts until USC_b reaches the transition with the same label. At this point both systems simultaneously perform the specified labelled transition.

On choice gateways we will impose a semantic interpretation where we maintain the non-deterministic choice inherent in the definition of UML Statechart gateways,[6] but supplemented with probabilistic selection. At a choice gateway, execution proceeds by non-deterministic selection of an outgoing transition from a choice gateway; these transitions are identified by labels from the set $\mathbf{L}$ which are assigned to specific transitions by means of the function lab introduced in Definition 1. Note that the well-formedness condition **G2** ensures all transitions are labelled at these points. Several transitions may have the same label and in this case the choice of a specific transition is made probabilistically. This effectively captures the behaviour of a process which involves an actor making a deliberate choice, and selected choice has different possible outcomes not under the control of the actor. We will assign probabilities to outgoing transitions from a choice gateway by means of the following function:

Definition 3 (BPD Gateway Flow Probability Function) *Given a BPD, a decision gateway probability function is a partial function* $\mathcal{P} : \mathcal{S} \times \mathbf{L} \rightarrow [0,1]$ *which for a node* $g \in \mathbf{G^D}$ *and label* $l \in \mathbf{L}$*, assigns probabilities to all outgoing sequence flows* (g,x)*, such that for a given* l*:*

$$\sum_{\forall x \in \mathsf{out}(g)} \mathcal{P}((g,x),l) = 1$$

Recall that well-formedness condition **G2** ensures that all transitions from a choice gateway are labelled. Several outflows may, however, have the same label. Definition 3 ensures that all choice gateways have an associated probability and that the sum of all probabilities for a given label l is 1.

For quantitative analysis of models, we add numerical data to our models by using a reward function which associates positive real numbers with states in a UML Statechart.

Definition 4 (USC Task Reward Function) *For a UML Statechart a reward function for a state $s \in \mathbf{S}$ is a partial function $\mathcal{R} : \mathbf{T} \rightarrow \mathbb{R}_{\geq 0}$.*

$\mathcal{R}$ captures the notion that certain nodes have some reward or cost associated with the state. There is no practical distinction between costs and rewards, and we use these annotations to keep track of whichever quantities may be of interest in a process, such as execution time or energy consumption. We may associate as many reward structures as we wish with a given UML Statechart, so that a single task may have multiple different numerical properties which are incremented when the task is performed, to achieve this we augment the R operator with a label.

III. MODELLING EXAMPLE

While the methods presented in this paper are applicable to a wide range of systems, we have chosen an example arising from problems encountered by an industrial partner in the healthcare sector.

Figure 2 illustrates a simplified scenario where a strategy must be devised for the actions of a robot arm moving materials for preparing drugs between different subcomponents. This system consists of 4 processes each represented as an individual Statechart. The "pharmacy robot" process drives the operation of this system and makes a non-deterministic choice between drugs which can potentially be manufactured. Manufacturing each drug involves a specific sequence of operations performed by separate sub components; each of these performs steps which have delays which are stochastically chosen. Synchronization between the different Statecharts is performed via the **[***label***]** constructs, in the fashion detailed in Section II.B. A number of states are annotated with reward structures tracking time used and energy expended.

Note that this system has 2 key points where a non-deterministic choice must be made between several options. Namely in the choice of which drug to manufacture and, when heating drugs, a choice between *normal* or *low-power* heating. In this system there is a safety requirement that shaking must never occur while loading of a drug is taking place, as the vibrations caused by shaking could lead the system to malfunction. Production of a specific batch of drugs (e.g. 2 doses of A, 1 of B and 3 of C) should be sequenced so that production takes place as quickly as possible and using the minimum amount of energy, while observing the safety requirements.

IV. STOCHASTIC MODEL CHECKING

The goal of this work is to transform a UML model into a *Markov decision process*[9] (MDP) which is amenable to formal statespace analysis. These states represent possible configurations of the system being modelled with probabilistic state transitions being combined with non-deterministic choices between several discrete probability distributions over successor states. Model checking allows for the efficient exploration of the entirety of this space with a temporal logic employed to select sets of states of interest, and offers the possibility of verifying many properties of a system. In this paper we will specifically use this capability to select sets of paths through the state space that represent different strategies; each path is then checked to ensure that given safety criteria are observed and the values of rewards of interest are computed.

IV.A. Translating a UML Statechart into PRISM code

In our approach, UML Statechart models are mapped directly into the guarded command language used by PRISM. The mapping, which focuses on the control flow structure of the model, involves decomposing a UML Statechart into sub-processes which are individually mapped to PRISM code, with appropriate synchronization constructs generated to maintain the same control flow. For reasons of space the details on this approach are omitted, however they are highly similar to our previous work with the *BPMN* language.[8] Combined this approach has an has an upper complexity bound of $O(n^3)$. The soundness of this algorithm can be shown by structural induction.

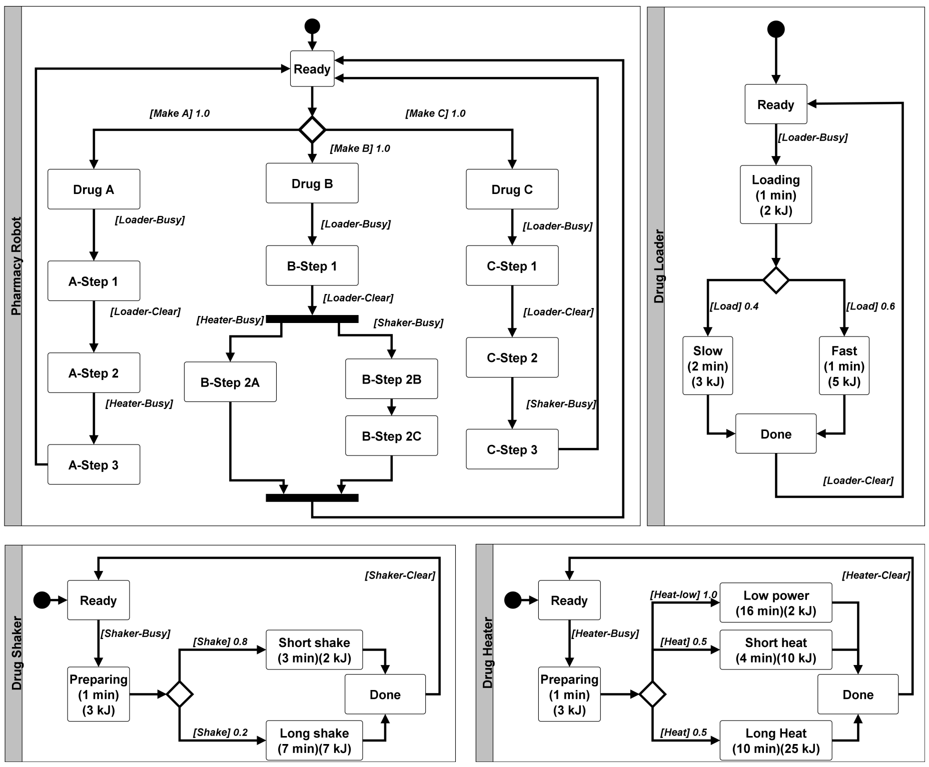

Fig. 2. Annotated UML Statecharts of a pharmacy automation system and surrounding environment.

V. STRATEGIES

Within this paper a sstrategy is defined as the *sequence of actions to be taken from an initial state in the system to obtain the optimum of one or more quantitative properties associated with a model*. This can be combined with a set of constraints on the process. These constraints would typically capture safety properties, but can be freely defined using the logic PCTL to express properties that must hold when executing the strategy.

For example suppose one was considering the medical robot from Section III which is capable, by means of making a non-deterministic choice, of producing either drug A, B or C. In the case when three doses of drug A, two doses of drug B and one dose of drug C are required. The possibilities for sequencing these actions are illustrated in in Figure 3, where each possible strategy is a path from the root to a terminal node.

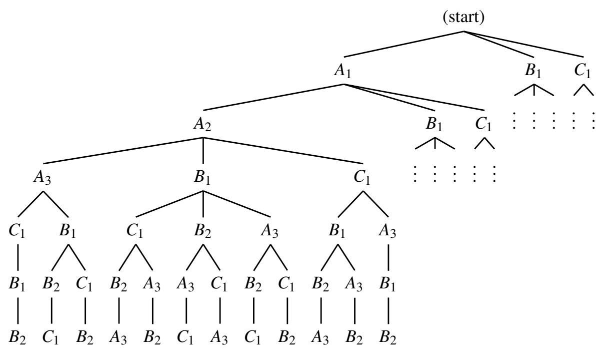
(start)
A_1 B_1 C_1
A_2 B_1 C_1
A_3 B_1 C_1
C_1 B_1 C_1 B_2 A_3 B_1 A_3
B_1 B_2 C_1 B_2 A_3 A_3 C_1 B_2 C_1 B_2 A_3 B_1
B_2 C_1 B_2 A_3 B_2 C_1 A_3 C_1 B_2 A_3 B_2 B_2

Fig. 3. Illustration of the possible strategy possibilities for a combination of 3 doses of drug A, 2 doses of drug B and 1 dose of drug C.

V.A. Strategy Specification

Generation of a strategy requires the resolution of the points of non-determinism under the control of agents in the model. We will employ PRISM's capability for adversary generation[11] which generates an induced *discrete time Markov chain* (DTMC)[9] on the generated state-space that equates to evaluating the best- or worst-case choice of actions at all decision points that satisfy a chosen Probabilistic Computation Tree Logic (PCTL)[11] constraint. The PCTL property specification language which is based on classical continuous stochastic logic extended to probabilistic quantification of described properties. An implementation of the PCTL logic is employed by the PRISM model checker.[11]

While this logic allows reasoning about a wide range of system properties,[5] we will employ PCTL queries to filter out paths in the state-space generated by the PRISM model checker for a UML Statechart. Specifically, PCTL queries are used to define the safety properties which we require for the system and the specific tasks that we want performed as part of running the system, and to determine the cumulative mean rewards values along that path. Determining an adversary, also known as a *strategy*, for a model with non-deterministic choices as shown in Figure 3 determines the effect these possible sequences of actions will have on the reward and probability values encoded in a PCTL query of interest. As PRISM quantifies over all possible adversaries, i.e. all possible resolutions of nondeterminism in the model it is ensured that there exists within these possible resolutions a set of resolutions for which a models associated reward and probability values which take on minimum or maximum values. Given a set of tasks that must be executed as part of running the system, solving a strategy problem by means of model checking involves determining which strategy optimises the execution of the system.

Adversary properties for a single objective are specified in the standard fashion for PRISM property queries described by Forejt et. al. in.[11] Multi-objective adversary properties are specified in PRISM 4.1 by means of the `multi(...)` keyword. This keyword allows for a comma separated list of separate queries to be defined and may only be employed when specifying an adversary. If a multi-objective property contains a single unbound `?` objective then an adversary is determined which achieves the minimum possible probability of reaching a *Load* state, from which the probability of reaching *Error* is less than 0.1, an example of this goal, expressed in PCTL, is shown in eq. (1).

$$\texttt{multi}(\mathsf{P}_{\mathsf{min}}{=}?[\mathsf{F}\ \mathit{Load}], \mathsf{P}{<}\ 0.1[\mathsf{F}\ \mathit{Error}]) \tag{1}$$

A C operator is allowed in adversary generation which calculates the total cumulative value of a reward structure in a multi-objective property query. This value is the total of the reward value accumulated along all paths as opposed to simply the specific path of the adversary as is the case with the F operator. For example eq. (2) expresses: which adversary ensures that the expected cumulative value of the reward structure *time* is minimised while ensuring that the expected cumulative value of reward structure *energy* is below 7.2.

$$\texttt{multi}(\mathsf{R}_{\mathsf{min}}(\mathit{time}){=}?[\mathsf{C}], \mathsf{R}_{\mathsf{min}}(\mathit{energy}){<}\ 7.2[\mathsf{C}]) \tag{2}$$

Further, possibilities are for specification of multiple adversaries along with an extensive description of the computation of adversaries is available in.[13] Note that while individual adversaries are generated on the same basic state-space, PRISM allows *symmetry*[24] and *partial order*[25] reduction to be employed when searching this space and allows our approach to presently scale to the feasible verification of complex properties of large systems (up to 10^{10} states[2]).

V.B. Strategy Generation

The adversary generation feature in PRISM produces an induced DTMC over an MDP which optimises the properties of interest by appropriate resolution of points of non-determinism in a model. This induced DTMC that equates to evaluating the best- or worst-case choice of actions at all decision points that satisfy a chosen PCTL constraint. Having produced such a DTMC the sequence of actions present in the DTMC record the strategy and can readily be mapped back to the source MDP, and in turn UML model, to highlight the optimal strategy. Where multiple strategies may exist, quantitative MDP model checking methods allow for the selection of an arbitrary optimal strategy. Finally, in the case where no possible strategy exists checking will return an empty (zero state) DTMC.

The specific query to be produced to generate the possible strategies for a Core BPMN process involve defining a multiset **A** of actions which must be performed as part of the strategy, with multiple occurrences of the same element encoding that the action must be performed multiple times. Combined with a PRISM PCTL query **C** encoding constraints on the strategy, this may possibly be empty if no additional constraints are defined. Given **A** and **C**, the strategy generation query is constructed using Algorithm 1.

Algorithm 1 simply produces a PCTL query Q. When constructing Q each element $a \in \mathbf{A}$ is combined using the PCTL *until* operators, if more than one occurrence of a is present in A, and using the PCTL *finally* operator in the case when a only occurs once. For each a each of the sub-queries are combined using conjunction. Finally, each of the constraints $c \in \mathbf{C}$ are combined with Q by means of conjunction. Note that Algorithm 1 omits the addition of square and rounded brackets need to build PRISM queries and that in line 5 an additional U must be omitted on the final run of the *for* loop. Having constructed the Q a PRISM adversary generation query can be constructed simply as a multi-objective adversary serach using a query of the form `multi(` $P_1[Q], P_2[Q], \cdots, P_m[Q]$`)`, where $P_1, P_2, \cdots, P_m$ are the PCTL reward or probability operators for which the adversary is to be optimised with respect to.

Note that developing a strategy involves resolving all point of nondeterminism so actions which could potentially be included in the strategy but which are not included in the set **A**, i.e. $\mathbf{A}^{\complement}$, may, or may not, be included in the strategy, at any point, depending on what produces the optimal adversary in terms of the strategy constraints. The complexity of Algorithm 1 is $O(nm)$ where n is the cardinality of **A** and m is the is the cardinality of **C**.

Algorithm 1: PCTL strategy query generation.

```
Input: A set $\mathbf{A}$ of actions and a set $\mathbf{C}$ of constraints.
Output: A PCTL query $Q$.
forall $a \in \mathbf{A}$ do
    $m \leftarrow v(a)$                    // where $v(x)$ determines the multiplicity of $x$
    if $m > 1$ then
        for $i \leftarrow 0$ to $m$ do
            $Q \leftarrow Q \mid a \, \mathsf{U}$      // where | denotes concatenation
    else
        $Q \leftarrow Q \mid \mathsf{F}\, a$
    $Q \leftarrow Q \mid \wedge$
forall $c \in \mathbf{C}$ do
    $Q \leftarrow Q \mid \wedge c$
return $Q$
```

VI. SCHEDULE GENERATION EXAMPLE

In the case of the medical robot from Section III we may wish to create a strategy for the robot arm that would produce a sequence of drugs say three doses of drug *A*, two doses of drug *B* and one dose of drug *C*. This should be done while observing the a single safety constraint **C** that there may never be any shaking taking place while a drug is being loaded, i.e. no states where both of these properties are simultaneously true are traversed. This safety constraint set is expressed in PCTL as: $\mathsf{G}!(\mathit{[Shake]} \wedge \mathit{[Load]})$. Further, it should be noted that when producing *A* whether there is a choice between choosing *low-power* or *normal* heating choices, each with a different time and energy use trade-off, we do not constrain these and allow free choice between them in determining a strategy. The goal for the developed strategy is to observe these constraints while minimising execution time and energy used, i.e. the accumulated value of the *time* and the *energy* rewards along the chosen path are the smallest possible, with equal weight being given to minimising both rewards.

To determine the strategy, we take the annotated UML model and apply the translation approach of sketched in Section IV.A to determine the statespace of the medical robot which is shown in annotation-free form in Figure 4. The initial state is represent by the black dot and the statespace is characterised by 3 large loops which correspond to the manufacture of each of the 3 drugs. The high complexity of the manufacture of drug B is clear in the larger number of nodes and transitions that form this loop. Solution of strategy generation problems requires choosing the correct sequence of choices, marked by black triangles, to reach the strategy goals.

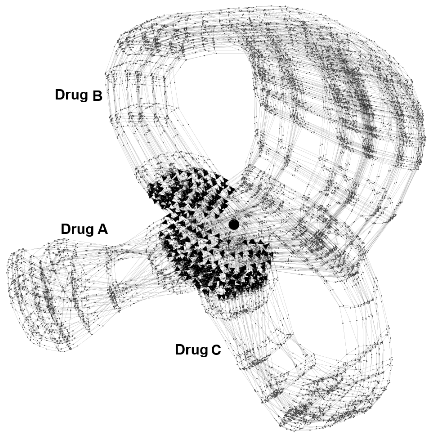

Fig. 4. Generated statespace for Section III (Annotations removed, 3080 States, 10999 Transitions).

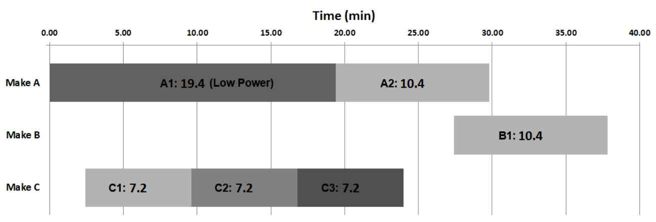

Fig. 5. Generated minimal time/energy usage strategy for the example from Section III (37.4 minutes, 98.3 kJ).

Next, to determine a strategy to meet the specific goals outlined in the previous we use PRISM to perform the following query:

$$\texttt{multi}\begin{pmatrix} \mathsf{R}_{\mathsf{min}}(\mathit{time}){=}?[(a \,\mathsf{U}\, a \,\mathsf{U}\, a) \wedge (b \,\mathsf{U}\, b) \wedge (\mathsf{F}\, c) \wedge (\mathsf{F}\, \mathit{Fail}\, \mathsf{X}\, \mathit{Reset})], \\ \mathsf{R}_{\mathsf{min}}(\mathit{cost}){=}?[(a \,\mathsf{U}\, a \,\mathsf{U}\, a) \wedge (b \,\mathsf{U}\, b) \wedge (\mathsf{F}\, c) \wedge (\mathsf{F}\, \mathit{Fail}\, \mathsf{X}\, \mathit{Reset})] \,) \end{pmatrix} \tag{3}$$

This query will explore the entire statespace shown in Figure 4, discarding patch which violate constraints or for which reward values exceed already determined minimums. Hence this approach ensure that optimal adversaries with respect to all possible orderings of the actions which must be performed as part of the strategy are generated. In this case there exists a unique strategy shown in fig. 5 with an expected mean time to completion 37.4 minutes, using 98.3 kJ. In this solution the robot chooses to begin production by manufacturing 1 dose of drug *A* and making use of the lower power heating setting in its production. Once loading is complete for drug A, manufacture of drug *C* is started and repeated until the loading of the 3rd dose of *C*. Then a second dose of *A* is started /suing normal power mode) and 2.4 minutes (the mean time needed to load B) before this is completed, production of *B* is started.

VII. CONCLUDING REMARKS

Our work presents a method to allow the automatic derivation of the optimal actions a system should perform to achieve desired goals. This can be crucial in forming system design as it suggests the fashion in which a system will be employed and can help focus testing and verification efforts. When applied to existing systems, these methods can be employed to optimise the systems' behaviour.

REFERENCES

1. J. C. Knight, "Safety critical systems: Challenges and directions", in *Software engineering, 2002. icse 2002. proceedings of the 24rd international conference on*, ser. ICSE '02, May 2002, pp. 547–550.
2. M. Kwiatkowska and D. Parker, "Advances in probabilistic model checking", in *Software safety and security - tools for analysis and verification*, T. Nipkow, O. Grumberg, and B. Hauptmann, Eds., ser. NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 33, IOS Press, 2012, pp. 126–151.
3. M. E. Fagan, "Design and code inspections to reduce errors in program development", *IBM systems journal*, vol. 15, no. 3, pp. 182–211, 1976, ISSN: 0018-8670.
4. P. P. Boca, J. P. Bowen, and J. I. Siddiqi, *Formal methods: State of the art and new directions*, First. Berlin, Heidelberg: Springer-Verlag, Sep. 2009, ISBN: 9781848827356.
5. C. Baier and J.-P. Katoen, *Principles of model checking*. Cambridge MA, USA: The MIT Press, 2008, ISBN: 9780262026499.
6. Object Management Group, "OMG Unified Modeling Language (OMG-UML), infrastructure", Object Management Group, Needham MA, USA, Standards Document formal/2011-08-05, Aug. 2011. [Online]. Available: `http://www.omg.org/spec/UML/2.4.1/`.
7. C. Lange, M. Chaudron, and J. Muskens, "In practice: UML software architecture and design description", *Ieee software*, vol. 23, no. 2, pp. 40–46, Mar. 2006, ISSN: 0740-7459.
8. L. Herbert and R. Sharp, "Model-checking business processes", in *Advances in computational sciences and information in engineering*, ser. ACIER book series, (Accepted for publication), ASME Press, 2014.
9. D. J. White, *Markov decision processes*. New Jersey, USA: John Wiley & Sons, 1993, ISBN: 9780471936275.
10. D. Parker. (Dec. 2013). Prism website, [Online]. Available: `http://www.prismmodelchecker.org/`.
11. V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker, "Automated verification techniques for probabilistic systems", in *Formal methods for eternal networked software systems*, ser. Lecture Notes in Computer Science, M. Bernardo and V. Issarny, Eds., vol. 6659, Berlin, Heidelberg: Springer-Verlag, 2011, pp. 53–113, ISBN: 9783642214547.
12. V. Forejt, M. Kwiatkowska, G. Norman, D. Parker, and H. Qu, "Quantitative multi-objective verification for probabilistic systems", in *Tools and algorithms for the construction and analysis of systems*, ser. Lecture Notes in Computer Science, P. A. Abdulla and K. M. Leino, Eds., vol. 6605, Springer Berlin Heidelberg, 2011, pp. 112–127, ISBN: 978-3-642-19834-2.
13. V. Forejt, M. Kwiatkowska, and D. Parker, "Pareto curves for probabilistic model checking", in *Automated technology for verification and analysis*, ser. Lecture Notes in Computer Science, S. Chakraborty and M. Mukund, Eds., Springer Berlin Heidelberg, 2012, pp. 317–332, ISBN: 978-3-642-33385-9.
14. J. Lilius and I. P. Paltor, "Formalising UML state machines for model checking", in *≪uml≫'99 - the unified modeling language*, ser. Lecture Notes in Computer Science, R. France and B. Rumpe, Eds., vol. 1723, Berlin, Heidelberg: Springer-Verlag, 1999, pp. 430–444, ISBN: 9783540667124.
15. C. Canevet, S. Gilmore, J. Hillston, M. Prowse, and P. Stevens, "Performance modelling with the unified modelling language and stochastic process algebras", *Iee proceedings - computers and digital techniques*, vol. 150, no. 2, pp. 107–120, Mar. 2003, ISSN: 1350-2387.
16. D. N. Jansen and H. Hermanns, "QoS modelling and analysis with UML-statecharts: The stocharts approach", *ACM SIGMETRICS performance evaluation review*, vol. 32, pp. 28–33, 2005.
17. A. J. Wijs, J. C. van de Pol, and E. M. Bortnik, "Solving scheduling problems by untimed model checking: The clinical chemical analyser case study", *International journal on software tools for technology transfer*, vol. 11, no. 5, pp. 375–392, Oct. 2009, ISSN: 1433-2779.
18. A. Basu, S. Bensalem, D. Peled, and J. Sifakis, "Priority scheduling of distributed systems based on model checking", *Formal methods in system design*, vol. 39, pp. 229–245, 3 2011, ISSN: 0925-9856.
19. T. C. Ruys, "Optimal scheduling using branch and bound with spin 4.0", English, in *Model checking software*, ser. Lecture Notes in Computer Science, T. Ball and S. K. Rajamani, Eds., vol. 2648, Springer Berlin Heidelberg, 2003, pp. 1–17, ISBN: 978-3-540-40117-9.
20. G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager, "Minimum-cost reachability for priced time automata", English, in *Hybrid systems: Computation and control*, ser. Lecture Notes in Computer Science, M. Benedetto and A. Sangiovanni-Vincentelli, Eds., vol. 2034, Springer Berlin Heidelberg, 2001, pp. 147–161, ISBN: 978-3-540-41866-5.
21. F. Giunchiglia and P. Traverso, "Planning as model checking", in *Recent advances in ai planning*, ser. Lecture Notes in Computer Science, S. Biundo and M. Fox, Eds., vol. 1809, Springer Berlin Heidelberg, 2000, pp. 1–20, ISBN: 978-3-540-67866-3.
22. M. von der Beeck, "A structured operational semantics for UML-statecharts", *Journal of software and systems modeling*, vol. 1, no. 2, pp. 130–141, 2 Dec. 2002, ISSN: 1619-1366.
23. J. Jürjens, "A UML statecharts semantics with message-passing", in *Proceedings of the 2002 ACM symposium on applied computing*, ser. SAC '02, New York, NY, USA: ACM, 2002, pp. 1009–1013, ISBN: 1-58113-445-2.
24. M. Kwiatkowska, G. Norman, and D. Parker, "Symmetry reduction for probabilistic model checking", in *Proceedings of 18th international conference on computer aided verification (cav'06)*, T. Ball and R. Jones, Eds., ser. Lecture Notes in Computer Science, vol. 4114, London, UK: Springer-Verlag, 2006, pp. 234–248.
25. M. Größer, G. Norman, C. Baier, F. Ciesinski, M. Kwiatkowska, and D. Parker, "On reduction criteria for probabilistic reward models", in *Proceedings of the 26th international conference on foundations of software technology and theoretical computer science*, ser. FSTTCS'06, Berlin, Heidelberg: Springer-Verlag, 2006, pp. 309–320.