A MODULAR AND HIERARCHICAL MODELLING APPROACH FOR STOCHASTIC CONTROL

Alexander Gouberman, Martin Riedl, Markus Siegle
Department of Computer Science
Bundeswehr University Munich, Germany
email: {alexander.gouberman, martin.riedl, markus.siegle}@unibw.de

ABSTRACT
We propose a novel modelling concept for stochastic control on systems which are hierarchically composed of subsystems with discrete states and stochastic continuous time. The global control structure (called decision tree) is based on the model hierarchy and can be defined by an interleaving of local control with concurrency. For model specification we review the language LARES which comprises an object-oriented modelling design in order to specify modular and hierarchically structured stochastic systems. In order to embed the control structure into the LARES framework we describe the language extension LARES.de. The main focus of the paper is a transformation to a Markov Decision Process induced by an agent-based view on the control structure. This defines the concrete language semantics and makes state-based system optimization accessible.

KEY WORDS
stochastic control, stochastic modelling, dynamic modelling, MDP, dependability model, fault tolerance

1 Introduction

In highly dependable or safety-critical systems, where components are subject to failures, the question of optimal control arises. In which situations should preventive maintenance be performed? If several components have failed, which one should be repaired first in order to maximize the lifetime of the overall system? Markov decision processes (MDP) constitute a suitable domain in which such problems can be treated. However, existing modelling formalisms for MDPs are not suited very well to specify large reconfigurable systems, where many components evolve concurrently with the possibility of different interaction patterns among them.

Therefore, in this paper, we present a modular and hierarchical modelling formalism, which enables its users to specify models of even very complex real systems in a clear and concise way. Our new formalism is an extension of the existing LARES specification language [**?**], which up to now was restricted to the modelling of fully stochastic systems. We chose LARES as a starting point since it implements an object-oriented modelling concept, separating abstract descriptions from their concrete instantiations, and using modularity and hierarchy together with scoping and interface-based information flow mechanisms. Furthermore, the LARES framework is very flexible, easily enabling different kinds of transformation and language extensions. In this paper, we present our own extension, called LARES.de, by introducing decisions which interleave with the information flow and conserve modularity. We also present a semantics which transforms LARES.de models to an MDP.

The remainder of the paper is organized as follows: In Section 2 we review the LARES language and construct a running example step by step. Section 3 describes the semantics of LARES by transformation to a Markov chain. We introduce the concept of a forward tree which incorporates the information flow through the model. The main part in Section 4 describes the extension LARES.de and its transformation to an MDP. We show a novel hierarchical concept called decision tree which describes control and reactions in an interleaving manner. Section 5 briefly outlines related work and Section 6 concludes the paper.

2 The LARES formalism

A typical LARES model consists of `Behavior` and `Module` definitions: In general a `Behavior` describes a single automaton with finite state space, whereas a `Module` can instantiate `Behaviors` and define interaction between them. A `Behavior` B is defined by a discrete set of states S^B and and two types of labelled transitions

$$s \xrightarrow{g}_{\lambda} s' \quad \text{and} \quad s \overset{g}{\dashrightarrow}_{w} s'$$

where $s, s' \in S^B, g \in L^B$ is a guard label, $\texttt{true} \in L^B$ is a special label, $w > 0$ represents the weight of a Dirac transition which fires immediately and $\lambda > 0$ the rate parameter of an exponential distribution specifying the positive delay of the transition. In order to treat different transition distribution types homogenously, we define the set of all transitions T^B containing tuples (s, g, δ, s'), where $\delta \in \{\text{Dirac}(w), \text{Exp}(\lambda)\}$ denotes the distribution of the g-labelled transition from s to s'. Define further

$$S^B(g) := \left\{ s \mid \exists s' \, \exists \delta : \ (s, g, \delta, s') \in T^B \right\}$$

as states in S^B from which a g-labelled transition can take place, regardless of the transition type.

```
Behavior B(mu) {
  State A, F, T
  Transitions from A
    if <true> → F, delay exponential 0.1
  Transitions from F
    if <rep> → T, delay exponential mu
  Transitions from T
    if <true> → A, weight 9.0
    if <true> → F, weight 1.0
}
```

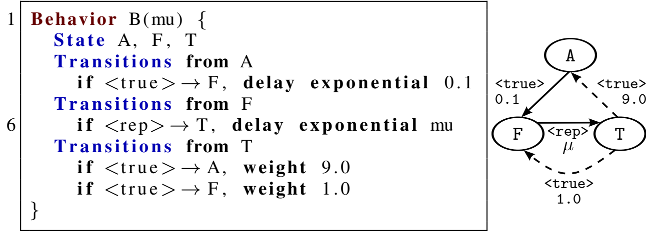

Figure 1. Definition of a Behavior and its automaton representation. Solid lines indicate transitions with exponential delay, whereas dashed lines indicate Dirac transitions.

As an example consider the Behavior definition in Figure 1 describing a component of a system which can be active (A), failed (F) or in the test state (T). If the component is active it can fail after an exponentially distributed time with rate 0.1. From the failed state the component can be repaired by triggering the guard label <rep> which takes an exponentially distributed time with parametrized rate mu moving the Behavior B to the test state in order to check if a repair was successful or not. From the local viewpoint of the Behavior B a repair is successful with probability 0.9, which is computed by normalizing the weights from the state T to define a probability distribution over states.

In order to build up a system specification out of Behavior definitions one can define an abstract Module definition which instantiates a set of Behaviors. Furthermore a Module can instantiate other Module definitions. In this way it is possible to establish a hierarchical tree structure of the model specification where the leaves of the hierarchy are given by instantiated Behaviors. The root module definition in this hierarchy has to be declared by a System keyword. As an example consider a system main which consists of three components C[1], C[2] and C[3], where C[1] and C[2] inherit their state space from Behavior B and C[3] is a container component which comprises two subcomponents SC[1] and SC[2] (which also inherit from B). Figure 2 shows a possible specification of this hierarchical model with LARES.

Since LARES is a specification language suited especially for hierarchically structured systems, it comes along with scoping constraints on the information flow through the model hierarchy. For this reason one can make assertions about states by logical expressions:

- Condition statements can be used in order to combine, aggregate and lift states or other conditions towards the root level of the system hierarchy.
- guards and forward statements can be used in order to trigger guarded transitions of instantiated Behavior definitions. A guards statement represents the entry point of a triggering event determined by a condition and forward statements propagate the event towards the addressed Behavior instances on the leaves.

```
System main {
  Instance C[1] of Comp(2.0) initially active
  Instance C[2] of Comp(5.0) initially active
  Instance C[3] of Cont(3.0) initially active

  Condition sysF = C[1].F & C[2].F & C[3].F

  C[1].F & !sysF guards C[1].<repair>
  C[2].F & !sysF guards C[2].<repair>
  C[3].F & !sysF guards C[3].<repair>
}

Module Comp(reprate) : B(mu=reprate) {
  Condition F = B.F

  Initial active = B.A

  forward <repair> to B.<rep>
}

Module Cont(reprate) {
  Instance SC[1] of Comp(reprate=reprate)
  Instance SC[2] of Comp(reprate=reprate)

  Condition F = SC[1].F | SC[2].F

  Initial active = SC[1].active, SC[2].active

  forward <repair> {
    if SC[1].F to SC[1].<repair>
    if SC[2].F to SC[2].<repair>
  }
}
```

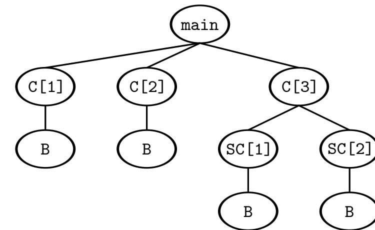

Figure 2. A hierarchically structured LARES model (top) with information flow through the instance tree (bottom): Behavior states are lifted towards the root main via Condition statements whereas guard labels are pushed towards the instantiated Behaviors (leaves) via forward statements.

In our running example (Figure 2) we specified a system which fails if all components C[1], C[2] and C[3] fail (described by the condition sysF). The container itself is a (series) system which fails if at least one of its subcomponents is failed. Initially all components are active. The guards statements are used in order to specify a repair process for failed components. In general, guards statements can be used in order to define interaction between instances by specifying a dependency for guarded transitions of one instance to states of other instances. In our case the label <repair> will be forwarded towards the addressed Behavior instance via forward statements. As specified above, if two components fail a race is induced between two concurring repair processes, meaning that both components are repaired in parallel.

3 Semantics of LARES

In the following, we describe the semantics of a LARES model by transformation to a Markov chain. [?] describes the whole transformation workflow which is implemented in Scala. This workflow consists of a sequence of transformation steps which roughly speaking resolve the model hierarchy, leading to a flat version of the LARES model. In this way

- each `Behavior` definition residing at a leaf of the instance tree is instantiated: by enumerating the leaves with $i = 1, \ldots, n$, a `Behavior` instance $B_i = (fqn, B)$ is created and characterized by its unique fully qualified name fqn induced by hierarchical instantiation (e.g. `main.C[3].SC[1].B`) and a reference to its `Behavior` definition B,
- all `guards` and `forward` statements are resolved to a set of `guards` statements (i.e. interactions, cf. Section 3.1) which directly trigger referenced guard labels in `Behavior` instances B_i.

Instead of resolving the `guards` and `forward` statements directly, we add in this paper an intermediate transformation step by defining a so called “forward tree”. The LARES.de extension in Section 4 will build upon this notion.

In the following we will use for shorthand notation $B_i.g$ resp. $B_i.s$ in order to reference a guard label $g \in L^B$ resp. a state $s \in S^B$ of a `Behavior` instance $B_i = (fqn, B)$. We define similarly $L^{B_i} := \{B_i.g \mid g \in L^B\}$ and $\widehat{L} := \bigcup_{i=1}^{n} L^{B_i}$ as the set of referenced guard labels for the instance B_i resp. the set of all referenced guard labels in the LARES model. Analogously, let $S^{B_i} := \{B_i.s \mid s \in S^B\}$ and $\widehat{S} := \prod_{i=1}^{n} S^{B_i}$ (i.e. the cross product of all instantiated `Behaviors`) be the composed potential state space of the LARES model.

3.1 Forward tree

A `guards` or `forward` statement can be seen as a set of conditional reactions (C, R), where C is a logical expression on states of `Behavior` instances representing a Boolean function $C : \widehat{S} \to \mathbb{B}$. The reaction R represents an effect on other `forward` labels or guard labels $\widehat{L}$ which finally can influence state transitions. In our example (Figure 2) the `forward` statement in lines 29..32 creates the conditional reactions `(SC[1].F, SC[1].<repair>)` and `(SC[2].F, SC[2].<repair>)`. More generally, since it is also possible to model label synchronisation with LARES, R is an abstract syntax tree with terminal nodes representing labels and inner nodes representing operators on labels. Typically, label operators are given by different views on synchronisation of transitions. So far the operators `sync`, `maxsync` and `choose` have been implemented [?]. For simplicity and since the label operators are out of scope of this paper, we will not enlighten the concrete semantics of these operators in this paper in full detail. In order to keep the following algorithms as simple as possible we only deal with the synchronisation operator `sync`. We will not use any label operators in examples in this paper, s.t. the algorithms can be understood more easily.

Roughly speaking a “forward tree” $\mathcal{F}$ incorporates the reaction flow in the instance tree towards `Behavior` instances. Figure 3 shows the forward tree for our running example.

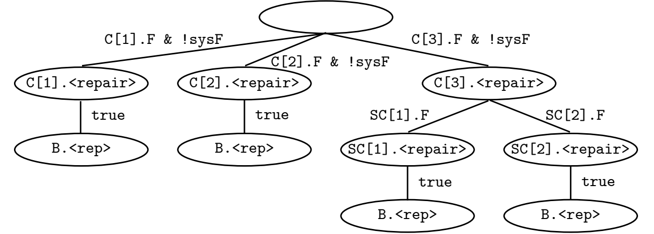

Figure 3. Forward tree for the LARES example model

A forward tree $\mathcal{F}$ defines in its leaf nodes possible transitions (in `Behavior` instances), which can be guarded by some guard label. The inner nodes can have different node types: branching nodes coming from `forward` statements and synchronisation nodes coming from a `forward to sync(...)` statement. The root node is of branching type and realizes all `guards` statements. Some subtrees of $\mathcal{F}$ describe an “interaction” between `Behavior` instances, comprising branching or synchronisation of conditional reactions. The conditions on edges of these subtrees represent restrictions of interactions. If there are no synchronisation nodes these subtrees are just paths in $\mathcal{F}$.

Definition (Interaction): Let $\mathcal{B} = \{B_i \mid i = 1, \ldots, n\}$ be the set of `Behavior` instances of a LARES model with $\widehat{S} = \prod_{i=1}^{n} S^{B_i}$ and $\widehat{L} = \bigcup_{i=1}^{n} L^{B_i}$. An interaction on $\mathcal{B}$ is a tuple $\gamma = (C, R)$ where $C : \widehat{S} \to \mathbb{B}$ represents a logical expression and $R \subseteq \widehat{L}$ is a set of guard labels.

Note that we use the same notation (C, R) for conditional reactions and interactions, since subtrees of $\mathcal{F}$ consisting of conditional reactions (represented as nodes and edges) are “flattened” into interactions. Roughly speaking an interaction describes a transition on the composed potential state space $\widehat{S}$: The conditional part C represents a constraint on composed source states while the reactive part $R \subseteq \widehat{L}$ describes a synchronous transition on a subset of `Behavior` instances to a composed target state.

Reachability analysis needs to be performed, in order to generate the state graph. In a first step we “flatten” $\mathcal{F}$ to a set $\mathcal{I}^{grd}$ of (guarded) interactions. The second step performs a depth-first search on the composed potential state space $\widehat{S}$ for generating the reachable subset of $\widehat{S}$ by using $\mathcal{I}^{grd}$ together with the concurrent unguarded process.

3.2 Flattening the forward tree

The detailed flattening procedure is described in Algorithm 1 and returns a set of (guarded) interactions

$$\mathcal{I}^{grd} := \textsc{Flatten}(\mathcal{F}).$$

Algorithm 1 Flattening the forward tree

Input: subtree T of the forward tree $\mathcal{F}$
Output: interactions $\mathcal{I}^{grd} = \{(C_k, R_k) \mid k \in K\}$

1: **function** $\textsc{Flatten}(T)$
2: $v :=$ root of T
3: **if** v is terminal **then**
4: let $B_i.g$ be the guard label referenced by v
5: $M := \{(C, B_i.g) \mid C \in S^{B_i}(g)\}$
6: **else**
7: **for all** children v' of v **do**
8: $C :=$ condition on edge (v, v')
9: $T_{v'} :=$ subtree of T with root v'
10: $M_{v'} := \textsc{Flatten}(T_{v'})$
11: $M_{v'} := \{(C \wedge C_k, R_k) \mid (C_k, R_k) \in M_{v'}\}$
12: **if** v is branching node **then**
13: $M := \bigcup \{M_{v'} \mid v' \text{ child of } v\}$
14: **else if** v is synchronisation node **then**
15: $P := \prod \{M_{v'} \mid v' \text{ child of } v\}$
16: $M := \bigcup \{\textsc{And}(\gamma) \mid \gamma \in P\}$
17: $M := \{(C, R) \in M \mid C \not\equiv false\}$ ▷ filter M
18: **return** M

In order to explain this algorithm, let's assume for the moment that there are no synchronisation nodes, i.e. all inner nodes of $\mathcal{F}$ are branching nodes. In this case for each leaf node of $\mathcal{F}$ an interaction $\gamma = (C, R)$ is generated as follows: In lines 3..5 the referenced guard label $B_i.g$ in each leaf node is resolved by extracting $S^{B_i}(g)$ (i.e. source states of `Behavior` instance B_i under guard label g) into conditions C. The reactive part consists only of the guard label $B_i.g$ itself. In lines 7..11 each parent node v conjuncts to the conditional part C_k of its children v' the condition C residing on the edge (v, v') and leaves the reactive part R_k untouched. In the case that for each referenced guard label $B_i.g$ there is only one source state (i.e. $|S^{B_i}(g)| = 1$), each leaf node would generate exactly one interaction. In general $|\mathcal{I}^{grd}|$ is the sum over all $|S^{B_i}(g)|$ with $B_i.g$ a referenced guard label (if there are no synchronisation nodes). Applying the flattening procedure to our running example we get (modulo namespace, resolution of `Condition` statements and tautological simplifications of the logical expression C):

(`SC[1].F & (C[3].F & !sysF)`,
{`main.C[3].SC[1].B.<rep>`}) $\in \mathcal{I}^{grd}$

In order to deal with synchronisation nodes we lift the conjunction of logical expressions on interactions: Let $\gamma = (\gamma_1, \ldots, \gamma_n)$ with $\gamma_i = (C_i, R_i)$. Define

$$\textsc{And}(\gamma) := \{(C_1 \wedge \cdots \wedge C_n, R_1 \cup \cdots \cup R_n)\}.$$

A synchronisation of two guard labels $B_1.g_1$ and $B_2.g_2$ induces a synchronous state transition of both instances B_1 and B_2 if and only if both of them are able to trigger the guards g_1 resp. g_2. This fact is realized by applying $\textsc{And}$ on both interactions $\gamma_i = (C_i, \{B_i.g_i\})$. In this case the reactive part is the set $\{B_1.g_1, B_2.g_2\}$ which is responsible for the synchronicity of the transition.
In order to handle the concurrent unguarded process, i.e. transitions in `Behavior` instances triggered by the guard label `<true>`, we define the unguarded interactions

$$\mathcal{I}^{ungrd} := \bigcup_{i=1}^{n} \{(C, R) \mid C \in S^{B_i}(\texttt{true}), R = \{B_i.\texttt{true}\}\}$$

and the complete set of interactions as

$$\mathcal{I} := \mathcal{I}^{grd} \cup \mathcal{I}^{ungrd}.$$

3.3 Reachability for LARES

Each interaction $\gamma = (C, R) \in \mathcal{I}$ is resolved to a set of transitions between composed states $s \in \widehat{S}$ as described in Algorithm 2. If $s = (s_1, \ldots, s_n)$ satisfies C then a guard label $B_i.g \in R$ induces a set of transitions from $s_i \in S^{B_i}$ to some state $s'_i \in S^{B_i}$ with distribution type $\delta_i \in \{\text{Dirac}(w), \text{Exp}(\lambda)\}$. All instances which do not synchronize remain in their state and are assigned an unspecified distribution NA. Function $\textsc{syncDistr}(\delta_1, \ldots, \delta_n)$ computes the "joint" (delay) distribution for the synchronized state transition out of component-wise distributions δ_i. Since synchronisation is out of scope in this paper we omit the concrete definition for $\textsc{SyncDistr}$. Originally the LARES semantics is based on the stochastic process algebra CASPA [**?**] and therefore carries over its synchronisation semantics [**?**].
The reachability analysis (cf. Algorithm 3) filters out all transitions with unspecified distributions NA, since they do not contribute to a fully specified state transition. Lines 5..6 describe the so-called maximal progress assumption [**?**, **?**], which indicates that Dirac transitions are not delayed and therefore any concurring (exponentially) delayed transition is discarded from s. Finally all remaining transitions are applied to the state s and s is marked as done and the reachability recurses on states on the fringe of the search space.
As an example, Figure 4 shows the application of the reachability algorithm to our running example (Figure 2) from the viewpoint of the modeller. Here a state s is encoded by:

$$s = (s_1, s_2, s_3, s_4) \in S^{\mathsf{C[1]}} \times S^{\mathsf{C[2]}} \times S^{\mathsf{SC[1]}} \times S^{\mathsf{SC[2]}},$$

e.g. $s = (\mathtt{A}, \mathtt{F}, \mathtt{F}, \mathtt{A})$ represents that the components `C[1]` and `SC[2]` are active whereas `C[2]` and `SC[1]` are already failed. From this state the conditions of two `guards`

Algorithm 2 Generation of transitions

Input: state $s = (s_1, \dots, s_n)$, interaction $\gamma = (C, R)$
Output: set of transitions between composed states

1: **function** GENERATETRANSITIONS(s, γ)
2: **if** s satisfies C **then**
3: **for** $i = 1, \dots, n$ **do**
4: **if** $\exists B_i.g \in R$ **then**
5: $t_i = \left\{(s'_i, \delta_i) \mid (s_i, g, \delta_i, s'_i) \in T^{B_i}\right\}$
6: **else** $t_i := \{(s_i, NA)\}$
7: $t := \left\{ (s', \delta) \;\middle|\; \begin{array}{l} s' = (s'_1, \dots, s'_n), (s'_i, \delta_i) \in t_i, \\ \delta = \text{SYNCDISTR}(\delta_1, \dots, \delta_n) \end{array} \right\}$
8: **else** $t := \emptyset$

Algorithm 3 Reachability

Input: (initial) state $s = (s_1, \dots, s_n)$, interactions $\mathcal{I}$
Output: reachable state space

1: **function** REACHABILITY($s, \mathcal{I}$)
2: mark s as done
3: $t := \bigcup_{\gamma \in \mathcal{I}}$ GENERATETRANSITIONS(s, γ)
4: $t := \{(s', \delta) \in t \mid \delta \neq NA\}$
5: $t^{Dirac} = \{\ (s', \delta) \in t \mid \delta = \text{Dirac}(w)\ \}$
6: **if** $t^{Dirac} \neq \emptyset$ **then** $t := t^{Dirac}$
7: $M := \{s\} \times t$
8: $N := \{s' \mid (s', \delta) \in t,\ s' \text{ not done}\}$
9: **return** $M \cup \bigcup_{s' \in N}$ REACHABILITY($s', \mathcal{I}$)

statements are satisfied which finally trigger the guard label `<rep>` in the referenced `Behavior` instances. (This corresponds to a subset of paths of the forward tree $\mathcal{F}$ (as in Figure 3).) By triggering these labels the corresponding `Behavior` instances can move to the test state `T`. Additionally there is a concurrent unguarded process running in which the two active components `C[1]` and `SC[2]` can still fail. Figure 4 bottom shows all potential transitions from the state `(T,T,F,F)`. In this case, by the maximal progress assumption, the exponentially delayed repair transitions will not be possible.
As a last step, the result of the reachability algorithm is transformed to a DTMC or a CTMC. The reachability returns the (reachable) state space $S \subseteq \widehat{S}$ with exponentially delayed transitions and immediate Dirac transitions between composed states. To obtain a DTMC, all exponentially delayed transitions can be embedded or uniformized and all weights of Dirac transitions normalized to discrete probability distributions [**?**]. E.g. in Figure 4 bottom a transition to the state `(A,T,F,F)` will be taken with probability 0.45. For transforming the reachability result to a CTMC, vanishing states have to be eliminated leading to a smaller state space [**?**, **?**].

4 Decision Extension: LARES.de

As we have seen from the running example in the last chapter, a failed component will be immediately repaired, regardless of when the component failed or how many other

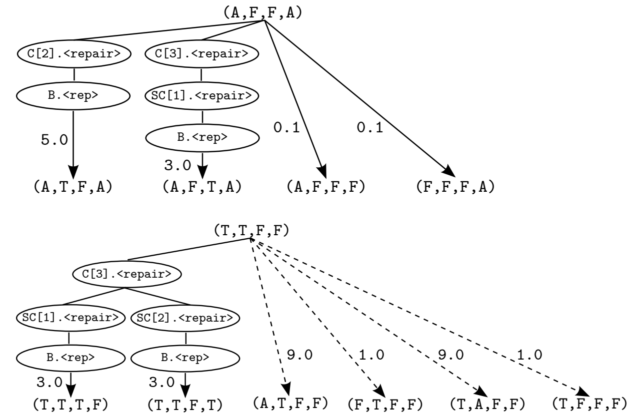

Figure 4. Forwarding labels through the model hierarchy for the running example model. Arrows with solid lines indicate exponential transitions and dashed lines indicate Dirac transitions.

components need to be repaired in parallel. Assume now that we want to model that only one component can be repaired at a time. Furthermore, in a state in which several components are failed we want to find an optimal repair assignment: Which component shall be repaired first in order to optimize some system measure (e.g. the life time of the system)? These and similar kinds of problems can be typically modelled by Markov Decision Processes, which is a well-established formalism for state-based optimization.

Definition (MDP): A Markov Decision Process (MDP) is a structure (S, Act, e, P, ν, R) with discrete state space S, initial distribution ν over S, discrete action set Act with enabling function $e(s) \subseteq Act$ for a state s, transition probability $P(s' \mid s, a)$ for moving from s by taking action $a \in e(s)$ to s' and gaining a reward $R(s, a) \in \mathbb{R}$.

The semantics of an MDP can be roughly described through the view of an agent who, being in state s, decides for action a in order to optimize some objective. Depending on the type of his objective he can use the reward in order to evaluate his chosen action a. Typical types for objectives coming along with MDPs can be classified by the length of the horizon (finite or infinite) and the type of reward accumulation (discounted, undiscounted or average) [**?**]. There is a plethora of optimization methods which can be roughly classified in dynamic programming (value iteration and policy iteration), linear programming and reinforcement learning [**?**, **?**].
Since LARES offers modularity and hierarchy concepts it is well-suited to be extended towards MDPs. Due to the exponential delays used in the LARES language, one would obtain a continuous-time model, which however can be mapped on a discrete-time MDP. In the following, we extend LARES with an action structure.
In order to specify actions, we enrich the forwarding

mechanism of LARES by adding the modifier keyword `decision` to `guards` and `forward` statements. In this way the modeller can define a control structure on the level of modules. In order to show the arising semantics we modify our running example as shown in Figure 5. From this specification a so-called "decision tree" $\mathcal{D}$ is constructed as can be seen in Figure 6.

```
System main {
  ...
  Condition oneF = C[1].F | C[2].F | C[3].F

  oneF & !sysF guards decision <dorep>
  oneF & !sysF guards decision <norep>

  forward <dorep> {
    if C[1].F to decision <repC[1]>
    if C[2].F to decision <repC[2]>
    if C[3].F to decision <repC[3]>
  }

  forward <norep> {}

  forward <repC[1]> to C[1].<repair>
  forward <repC[2]> to C[2].<repair>
  forward <repC[3]> to C[3].<repair>
}

Module Comp(reprate) : B(mu=reprate) {
  ...
  forward <repair> to B.<rep>
}

Module Cont(reprate) {
  ...
  forward <repair> {
    to decision <repSC[1]>
    to decision <repSC[2]>
  }

  forward <repSC[1]> if SC[1].F to SC[1].<rep>
  forward <repSC[2]> if SC[2].F to SC[2].<rep>

}
```

Figure 5. The running example extended by decisions

A decision tree extends the notion of forward tree with a new type of edges representing decisions. The (standard) edges carried over from the forward tree will be called concurrent edges.
Interpretation of the decision tree in Figure 6: If the system is working and some component `C[i]` is failed (the condition `oneF & !sysF` is satisfied) the decisions `<dorep>` and `<norep>` get activated. In contrast to concurrent edges which roughly speaking (finally) induce a probabilistic choice between the outgoing paths in $\mathcal{D}$, these decisions behave non-deterministically. This means that from the perspective of an agent residing in some system state and faced with a set of decisions, he has to "decide" which path in $\mathcal{D}$ to follow. In case the subtree with root `<dorep>` is chosen and depending on which component `C[i]` has failed, the decisions `<repC[i]>` are activated. In this way the control for the repair process is refined, i.e. the agent has to decide as next which of the components `C[i]` shall be repaired. The chosen label `<repC[i]>` is forwarded by `C[i].<repair>` from the `main` module to

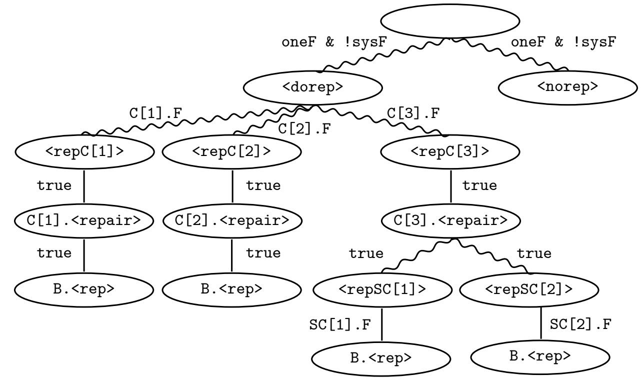

Figure 6. Decision tree $\mathcal{D}$ for the LARES.de running example. Concurrent edges are drawn by solid lines whereas decisions are represented by curly lines.

the component modules `Comp` resp. `Cont`, which can define on their own a control process, as it is done in the container module `Cont`. In this way the modularity aspects of LARES can be used in order to refine decisions inside submodules.
Normally, the decision to repair one of the components `SC[i]` should be only possible if the component itself has failed. However, the corresponding conditions `SC[i].F` can be found in Figure 5 lines 33..34 and not in lines 29..30 as one would expect. This is due to didactical reasons only, in order to show a semantical difference in both specification possibilities. Let's assume that `SC[1]` has failed and `SC[2]` is active. As specified, both decisions `<repSC[1]>` and `<repSC[2]>` get activated. If the agent chooses `<repSC[1]>` then the guard label `SC[1].B.<rep>` will be finally triggered since the condition `SC[1].F` is satisfied and a state transition for repair can take place. However a choice for `<repSC[2]>` will not induce an additional state transition since `SC[2].F` is not satisfied. This means that by choosing `<repSC[2]>` in such a state, only the concurrent unguarded process could take place (i.e. some other component fails). In the other case, if both of these conditions were specified in lines 29..30, as expected, then `<repSC[2]>` would not be selectable for the agent.
In order to define the concrete semantics of LARES.de we transform it to an MDP. Therefore we derive from the decision tree a set of actions $a \in Act$ together with their enabling function $e : S \to \mathcal{P}(Act)$. Roughly speaking an action a is represented as a subtree R of $\mathcal{D}$ s.t. each inner node v in R has at most one outgoing decision. We explain the transformation by dividing it into the 3 steps: completion, splitting and enabling.

4.1 Completion

This step is a sort of preprocessing for the main splitting part. Consider in our example that we change the condition `oneF` to be `true` in every composed state. Then from

state (A,A,A,A) the agent could still choose the decision <dorep>, but no subdecision <repC[i]> would be activated. Therefore the agent would be stuck in the node <dorep> and wait until some unguarded transition happens. In order to handle such cases in the splitting part consistently we compute the so-called "completion" $\mathcal{D}^c$ of the decision tree $\mathcal{D}$. This means that we append to each inner node a child node <noDec> which incorporates the possibility to choose no decision. Thereby the agent can decide for <noDec> if and only if there is no other decision activated: For a non-terminal node v of $\mathcal{D}$ with children v_k $(k \in K)$, reached by some decision edge labelled with the condition C_k, define

$$c := \bigwedge_{k\in K} \overline{C_k} = \overline{\bigvee_{k\in K} C_k}$$

(bar indicates logical negation equivalent to the LARES !-operator) as the condition of the decision edge $(v, <\texttt{noDec}>)$. Figure 7 illustrates the completion procedure by example.

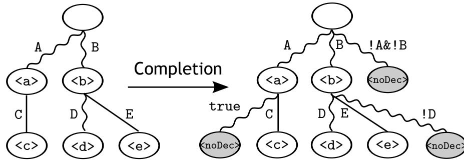

Figure 7. Completion of a decision tree: Each inner node gets a <noDec> child completing the logical expressions for decisions to $true$.

4.2 Splitting

The splitting of the completed decision tree $\mathcal{D}^c$ to a set of split decision subtrees R of $\mathcal{D}^c$ is shown in Algorithm 4.
General idea behind this algorithm: The semantics distinguishes between concurrent edges and decision edges in the following way: If an agent resides in some node v and is faced with a set of decisions then the current subtree T (with root v) is split into a set of trees, s.t. there is exactly one outgoing decision edge from v. Therefore all already split subtrees $R_{v'}$ (of a child v' of v) which are reached by some decision edge are collected into a set $\mathbf{D}$ (lines 10..11). All concurrent edges induce a branching, since there is no decision yet visible and might possibly appear later (in some subtree of $\mathcal{D}^c$). Since the splitting of concurrent edges is left yet open we take the product $\mathbf{C}$ over all these concurrent subtrees (lines 12..13). Therefore each combination $p \in \mathbf{P} = \mathbf{D} \times \mathbf{C}$ (line 14) consists of exactly one subtree which is directly reached by a decision edge from v and a set of subtrees reached by some concurrent edge.

Algorithm 4 Splitting of a decision tree $\mathcal{D}$

Input: subtree T of a decision tree $\mathcal{D}$
Output: set of subtrees R of the decision tree $\mathcal{D}$

1: **function** SPLIT(T)
2: $v :=$ root of T
3: **if** v is terminal **then**
4: $R := \{v\}$
5: **else**
6: $\mathbf{D} := \emptyset,\ \mathbf{C} := \{()\}$
7: **for all** children v' of v **do**
8: $T_{v'} :=$ subtree of T with root v'
9: $R_{v'} :=$ SPLIT($T_{v'}$) ▷ $R_{v'}$ is a set of split trees
10: **if** (v, v') is decision edge **then**
11: $\mathbf{D} := \mathbf{D} \cup R_{v'}$
12: **else** $((v, v')$ is concurrent edge)
13: $\mathbf{C} := \mathbf{C} \times R_{v'}$
14: $\mathbf{P} := \mathbf{D} \times \mathbf{C}$
15: $R :=$ append each element of $\mathbf{P}$ as a subtree to v
16: carry over all node and edge labels from T to R
17: **return** R

Definition (Action set): Let $\mathcal{D}$ be a decision tree of a LARES.de model and $\mathcal{D}^c$ its completion. For the transformation towards an MDP we define the action set

$$Act := \text{SPLIT}(\mathcal{D}^c).$$

An action $a \in Act$ is represented (by recursion) as a subtree of $\mathcal{D}^c$ s.t. from each inner node there is exactly one outgoing decision.
In order to understand this algorithm in more detail we show its application on two examples. We denote a tree T with root v and children $v'_1, \ldots, v'_n$ recursively through the linearized notation $v(v'_1, \ldots, v'_n)$. In the following examples we didn't make explicit which of the nodes are <noDec> nodes, but we assume the trees are completed.

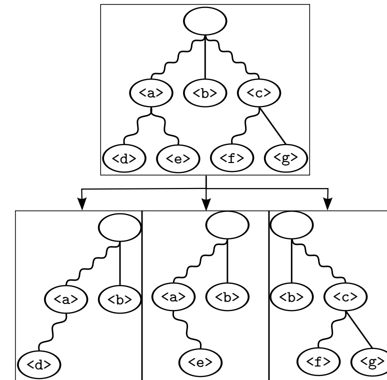

Figure 8. Splitting example 1: A completed decision tree $\mathcal{D}^c$ splits into 3 actions

Example 1: (cf. Figure 8) Being in some state, an agent is confronted with decisions <a> and <c>, where decision <a> refines into the subdecisions <d> and <e>. Therefore

calling SPLIT on $T = \texttt{<a>}(\texttt{<d>}, \texttt{<e>})$ returns the split subtrees

$$R_{\texttt{<d>}} = \{\texttt{<d>}\} \text{ and } R_{\texttt{<e>}} = \{\texttt{<e>}\}\,.$$

Since both nodes `<d>` and `<e>` are reached by a decision it follows that $\mathbf{D} = R_{\texttt{<d>}} \cup R_{\texttt{<e>}} = \{\texttt{<d>}, \texttt{<e>}\}$ and $\mathbf{C} = \{()\}$ is the set containing the 0-tuple (representing an empty product). The split trees computes to $\mathbf{P} = \mathbf{D} \times \mathbf{C} = \{(\texttt{<d>}), (\texttt{<e>})\}$ and each of them is appended to `<a>`. On the root level (calling SPLIT on $T = \mathcal{D}^c$) we get similarly the split trees

$$R_{\texttt{<a>}} = \{\texttt{<a>}(\texttt{<d>}), \texttt{<a>}(\texttt{<e>})\}\,,$$
$$R_{\texttt{<b>}} = \{\texttt{<b>}\} \text{ and } R_{\texttt{<c>}} = \{\texttt{<c>}(\texttt{<f>}, \texttt{<g>})\}$$

Therefore $\mathbf{D} = \{\texttt{<a>}(\texttt{<d>}), \texttt{<a>}(\texttt{<e>}), \texttt{<c>}(\texttt{<f>}, \texttt{<g>})\}$ and $\mathbf{C} = \{\texttt{<b>}\}$, s.t.

$$\mathbf{P} = \mathbf{D} \times \mathbf{C} = \{(\texttt{<a>}(\texttt{<d>}), \texttt{<b>})\,, (\texttt{<a>}(\texttt{<e>}), \texttt{<b>})\,, (\texttt{<c>}(\texttt{<f>}, \texttt{<g>}), \texttt{<b>})\}$$

and the returned result R as illustrated in Figure 8.

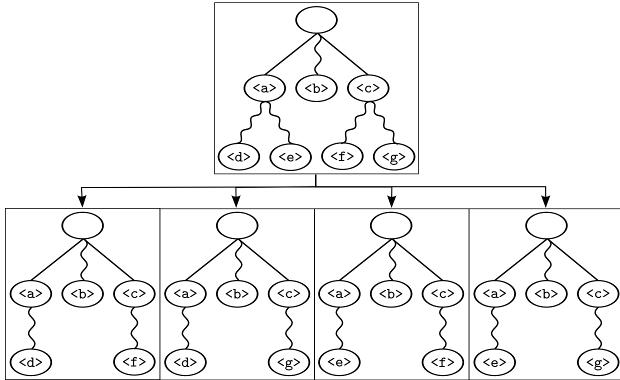

Figure 9. Splitting example 2: Concurrent decisions can be modelled by specifying an intermediate layer with concurrent edges.

Example 2: (cf. Figure 9) This example shows that decisions can also be taken concurrently by mixing them with concurrent edges: Being in some state, an agent chooses the single available decision `<b>`. Since the nodes `<a>` and `<c>` are reached concurrently there are possibly some decisions inside their subtrees. The agent can not decide to which node to move, thus this decision is left open by creating a branching of these nodes. Such a branching induces the possibility to choose concurrently decisions residing in subtrees of the branched nodes. In an MDP setup this corresponds to taking multiple actions simultaneously.
Applying this idea to our example, calling SPLIT on the whole decision tree $\mathcal{D}^c$ returns

$$R_{\texttt{<a>}} = \{\texttt{<a>}(\texttt{<d>}), \texttt{<a>}(\texttt{<e>})\}\,,$$
$$R_{\texttt{<c>}} = \{\texttt{<c>}(\texttt{<f>}), \texttt{<c>}(\texttt{<g>})\} \quad \text{and} \quad R_{\texttt{<b>}} = \{\texttt{<b>}\}$$

s.t. $\mathbf{C} = R_{\texttt{<a>}} \times R_{\texttt{<c>}}$ gives all combinations of already split subtrees and $\mathbf{D} = \{\texttt{<b>}\}$. The result R given by appending all combinations of $\mathbf{D} \times \mathbf{C}$ to the root node is shown in Figure 9.
Typical "basis" forms of actions already shown in the examples (Figure 8 and 9) are sequences of decisions and branching of decisions. These basis actions can be arbitrarily combined to more complex actions, e.g. branching of sequences of decisions. Since decision trees extend forward trees from LARES it is also possible to synchronize reactions (e.g. state transitions). Thus it is possible to define both

- actions with synchronous effects (e.g. `sync` node following a decision) and
- synchronous actions (decisions residing in subtrees of a `sync` node).

4.3 Enabling

We still need to define the enabling function $e : S \to \mathcal{P}(Act)$ which describes for each state $s \in S$ which actions $e(s) \subseteq Act$ can be chosen in that state. Syntactically the enabling of an action is given by conditions in `guards decision` resp. `forward to decision` statements. By looking at the running example in Figure 5 together with the arising decision tree in Figure 6 the decision `<dorep>` is activated if the condition `oneF & !sysF` holds. If this is the case then `<repC[1]>` is activated if additionally the condition `C[1].F` holds. But there are also concurrent edges which finally can end up in further decision edges as can be seen in the edge between `<repC[3]>` and `C[3].<repair>`.

Definition (Enabling): Let s be a composed state and $a \in Act$. We say that

- a is partially enabled in s if s fulfills all conditions residing on decision edges of a. Denote $e^p(s)$ as all partially enabled actions a in s and e^p_a as the predicate describing the partial enabling.
- a is (completely) enabled in s if $a \in e^p(s)$ and a induces at least one transition in the composed model. Denote $e(s)$ as all enabled actions a in s.

As an example Table 1 shows the partial enablings e^p_a after splitting the completed tree in Figure 7.

action a	partial enabling e^p_a
`<a>(<noDec>,<c>)`	$A \wedge true$
`<b>(<d>,<e>)`	$B \wedge D$
`<b>(<noDec>, <e>)`	$B \wedge \overline{D}$
`<noDec>`	$\overline{A} \wedge \overline{B}$

Table 1. Action set together with its partial enabling for the completed decision tree in Figure 7

In order to motivate the separation between partial enabling and complete enabling consider Figure 7. Imagine that an agent is in some state s which satisfies the condition $\overline{A} \wedge \overline{B}$ and that there is no concurrent unguarded process active in s. Then the agent would choose the single partially enabled action `<noDec>` and no transition to some target state s' could take place. Therefore in the MDP context there would be a state-action deadlock. In order to resolve this situation, an action a is defined as (completely) enabled in s if there really is some transition possible. On the other hand, if for all $a \in e^p(s)$ no transition is possible from s then there are no enabled actions in s, i.e. $e(s) = \emptyset$ and thus s is absorbing.

Proposition: The partial enabling covers the whole composed potential state space, i.e.

$$\forall s \in \widehat{S}\ \exists a \in Act:\ a \in e^p(s).$$

The meaning of this proposition is that the intuitive approach for hierarhical specification of actions by the modeller can be actually represented through the cooperation of the three steps: completion, splitting and enabling. Concretely, by applying these steps, it is not possible to create a deadlock state in which the agent gets stuck and can not choose any (partially enabled) action.

Proof. Let $s \in \widehat{S}$ and v some inner node of $\mathcal{D}^c$ with tree $T_v := v(v'_1, \ldots, v'_d, v''_1, \ldots, v''_c)$ s.t. $v'_k,\ k = 1, \ldots, d$ ($d = |\mathbf{D}|$) are reached by some decision edge from v in $\mathcal{D}^c$ and $v''_j,\ j = 1, \ldots, c$ by some concurrent edge. By completion the disjunction over all conditions on (v, v'_k) is tautologically $true$. The splitting divides T_v into $d \cdot |\mathbf{C}|$ subtrees $R_k = v(v'_k, v''_1, \ldots, v''_c)$ with exactly one decision edge (v, v'_k) (cf. Algorithm 4). Therefore there exists $k \in \{1, \ldots, d\}$ s.t. s satisfies the condition on edge (v, v'_k) of R_k. Note that for $j \in \{1, \ldots, c\}$ the node v''_j is the same in each R_k but the subtrees from v''_j are different ($|\mathbf{C}|$ possible combinations). We need also the following distributive law for two sets X and Y of logical expressions:

$$\bigvee_{(x,y) \in X \times Y} (x \wedge y) = \left(\bigvee_{x \in X} x \right) \wedge \left(\bigvee_{y \in Y} y \right).$$

Transfering to our case with c sets $A_j,\ j = 1, \ldots, c$ of logical expressions given by conditions on decision edges in subtrees $T_{v''_j}$ of v''_j we have that by induction the partial enabling of $T_{v''_j}$ is given by some conjunction $x_1 \wedge \cdots \wedge x_c,\ x_j \in A_j$. Furthermore $\bigvee_{x_j \in A_j} x_j = true$ for all j, since $T_{v''_j}$ is completed. Thus by distributive law

$$\bigvee_{(x_1, \ldots, x_c) \in A_1 \times \cdots \times A_c} (x_1 \wedge \cdots \wedge x_c) = true.$$

Therefore there is some split concurrent subtree which partial enabling is also satisfied in s. □

Remark: In case actions are modelled by branching decisions, then in the worst case the number of actions grows exponentially with the number of decisions. If such an exponential increase in the number of actions is not directly intended by the modeller, actions can be discarded by checking the satisfiability of partial enablings.

4.4 Reachability for LARES.de

Each action $a \in Act$ induces a forward tree $\mathcal{F}_a$ by changing all decision edges in a to concurrent edges. A state s' is reachable from s if there exists $a \in e(s)$ s.t. applying $\mathcal{F}_a$ to s reaches s'. The reachability (Algorithm 5) for a LARES.de model slightly modifies the reachability (Algorithm 3) for a LARES model: The interactions $\mathcal{I}$ are replaced by "enabled interactions" $(a, \mathcal{I}_a, e^p_a)$ consisting of the concrete action $a \in Act$, interactions $\mathcal{I}_a := \text{FLATTEN}(\mathcal{F}_a) \cup \mathcal{I}^{ungrd}$ and its partial enabling e^p_a.

Algorithm 5 Reachability for LARES.de

Input: (initial) state $s = (s_1, \ldots, s_n)$ and enabled interactions $\mathcal{I} := \{(a, \mathcal{I}_a, e^p_a) \mid a \in Act\}$
Output: reachable state space
1: **function** REACHABILITY$(s, \mathcal{I})$
2: mark s as done
3: $M := \emptyset$
4: **for all** $a \in Act$ with $e^p_a(s) = true$ **do**
5: $t := \bigcup_{\gamma \in \mathcal{I}_a}$ GENERATETRANSITIONS(s, γ)
6: $t := \{(s', \delta) \in t \mid \delta \neq NA\}$
7: $t^{Dirac} = \{\ (s', \delta) \in t \mid \delta = \text{Dirac}(w)\ \}$
8: **if** $t^{Dirac} \neq \emptyset$ **then** $t := t^{Dirac}$
9: $M := M \cup (\{(s, a)\} \times t)$
10: $N := \{s' \mid (s', \delta) \in t,\ s' \text{ not done}\}$
11: **return** $M \cup \bigcup_{s' \in N}$ REACHABILITY$(s', \mathcal{I})$

Note that Algorithm 5 uses the partial enabling e^p_a as an input but implicitly uses the semantics of the complete enabling: If $t = \emptyset$ for some $a \in e^p(s)$ then the product $\{(s, a)\} \times t$ in line 9 gets empty and thus a is not visible in s in the transformed MDP.

5 Related work

MDPs are often used in the field of AI for describing actions (e.g. in planning domains) with probabilistic effects. Typical specification languages used there are PPDDL [?], PPDDL+ [?], SPUDD [?] and RDDL [?]. We shortly describe their expressivity and explain why we couldn't build upon these formalisms.
SPUDD is based on a symbolic representation of action-dependent dynamic Bayes nets (DBNs). However, time is discrete, actions can not be taken concurrently and there is no enabling structure for actions, s.t. all actions can be executed in every state.
RDDL builds upon PPDDL and extends its expressivity

by composing state spaces through so-called fluent variables. It further allows for some features also expressible with LARES.de, like concurrent actions and intermediate layers for transition control. Further features are intended for planning domains which are not expressible with LARES.de, like partial observability and first-order expressions. A subset of RDDL can be transformed to SPUDD in order to generate an MDP model, thus losing the enabling structure. Furthermore, RDDL lacks the modelling of continuous stochastic delay and reward structure for counting the sojourn time in states.
PPDDL+ closes these shortcomings by incorporating stochastic delays but fails for modelling more complex action structures. Furthermore, the tool Tempastic-DTP [**?**] works on a subset of PPDDL+ and does not allow (among others) for undiscounted measures and non-binary state variables.
All these formalisms do not support separation between abstract descriptions and concrete instantiations, thus the degree of modularity is restricted.

6 Conclusion and Future work

We have introduced the modelling concept of a forward tree which abstracts stochastic transitions by concurrent conditional reactions. We extended this structure in an intuitive way to the so-called decision tree in order to specify stochastic control by mixing controllable reactions with concurrent reactions. In order to perform analysis and optimization on the specified systems, we defined a transformation to an MDP which is based on the view of an agent acting in the decision tree. This approach shows several advantages for modelling control, e.g. flexible description of different types of actions comprising sequences, branching or synchronisation of actions and an enabling structure for actions. We further presented and motivated LARES and LARES.de as modular and hierarchical specification languages which allow to embed these tree structures in a cooperative way with object-oriented modelling concepts.
We haven't shown in this paper the already existing hierarchical definition of the reward structure which allows for different kinds of model measures. The implementation of LARES.de is almost finalized. Therefore we plan in a forthcoming paper to mix the decision and reward extensions and show the modelling advantages on a bigger case-study. Moreover we plan to define a transformation to a continuous-time MDP by eliminating vanishing state-action pairs on the level of LARES.de such that the state space of specific models may be reduced dramatically.

7 Acknowledgements

We woud like to thank Deutsche Forschungsgemeinschaft (DFG) who supported this work under grant SI 710/7-1.

References

[1] J. Bachmann, M. Riedl, J. Schuster, and M. Siegle. An Efficient Symbolic Elimination Algorithm for the Stochastic Process Algebra Tool CASPA. In *SOFSEM*, pages 485–496, 2009.

[2] H. Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*, volume 2428 of *Lecture Notes in Computer Science*. Springer, 2002.

[3] J. Hoey, R. St-Aubin, A.J. Hu, and C. Boutilier. SPUDD: Stochastic Planning using Decision Diagrams. In *Proceedings of Uncertainty in Artificial Intelligence*, pages 279–288, 1999.

[4] M. Kuntz, M. Siegle, and E. Werner. Symbolic Performance and Dependability Evaluation with the Tool CASPA. In *FORTE Workshops*, pages 293–307, 2004.

[5] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. Wiley, 1995.

[6] M. L. Puterman. *Markov Decision Processes*. John Wiley & Sons INC., 1994.

[7] M. Riedl, J. Schuster, and M. Siegle. Recent Extensions to the Stochastic Process Algebra Tool CASPA. In *QEST*, pages 113–114, 2008.

[8] M. Riedl and M. Siegle. A LAnguage for REconfigurable dependable Systems: Semantics & Dependability Model Transformation. In *Proc. 6th International Workshop VECOS'12*, pages 78–89, 2012.

[9] S. Sanner. Relational Dynamic Influence Diagram Language (RDDL): Language Description. `http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf`, 2010.

[10] O. Sigaud and O. Buffet, editors. *Markov Decision Processes and Artificial Intelligence*. Wiley, 2010.

[11] H.L.S. Younes. Extending PDDL to Model Stochastic Decision Processes. In *Proc. ICAPS Workshop on PDDL*, pages 95–103, 2003.

[12] H.L.S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2005.

[13] H.L.S. Younes, M.L. Littman, D. Weissman, and J. Asmuth. The first probabilistic track of the international planning competition. *J. Artif. Intell. Res. (JAIR)*, 24:851–887, 2005.