Managing Non-functional Uncertainty via Model-Driven Adaptivity

Carlo Ghezzi*, Leandro Pinto*, Paola Spoletini†, and Giordano Tamburrelli*
* DeepSE, Politecnico di Milano, Italy. {carlo.ghezzi|leandro.pinto|giordano.tamburrelli}@elet.polimi.it
† Università dell’Insubria, Italy. paola.spoletini@uninsubria.it

Abstract—Modern software systems are often characterized by uncertainty and changes in the environment in which they are embedded. Hence, they must be designed as adaptive systems. We propose a framework that supports adaptation to non-functional manifestations of uncertainty. Our framework allows engineers to derive, from an initial model of the system, a finite state automaton augmented with probabilities. The system is then executed by an interpreter that navigates the automaton and invokes the component implementations associated to the states it traverses. The interpreter adapts the execution by choosing among alternative possible paths of the automaton in order to maximize the system’s ability to meet its non-functional requirements. To demonstrate the adaptation capabilities of the proposed approach we implemented an adaptive application inspired by an existing worldwide distributed mobile application and we discussed several adaptation scenarios.

I. INTRODUCTION

Modern software systems are characterized by an increased complexity, for example, in terms of size as well as geographical distribution. Moreover, engineers increasingly design systems by relying on components operated by third-party organizations. All these factors introduce many sources of uncertainty in designing software that should guarantee certain quality requirements. For example, a component operated by a third-party organization has an uncertain execution time and failure rate. This means that, at design time, engineers can make certain assumptions, but these may be invalidated at run time. Indeed, uncertainty may subvert design-time assumptions, jeopardizing the system’s ability to meet its requirements. The execution time of a remote component may increase unexpectedly affecting the response time of the overall system or, similarly, a component failure may affect the system’s reliability. Both cases may lead to potential violations of non-functional requirements that cannot be tolerated.

To cope with uncertainty, software can be designed as an adaptive system [4]. In practice, engineers typically design systems by explicitly programming alternative behaviors and by heavily using exception handling techniques to adapt the execution to detected changes in the environment that reify the sources of uncertainty. This is quite hard per-se and cannot be done by inexperienced developers. In addition, using this approach, such alternative behaviors cannot be kept separated from each other and from the exception handling code. As a result, the architecture and the code are hard to understand and maintain.

To overcome this limitation, we present ADAM: a model-driven framework conceived to support the development and execution of software that tolerates manifestations of uncertainty by self-adapting to changes in the environment, trying to do its best to satisfy certain non-functional requirements. In particular we support adaptation aimed at mitigating the non-functional uncertainty concerning (1) response time and (2) faulty behavior of components integrated in a composite application. The proposed solution models the system as a workflow of abstract functionalities. For each of them, one or more implementations are provided.

The approach derives then a finite state automaton augmented with probabilities in which each state of the automaton represents an implementation of an abstract functionality of the system, while paths represent all the possible execution flows of the system. The system execution is performed by an ad-hoc interpreter that navigates the automaton state by state and invokes the implementations associated with the states it traverses. The interpreter is responsible for driving and adapting the execution by choosing among alternative paths of the automaton in order to maximize the system’s ability to meet its non-functional requirements.

With this paper we contribute to research in self-adaptive systems in two distinct ways:

1) We lay the foundations of a model-driven methodology that supports the design of systems in terms of abstract functionalities and delegates the execution to an optimizing interpreter, which self-adapts to changes by selecting the implementation variants to bind to abstract functionalities in order to satisfy non-functional requirements;

2) We present a novel technique to support non-functional adaptation that exploits Probability Theory and Probabilistic Model Checking [1]. The proposed technique computes and assigns a probability to each possible execution flow of the system indicating their likelihood to meet the desired non-functional requirements. In our approach the interpreter relies on such values to drive the execution;

The remainder of the paper is organized as follows. Section II introduces a reference example we use throughout the paper. Section III describes the ADAM approach in its essential steps, presenting it also applied to the example,
highlighting several relevant scenarios. Section IV discusses its advantages w.r.t. the state of the art. Section V evaluates the performance of ADAM in several scenarios of growing complexity. Finally, Section VI discusses related work, while Section VII draws some concluding remarks.

II. THE SHOPREVIEW APPLICATION

ShopReview (hereafter referred to as SR) has been selected as the adaptive application used throughout the paper to explain our approach. SR is a mobile application for smartphones inspired by ShopSavvy,2 an existing worldwide distributed application, which allows users to share data concerning a commercial product or query for data shared by others. Users may use SR to publish the price of a product they have found in a certain shop and, in response, the application provides the users with more convenient prices offered by: (i) nearby places (i.e., LocalSearch) or (ii) on-line stores (i.e., WebSearch). The unique mapping between the price signaled by the user and the product is obtained by exploiting the product’s barcode. The application starts by asking the user to scan a barcode with the smartphone camera and to type the price of the product whose barcode has been scanned. In response to these inputs the applications: (1) recognizes the barcode number in the scanned image, (2) looks up online the product associated to the barcode number, (3) retrieves the user location, (4) performs the WebSearch as well as the LocalSearch, (5) displays the obtained results, and (6) allows the user to publish the price of the product to be used by searches issued by other users.

Furthermore, we decide to implement the code for recognizing the barcode from a picture acquired through the camera, which runs an ad-hoc developed component that encapsulates an image recognition algorithm. Since such component executes correctly only on devices with an autofocus camera and does not work properly on other devices, this choice would limit the usability of our application. For this reason, SR, as for the original ShopSavvy app, is designed to provide also an alternative recognition algorithm. Indeed, it detects if the camera on the current device has autofocus and, if not, it invokes an external service to process the acquired image with an ad-hoc blurry decoder algorithm. In both cases, if the barcode cannot be recognized (e.g., the image is blurred), the application asks the user to take the picture again.

In addition, let us assume that SR must satisfy the non-functional requirements listed in Table I. R1, for example, is a performance requirement typically imposed by several marketplaces of mobile applications.3 Notice that, in our application, several features contribute—at large—to usability (R2). For example, to retrieve the user location, the GPS is preferable over the NPS4 because of its increased precision. Similarly, presenting a sorted list (by price and distance) of the results of WebSearch and LocalSearch, respectively, also increases the app’s usability.

ADAM supports two classes of non-functional requirements: (1) Threshold-Based and (2) Max/Min. The former comprise requirements in the form: \( m \leq th \) or \( m \geq th \), where \( m \) is a non-functional metric (e.g., response time, energy, etc.) and \( th \) is a threshold (see R1). The latter requires to minimize/maximize a non-functional metric (see \( R2 - R3 \)).

SR is subject to several sources of uncertainty and, as a consequence, run-time conditions may jeopardize the application’s ability to meet its non-functional requirements. For example, engineers may design SR to be compliant with R1 by estimating or experimentally measuring the response time of the component implementing the WebSearch functionality, which invokes a remote back-end. However, such response time may increase unexpectedly during operation because of the network latency or other external factors, causing the system to violate R1. Such unexpected behavior, if left unmanaged, may cause an unsatisfactory user experience or, in the worst case, the rejection from the marketplace. Thus, even with such a simple example, uncertainty management is fundamental for a successful system design. It is important to notice that the concepts illustrated through this paper apply seamlessly to larger and more complex systems where the benefits of our approach are even more relevant.

III. THE ADAM APPROACH

ADAM is a model-driven framework conceived to support the development and run-time operation of self-adaptive systems, which can react to (1) unexpectedly higher response time or (2) unexpected faults of the parts they rely upon.

Let us start by giving an overview of the approach, as illustrated in Figure 1. First, developers provide a model of the system in terms of abstract functionalities organized in a workflow. In particular, we support systems modeled by Activity Diagrams [8]. For each abstract functionality, developers also provide one or more target implementations, annotated with a description of their non-functional behaviors. Annotations may concern, for example, the expected execution time, cost, or the impact on energy consumption or on the application’s usability. The supplied target implementations display different non-functional qualities. The goal is to be able to develop dynamic composition of target implementations that best match the overall application’s requirements. We assume target implementations to be stateless components.

Given these inputs and a set of non-functional requirements the system has to meet, the approach relies on two distinct tools: (1) the Generator and (2) the Interpreter. The Generator analyzes the Activity Diagram and the corresponding annotated target implementations and generates a finite state

<table>
<thead>
<tr>
<th>Description</th>
<th>Metric</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. After an input the application shall respond in at most 3s.</td>
<td>Response Time (RT)</td>
<td>Threshold-Based</td>
</tr>
<tr>
<td>2. Maximize application usability</td>
<td>Usability (U)</td>
<td>Max</td>
</tr>
<tr>
<td>3. Minimize battery consumption</td>
<td>Energy Consumption (E)</td>
<td>Min</td>
</tr>
</tbody>
</table>
automaton, called Embedded Model (EM). In the EM, each state represents an implementation of an abstract functionality of the system, while paths represent all the possible execution flows. The Interpreter is, instead, in charge of executing the system by navigating the automaton state-by-state and by invoking the chosen target implementations associated to the states it traverses. In particular, it is responsible for driving and adapting the execution by choosing among alternative paths of the automaton in order to maximize the system’s ability to meet its non-functional requirements. By this we mean that the Interpreter first measures the effects of non-functional uncertainty (e.g., the response time of invoked functionalities) and consequently chooses the most convenient path in the EM to maximize the likelihood of meeting all the system’s requirements. This way, if the Interpreter detects that the current execution is slower w.r.t. a certain performance requirement, it may autonomously decide to drive the execution by choosing a specific (fast) path in the EM that guarantees the compliance with the performance requirement. The approach comprises the following steps: (1) Modeling, (2) Transformation, (3) Model Manipulation, and (4) Execution. Hereafter, we describe each of them in detail. For each step, we also illustrate it referring to the SR example.

A. Modeling

As previously introduced, the system is initially conceived in terms of abstract functionalities and modeled by one or more UML Activity Diagrams, which organize them in workflows. For each abstract functionality, engineers also provide one or more corresponding alternative target implementations. The design methodology to derive the set of target concrete functionalities for each abstract one, given the overall requirements and an uncertainty mitigation policy, is out of scope of the present paper. We observe, however, that designing systems in terms of alternative implementations corresponds to an approach already used for complex software systems, even if informally. For example, in mobile applications, the user location is typically obtained by relying on two alternatives: (1) the GPS sensor or (2) the NPS. Clearly, every abstract functionality needs at least one corresponding implementation. In addition, while modeling, engineers are allowed to annotate a subset of the abstract functionalities as Optional. Usually, optional functionalities are not essential for the correctness of the final result, but may, however, affect usability. If necessary, they are sacrificed to accomplish more important goals. As illustrated in the example, each non-functional requirement predicates over a certain non-functional metric. As a consequence, each implementation is annotated with the impact it has w.r.t. these metrics. For example, an implementation of an abstract functionality with an expected response time of 2 seconds is annotated with responseTime=2s. Concrete implementations that require user interaction cannot be annotated with an impact on response time, since they depend on the user’s think time. They are therefore annotated with @UI, whose meaning will become clear later on. Notice that the annotation process occurs for each requirement metric on all the implementations. Finally, ADAM requires engineers to annotate each branch of decision nodes in the UML Activity Diagram with the expected probability that an execution of the system may take that branch. When not specified, branches are considered to have the same probability.

**Modeling the SR Application**. The modeling step applied to the SR example may produce the Activity Diagram illustrated in Figure 2. For each abstract functionality, one or more concrete implementations are provided. For instance, concerning ProductLookup, which translates a barcode into a product name, SR relays on a remote service (e.g., searchupc.com) as one of the possible implementations. Alternatively, the application may ask the user to directly provide the product’s name. As for searching the Web for more convenient prices, SR relies on a primary remote service (e.g., shopzilla.com) and on complementary services, represented by the abstract functionalities WebSearch and SecondaryWebSearch, respectively. Note that the SecondaryWebSearch is annotated as an Optional functionality to represent the fact that it may be omitted at runtime, if necessary. Similarly, the ResultOrdering functionality, which sorts the results of WebSearch and LocalSearch by price and distance, respectively, has been annotated as optional.

Concrete implementations are provided by Java methods using the ad-hoc annotation @Implementation to refer to the abstract functionality they implement. Moreover, the annotation @Impact is used to specify the impact the implemen-
The translation of Activity Diagrams into an MDP is performed by the Generator tool. It first translates each abstract functionality into a simple MDP with an initial state, a final state, and as many intermediate states as the number of implementations associated with the abstract functionality. In addition it adds a non-deterministic transition from the initial state to each intermediate state that are, in turn, connected to the final state. The resulting MDPs are then merged in a single MDP which represents the translation of the overall Activity Diagram. Moreover, annotations attached to implementations are also propagated into the EM by annotating each state in the model with the same impact numbers as its corresponding implementation. Notice that connections which participate in decision nodes (i.e., connections labeled with probabilities) are translated in the MDP as probabilistic transitions labeled with the same probabilities as the originating connections.

Transforming the SR Activity Diagram. As we said above, we first translate each abstract functionality into a simple MDP with an initial state, a final state, and as many intermediate states as the number of implementations associated to the abstract functionality. For example, the ProductLookup functionality, which has two corresponding implementations, results in the MDP in Figure 3(a).

A functionality annotated as optional also requires the introduction of a direct transition from the initial state to the final one. For example, if we have two alternative implementations of the SecondaryWebSearch, which differ in the remote service they invoke, we obtain the MDP in Figure 3(b).

The resulting MDPs are then composed together as follows. Two MDPs corresponding respectively to two subsequent nodes in the Activity Diagram are composed by merging the final state of the first one with the initial state of the second one. By merging such states we generate a symbolic state. This process is exemplified in Figure 3(c), where symbolic states are highlighted in grey and labeled with a letter. Decision nodes are translated similarly. The MDP corresponding to a functionality that precedes a decision node has its final state merged to the initial state of all the MDPs corresponding to the functionalities subsequent to the decision node. The merging process generates a symbolic state as aforementioned.

Concerning nodes that also map to other diagrams, as for the Location node, we first translate the node (as explained so far) considering only its implementations (e.g., the GPS alternative). Subsequently, we add, as an alternative path from the initial to the final state, the MDP obtained by translating the other diagrams (e.g., the NPS sub-diagram).

The translation process applied to the Activity Diagram of the SR application results in the EM reported in Figure 3(d). For instance, the functionality Location is represented in the MDP by state $6_1$ (i.e., the GPS alternative) and states $12 = f − 13$ (i.e., the translation of the NPS nested diagram).

At this stage we propagate the annotations attached to the implementations into the EM. In Figure 3(d) we indicate them with $RT$, $E$, and $U$ for response time, energy consumption, and usability, respectively. For example, state $5_1$, which cor-

Listing 1. Implementations for the ProductLookup functionality.

```java
@Implementation(name="ProductLookup")
@Impact (metrics={"responseTime", "energy", "usability"}, values={1, 2, 1})
public String automaticProductLookup(String barcode){
    //invoke http://searchupc.com/
}

@Implementation(name="ProductLookup")
@Impact (metrics={"energy", "usability"}, values={1, 0})
public String manualProductLookup(String barcode){
    //ask the user to insert the product name
}
```

The last step in the modeling phase concerns the probability annotations of decision nodes. Decision nodes guarded by the conditions hasAutoFocus and recognized have been annotated with the probabilities reported in Figure 2. They express, respectively, the fact that 40% of mobile devices do not have an autofocus camera and, on average, 70% of barcodes contained in acquired pictures are correctly recognized.

B. Transformation

Activity Diagrams are translated into a formal representation, the Embedded Model (EM), which is a Markov Decision Process (MDP) [18]. MDPs are finite state machines augmented with probabilities and non-deterministic transitions. The formal concepts concerning MDPs needed to understand the rest of the paper are summarized in the Appendix.
responds to the \textit{automaticProductLookup} implementation (see Listing 1), is annotated with its impact in terms of response time (i.e., 0.5s), energy consumption (i.e., 2), and usability (i.e., 1). Since symbolic state are artificially generated by the translation process they are annotated with neutral values: \(RT = 0\), \(E = 0\), \(U = 0\). Notice that, by construction, the obtained EM represents all the possible execution flows of the system in terms of target implementations. Indeed, starting from its initial state, the MDP has multiple alternative paths towards the final state. The translation process performed by the Generator hides the complexity of MDPs to developers. A formal description of the automatic translation algorithm is not given here for space reasons. It is based on the automatic translation of an annotated Activity Diagram into a Markov process that was presented in our previous work (i.e., [13]).

\subsection*{C. Model Manipulation}

The annotations attached to the states of the EM represent the impact of the corresponding implementation on quality metrics. Formally, this information corresponds to \textit{rewards} in the MDP formalisms (see the Appendix). It can be used to compute the minimum and maximum cumulative rewards (indicated as \(minR(s)\) and \(maxR(s)\)) from each state \(s\) to the final state in the model and for each quality metric. The computation of such cumulative rewards may be arbitrarily complex because of three characteristics of the model: (1) loops, (2) probabilities attached to transitions, (3) a large number of alternative paths. We rely on a probabilistic model checker, such as PRISM [14], to compute them. Given these premises, we manipulate the model by replacing impact numbers attached to each state \(s\) with an interval (\(minR(s), maxR(s)\)) for every requirement metric of the system. It is important to notice that such intervals represent forecasts of the impacts necessary to complete the execution (i.e., reach the final state) starting from a specific state \(s\) of the model. At execution time, such values are used by the Interpreter to select the most appropriate path towards the final state, as illustrated in Section III-D. Figure 4 illustrates the cumulative rewards obtained by exploiting PRISM for some states of the EM. Notice that, when cumulative rewards are computed for response time, all the states characterized by user interaction (i.e., whose corresponding implementations are annotated with \(\emptyset\cup1\)) are considered as final states of the EM together with the original final states. Indeed, the requirements concerning response time (e.g., R1) predicate over the portions of the system in which the computation occurs autonomously, i.e., without user input.

\begin{center}
\textbf{Manipulating the SR Model.} At this stage, we manipulate each state \(s\) of the model in Figure 3(d) by replacing impact numbers with intervals in the form \(\langle minR(s), maxR(s)\rangle\), for every requirement metric, obtained by running the probabilistic model checker, as explained above. For example, let us focus on state \(s_a\) and usability. In this case the model checker yields the following values: \(\langle 4; 6\rangle\). These values indicate that an execution reaching state \(s_a\) will have an additional usability impact value in the interval \(\langle 4; 6\rangle\) to reach the final state. Similarly, for the response time and energy consumption we obtain \(\langle 2.9; 4.1\rangle\) and \(\langle 8; 11\rangle\), respectively.
\end{center}
D. Execution

At run time, given the annotated EM, the Interpreter is in charge of executing the application, by navigating through the state space. It invokes the corresponding implementation and performs two additional tasks. First, it keeps track of the cumulated impact of all the quality metrics. For example, assuming that the Interpreter invoked two implementations that executed in 1s and 0.5s, its aggregated response time is 1.5s. The run-time data structure that contains all these aggregated values is called the Execution Context. Second, it selects one of the alternative paths in the model. The choice among alternative paths occurs if the current state has many outgoing non-deterministic transitions, i.e., the next functionality to be executed has many corresponding implementations and/or is optional.\(^5\) The choice is performed by the Interpreter as described next.

Given a system with a set of non-functional requirements \( R = \{r_1, ..., r_n\} \) and an execution that reached state \( s\) characterized by a set \( T = \{t_1, ..., t_k\} \) of non-deterministic outgoing transitions, the Interpreter computes a Probability of Success for each alternative transition \( t_i \). Such probability \( P_{t_i} \) represents the likelihood that the system may complete the execution meeting all the requirements by taking transition \( t_i \).

Let \( s_i \) be the destination state of each outgoing transition \( t_i \) and let us focus initially on a threshold-based requirement \( Req \) that predicates on response time, i.e., \( RT \leq th \). We know from the interval associated to each state \( s_i \) that an execution through that state has an expected response time included in the interval \((a_i, b_i)\), as computed in the model manipulation step of the approach. In addition, we know from the Execution Context that, to reach state \( w \), the execution already cumulated a response time \( c_{RT} \) (i.e., the execution spent \( c_{RT} \) seconds to reach state \( s_i \)). Given these premises, we model the response time associated with the execution from state \( s_i \) to the final state as a random variable \( x_{s_i} \) uniformly distributed over the interval \((a_i, b_i)\) (i.e., \( x_{s_i} \sim U(a_i, b_i) \)). In this setting, the ability of the system to meet \( Req \) boils down to choosing a transition towards a state \( s_i \) such that the response time from that state to the final one is less than the requirement threshold decreased by the time already consumed to reach the current state: \( x_{s_i} \leq th - c_{RT} \). In particular, the probability that this may occur (i.e., \( P(x_{s_i} \leq th - c_{RT}) \)) corresponds to the area in the Uniform Distribution comprised among \( a_i \) and \( th - c_{RT} \):

\[
P_{t_i,RT} = P(x_{s_i} \leq th - c_{RT}) = \frac{(th - c_{RT}) - a_i}{b_i - a_i}
\]

Such value corresponds to the probability of success for transition \( t_i \) concerning only response time. In particular, three cases may occur that yield to a probability of success, as illustrated in Figure 5. Intuitively, if the threshold decreased by \( c_{RT} \) is greater than the interval that contains the expected values of response time, probability of success is one. Conversely, if

\(^5\)States with multiple outgoing transitions labeled with probabilities do not represent a decision point among alternative paths since they correspond to decision nodes in the Activity Diagram and the next state in the execution is selected by evaluating run-time conditions (e.g., hasAutoFocus for SR).

![Fig. 5. Probability of Success.](image)

the threshold decreased by \( c_{RT} \) is smaller than the interval, the probability is zero. In all the other cases the probability is in \([0,1]\) accordingly to the relative position of the interval and the quantity \( th - c_{RT} \). Notice that requirements in the form of \( RT \geq th \) are processed similarly by inverting the area considered in computing the probability of success. We repeat this procedure for all requirements in \( R \). In particular, if we have \( n \) requirements and \( k \) possible outgoing transitions, the Interpreter builds a \( n \times k \) matrix in which each element \((i,j)\) represents the probability that choosing transition \( t_i \) would yield to an execution compliant with requirement \( r_j \) (i.e., probabilities of success \( P_{t_i,r_j} \)). The Interpreter is parametrized with a vector \( w = w_1, ..., w_n \) of weights. Such vector is used to aggregate the probabilities of success by computing a weighted average of the values in each column of the matrix. The result is a single value for each outgoing transition, which represents the probability of satisfying all the requirements according to the weight: \( P_{t_i} = \sum_{1 \leq j \leq n} (w_j \times P_{t_i,r_j}) \). The Interpreter proceeds the execution by choosing the transition \( t_i \) with the highest \( P_{t_i} \). Notice that weights are used by the designer to prioritize requirements.

**Executing the SR Application.** Let us consider requirement R1 and let us assume that the Interpreter reached state \( i \) (see Figure 4). In this scenario, it has to choose among three outgoing transitions (i.e., \( 9_a, 9_b \) and \( j \)). The probability of success of the transition towards state \( 9_c \) corresponds to the probability that the execution terminates within 3s (R1). We know from the interval associated to state \( 9_a \) that the execution through that state will terminate with a response time in the interval \((0.5s; 1.1s)\). At this stage, we model the response time of the execution through state \( 9_a \) as a random variable \( x_{9_a} \) uniformly distributed over the interval (i.e., \( x_{9_a} \sim U(0.5, 1.1) \)). Furthermore, let us assume that the Execution Context contains a cumulated value for the response time equal to \( c_{RT} = 1.95s \). The Interpreter computes the probability that the response time will be lower than the requirement threshold decreased by the value in the Execution Context (i.e., \( 3s - 1.95s = 1.05s \)) as the area in the Uniform Distribution in the interval \((0.5s; 1.05s)\):

\[
P(x_{9_a} \leq 3s) = \frac{1.05 - 0.5}{1.1 - 0.5} = 0.92
\]

Similarly, for the outgoing transitions to state \( 9_b \) we have \( P(x_{9_b} \leq 3s) = 0.75 \). Finally, for the outgoing transitions to state \( j \) we have \( P(x_j \leq 3s) = 1 \) (i.e., the third case of Figure 5).
Let us consider now requirement $R2$, which is a maximization requirements and thus needs a slightly different process. In this case the probability of success concerning usability associated with the transition toward state $9_a$ (i.e., $(P_{9_a,U})$) corresponds to the probability that the usability value obtained by choosing it is greater than the usability value obtained with the transitions to state $9_b$ and to state $j$. We know that by choosing the transition to state $9_a$ usability would be in the interval in $\langle 2; 3 \rangle$, while choosing the transitions to state $9_b$ and $j$ we obtain a usability value in the intervals $\langle 2; 3 \rangle$ and $\langle 1; 2 \rangle$, respectively. By modeling the usability of the execution through state $9_a$ as a random variable $y_{9_a}$ uniformly distributed over the discrete interval (i.e., $y_{9_a} \sim U(2, 3)$) and the usability of the execution through states $9_b$ and $j$ with random variables $y_{9_b}, y_j$ uniformly distributed over their intervals (i.e., $y_{9_b} \sim U(2, 3)$ and $y_j \sim U(1, 2)$), we obtain that the probability of success associated with the transition toward state $9_a$ (w.r.t. $R2$) is:

$$P_{9_a,U} = P(y_{9_a} \geq y_{9_b} \land y_{9_a} \geq y_j) = \sum_{(i,k,l) \in A} P(y_{9_a} = i \land y_{9_b} = k \land y_j = l)$$

where: $A = \{(i,k,l) | i \in \langle 2; 3 \rangle, k \in \langle 2; 3 \rangle, l \in \langle 1; 2 \rangle, i \geq k, i \geq l\}$. Considering $y_{9_a}, y_{9_b}$ and $y_j$ are independent variables we obtain $P_{9_b,U} = 0.75$. Similarly, we obtain: $P_{9_b,U} = 0.75$ and $P_{j,U} = 0.13$. Finally, for requirement $R3$ (i.e., a minimization requirement) we can apply a similar approach, with obvious changes, obtaining $P_{9_a,E} = 0$, $P_{9_b,E} = 0$ and $P_{j,E} = 1$.

Let us configure the Interpreter with the following weight vector $w = \{0.4, 0.4, 0.2\}$, which corresponds to requirements $R1$, $R2$, and $R3$ respectively. This choice prioritizes $R1 – 2$ over $R3$. Given these weights the probabilities of success are: $P_{9_a} = 0.67$, $P_{9_b} = 0.6$, and $P_j = 0.65$. In this setting, the Interpreter proceeds in the execution towards state $9_a$.

Notice that the approach, as described so far, uses only uniform distributions. However, the concepts illustrated in the paper apply seamlessly to other distributions that may be adopted—irrespective of the time spent by the computation to reach state $i$. The adaptive execution supported by ADAM would, instead, select the transition towards state $9_a$ in a scenario where state $i$ is reached in $1.95s$ (i.e., $c_{RT} = 1.95s$), and select transition $j$ in the case where state $i$ is reached in $2s$ (i.e., $c_{RT} = 2s$). In the latter case, with $c_{RT} = 2s$, the probability of success associated with outgoing transitions from state $i$ changes. In particular, by repeating the calculations illustrated in the previous section, we obtain the following values: $P_{9_a} = 0.63$, $P_{9_b} = 0.57$, and $P_j = 0.65$. As a consequence, the Interpreter proceeds the execution by selecting transition $j$ (i.e., the highest probability).

This choice is reasonable since the new context (i.e., $2s$) indicates a slower execution. In this situation the probability of success of the outgoing transitions $9_a$ and $9_b$ decreases, making the option that skips the SecondaryWebSearch functionality (i.e., transition towards $j$) preferable. Indeed, skipping this functionality speeds up such slower executions, minimizing the risk of violating $R1$. Conversely, in faster executions, transition $j$ has a lower probability of success because of its smaller contribution to usability w.r.t. $9_a$ and $9_b$. The same adaptation mechanisms apply to all of the EM non-deterministic choices; for example, the choice between the GPS and NPS, which may require different execution times and offer different degrees of usability and energy consumption.

A traditional strategy to manage uncertainty consists of designing systems by explicitly programming alternative behaviors and by heavily using exception handling techniques to adapt the execution to the manifestations of uncertainty. Let us consider the case of the alternative implementations of the Location functionality. A traditional adaptive implementation would require engineers to write an ad-hoc adaptation logic (e.g., cascaded if-elses to choose between the two implementations) and exception handling constructs aimed at managing the potential failures of both the alternatives. Such logic, not only results in convoluted solutions, but is also intertwined with the application logic yielding to code that is hard to read and maintain. Conversely, ADAM applications do not require such adaptation logic (see Listing 1) and delegate the adaptation management to the framework tools. As a consequence of this simplification, the implementations in the form of distinct annotated methods result in code that is easy to read, write, maintain, and evolve. Furthermore, ADAM also increases reusability, since the same functionality implementations can be reused across different applications.

ADAM also offers another source of adaptation against uncertainty. In Section I we stated that ADAM not only manages non-functional uncertainty in terms of higher response time, but also handles unexpected faults. Indeed, the system execution managed by the Interpreter introduces a useful self-healing feature in ADAM applications. As soon as the Interpreter catches a run-time exception while invoking a functionality implementation, it redirects the execution—if possible—towards another EM path that allows the system to complete successfully. For example, if invoking the functionality associated with state $9_a$ the Interpreter catches an exception indicating that the underlying Web service is unavailable, it may backtrack the execution to the previous state (i.e., $i$) and redirect the execution through an alternative path (e.g., $9_b$). Even if this is a simple self-healing mechanism, it is useful to maximize the system reliability in many scenarios. For
example, in the mobile domain, the GPS may fail unexpectedly and the NPS may be executed alternatively and transparently.

Finally, ADAM also achieves a clear separation among the different aspects of the application: from the more abstract ones, captured by Activity Diagrams, to those closer to the technical domain, captured by the implementations. By relying on such a sharp separation of concerns, developers may first model the features they want to introduce in the system, ignoring how they will be implemented later on. Let us consider again the Location functionality of the SR application. In the inception phase, developers only focus on the fact the system needs such feature and connect it to the other features by relying on the Activity Diagram. Later on, they can implement a first prototype that leverages NPS and the manual input of the user realizing that this solution needs to be improved in terms of usability. The applications may gradually evolve, by adding other implementations for this feature (e.g., the GPS). This process, in which the system design is decoupled from the implementation, as enforced by the proposed approach, is a widely recognized best practice in Software Engineering.

It is important to notice that the advantages provided by the ADAM approach are obtained transparently w.r.t. to engineers, who only have to produce one or more Activity Diagrams and their corresponding implementations. Even the complexity concerning the EM and MDPs is managed behind the scenes by the ADAM tools.

V. VALIDATING THE APPROACH

ADAM is implemented a publicly available open-source tool. Although our approach is general and applies with limited technological modifications to other languages, we focused on the Java language for our prototype. In this Section, we discuss the validation of the ADAM approach, focusing on its run-time overhead and scalability. The validation has been carried out by performing a large simulation campaign. Hereafter, we report the most significant results and we refer the reader to our prototype implementation for the replicability of the presented data. Since ADAM requires additional run-time computation, we start by discussing the overhead imposed to navigate and execute the model. This includes the computation of the probabilities of success for each alternative. We developed an application that automatically generates different models, and we used the ADAM prototype to execute them. To test its performance under different situations, we varied the number of abstract functionalities that are part of the model and the number of alternative implementations bound to them. Finally, we also varied the number of non-functional requirements present in the model to be satisfied during execution. Our evaluation was carried out on the following hardware setting: i5-540M processor, 4GB of RAM, Ubuntu Linux 11.10, and Oracle Java Runtime Environment version 1.6.0_26. Moreover, each of the experiments was repeated at least 30 times. The first experiment investigates the overhead imposed by ADAM to execute a simple model where each abstract functionality is associated with a single implementation. To calculate the overhead, we compared an ADAM execution with an equivalent (hard coded) sequence of method calls to the same implementations. We have varied the number of used abstract functionalities (and, consequently, the number of method calls) from 10 to 1000. The measured results are reported in Figure 6(a). These results show that the way ADAM models are navigated and executed introduces a negligible overhead, around 2.35%. For instance, for a large model with 500 abstract components, the execution takes in average 25.65s, while the Java execution with the same number of method calls requires 25.06s. To further investigate the overhead imposed by ADAM we extended the previous experiment as follows. Let us consider a base scenario with the following parameters: 500 abstract functionalities, each with 3 alternative implementations, annotated with different impacts; 6 non-functional requirements, 3 of which are threshold based and the remaining 3 are min/max requirements.

Figure 6(b) shows how the size of the model affects the overall execution time. In this experiment, we consider the base scenario increasing the size of the model from 10 to 1000. Note that these results differ from Figure 6(a) because they include also the time to compute and choose the best alternative. In general, the experiments show an overhead of approximately 4 – 5%, which we still claim to be reasonable. For instance, for a large model of 500 abstract functionalities, the execution time increases from 25.06s to 26.15.

Figure 6(c) shows the time required to execute an ADAM model in our base scenario with an increasing number of alternatives bound to each abstract functionality from 1 to 5. Note that we plotted with an horizontal black line the average time for the Java execution. Even in the worst scenario, in which we have 5 alternatives for each abstract functionalities, the overhead compared to the Java execution reaches an acceptable value of 9%. Consider that this testbed included a very large number of alternatives, 2500 in total, which is far from what we expect from a realistic application. If we consider that we typically have two alternatives, the average execution takes 25.83s instead of 25.06s.

Finally, Figure 6(d) shows, instead, how the execution time is affected, in our base scenario, by increasing in the number of requirements, which we varied from 2 to 10. Note that the ADAM Interpreter scales very smoothly, allowing applications to add multiple requirements without decreasing the ADAM performance. In general, from the above assessment we may conclude that the ADAM approach is feasible and its use introduces an acceptable (often negligible) overhead in the execution time of the overall application, especially when compared to the advantages as discussed in the Section IV.

VI. RELATED WORKS

Many existing works address the problem of uncertainty management and investigate adaptive software systems. The requirements engineering community has been particularly

---

http://code.google.com/p/adam-java/

---

7All the methods have a default implementation which sleeps for 50ms.
active w.r.t. this research challenge [20]. For example RELAX [24], a requirements language for adaptive systems, explicitly addresses uncertainty by enabling engineers to capture uncertainty in the requirements definition. Differently from our approach, RELAX achieves adaptation by relaxing non-critical requirements instead of relying on alternative or optional functionalities as in the ADAM approach. Souza et al. [21], instead, enable adaptation by conceiving systems in control loop in which a set of system parameters (i.e., variability points) is tuned at run-time to satisfy the system requirements. Finally, Wang et al. [23] describe a framework that exploits software variability and goal models to allow self-repair in cases of failure. From a model-driven perspective, we can mention [17] and [16]. The former focuses on functional adaptations and investigates the adoption of aspect oriented programming to weave new system configurations together with run-time models used to validate them. The latter also exploits aspect oriented programming, but focuses on non-functional properties of systems. It describes a middleware for run-time adaptation that chooses between alternative application variants. In addition, concerning the concept of Embedded Model, we can mention the work by Balz et al. [2], [3], that describes an approach to embed full model semantics into source code. In this case, the ultimate goal of the authors is to support the synchronization between the model and the implementation. This approach may be considered as an alternative solution to implement the ADAM concepts. From an architectural viewpoint, we may mention the foundational work on a three-layer architecture for software adaptation, described in [15], [22], which focuses mainly on functional adaptation. Cheng et al. in [5] investigate instead architecture-based adaptation through resource prediction, that is similar to the non-functional forecasts obtained via probabilistic model checking in ADAM, even if at architecture level. In addition, concerning monitoring, it is important to mention the work by Ehlers et al. [6], that propose an approach to localize performance Anomalies and enable adaptations through a rule-based expert system even if they focus only on response time. In addition, Fleurey et al. [11] define the impact of features (i.e., functionalities), as well as high-level adaptation rules to choose among them according to the context and achieve an adaptive behavior. In the same way, Floch et al. [12] adopt utility functions to determine which component implementation should be selected according to the context. The work by Ramirez et al. [19] represents, instead, an approach that may used to complement the ADAM solution. Indeed, it allows the automatic discovery of combinations of environmental conditions that produce requirements violations and latent behaviors in an adaptive system. Such approach may be used in ADAM to support engineers in designing the alternative implementations needed to face the discovered environmental conditions. All the mentioned approaches differ from ADAM in many aspects. Some of them, for example, do not consider non-functional aspects. Others, even if they focus on non-functional adaptation, do not offer the same degree of flexibility provided in ADAM that, for example, not only optimizes the non-functional behavior of the system, but also maximizes its reliability, as described in Section IV.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we presented ADAM, a novel approach that supports the effective development and operation of adaptive systems. To demonstrate its advantages, we used the proposed approach to implement a realistic mobile application and we assessed the overhead introduced by the approach and its scalability by performing a simulation campaign. In addition, to
encourage the adoption of the proposed approach and to allow
the replication of experiments, the ADAM implementation
has been released as an open-source tool. Finally, ADAM
currently supports parallelism only in the implementation
methods and not at the Activity Diagram level. We plan to
overcome this limitation in our future work.

APPENDIX

In this section we provide a brief introduction to the mathe-
matical concepts used throughout the paper: Markov Decision
Processes (MDPs) and rewards. MDPs are non-deterministic
Kripke structures with probabilistic transitions among states.
Formally, an MDP over a set of atomic propositions \( AP \) is a
tuple \( \langle S, s_0, F, \text{Steps} \rangle \), where:

- \( S \) is a finite set of states, \( s_0 \in S \) is the initial state and
  \( F \subseteq S \) is the set of final states;
- \( \text{Steps} \) is the transition function, such that for each \( s \in S \),
either \( \text{Steps}(s) = \text{Dist}(S) \) or \( \text{Steps}(s) \in 2^S \), where
\( \text{Dist}(S) = \{(s_1, p_1), \ldots, (s_k, p_k)\} \), with \( \sum_{j=1}^{k} p_j = 1 \),
is a discrete probability distribution over \( S \).

Notice that, if for a final state \( s_F \in F \), \( \text{Steps}(s_F) = \{(s_F,1)\} \), the self-edge is often not represented graphically.
MDPs may be augmented with rewards, through which one
can quantify a benefit (or loss) due to the residence in a specific
state. A reward \( \rho : S \to \mathbb{R}_{\leq 0} \) is a non-negative value assigned
to a state. They can represent information such as average
execution time, power consumption or usability. In MDPs,
each state \( s \) can be also annotated with rewards cumulated
from \( s \) to a final state \( s_F \). Given a state \( s \), there may be many
paths that connect it to one of the final state \( s_F \). Each of
these paths cumulates as reward the sum of the rewards of
the states in the path. If \( s \) is such that \( \text{Steps}(s) = \text{Dist}(S) \),
the overall cumulated reward for \( s \) is given by the weighted
sum w.r.t. \( \text{Dist}(s) \) of the path cumulated rewards for the
paths starting from \( s \). However, this quantity becomes an
interval because of the non-deterministic transitions. Indeed,
all the non-deterministic choices have associated a cumulative
rewards. To represent all these alternatives, each state \( s \) is
annotated with the interval \( \langle \min R(s), \max R(s) \rangle \) in which
all the non-deterministic cumulative rewards are comprised.
These values are computed as follows. If \( s \) is a final state,
\( \min R(s) = \rho(s) \), instead, if \( s \in S \setminus F \), when \( \text{Steps}(s) = \{(s_1, p_1), \ldots, (s_k, p_k)\} \), i.e., the transitions
outgoing from \( s \) form a probability distribution, \( \min R(s) = \sum_{j=1}^{k} p_j \times\) 
\( \min R(s_j) \), and, when \( \text{Steps}(s) = \{s_1, \ldots, s_m\} \), i.e., the
transitions outgoing from \( s \) represent a non deterministic
choice, \( \min R(s) = \min_{i \in [1, \ldots, m]} \min R(s_i) \). If one of the
paths on which the cumulative reward is computed contains
loops, the above equations recall themselves recursively. Iter-
avative numerical methods are used to solve these equations.
The termination condition of such methods is obtained by
checking when the maximum difference of the solutions from
successive iterations drops below a fixed threshold. The upper
bound of the interval, \( \max R(s) \), is computed analogously.
A comprehensive in-depth treatment of MDPs be found in [1].

ACKNOWLEDGMENTS

This research has been funded by the EU, Programme
IDEAS-ERC, Project 227977-SMScom and FP7-PEOPLE-
2011-IEF, Project 302648-RunMore.

REFERENCES

program code. In Proceedings of the 1st Workshop on Behaviour
B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. Software engineering
for self-adaptive systems: A research roadmap. Software Engineering
architecture-based self-adaptation through resource prediction. Software
software system monitoring for performance anomaly localization. In
ICACM. ACM, 2011.
model checking. In ICSE 2011. ACM.
software: continuous compliance of non-functional requirements. Formal
supporting specification, simulation and execution of dynamic adaptive
systems. Model Driven Engineering Languages and Systems, 2009.
Using architecture models for runtime adaptability. Software, IEEE,
prediction of service compositions through probabilistic model checking.
for automatic verification of probabilistic systems. TACAS, 3920, 2006.
F. Eliassen. Construction and execution of adaptable applications using
an aspect-oriented and model driven approach. In Distributed
[17] B. Morin, O. Barais, G. Nain, and J. Jezequel. Taming dynamically
exploring how uncertainty impacts behavior of dynamically adaptive sys-
Requirements-aware systems: A research agenda for re for self-adaptive
[21] V. Silva Souza, A. Lapouchnian, and J. Mylopoulos. System iden-
tification for adaptive software systems: a requirements engineering
[22] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to
[23] Y. Wang and J. Mylopoulos. Self-repair through reconfiguration:
A requirements engineering approach. In Proceedings of the 2009
IEEE/ACM International Conference on Automated Software Engineer-
language to address uncertainty in self-adaptive systems requirement.