Providing Assurances for Self-Adaptation in a Mobile Digital Storytelling Application Using ActivFORMS

Danny Weyns, Stepan Shevtsov, Sabri Pllana
AdaptWise Research Group, Department of Computer Science
Linnaeus University, Växjö Campus, Sweden
danny.weyns@lnu.se, stepan.shevtsov@lnu.se, sabri.pllana@lnu.se

Abstract**—***Self-adaptability enables a system to adapt itself to changes in its execution conditions and user requirements in order to achieve particular quality goals. However, assuring that the adaptation goals are satisfied poses complex challenges. We recently developed the ActivFORMS approach that aims to tackle some of these challenges, but further research is required to evaluate the approach. This paper presents the results of a study in which we applied ActivFORMS to a mobile storytelling application that employs a social recommender. The initial version of the application used a static recommender that could not deal with changing environment conditions, or take into account preferences of users. To that end, we added a self-adaptive layer on top of the application. The study results show that self-adaptation significantly increases the quality of recommendations compared to the initial version by: (1) enabling the social recommender to adapt to the quality of user input and unavailability of the GPS service, and (2) making the recommender adaptive to user preferences. Providing guarantees for these adaptation goals is crucial in this domain from a business perspective. The study results show the feasibility and effectiveness of ActivFORMS for a practical application; but they also underpin the need for an integrated verification approach for self-adaptive systems that combines offline with online verification.*

Keywords-self-adaptation, executable formal models, assurances

I. Introduction

Today customers expect particular levels of qualities from software systems, such as reliability, performance, flexibility, and scalability. However, contemporary software systems must operate under changing conditions, such as dynamics in the availability of resources, changing system goals, etc. As these dynamics may be difficult to predict, software engineers have to deal with incomplete knowledge at design time. Consequently, providing guarantees to assure the required qualities at runtime is complex. Self-adaptation is widely considered as an effective approach to cope with this complexity [17][10][9]. Self-adaptation allows a system to dynamically react to changes in requirements and adapt itself to maintain system goals.

In this research we focus on architecture-based self-adaptation, which provides a suitable level of abstraction to handle system dynamics that involve adaptation of components and their relations [16][11][15]. Central in architecture-based self-adaptation is the separation between the domain concerns, which are part of the managed system, and the adaptation concerns, which are part of the managing system. A well-known approach to structure the managing system is by means of a feedback loop divided in four components: Monitor, Analyze, Plan, and Execute [6]. These components share a common Knowledge base (hence, MAPE-K) that may contain data about the managed system, environment, adaptation goals, and working data that can be used by the MAPE components.

One of the key problems in engineering self-adaptive systems is providing guarantees that the adaptation goals are satisfied at runtime regarding of dynamics in the environment or the managed system [9]. Recent research in this direction suggests using formal models to tackle the problem, e.g., [18][22]. Existing approaches primarily focus on formal models of the Knowledge part of MAPE-K. Little attention has been devoted on formalizing the MAPE elements and providing guarantees about the behavior of the adaptation components themselves. E.g., important properties of a self-healing system may be: does the analysis component correctly identify errors based on the monitored data, or does the execute component perform repair actions in the correct order? Guaranteeing such properties is important to assure proper adaptation capabilities.

To tackle this problem, we recently developed an innovative approach called ActivFORMS (Active Formal Models for Self-adaptation) [7]. ActivFORMS uses an integrated formal model of the MAPE-K feedback loop. This formal model can be executed directly on a virtual machine, and interact with the managed system via probes and effectors. As a result, the properties that are derived from system requirements, and are verified before execution, are guaranteed at runtime. We have tested ActivFORMS for a number of simple self-adaptive systems, but more research is required to evaluate the approach.

This paper presents the results of a study in which we have applied ActivFORMS to a practical mobile storytelling application that employs a social recommender. The initial version of the application used a static recommender that could not deal with issues such as unavailability of GPS or user preferences. To deal with these issues, we added a self-adaption layer to the application. The results show that self-adaptation significantly increases the quality of recommendations of the application. Providing guarantees for the adaptation goals is crucial in this domain from a business perspective. The study shows the feasibility of ActivFORMS for a practical application, but also points to some issues that require further research.

The remainder of the paper is structured as follows. Section II briefly describes ActivFORMS and discusses related efforts. In Section III, we introduce the storytelling application with social recommender and discuss its inflexibility problem. Section IV explains how we tackled this problem by adding a self-adaptive layer to the applicaton. In Section V, we elaborate on the design of the MAPE-K feedback loop that realizes self-adaptation. Section VI discusses verification of required system properties. In Section VII, we evaluate the self-adaptive solution by comparing it to the initial version. Finally, conclusions and directions for future work are presented in Section VIII.

II. ACTIVFORMS APPROACH

ActivFORMS follows the three-layered reference model proposed by Kramer and Magee [11], see Figure 1. The bottom layer comprises the managed system that implements domain-specific functionality. ActivFORMS is responsible for adaptation of the managed system comprising two layers: Active Model and Goal Management. Active Model realizes a MAPE-K feedback loop that monitors the managed system and adapts it according to some adaptation goals. The MAPE-K feedback loop consists of a formal model that is directly executed by a virtual machine, taking input from probes and performing adaptation actions via effectors.

Goal Management comprises a tree-based goal model where nodes have associated MAPE-K models to realize adaptations. Goal management monitors goals via the virtual machine. When a goal violation is detected, the models associated with an alternative goal that matches the changing conditions are used to update the deployed models via the virtual machine. Goal models can be updated at runtime. We refer the interested reader to [7]. In this paper, we focus on the Active Model.

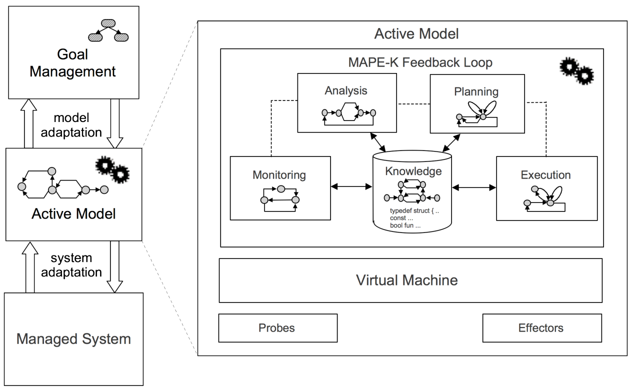


Figure 1. The ActivFORMS approach

A. *Active Model*

In our research we model a MAPE-K feedback loop as a network of timed automata. A timed automaton is a finite-state machine extended with clock variables. Automata can communicate through shared data or by sending signals. Signals being sent are marked with "!" and received signals are marked with "?" For the specification of required properties, we use Timed Computation Tree Logic (TCTL). TCTL allows checking individual states of the state space of the system as well as traces over the state space. The latter makes it possible to verify reachability, safety, and liveness properties.

The ActivFORMS virtual machine is able to initiate and execute a formal model and supports interaction with the managed system via probes and effectors. In addition, the virtual machine supports verification of goals at runtime (currently only verification of goals based on actual state), as well as live updates of the formal model. The latter functionalities are not further studied here. We refer the interested reader to [7].

The execution of the active model conforms to the semantics of networked timed automata. Internally, the virtual machine transforms the MAPE-K formal model to a graph representation. The model can be triggered by input from the managed system/environment or by time. Time triggering is based on an internal clock of the virtual machine. The ratio between a clock tick and real time can be configured in the virtual machine.

The virtual machine runs on a JVM. To support engineers, ActivFORMS provides a set of Java classes to implement probes and effectors. Probes track the managed system and possibly the environment and transfer data to Monitor automata of the MAPE-K feedback loop, while Effectors transfer actions generated by Execution automata to the managed system.

B. *Related Efforts*

A vast body of work on formal approaches for self-adaptive systems exists; [20] provides a recent overview of the state of the art. Here we discuss some key references and highlight the types of applications the approaches have used for validation.

[13] presents an approach to create and verify formal models for adaptive systems using Petri Nets. The models can automatically be transformed into executable programs. The approach is applied to a GSM-based audio streaming protocol. As a result, development time and reliability of the system improved. [21] uses discrete time Markov chains to design models of the system and environment and a Bayesian learner to adapt uncertainties. A feedback loop detects requirement violations and modifies the system goals accordingly. The approach is demonstrated in a simple Web-service application setting. [22] presents ADAM, a tool that transform UML diagrams into probabilistic decision models that are used at runtime to guarantee optimal performance. The approach is used for a ShopReview mobile application and compared to Java implementations using simulation. The results show that the approach supports engineers with creating reliable software and decreases development time, but there is some overhead compared to "hard coded" solutions.

In contrast to existing work, ActivFORMS directly executes the formal models to realize self-adaptation. We apply the approach to a practical application, provide formal guarantees, and compare the adaptive version of the application with the initial non-adaptive version.

III. DIGITAL STORYTELLING WITH SOCIAL RECOMMENDER

This section describes the initial, non-adaptive mobile digital storytelling application, and explains the inflexibility problem of the social recommender.

A. *mDS-SR application*

Mobile Digital Storytelling with Social Recommender (mDS-SR) is a native iOS application based on a storyboard technology [3]. The application provides functionality for creating digital stories on a mobile device and sharing them. Concretely, mDS-SR allows:

- Starting a new story or selecting an existing one;
- Adding and rearranging images from a local collection on a canvas so they form a sequence;
- Viewing similar photos (recommendations) from Flickr, and inserting them into a sequence;
- Sharing photos to Flickr so that they can be used in the stories of other people;
- Recoding audio, adding a soundtrack or tags to a story;
- Creating a narrative (a video) out of the story and sharing it to YouTube or by e-mail.

The mDS-SR application can be used as a tool for collaborative learning, to share touristic experiences, etc.

B. *Social Recommender*

The social recommender supplements story creation and gives a possibility to inherit experience of other users that used the application in similar situations. In general, recommender systems provide suggestions with the help of collaborative filtering, content-based filtering, or a combination of these two methods [1]. The choice for filtering methods depends on the goals of application at hand. Collaborative filters work better when there is a big amount of data available about users, their preferences and community behavior. Content-based filters on the other hand, are very good in recommending new/unrated items and satisfy people with unique preferences [2].

For the mDS-SR application, we use content-based filtering. The choice was based on information gathered during requirements elicitation. Most recommendation systems with content-based filtering rely on tags and standard item data (title, rating, creation date, etc.) when calculating recommendations. mDS-SR also relies on it, but in addition, we added a new parameter in the recommender: it compares the geographical location of the user with the location where the recommended photo was created or uploaded. As for the source of suggested content, we adopted the idea of Guy & Carmel [4] to combine social media with recommender systems, hence social recommender. In particular, mDS-SR uses Flickr as a social service for providing recommendations. Flickr provides an API to send a query for recommendations with parameters (geo-location, text for search, etc.). Flickr then responds with matching photos, i.e., recommendations. In addition, Flickr can sort photos according to relevance, interestingness, or the date when photos were taken or uploaded. Flickr calculates relevance and interestingness based on protected algorithms.

As a result, the social recommender of the mDS-SR application uses a combination of geo-locations, story title, tags, and sorting methods to provide social recommendations. By combining subsets of these parameters, a variety of recommender algorithms can be implemented. In the mDS-SR application, we selected five of them based on initial testing and feedback from participants that used the application in the field. Table I lists the algorithms with their key properties. When a user starts working with the application, the recommender loads 20 photos generated by the recommender algorithms, four photos by each algorithm. The photos are shown in an interleaved way, such that they have approximately equal chances of being chosen.

TABLE I. ALGORITHMS IN MDS-SR

#	The algorithm	Sort by	Gps based	Text based	Tag based
1	Search for story tags in tags of photos	relevance			+
2	Search for story name in picture title, description, tags of photos	relevance		+	
3	Search in current geo location for photos containing story name in photo title, description or tags	relevance	+	+	
4	Search in current geo location for photos uploaded before 2013-05-30 (first experiment with the app).	posting date, descending	+		
5	Search current geo location for photos uploaded after 2000-01-01.	interesting-ness, descending	+		

Figure 2. shows the user interface of the mDS-SR application. The left hand side shows the UI for naming and tagging a story; the right hand side shows the UI for working with images and the recommender.

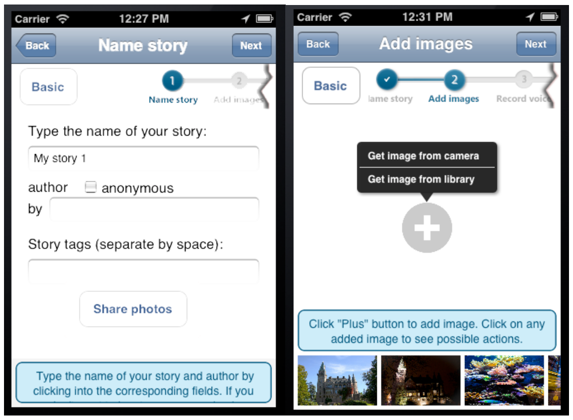

Carrier
12:27 PM
Back
Name story
Next
Basic
Name story
Add image
Type the name of your story:
My story 1
author by
anonymous
Story tags (separate by space):
Share photos
Type the name of your story and author by clicking into the corresponding fields. If you
Carrier
12:31 PM
Back
Add images
Next
Basic
lame story
Add images
Record voic
Get image from camera
Get image from library
Click "Plus" button to add image. Click on any added image to see possible actions.

Figure 2. Initial mDS-SR application: naming and tagging story (left), working with images and recommender (right).

C. *Problem description*

The recommender algorithms of the mDS-SR application are statically defined, but they rely on dynamic parameters. For example, if the Global Positioning System (GPS) is turned off or the system is not available, all algorithms that are based on GPS will provide no recommendations. Hence, only 8 photos will be visible instead of the intended 20 photos (as three algorithms depend on GPS, see Table I). A similar situation occurs when a user gives a very complex name to a story for which Flickr cannot find matching photos. In this case, algorithms that are text-based (see Table I) will not provide recommendations. Furthermore, due to the use of a fixed set of algorithms, the social recommender is not able to adapt dynamically to user preferences. This inflexibility of the social recommender has a negative effect on user experience and consequently may affect the competitiveness of the application on the market.

IV. SELF-ADAPTIVE SOCIAL RECOMMENDER

We start this section with listing the requirements for self-adaption to tackle the flexibility problem. Then, we give a high-level overview of the architecture of the self-adaptive system. The detailed design and verification are presented in the following sections.

A. *Requirements for self-adaptation*

Self-adaptation should satisfy the following requirements:

R1: The self-adaptive system should work correctly, i.e., without errors, deadlocks, or time locks.

R2: When GPS is not available or Flickr provides no response to a query, self-adaptation should switch off the algorithms that require these resources and provide recommendations from alternative algorithms; always 5 in total.

R3: The social recommender should be able to adapt dynamically to user preferences; i.e., the more an algorithm is selected the higher position it should get in the recommender.

To ensure sufficient recommendation algorithms and provide the required flexibility, we added a set of six extra algorithms to the mDS-SR application (# 6-11 in Table II).

TABLE II. ADDITIONAL ALGORITHMS FOR MDS-SR

#	The algorithm	Sort by	Gps based	Text based	Tag based
6	Search for story tags in picture title, description and tags of photos	interesting-ness, descending			+
7	Search in current geo location for photos containing story tags in tags of photos	relevance	+		
8	Search in current geo location for photos containing story name in photo title, description or tags and taken before 2010-01-01	taken date, ascending	+	+	
9	Search for story tags in picture title, description and tags of photos, tag mode "all"	taken date, ascending			+
10	Search for first story tag in picture title, description and tags of photos	taken date, descending			+
11	Search for first word from story title in picture title, description and tags of photos	relevance			+

The choice for the algorithms is based on pilot tests with users. The total set of algorithms offers 5 tag-based algorithms that do not depend on GPS or Flickr response. Furthermore, the additional algorithms can provide richer content than algorithms #1-5. Tracking the choices of recommended content and selecting algorithms based on that enables the system to adapt the recommender algorithms to the preferences of the user.

B. *High-level architecture of the self-adaptive layer*

Figure 3. shows the high-level architecture of the mDS-SR application with self-adaptation. We first focus at the bottom part that contains the main components of the initial digital storytelling application.

Central to the mDS-SR application is the *Local Storage* that maintains the data of all stories created by the user as well as the recommender algorithms that can be used by the different components. Any data of a story changed by a component automatically updates the story data in the repository. The *Start New Story* component provides functionality to a user to create a new narrative, which includes adding a story name, tags, and author. The *Select Existing Story* component allows a user to select a previously created story and update basic story data. The *Edit Story* component offers different functions to the user to work with story content, such as selecting images and adding sound. Edit Story can interact with the GPS sensor and Flickr when it uses recommender algorithms that require these resources. The *Play Story* component provides functionality to finalize the creation of a story, i.e., producing, watching and sharing a video-narrative. Play Story can interact with YouTube that serves as a platform for sharing stories.

We now look at the self-adaptive layer that is added on top of the application to tackle the inflexibility problem described in Section 3. Here we give a general overview of the self-adaptation components; the detailed design is discussed in the following section. The self-adaptation layer consists of a MAPE-K feedback loop, which is local to every mobile device. The *Knowledge* repository maintains data relevant to self-adaptation, including representations of the different algorithms with ratings that represent the user preferences, the status of the GPS, data about the latest response of Flickr, data about plans, etc. MAPE components have read/write access to the repository. The *Monitor* component uses a *Probe* to monitor changes in the story title, user interactions with the recommender interface (selecting a photo or absence of actions during a certain period), GPS status, and the number of photos provided by the used algorithms. Based on any of these events, the *Analyze* component analyses the situation. If GPS is not available or Flickr does not provide a response to a query, *Plan* is triggered to change the used recommender algorithms. Based on the input provided by the user, Analyze may adapt the order of the recommender algorithms, which then in turn triggers the Plan component to change the used recommender algorithms. Finally, the *Execute* component communicates with the Effector to modify the algorithms used by the social recommender of the application according to the developed plan.

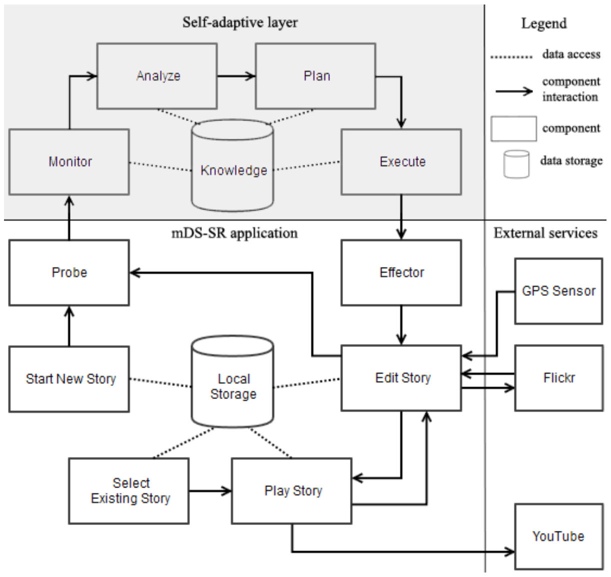


Figure 3. Architecture of mDS-SR application with a self-adaptation layer

V. SELF-ADAPTATION DESIGN

We now explain the detailed design of the self-adaptation layer. Subsequently, we discuss the probe, the MAPE-K components, and the effector. We conclude with explaining how ActivFORMS is configured and started to apply self-adaptation.

A. *Probe*

The probe gathers the data that is required by the MAPE-K loop to realize self-adaptation. Probe connects the Monitor with the mDS-SR application using the following signals:

- gpsUpdated: signal that says when the status of the GPS changes (on/off);
- recomLoaded: signal that indicates that new recommendations (photos) from Flickr are loaded in the application interface;
- textUpdated: signal that indicates changes in story title;
- flickrResp: signal that provides the response of Flickr (number of photos) to a query with the story title;
- photoAdded: signal that indicates that a user clicked on any recommended photo; the signal includes the algorithm ID of the clicked photo;
- noPhotos: signal that says that a used algorithm did not provide a single recommendation; the signal includes the algorithm ID.

An excerpt of the Probe class implementation is shown in Code block 1. The virtual machine requires a unique identifier for each signal that is used by a probe to communicate between the application and the formal model of the MAPE-K feedback loop. Identifiers are matched in the Probe constructor, e.g. the identifier *signalGpsUpdated* provided by the application is matched with the *gpsUpdated* signal used in the formal model. The ActivFORMS virtual machine (ActivFORMSEngine) provides a *Send* function that allows the probe to communicate with the formal model. The Send function takes three parameters: channel identifier, the probe object, and data to be send (in string format). The third parameter is optional.

Code block 1. Part of mdsProbe class

```
public class mdsProbe implements Synchronizer {
int signalGpsUpdated;
int signalTextUpdated;
ActivFORMSEngine engine;
  public mdsProbe (ActivFORMSEngine gotEngine)  {
        this.engine = gotEngine;
        signalGpsUpdated = engine.getChannel("gpsUpdated");
        signalTextUpdated = engine.getChannel("textUpdated");
        … //match other signals with their identifiers
  }
   public void sendTextSignal() {
        engine.send(signalTextUpdated, this);
   }
   public void sendGpsSignal(int gpsIsWorking) {
        String gps = Integer.toString(gpsIsWorking);
        engine.send(signalGpsUpdated, this, "GPSactive="+gps);
   }
}
```

A concrete implementation example is shown in Code block 2. The probe can be triggered via the *textsChanged* or *gpsState* functions. This first one is triggered when the user changes the story title; the second one when the GPS turns on/off. Both functions send a signal to the formal model; when the GPS changes, the status is also sent (0 for “off”, 1 for “on”).

Code block 2. Creating a Probe, sending signals

```
probe = new mdsProbe(engine);
public void textsChanged () {
         probe.sendTextSignal();
 }
public void gpsState (int gpsIsWorking) {
         probe.sendGpsSignal(gpsIsWorking);
}
```

B. MAPE-K Behaviors

To model the MAPE-K behaviors, we used a set of formal templates for self-adaptive components [14] and to model and test the automata, we used the Uppaal [8] tool.

(1) KNOWLEDGE

The knowledge that is shared by the MAPE behaviors (see Figure 3.) is structured in three parts (see Code block 3). First, the knowledge of the managed system comprises a representation of the recommender algorithms, each with an identifier, a rating, and three Booleans indicating that the algorithm is in use, whether it is GPS based, or text based. This part also maintains knowledge about the number of algorithms in use, the ID of the algorithm picked, and the ID of the algorithm that did not provide recommendations from Flickr. Second, knowledge keeps track of information in the environment, i.e., the status of GPS and the response of Flickr to a query with story title. Third, knowledge maintains data about the adaptation process itself, i.e., a flag (Boolean) that indicates that text-based algorithms can be used, a flag that indicates that the recommender has been updated, the ID of the plan that is selected for adaptation, and the number of currently used recommender algorithms (which should be 5 at all times).

Code block 3. Knowledge

```
//Managed System knowledge
typedef struct {
int ID;  int Rating;  bool Used;  bool GpsBased; bool TextBased;
} RecAlgorithm; //an algorithm
const int algTotal = 11; //number of algorithms
const int algToUseInRec = 5; //number of simultaneously used algs
RecAlgorithm algs[algTotal]; //array of algorithms
int algChosen; // algorithm ID of a picked photo
int algNumber; // ID of algorithm that provide no recommendations

//Environment knowledge
int Resp; // Flickr response to a query with story title
bool GPSactive=YES; // GPS status

//Adaptation knowledge
bool AnswerOnText=1; // flag indicating that text based algs can be used
bool recomUpdated; // calculations are finished, recommender updated
int PlaN; // ID of executed plan
int curUsedAlgsNumber=5; // number of currently used algs
```

(2) MONITOR

The Monitor component consists of four independent processes that handle the following monitoring tasks:

- MonitorGPS: monitors the status of the GPS;
- MonitorText: monitors changes of the story title;
- MonitorNoActivity: tracks whether a user clicks or not on new loaded photos within a given time period;
- MonitorAlgs: monitors when a photo is clicked, and whether an algorithm does not provide recommended photos in response to a query; the monitor updates the ratings of the algorithms accordingly.

Using different monitors separates concerns and allows the behaviors to work in parallel. The latter is required, e.g., to track the GPS status (MonitorGPS) in parallel with monitoring user clicks (MonitorNoActivity).

MonitorGPS

The MonitorGPS behavior (Figure 4.) is responsible for tracking the availability of GPS. The monitor receives the *gpsUpdated* signal when the GPS status changes. It then updates the knowledge (*GPSActive=!GPSactive*) and triggers the Analyze behavior (*analyzeGPS!*).

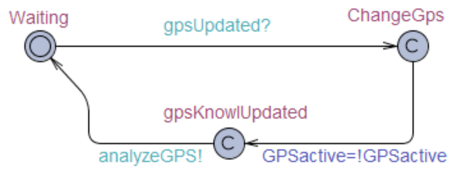


Figure 4. MonitorGPS behavior

MonitorNoActivity

When the application loads recommended content (*recomLoaded?*), the *MonitorNoActivity* behavior (Figure 5.)

starts monitoring interactions of the user with the new content (*WaitingForActions*). If none of suggested photos is clicked within 30 ticks of the clock it is assumed that the user is not interested in the content. Hence, the rating of all used algorithms is decreased (*decreaseRatingsof UsedAlgs(1)*) and the analyzer is triggered (*analyzeRatings!*).

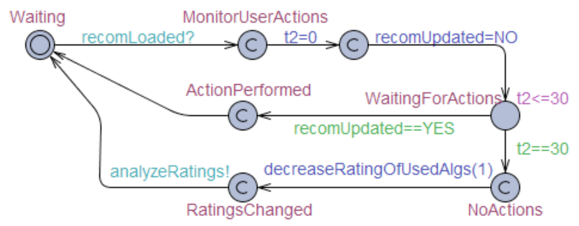


Figure 5. MonitorNoActivity behavior

MonitorAlgs

The *MonitorAlgs* behavior (Figure 6.) consists of two branches triggered by signals from the application. First, the *noPhotos* signal is received when an algorithm in use does not provide a single recommended photo in response to a query. The ID of the algorithm, which is received together with the signal from application, is used to decrease the rating of the algorithm (*decreaseRating(algNumber,2)*). Then the analyzer is triggered (*analyzeRatings!*). Second, when the user clicks a recommended photo, the *photoAdded* signal is received. As a result, the rating of the algorithm that provided the suggestion is increased (*increase(algChosen,2)*). Next the analyzer is triggered.

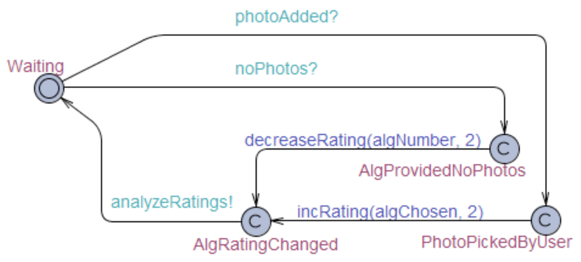


Figure 6. MonitorAlgs behavior

MonitorText

When *MonitorText* (Figure 8.) is triggered by a text update in the application (*textUpdated?*), it generates a query with story title as main parameter and sends it to Flickr (*flickrQuery!*).

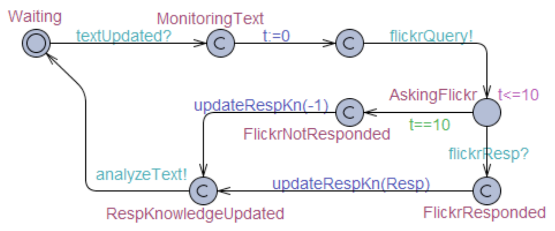


Figure 7. MonitorText behavior

In case Flickr does not respond within 10 clock ticks, the knowledge (*AnswerOnText*, see Code block 3) is updated with a failure signal (*updateRespKn(-1)*). Otherwise, the knowledge is updated with the number of available photos returned from the query (*updateRespKn(Resp)*). Monitoring the response of Flickr based on the story title as a main search parameter is important for selecting proper algorithms for the social recommender.

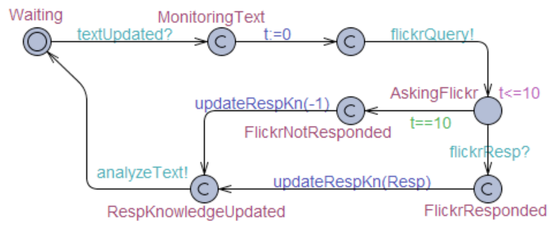


Figure 8. MonitorText behavior

(3) ANALYZE

The Analyze behavior (Figure 9.) consists of three branches (from *Waiting*) triggered by signals from the monitor processes.

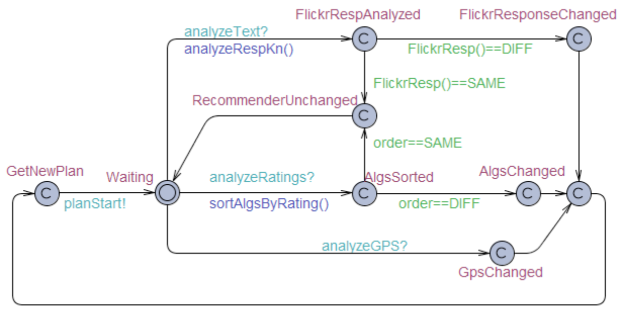


Figure 9. Analyze behavior

First, when the GPS status changes (*analyzeGPS* signal) the analyzer immediately triggers the Plan behavior to start planning (*planStart!*). Second, changes in texts of a story may or may not imply adaptations in the used algorithms. When the Analyzer receives the *analyzeText?* signal it starts analyzing (*analyseRespKn()*) the response that the *MonitorText* behavior received from Flickr (see MonitorText above). If Flickr provided no photos to the query, its response is considered "bad," otherwise; it is considered "good." The current response of Flickr (good or bad) is compared with the previous response. If the response is the same, no changes of the algorithms are required and the Analyze behavior returns to the *Waiting* state. In the other case (*FlickrResponseChanged*) the Plan behavior is triggered. Third, when the Analyze behavior receives the *analyzeRatings* signal, it sorts the recommender algorithms by rating (*sortAlgByRating()*) and then checks whether the order of the algorithms has changed. If the order has not changed (*order==SAME*) the analyzer returns to the *Waiting* state; otherwise (*order==DIFF*), the Plan behavior is triggered to start changing the algorithms used by the application.

(4) PLAN

The Plan behavior (Figure 10.) creates a plan for adapting the recommender algorithms depending on GPS availability and the response of Flickr to a query with a story title. If GPS is not available (*GPS==OFF*) and there are no photos provided by Flickr (*photosInRespToStoryTitle==0*), a plan that only includes algorithms with tags (*PlaN=Count_only_on_tags*) is generated.

If photos are returned (*photosInRespToStoryTitle>0*), a plan that depends on text (*PlaN=Count_only_on_text*) is generated. If GPS is available, but Flickr does not provide recommendations (*photoinRespToStoryTitle==0*), a plan that only depends on GPS (*PlaN=Count_only_on_gps*) is generated. If both services are available, a plan is generated that uses all available algorithms (*PlaN=Count_only_on_gps_and_text*). Once the plan is generated, the Execute behavior is triggered (*exec!*).

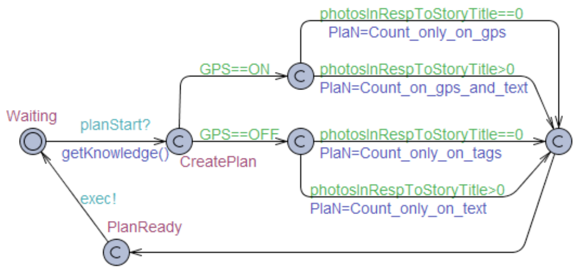


Figure 10. Plan behavior

(5) Execute

When the Execute behavior (Figure 11.) is triggered (*exec?*) it changes the *used* flags (see Code block 3) of the algorithms in the Knowledge based on their rating and the selected plan using the *changeAlgs(int PlanN)* function (see Code block 4).

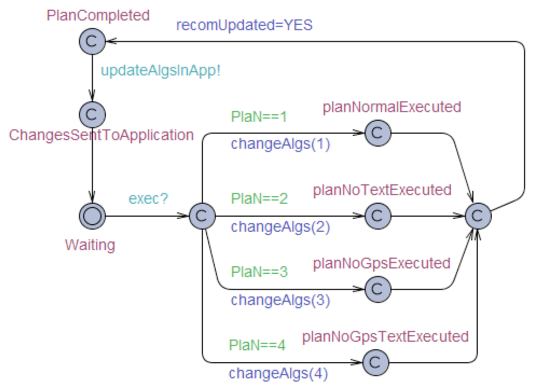


Figure 11. Execute behavior

Depending on the plan selected by the Analyzer the recommender algorithms are changed by the Executor (PlaN==4 for *planNoGPSTextExecuted*, *PlaN==3* for *planNoGPS-Executed*, *PlaN==2* for *planNoTextExecuted*, *PlaN==1* for *planNormalExecuted*). [1] Subsequently, the Execute behavior updates the recommendation status (*recomUpdated=YES*) and invokes the update of the algorithms in the application (*updateAlgsInApp!* via the Effector, see below).

Code block 4. changeAlgs function

```
void changeAlgs(int PlanN){
  algsUsedInRec:=0;
  for (i:=0; i<algTotal; i++) {
       algs[i].Used=0;
       if (PlanN== Count_on_gps_and_text
        || (PlanN==Count_only_on_gps && algs[i].TextBased==0)
        || (PlanN==Count_only_on_text && algs[i].GpsBased==0))
        || (PlanN==Count_only_on_tags && algs[i].TextBased==0
        && algs[i].GpsBased==0) {
          algs[i].Used=1;  algsUsedInRec++;
        }
     }
    curUsedAlgsNumber=algsUsedInRec; //needed for verification
}
```

[1] PlaN==1 corresponds to Count_only_on_gps in the planner, PlaN==2 corresponds to Count_on_gps_and_text, PlaN==3 corresponds to Count_only_on_text, PlaN==4 corresponds to Count_only_on_tags

C. *Effector*

The Effector (Code block 5) invokes the actions of the Execute behavior to the mDS-SR application. Similar to Probe it associates a unique identifier to each channel through which the Executor behavior communicates (e.g., *updateAlgs= engine.getChannel("updateAlgsInApp")*). Each channel is registered in the engine to receive particular data from the model (e.g., *engine.register(updateAlgs, this, "algs")*). To trigger a Flickr query there is no need to read data from the model; hence, the channel is registered to receive a plain signal without extra data (*engine.register(flickrQuery, this)*. The effector uses the *receive* function to communicate data from the formal model to the application. The function is triggered when the formal model sends a signal to the application. The receive function takes two parameters: a channel ID and the data received via that channel. The received data is formatted as a String-Object hash map and the application needs to be instrumented to process this format. When the application receives data via the channel associated with *updateAlgs* it processes the parameters of the recommender algorithms (rating, usage flag) and updates the recommendations of the application accordingly. In case a *flickrQuery* signal is received, a Flickr query is invoked (Probe tracks the response).

Code block 5. Part of mdsEffector class

```
mdsEffector effector = new mdsEffector(engine, this); // in MDS class

public class mdsEffector implements Synchronizer {
int updateAlgs;
int flickrQuery;
ActivFORMSEngine engine;
MDS mds;
public mdsEffector(ActivFORMSEngine engine, MDS mainMDS) {
      this.engine = engine;
      this.mds = mainMDS;
      updateAlgs= engine.getChannel("updateAlgsInApp");
      engine.register(updateAlgs, this, "algs");
      flickrQuery = engine.getChannel("flickrQuery");
      engine.register(flickrQuery, this);
  }
@Override
public void receive (int channelId, HashMap<String, Object> recData) {
  if (channelId == updateAlgs) {
        mds.parseData(recData);
  }
  else if (channelId == flickrQuery)  {
        mds.evaluateText();
  }
}
```

D. *Initialization and Starting ActivFORMS*

To execute the formal model, the ActivFORMS engine needs to be initialized and started (Code block 6). The *mDS.xml* file contains the MAPE-K model produced by Uppaal. Clock variables progress with ticks that need to be associated with real time units. In the mDS-SR application, we associated 1000 ms with a tick (*engine.setRealTimeUnit(1000)*). Finally, the ActivFORMS virtual machine can be started (*engine.start()*).

Code block 6. Setting ActivFORMS engine

```
ActivFORMSEngine engine;
engine = new ActivFORMSEngine("mDS.xml");
engine.setRealTimeUnit(1000); //1 clock tick = 1 second
engine.start();
```

VI. Verification

Before the formal model is deployed for execution, the self-adaptive system must be verified to assure requirements R1 to R3 (see Section 3). We explain the models of the managed system and environment used for verification. Then we discuss the verification of properties derived from the requirements.

A. *Models of the Managed System and Environment*

To verify the MAPE-K loop, we need proper models of the managed system and the environment that represent the relevant behavior related to self-adaptation. The Environment behavior (Figure 12.) comprises two parts: (i) it randomly changes the status of the GPS and sends a corresponding signal to the managed system (*gpsChanged!*), and (ii) when a query to Flickr is requested (by the MonitorText behavior), it either returns a response (*flickrResp!*) or not (*NoFlickrResp*).

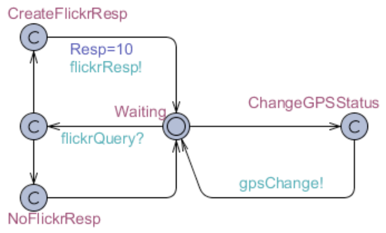


Figure 12. Environment behavior

ManagedSystem (Figure 13.) comprises three parts: (i) it updates the GPS status and signals the monitor (gpsUpdated!), (ii) it randomly generates clicks on photos (*photoAdded!*) or updates of story title (*textUpdated!*), and (iii) it randomly emulates that an algorithm has generated no photos (*noPhotos!*).

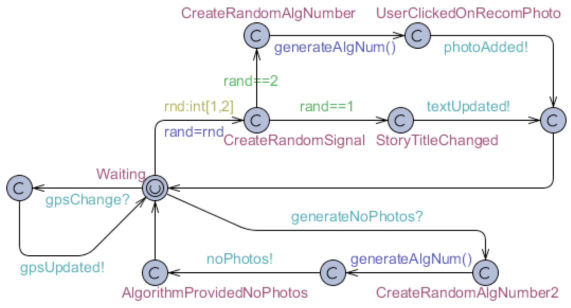


Figure 13. ManagedSystem behavior

B. *Verifying the System Requirements*

Requirement R1 requires that the self-adaptive system works correctly, i.e., without errors, deadlocks, or time locks. To guarantee that, we verified several properties:

```
P1.  A[] not deadlock
P2. MonitorGPS.ChangeGPS --> Analyze.GpsChanged
P3. Analyze.GpsChanged --> Plan.CreatePlan
P4. Plan.CreatePlan --> Execute.PlanCompleted
```

Property P1 (provided by Uppaal) allows checking deadlock freeness of a model. In addition, we have verified a series of properties; here we present three examples that show correctness of the interaction between MAPE components. Property P2 guarantees that if the monitor changes the status of the GPS, the Analyzer will eventually detect this. P3 subsequently guarantees that the change will lead to a plan to deal with it. Finally, P4 guarantees that the plan is executed.

R2 requires that when GPS is not available or Flickr provides no response to a query, the self-adaptation system should switch off the algorithms that require these resources and provide recommendations from alternative algorithms (always 5 in total). To that end, we verified the following five properties:

```
P5. A[] Execute.planNoTextExecuted imply algs[2].Used==NO
P6. A[] Execute.planNoGpsTextExecuted imply algs[2].Used==NO
P7. A[] Execute.planNoGpsExecuted imply algs[3].Used==NO
P8. A[] Execute.planNoGpsTextExecuted imply algs[3].Used==NO
P9. A[] curUsedAlgsNumber==5 //counted in Execute (Code block 4)
```

Property P5 guarantees that if no response if provided by Flickr, algorithm with ID 2 will not be used. Property P6 additionally guarantees that if the GPS is off the same algorithm will not be used. P7 and P8 guarantee similar properties for the GPS-based algorithm with ID 3. We verified the correctness of all variants for the other algorithms. Finally, P7 guarantees that there are always five algorithms used by the social recommender, independent of available external services.

R3 requires that the more an algorithm is selected the higher position it should get in the social recommender. To guarantee this requirement, we specified two properties. As an algorithm gets a position in a recommender depending on it's rating (number of times picked by a user), we verified property P10 that guarantees that the algorithms are sorted according to their rating after execution. In addition, property P11 guarantees that the rating of an algorithm that provides a recommendation and is selected by the user always increases by a given reward.

```
P10. A[] Execute.ChangesSentToApplication imply
        forall(i:int[0,algTotal-2]) algs[i].Rating>=algs[i+1].Rating
P11. ManagedSystem.UserClickedOnRecomPhoto -->
        MonitorBasic.AlgRatingChanged &&
          algChosenPrevRating = algs[algChosen].Rating-reward
```

The verification times are summarized in the Table III.

TABLE III. Verification time of system properties

Property	Verification time, sec.	Property	Verification time, sec.
P1	125	**P7**	62
P2	71	**P8**	62
P3	66	**P9**	56
P4	68	**P10**	54
P5	63	**P11**	77
P6	60		

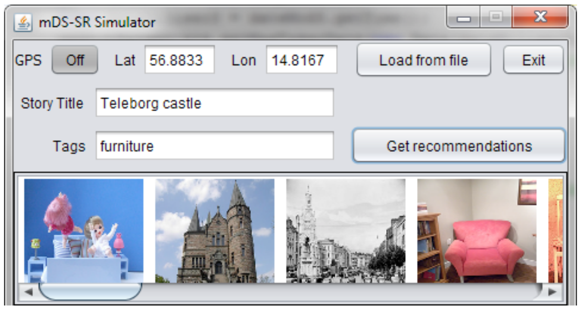
As the models of the environment and the managed system do not cover all possible conditions, which would lead to an explosion of the state space, exhaustive verification would require additional verification at runtime.

VII. Evaluating the Application

We compare the quality of recommendations of the initial version of the application with the self-adaptive version using a concrete scenario. As ActivFORMS is Java-based, we developed a simulating environment around the social recommender to test self-adaptation. We start by introducing the simulator. Then we show how the recommender improves with self-adaptation using a scenario with changing conditions. We conclude with measurements of performance overhead.

A. *mDS-SR Simulator*

We developed a simulator for the social recommender that provides all the functionality to test self-adaptation. The simulator gathers all parameters that influence recommendations in a single window (see Figure 14.) and is able to show suggested content as a set of horizontally sliding images. The location and GPS status can be manually updated by the user (*Off* button) or imported from a file (*Load from file* button). Recommendations are directly received from Flickr (via the *Get recommendations* button).



GPS Off Lat 56.8833 Lon 14.8167

Story Title Teleborg castle

Tags furniture

Figure 14. Social recommender simulator written in Java.

The mDS-SR simulator runs in the same Java environment as ActivFORMS. The latter includes the ActivFORMS library, the formal models of the MAPE-K feedback loop, the code for initialization and starting ActivFORMS (Code block 6), the Probe (Code blocks 1 and 2), and the Effector (Code block 5).

B. *Adaptation Results*

To evaluate the quality of the adaptation, we introduce the notion of *selection probability* (*sp*) that is defined as follows:

$p_i = R_i / \sum_{k=1}^{N} R_k$, where:

N // total number of recommendation algorithms

R_i // rating of algorithm *i*

$sp = 100 * \sum_{k=1}^{Nu} p_k$, where

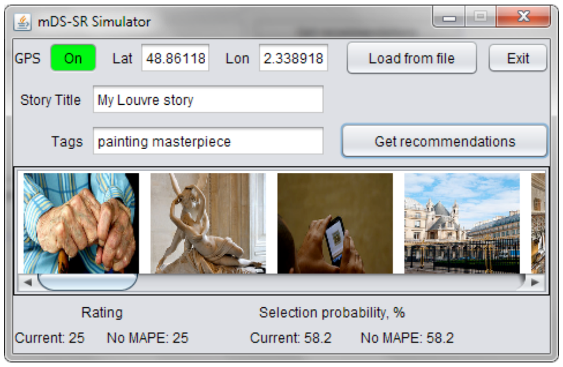
Nu // the number of algorithms used

The variable p_i represents the probability that a photo provided by algorithm *i* is selected. The probability depends on the algorithm rating R_i and is relative to the probabilities of all algorithms. The selection probability *sp* sums the probabilities of the algorithms that provide content for the user to select, multiplied by 100 to obtain a percentage. E.g., if all external services are available and the five best algorithms have a rating of 5 each, while the other algorithms have a rating of 3, the rating of the used algorithms is 5*5=25 (the total rating of all algorithms is 25+6*3=43) and the selection probability is:

$sp = 100*(R_1+R_2+R_3+R_4+R_5)/(R_1+R_2+\ldots+R_{11})$;

$= 58.2\ \%$

This situation is shown in Figure 15. The values for *Current* refer to rating and selection probability with adaptation, while values with *No MAPE* refer to the situation without adaptation.



GPS On Lat 48.86118 Lon 2.338918

Story Title My Louvre story

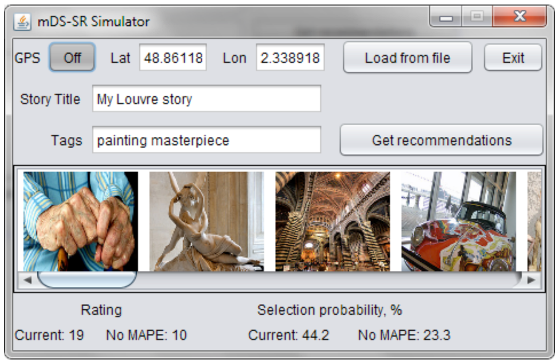
Tags painting masterpiece

Rating: Current: 25 No MAPE: 25

Selection probability, %: Current: 58.2 No MAPE: 58.2

Figure 15. The recommender in normal conditions (GPS is on, story title is sufficient for getting photos from Flickr in response)

When the GPS is turned off, some of the algorithms will no longer provide recommendations. This will affect the ratings and selection probability as shown in Figure 16. Both the total rating of the recommendations that are available and the selection probability are significantly better with adaptation (*Current*) as without (*No MAPE*).



GPS Off Lat 48.86118 Lon 2.338918

Story Title My Louvre story

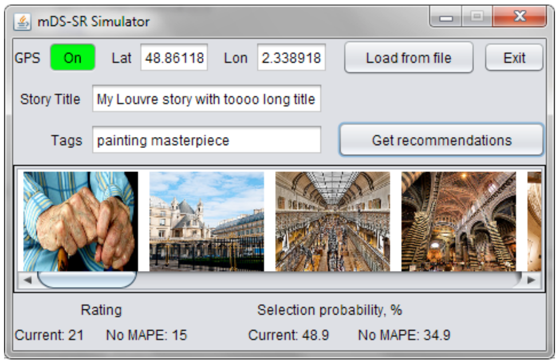
Tags painting masterpiece

Rating: Current: 19 No MAPE: 10

Selection probability, %: Current: 44.2 No MAPE: 23.3

Figure 16. The recommender with GPS turned off

When Flickr does not provide a response to a query of a story title (e.g., the title is too specific to find matching photos), the ratings and selection probability change as shown in Figure 17. The values show a further improvement with adaptation.



GPS On Lat 48.86118 Lon 2.338918

Story Title My Louvre story with toooo long title

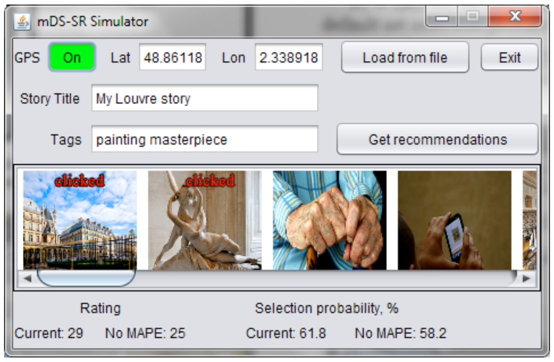
Tags painting masterpiece

Rating: Current: 21 No MAPE: 15

Selection probability, %: Current: 48.9 No MAPE: 34.9

Figure 17. The recommender with no Flickr response to a query with story title (the title is too specific so Flickr can't find matching photos)

The self-adaptive layer also ensures that the most picked algorithm gets the highest position in the recommender. For example, starting from default conditions (Figure 15.), when the user clicks on the fourth photo then the rating of the associated algorithm is increased from 5 to 7 and the picture will be moved to the first position (Figure 18.). The rating and selection probability increases respectively.



Lat 48.86118 Lon 2.338918

Story Title: My Louvre story

Tags: painting masterpiece

Rating — Current: 29 No MAPE: 25

Selection probability, % — Current: 61.8 No MAPE: 58.2

Figure 18. Recommendations change order based on user clicks

The scenario shows how self-adaptation improves the quality of the recommender for the user. Self-adaptation ensures that recommendations with higher rating and higher selection probability are available. In addition, it adapts the position of recommendations based on the user preferences over time.

We measured performance overhead for three types of adaptations: gpsUpdated! (changes GPS status), textUpdated! (changes of the story title) and photoAdded! (select recommended photos). The measured values (Table IV) show the minimum, average, and maximum time between initiating the adaptation and the point when algorithms are sorted (excluding loading pictures from Flickr) for 20 adaptations of each type. The initial test setting was: story title = My Louvre story, tags = painting masterpiece, latitude = 48.86, longitude=2.32, GPS ON, recommendations loaded once. Hardware: AMD Turion Dual-Core Mobile 2GHz, 2GB DDR2.

TABLE IV. PERFORMANCE OVERHEAD OF ADAPTATION

Adaptation Type	Overhead (ms) [min; average; max]
gpsUpdated!	[19.5; 30.8; 53.6]
textUpdated!	[19.2; 30.7; 42.2]
photoAdded!	[11.3; 29.3; 44.5]

The overall average overhead of 30.2 ms to realize adaptations is negligible for the mDS-SR application. However, for time-critical applications with possibly more complex adaptation scenarios the overhead may have substantial impact.

VIII. CONCLUSIONS AND FUTURE WORK

This paper contributes with a concrete application in which we have used ActivFORMS to realize self-adaptation. A self-adaptive layer was designed using a formal model of a MAPE-K feedback loop that was added to the social recommender of the digital storytelling application to improve its flexibility. We translated the adaptation requirements to formal properties that we verified before deployment. Having formal guarantees about the requirements increases the potential of the app on the market. We compared the quality of the social recommender with and without adaptation, which shows that self-adaptation improves the quality of the social recommender. As no coding was required, the application could be developed very effectively; it took the developer two month to design the self-adaptive system, provide assurances, and run it.

We also experienced some issues with ActivFORMS that are important for our future research. As exhaustive verification at design time is difficult or even impossible, some guarantees can only be obtained during runtime. In the mDS-SR application, the behavior of the user and availability of resources in the environment are difficult to predict. Consequently, the guarantees provided by offline verification are only valid for restricted models. We plan to extend ActivFORMS with enhanced support for verification at runtime. Furthermore, ActivFORMS requires expert knowledge to design and change the formal models. We are studying how to add a user-friendly modeling layer on top of the formal models that hides (most of the) the underlying formalism from the designers.

REFERENCES

[1] L. Chen, P. Pearl, “User evaluation framework of recommender systems,” 5th ACM conference on Recommender systems, 2011

[2] P. Cremonesi, et al., Hybrid algorithms for recommending new items, Information Heterogeneity and Fusion in Recommender Systems, 2011

[3] https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/WhatsNewXcode/Articles/xcode_4_2.html.

[4] I. Guy, D. Carmel, “Social recommender systems tutorial,” IMB Research-Haifa, Israel, 2011.

[5] R. de Lemos, D. Garlan, H. Giese, “Software engineering for self-adaptive systems: assurances,” Dagstuhl Seminar 13511, 2013.

[6] J. Kephart and D. Chess, “The vision of autonomic computing,” IEEE Computer Society, vol. 36, no. 1, 2003.

[7] U. Iftikhar, D. Weyns, “ActivFORMS: Active Formal Models for Self-adaptation,” SEAMS, 2014.

[8] G. Behrmann, A. David, P. Pettersson, W. Yi, and M. Hendriks, “UPPAAL 4.0,” in Quantitative Evaluation of Systems, 2006.

[9] R. de Lemos et al., “Software engineering for self-adaptive systems: a second research roadmap,” in Software Engineering for Self-Adaptive Systems II, LNCS 7475, Springer, 2012.

[10] B. Cheng et al., “Software engineering for self-adaptive systems: a research roadmap,” in Software Engineering for Self-Adaptive Systems, LNCS 5525, Springer, 2009.

[11] J. Kramer & J. Magee, “Self-managed systems: an architectural challenge,” Future of Software Engineering, 2007.

[12] D. Garlan et al. “Rainbow: architecture-based self-adaptation with reusable infrastructure,” IEEE Computer 37(10), 2004.

[13] J. Zhang & B. Cheng, “Model-based development of dynamically adaptive software,” ICSE, 2006.

[14] MAPE-K Formal Templates http://homepage.lnu.se/staff/digmsi/MFT

[15] D. Weyns, S. Malek, J. Andersson, FORMS: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems, ACM TAAS, 7(1), 2012

[16] P. Oreizy, N. Medvidovic, R. Taylor, Architecture-Based Runtime Software Evolution, ICSE 1998

[17] S. Dobson et al., F, A survey of autonomic communications, ACM TAAS, 1(2) pp. 223-259, 2006

[18] R. Calinescu et al., Dynamic qos management and optimization in service-based systems. IEEE TSE, 37(3):387–409, May 2011

[19] J. Tretmans. Formal methods and testing. chapter Model based testing with labelled transition systems, Springer-Verlag, 2008

[20] D. Weyns et al., A Survey on Formal Methods in Self-Adaptive Systems, Formal Methods for Self-Adaptive Systems, 2012

[21] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, “Model evolution by run-time parameter adaptation,” in ICSE, 2009.

[22] C. Ghezzi et. al., “Managing non-functional uncertainty via model-driven adaptivity,” in ICSE, 2013.