Computer Science and Information Systems 17(1):1–27 https://doi.org/10.2298/CSIS180829021T

Run-time Interpretation of Information System Application Models in Mobile Cloud Environments

Nikola Tanković[1] and Tihana Galinac Grbac[2]

[1] Faculty of Informatics, Juraj Dobrila University of Pula
nikola.tankovic@unipu.hr
[2] Faculty of Engineering, Juraj Dobrila University of Pula
tihana.galinac@unipu.hr

Abstract. Application models are commonly used in the development of information systems. Recent trends have introduced techniques by which models can be directly transformed into execution code and thus become a single source for application design. Inherently, it has been challenging for software developers to become proficient in designing entire systems due to the complex chain of model transformations and the further refinements required to the code generated from the models. We propose an architectural framework for building the distributed information system applications in which the application models are directly interpreted during execution. This approach shortens the evaluation cycles and provides faster feedback to developers. Our framework is based on a holistic application model represented as a graph structure complemented with a procedural action scripting language that can express more complex software behavior.
We present the implementation details of this framework architecture in a mobile cloud environment and evaluate its benefits in eleven projects for different customers in the retail, supply-chain management and merchandising domain involving 300 active application users. Our approach allowed engaging end-users in the software development process in the phase of specifying executable application models. It succeeded in shortening the requirements engineering process and automating the configuration and deployment process. Moreover, it benefited from the automatic synchronization of application updates for all active versions at the customer sites.

Keywords: model-driven development, MDD, cloud computing, information system, model interpretation, application graph model

1. Introduction

The information systems (IS) development process includes numerous repeating patterns such as constructing database schema, designing user interfaces for data display and manipulation, building communication services [1], etc. These and similar patterns arise during the process of human or semi-automatic translation of the system model artifacts to machine-executable code. Application modeling has become extremely relevant—not just for supporting analysis and design phases—but in serving as the primary source for automatic application generation [2]. Therefore, Model-Driven Development (MDD) advocates the automation of repetitive software development tasks.

In addition, such automation opens new opportunities for end-users to actively participate in the software development process, offloading the development tasks that were exclusive to software engineers [3]. Figure 1 depicts a generic scenario of how this is applied. The software developers can steer their focus on providing generic components and/or model transformation procedures that are actively reused through different application models defined by the end-user developers.

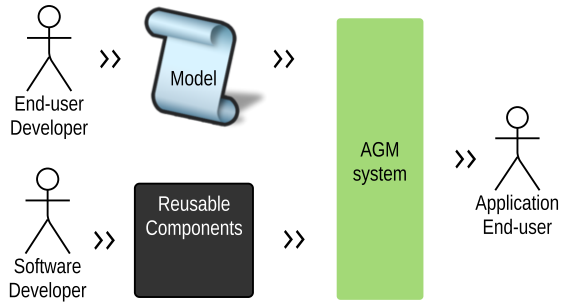


Fig. 1. Offloading the software developers by introducing modeling techniques understandable to end-users

Additionally, the cloud computing paradigm facilitates software provisioning by enabling software providers with large-scale infrastructure resources paid on a per usage basis [4]. The cloud services benefit from offloading software and system engineers due to the high level of automation.

This paper proposes an approach to alleviate the role of end-users in the development process of information systems. This goal is achieved by interpreting application models at runtime on a highly automated distributed cloud environment. We strive to gain benefits from: (1) enabling adaptive and rapid-feedback modeling in which changes can be made quickly and applied easily, (2) enabling easier software maintenance and distribution, and (3) runtime application version management with quick transition times between different end-application versions. Our approach is targeted primarily at small-to-medium-sized projects that require fast development cycles and a greater degree of exploration during the requirements-gathering phase.

To achieve model interpretation, we devised a technique that represents models at runtime as *directed graphs*, called the Application Graph Model (AGM). To supplement AGM for building more complex solutions, we provide a complementary action scripting language targeted to advanced users with a software programming skillset. This paper describes the AGM execution architecture and evaluates it with a concrete implementation for building mobile-enabled information systems [5] in the retail, supply chain management, and the merchandising domain.

The rest of this paper is organized as follows: Section 2 provides the rationale behind the model interpretation. Section 3 lays a conceptual foundation for interpreting models in the information system application domain. The implementation details are presented in Section 4 together with a discussion concerning our experiences of AGM usage in practice and an example application model. The limitations of our approach are disseminated in

Section 5. Related work is reviewed in Section 6. Finally, Section 7 concludes the paper and elaborates on the possibilities for further research.

2. Background

Two major strategies exist for transforming models to executable applications [6]: (1) the generative approach, in which models undergo a series of transformations that result in executable application code, and (2) the interpretative approach, in which models are exploited through runtime interpretation. Generative solutions yield better end applications performance-wise because runtime interpretation of models comes with additional execution cost. The existing strategies, such as the Model-Driven Architecture initiative (MDA) and the Eclipse Modeling Framework (EMF), focus primarily on the generative approach. Though application performance is superior, the generative approach requires the definition of a series of model transformation steps using different templates for compiling the higher-level model to lower-level programming code. Specifying such transformations can be extremely challenging [7]. Consequently, it involves a broader spectrum of highly specialized engineers. While such involvement is essential for large-scale software products, at the same time, such characteristics hinder large-scale MDD adoption.

Another ongoing research challenge associated with the generative approaches is the manual source-code refinement typically applied after the initial *model to code* transformations. While the code generated from models covers the majority of generic application functionalities, some parts of applications still require the implementation of specific business logic. These highly specific portions of applications are difficult to represent using high-level abstract models [8]. Therefore, specific functionality is often implemented after the initial code has been generated from the models. This post-model-generation code refinement requirement has several inherent drawbacks: (1) it impacts the synchronization between the model and the application source code, (2) it requires specialized software programming skills, and (3) it imposes a burden on model and application change and deployment management, because it requires additional housekeeping for each version of the model, transformation rules, code templates and customized code to maintain their compatibility [8]. To address these drawbacks, some solutions have proposed modeling specific functionalities by providing an abstract action domain-specific language [9,10] that is also transformed to concrete programming code through compilation. This approach works in keeping the models synchronized but raises other difficulties. For example, specifying model transformation rules requires expert knowledge and has a massive impact on the customer-engineer requirements negotiation process. Rapid prototyping becomes almost impossible: the time required for generating code, compiling, installing and restarting existing systems can range from several minutes to several hours [11].

3. AGM Solution

As we briefly discussed in a previous research paper [12], AGM reuses the *Object* and *Process* modeling approach [13] standardized in Automation systems and integration—the Object-Process Methodology, ISO/PAS 19450:2015. Object Process Methodology (OPM) is a holistic graphical modeling language applicable to a large variety of domains.

The main advantage of OPM is its ability to capture the dynamic and static aspects of a system within a single model. The OPM approach handles model complexity through refinement techniques [14] and alternative model views rather than splitting the modeling process using several semantically orthogonal modeling languages. The underlying model in OPM is holistic: it captures the complete end-solution within a single model.

Our solution builds upon the OPM research in several ways:

– We emphasize the holistic model idea derived from a reflexive meta-model; that is, a model that can describe itself. The *Meta-Object Facility (MOF)* [15] from the *Model-Driven Architecture (MDA)* initiative is also reflexive, but it is used to derive a set of semantically and not necessarily compatible models.
– We built the AGM meta-model in a manner similar to the OPM meta-model, but it possesses specialized constructs for defining mobile information system applications.

In the following subsections, we present the framework for realizing the runtime model interpretation, followed by an explanation of the AGM meta-model and, finally, the details of how model interpretation is achieved.

3.1. AGM Framework Architecture

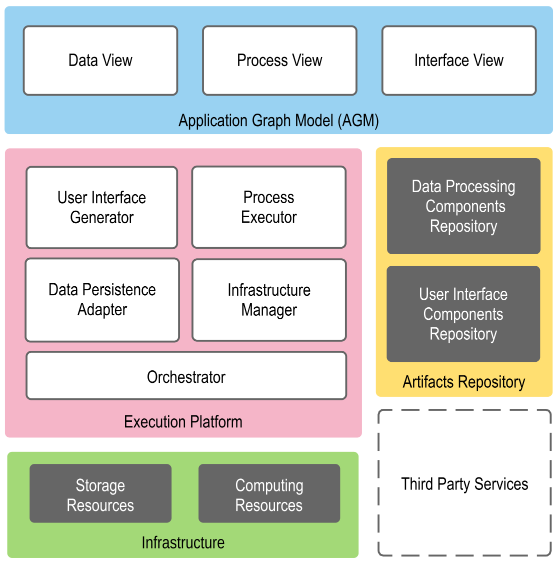


Fig. 2. AGM Framework Architecture

The AGM framework architecture for runtime interpretation is presented in Figure 2. The framework consists of an *application graph model (AGM)* interpreted by the *execution platform* that references components available at *artifacts repository*.

The *Application graph model (AGM)* provides a set of notations by which an end-user developer can specify the application models. An information system application model is constructed by using the *Data view*, *Process view* and *Interface view* notations in a holistic model. A *Data view* involves defining runtime invariant data structures, while the *Process view* and *Interface view* are assembled using constructs that reference artifacts stored in an artifact repository. These models respectively represent the graphical user interface (GUI) and the data-flow of the information system.

The *Execution platform* interprets the AGM model. It consists of a *User Interface Generator*, a *Process Executor*, a *Data Persistence Adaptor*, an *Infrastructure Manager* and an *Orchestrator*. The *User Interface Generator* interprets the part of the AGM model that defines GUI elements within the *Interface* view. The *Process Executor* interprets the information system application model and orchestrates the data-flow components available in the *artifacts repository*. The data-flow components are connected with the GUI components to represent information system data. The *Data Persistence Adapter* provides permanent data storage functionality based on structures defined in the information system application within a *Data view*. The *Infrastructure Manager* provisions the infrastructure resources required for the interpretation processes and data processing components. It ensures that non-functional requirements such as application performance are met by providing sufficient resources. Finally, the *Orchestrator* binds all the above components and switches control according to the current system state.

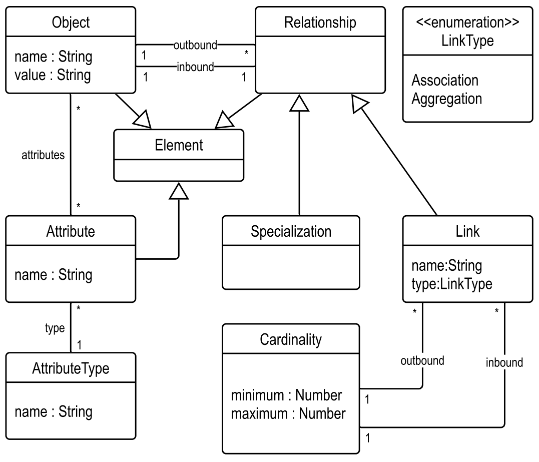
The *Artifacts repository* contains a set of reusable artifacts (generic parts of an information system application). These components are divided into *user interface*-related and *data processing*-related functionality. The *User interface* components represent application-specific information through a set of user interface (UI) elements that enable interaction with end-application users. Note that there could be several different types of UIs depending on the client platform (e.g. smartphone, tablet, or personal computer), and these components can be specialized for specific client platforms. *Data processing* components are used for data management. They provide real-time integration with external systems for persistent data storage.

The *Infrastructure* resources involved in provisioning information system applications can be divided into *computing* resources and *storage* resources. The *computing* resources provision the CPU-intensive parts of information systems such as data processing, while the *storage* resources are specialized for data management.

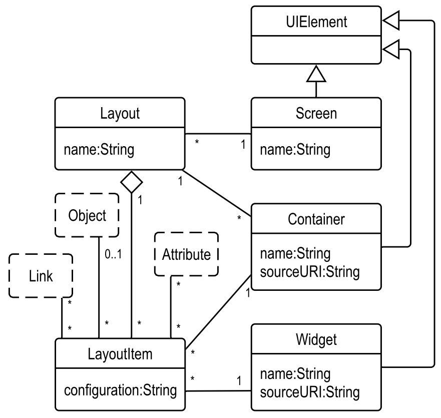
3.2. AGM meta-model

Figure 3 displays an AGM meta-model we developed for modeling information system applications. The AGM meta-model is extensible because of its reflexive nature—it completely defines itself [12,16]. This characteristic enables extending AGM capabilities beyond the three views described in the following subsections. For example, additional views could be defined for other types of interfaces (e.g., special resources and domain-specific devices). The meta-model is defined using UML class model semantics [17].

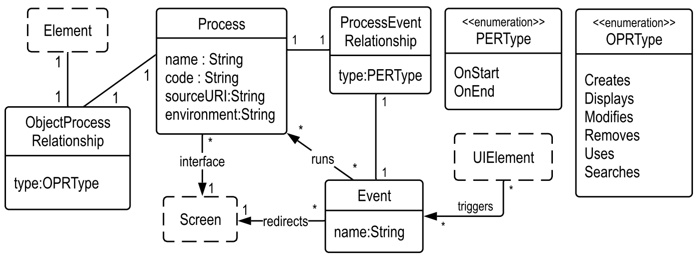
Data view A *Data view* defines the data structures from which different static artifacts can be derived (e.g. the information system database schema, web service interfaces, ...). It is derived both from the *Structure - Classes* specification contained within UML Superstructure model [17] and the OPM meta-model [13].



(a) Data view meta-model



(b) User interface view meta-model



(c) Process view meta-model

Fig. 3. AGM Meta-model

Each entity represented within the data structure is represented by an *Object* construct. Each *Object* can have multiple *Attributes* and *Relationships* to other objects. A *Relationship* can be a *Specialization*, representing attribute and link inheritance (as in object-oriented programming) or a *Link* representing connections between objects. The *Cardinality* quantifies the minimal and maximal number of *Objects* connected by *Links*. Unbounded cardinality is achieved by omitting a *maximum* quantity in the *Cardinality* object.

User interface view A *User interface view* defines a GUI used to manage the information and conduct business processes defined in a *Process view*. User interface generation combines information stored in the AGM with the current application context (e.g. currently running processes) and composes the graphical user interface using a reusable set of components, referred to as *Widgets* [18]. *Widgets* can be connected either to whole *Object*s or to their *Attribute*s and *Link*s through *LayoutItem*s which serve to select and order the displayed data. For example, *table-like* widgets can display entire *Object* instances, whereas *text-input* widgets display only certain *Attribute*s. A set of *LayoutItem* objects comprises a *Layout* that can be placed directly into an application *Screen* or into a *Container*. A *Container* is a user interface component used to organize information on *Screen*s, e.g. a *tabular* form with many *tabs*. The location of the source code of the *Widget* implementation is contained within *sourceURI* attribute.

Process view A *Process view* represents the modeling concepts used to implement dynamic application aspects. A *Process* either models a user activity within the application (which usually corresponds to real-life business process - or activity we wish to electronically document) or a background task that does not require user interaction (e.g. calculations and connection to remote services). A process can also run additional processes, all of which are contained in the application context within a *process stack*.

Every instantiated *Process* node is linked with an *Element* (*Object*, *Attribute* or *Link*) on which it operates through an *ObjectProcessRelationship*. The nature of this relationship is denoted by an *OPRType* enumeration: *Process*es can create, display, use, modify, remove and search for *Elements*. *Process*es can also emit *Event*s, which are represented by *ProcessEventRelationships* that can be a *PERType* (triggered on process start or end). User interface components or *Widget*s can also trigger *Events*, usually as a result of user interactions. Throughout the user interfaces, processes that involve end-application users are represented by the *Screen* constructs from an *Interface view*. These processes collect and display information to/from end-users through a series of user interface forms. In contrast, processes that are not linked with *Screen*s are considered as background processes. *Process* nodes can either directly represent generic components through their *sourceURI* attribute or contain action scripts used for expressing additional functionalities stored in a *Code* attribute. Currently, our proposal assumes that action scripts are written in an internal domain-specific language (DSL) [19], namely JS-DSL, that runs on top of existing procedural code that is also interpreted at runtime.

For the purposes of our evaluation case-study, the mobile cloud IS, or *Processes*, have an additional attribute intended to specify the *environment* in which they should be executed, which can be local (on the client side) or remote (in the cloud). Remote process execution is vital for maintaining a consistent and secure system since the scripts at the front-end are exposed and easy to manipulate.

3.3. AGM representation

To describe the methods and algorithms for interpreting the AGM, we first define it using a *directed property graph*. AGM thus becomes an ordered quadruple consisting of edges, vertices and mapping functions denoting their type (class from meta-model):

$$G_{AGM} = (V, E, T_E, T_V)$$

where V is a set of vertices $V = \{x_1, x_2, x_3, \ldots, x_n\}, x_i \in \mathcal{D}$, and $\mathcal{D}$ represents a set of concepts from the modeled domain (e.g. *customer*, *product*, ...); whereas $E \subseteq X \times X$ represents a set of ordered vertex pairs $E = \{(x_i, x_j) | i \neq j, x_i, x_j \in E\}$ that represent the edges (links) from x_i to x_j. Mapping functions are defined as $T_E : E \rightarrow \mathcal{M}_E$, which represents the node types in the AGM meta-model, and $T_V : V \rightarrow \mathcal{M}_V$, denoting the vertex types. According to the AGM meta-model defined in the previous section,

$$\mathcal{M}_E = \{Object, Process, Event, Layout, Screen \ldots\} \tag{1}$$

$$\mathcal{M}_V = \{Association, Aggregation, Inheritance, Uses, Modifies, \ldots\} \tag{2}$$

$Object$ nodes are the primary building blocks for the structural aspects of a system (*Data view*). Node types such as $Process$ and $Event$ represent dynamic (*Process view*) behaviors and node types such as $Layout$ and $Screen$ represent *User interface views*. AGM provides full modeling capability for the structural aspects of a system, including schemes for data storage and user interface definitions. The structural model is complete, meaning that the interpreter can execute them without requiring additional programming code to render database schema [20] or augment the user interfaces. For dynamic system definition, the AGM is supplemented by action scripts contained in $Process$ nodes, meaning that an additional mapping exists from each $Process$ node to executable artifacts or source code, if required.

The biggest advantage of representing models as holistic graphs is that we can use all the well-known concepts from graph theory, including graph traversal, graph matching, and querying for subgraphs. Our graph model interpreters use such capabilities to efficiently and reliably execute end applications.

We will demonstrate runtime traversal in a *Data view*. The Algorithm 1 is used to build a subgraph with the structural model defined for the requested *Object*. The input to the algorithm is the node that represents a certain *Object*, for which a complete list of attributes and links is extracted by traversing the AGM graph. The resulting subgraph can then be used to enable *Object* serialization (e.g., XML or JSON), or to generate SQL-language data manipulation queries to communicate with underlying databases.

```
Data: $G_{AGM} = (X, V)$, and starting node $x_s$ where $T_E\,(x_s) \in \{Object\}$
Result: $G'_{AGM} = (X', V')$ where $X' \subseteq X$ and $V' \subseteq V$ represent the complete structural
        model of concept $x_s$
begin
    $X' \leftarrow \{x_s\}, V' \leftarrow \emptyset$
    for $x \in X'$ do
        $S_{successors} \leftarrow \{x_j | x_j \in \Gamma^+(x), T_V(x, x_j) = Specialization\}$
        $X' \leftarrow X' \cup S_{successors}$
        for $s \in S_{successors}$ do
            $V' \leftarrow V' \cup (x, s)$
        end
    end
    for $x \in X'$ do
        $S_{successors} \leftarrow \{x_j | x_j \in \Gamma^+(x), T_V(x, x_j) \in$
          $\{Association, Aggregation, Attribute\}\}$
        for $s \in S_{successors}$ do
            $V' \leftarrow V' \cup (x, s)$
        end
    end
end
```

Algorithm 1: Traversal of subgraph containing the structural definition of a modeled concept

A similar concept is applied to extract interface compositions and their relationships from $Screen$, $Layout$ and $Widget$ nodes, and to conduct processes defined by $Process$ and $Event$ nodes.

3.4. Interpretation principles

Model interpretation consists of three interpreters, a server-side interpreter (SSI), a client-side process interpreter (CSPI), and a client-side user interface interpreter (CSUII) as depicted in Figure 4. The SSI serves as the *Process Executor* and is responsible for executing defined processes on the server side and for creating web service endpoints for bindings with client-side interpreters. On the client side, the CSPI serves both as a *Process Executor* and as an orchestrator for conducting processes locally in the client environment. The CSUII is a *User Interface Generator* used to render user interface (UI) elements. AGM models, when interpreted, are stored as graphs (see Section 3.3) in a *Graph DB*. To minimize the communication between client and server, and to enable the application to work in situations when the client is disconnected from the network, we also implemented a local database (*Local DB*) that contains serialized portions of the AGM model and application data.

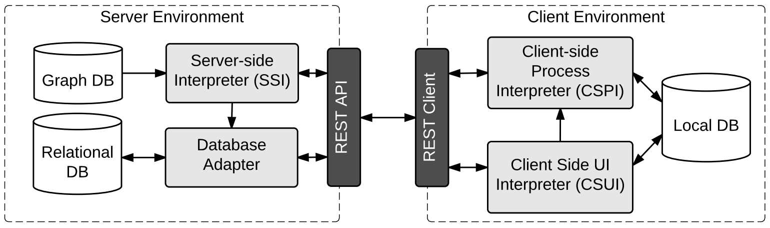

Server Environment
Graph DB
Server-side Interpreter (SSI)
Relational DB
Database Adapter
REST API
REST Client
Client Environment
Client-side Process Interpreter (CSPI)
Local DB
Client Side UI Interpreter (CSUI)

Fig. 4. A high-level view of the server-client architecture for interpreting the AGM

The server side includes a *Database Adapter* module to enable access to relational databases for manipulating data in persistent storage according to a defined *Data view* structure in the AGM. This module implements a *Data Persistence Adapter* from the framework presented in Figure 2. Because our approach supports runtime changes, holding relational schema solely within the *Relational DB* hinders instant adaptations to new models. Instead, every model change in a *Data view* yields incremental relational schema updates. The *Database Adapter* analyses model changes and issues updates to relational schema. However, because schema updates are a sensitive process that may result in data losses, it is possible to turn off automatic schema changes and rely on a semi-automatic approach after the initial application release.

Runtime model interpretation makes it possible to link stored data with a specific AGM model version. Multiple AGM model versions can be stored in *Graph DB* due to the continuous evolution of the modeled system. As an information system evolves from version to version, data structure mismatches may be introduced between the older and newer model versions.

The interpretation process transforms an AGM model into an information systems application at runtime. The resulting information system application may include a number of UI forms (we will use the term *application screens*), where users process presented information and decide on their next action. We refer to presentation and interaction between application users and *application screens* as *interpretation cycles*.

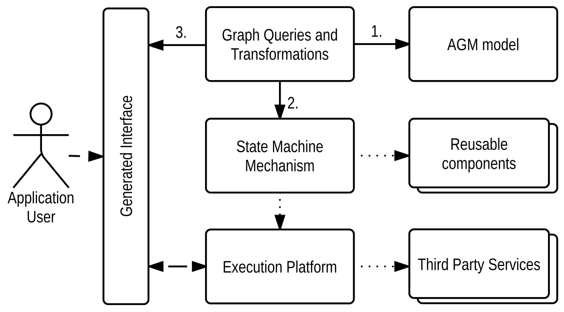


Fig. 5. AGM interpretation cycle

Figure 5 shows the steps that the AGM interpreter must take to execute each interpretation cycle. Each cycle begins by querying the AGM model for a definition of the UI and a set of possible user actions (Step 1). This definition is used to construct a state machine (SM) used to implement communication with the user (Step 2). Finally, the UI is generated for the application user (Step 3). Figure 6 shows a simplified version of the states and transitions for an SM. The SM is initialized to a start state that triggers UI rendering (*application screen*); then, it enters a *wait-for-user* state. For each $Process$ node and connected $Event$, a transition and a new state is added to SM and can be executed.

The algorithm 2 shows how a state machine is generated for each interpretation cycle. To provide user interaction for each *application screen*, the CSUII traverses AGM model from the *Screen* nodes in a *User interface view* to all connected *Process* and *Event* nodes that serve as input for defining state machine states and transitions. Three default states are always present: (1) an s_{RI} state in which the user interface is rendered, (2) a s_{WU} state that represents user *think time*, and (3) s_{NC}, which is a final state that occurs when the CSPI makes a switch to the next interpretation cycle. Additional states are defined for each process obtained while traversing the AGM. Transitions correspond to incident $Event$ nodes that trigger those processes.

The UI is constructed according to a *User interface view* in the AGM. It consists of multiple *Screen* nodes that define the appearance of each end-application UI component. Each *Layout* specifies a mapping between object attributes and widgets. It is important for widgets to be developed using *generic programming* approaches, allowing them to serve as templates that display different information based on linked $Attribute$s from $Object$s. We provide different *Widget* types according to the cardinality between an $Object$ and its $Attribute$ or $Link$.

```
Data: $s_{RI}$ - render interface state, $s_{WU}$ - wait for user state, $s_{NC}$ - next cycle state, and
      $P_e = \{(p, e)\}$ - set of process-event pairs from AGM for current cycle,
      $p \in P, e \in E$
Result: generated state machine $(\Sigma, S, s_0, \delta, F)$ where $\Sigma$ is set of transition events, $S$ is
        set of possible states, $s_0$ is a start state, and $\delta$ is a set of transitions $\delta : S \times \Sigma \rightarrow S$
begin
    $s_0 \leftarrow s_{RI}, F \leftarrow \{s_{NC}\}\ S \leftarrow \{s_{RI}, s_{NC}, s_{WU}\}, \Sigma \leftarrow \emptyset\ \delta \leftarrow \emptyset$
    for $(p, e) \in P_e$ do
        $S \leftarrow S \cup \{s_p\}$
        $\Sigma \leftarrow \Sigma \cup \{v_e\}$
        $\Sigma \leftarrow \Sigma \cup \{v_{finish}\}$
        if isInterfaceTriggered(e) then
            $\delta \leftarrow \delta \cup \{\delta(s_{WU}, v_e) \rightarrow s_p\}$
        end
        else if isDataTriggered(e) then
            $\delta \leftarrow \delta \cup \{\delta(s_{RI}, v_e) \rightarrow s_p\}$
        end
        if redirects(p) then
            $\delta \leftarrow \delta \cup \{\delta(s_p, v_{finish}) \rightarrow s_{NC}\}$
        end
        else if updatesInterface(p) then
            $\delta \leftarrow \delta \cup \{\delta(s_p, v_{finish}) \rightarrow s_{RI}\}$
        end
        else
            $\delta \leftarrow \delta \cup \{\delta(s_p, v_{finish}) \rightarrow s_{UW}\}$
        end
    end
end
```

Algorithm 2: Constructing a cycle state machine from a set of processes and events linked to each application screen

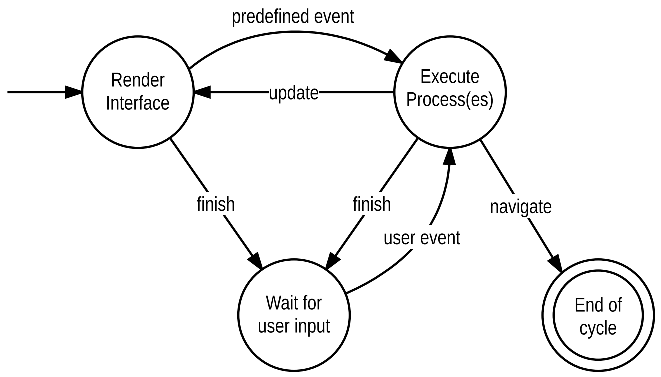


Fig. 6. A state machine is constructed for each *interpretation cycle*

4. Implementation and Evaluation

In this section, we describe the implementation of the presented framework for executing business applications in a mobile cloud context. IS applications contain numerous repeated patterns, which we have identified over the past ten years of professional software development[3] in the retail, supply-chain and merchandising domains. To automate the software development processes, we abstracted these patterns and now provide them as *generic components* through the AGM. Figure 7 shows an overview of our implementation. The *AGM interpreter* is divided into a *Mobile interpreter* and a *Cloud server interpreter*. The *Cloud server interpreter* additionally manages infrastructure resources using an *Infrastructure API* provided by the cloud provider. The main challenge in this type of implementation was the distributed nature of mobile cloud applications because it requires keeping the AGM models and application data synchronized in a distributed mobile execution environment.

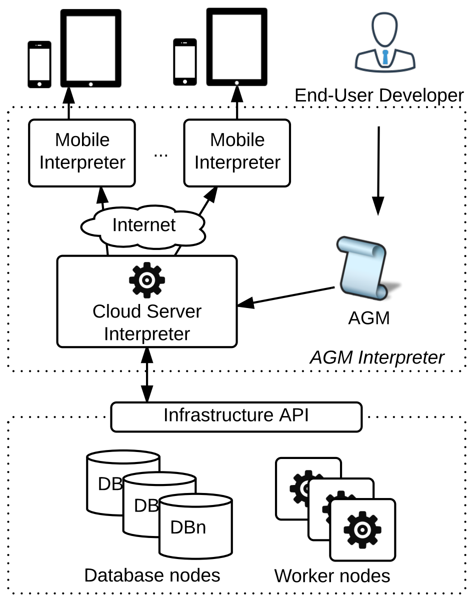


Fig. 7. AGM concept in the mobile cloud computing environment

Because AGM is a directed graph structure, we sought a semantically similar solution to manage it efficiently and reliably. We decided to use a graph database called Neo4j [21], which supports efficient queries because of its specialized graph-structure storage scheme. Neo4j provides a *property graph* model for storing data. This model allows each node and vertex to be associated with its own *key-value* data-store that can hold additional information. This capability was used to represent the mappings T_E and T_V from G_{AGM}

[3] One of the authors was associated with a Croatian software company *Superius d.o.o.*, dedicated to building mobile cloud information systems

as well as other additional values associated with certain model nodes (e.g., *Object* nodes contain names and values, and *Process* nodes can contain action script source code (see Figure 3).

Currently, mobile interpreters have been implemented for Android and iOS platforms using a hybrid development approach that combines native platforms with web application components [22]. The server-side interpreter is built as a Java application running on (but not limited to) *Apache Tomcat* application servers.

4.1. Data persistence

To manage stored data, we implemented $DatabaseAdapters$ for three relational databases: PostgreSQL, Oracle, and MSSQL. A generative tool is used to construct database schema from AGM models, allowing us to keep the database schema synchronized with the AGM models.

In the current implementation, the scalability and elasticity limitations of the relational databases are a drawback, since these relational databases do not typically provide elastic capabilities [23]. Our execution engine does not currently control database elasticity, hence the database components need to be provisioned according to planned workloads. Obviously, this is not something that end-user developer can achieve, and thus this steps requires specialized infrastructure personnel for on-premises usage of AGM. On the other hand, if one wishes to use public cloud providers, one can use a managed relational database such as Amazon RDS [4] or DigitalOcean PostgreSQL [5]. Achieving cloud-native automatically elastic persistence for AGM is a great future challenge, where we could also explore NoSQL solutions like document databases (e.g. MongoDB [6]).

4.2. Action Scripting Language Implementation

Process nodes from AGM are enriched with an action scripting language called *JS-DSL*. JS-DSL is an internal domain-specific language (DSL) developed on top of *JavaScript*. Figure 9 represents a block of JS-DSL code from one of our applications that sums up the total charges in an invoicing process. JS-DSL provides special constructs for accessing and manipulating user-level data. Figure 8 displays how JS-DSL can be used to customize user interfaces.

To create JS-DSL we followed the guidelines for creating internal DSLs proposed in [24], [25] and [19]. JS-DSL currently provides the following capabilities:

a) It can make runtime changes to UI widgets (e.g., emphasizing certain information using color, controlling widget behavior, and navigating through the application).
b) It can access data stored in background services (e.g., data persisted on a smartphone client or from remote servers),
c) It can apply a set of simple mathematical operators to data (e.g. sum, average, min, and max).
d) It can invoke remote third-party services.

[4] Amazon RDS, available at `https://aws.amazon.com/rds/`
[5] DigitalOcean cloud provider, available at `https://www.digitalocean.com/`
[6] MongoDB, available at `https://www.mongodb.com/`

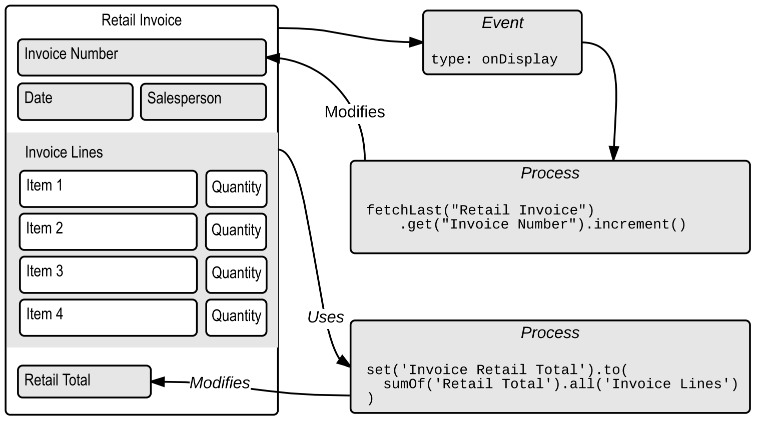


Fig. 8. Example of *Process* nodes combined with JS-DSL in specifying custom application behavior

```
set('Invoice Retail Total').to(
  sumOf('Retail Total').all('Invoice Line')
)
set('Lines Count').to(
  all('Invoice Line').count()
)
```

Fig. 9. A sample of JS-DSL source code for a $Process$ node.

When desired actions are not available within JS-DSL, designers can rely on classical JavaScript code, which exploits the benefits of adopting a DSL embedded in the underlying JavaScript language. Note that such extensions require a more advanced user skill set.

4.3. Defining AGM models

To construct application models and load them to an execution platform we provide an additional DSL, called AGM-DSL. AGM-DSL is a one-to-one textual representation of an AGM model. Each model view has associated AGM-DSL commands (e.g., a *Data view* is associated with a *DEF* command). A BNF specification for a *DEF* command is presented in Figure 10, and Figure 11 shows an example *DEF* command used in an application from the retail domain.

4.4. Reusable components

Client-side user interface interpreter (CSUII) interprets the AGM model and composes the user-interface using the reusable generic components - widgets. Widgets are designed to be the gatekeepers of complexity towards the end-user developers. They are engineered

$$\begin{aligned}
\langle\text{define statement}\rangle &\models \texttt{DEF}\ \textit{object}\ \langle\text{inheritance}\rangle\ \langle\text{newline}\rangle\ \langle\text{attribute list}\rangle \\
\langle\text{attribute list}\rangle &\models \langle\text{attribute}\rangle\ \langle\text{newline}\rangle\ \langle\text{attribute list}\rangle\ \mid\ \langle\text{attribute}\rangle \\
\langle\text{Inheritance}\rangle &\models\ :\ \textit{inheritedObject} \\
\langle\text{attribute}\rangle &\models \langle\text{tab}\rangle\textit{attribute name}\ :\ \textit{attribute type}\ \langle\text{card}\rangle\ \langle\text{nl}\rangle \\
\langle\text{card}\rangle &\models \langle\text{quantity}\rangle..\langle\text{quantity}\rangle\ \mid\ \langle\text{quantity}\rangle \\
\langle\text{quantity}\rangle &\models \textit{numeric value}\ \mid\ \star \\
\langle\text{nl}\rangle &\models \backslash n \\
\langle\text{tab}\rangle &\models \backslash t
\end{aligned}$$

Fig. 10. BNF specification for DEF command in AGM DSL

```
DEF Product :Resource
   Name: Name 1..1
   Unit Of Measure: Unit Of Measure 0..1
   Wholesale Price: Number 0..1
   VAT: Number 0..1
   Retail Price: Number 0..1
   Stock: Number 0..1
   Stock Date: Date 0..1
   Code: Number 0..1
   Weight: Number 0..1
   Tax: Number 0..1
   Package Weight: Number 0..1
   Pallet Weight: Number 0..1
   Barcode: Number 0..1
   Package Quantity: Quantity 0..1
```

Fig. 11. Excerpt from the AGM-DSL defining a product in a retail domain

with classical software engineering methods by professional teams to conform to the predefined component specification. AGM achieves extensibility through the development of new widget components paired with their meta-models - connection points to the rest of the AGM model. Some widgets like `InputTextWidget` have a single connection point (e.g. the attribute of the object that needs to be provided by the user), while some widget can have multiple connection points (e.g. `LabelWidget` can represent multiple objects' attributes).

Widget component-model interface is derived from the *port-based interface* component-model [26]. The deviation from port-based interfaces is the introduced connection between components and the meta-data. This enables turning generic components into specific representations based on the context. The widget interface is displayed in Figure 12. Each widget implementation is complemented with an AGM node inherited from a $Widget$ node. The meta-data connection within I_M are referencing the *data-view* elements (Fig. 3a) refined through *LayoutItem* nodes. The data that flows through data-

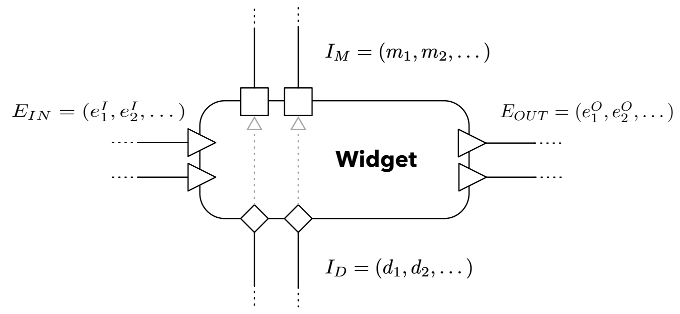

$I_M = (m_1, m_2, \dots)$
$E_{IN} = (e^I_1, e^I_2, \dots)$
$E_{OUT} = (e^O_1, e^O_2, \dots)$
Widget
$I_D = (d_1, d_2, \dots)$

Fig. 12. Interface for UI components - Widgets

interface I_D conforms to *data-view* elements from I_M. The set of input and output events (E_{IN}, E_{OUT}) is also linked to *Event* nodes from the *process-view* model (Fig. 3c).

Currently, widget components are implemented in JavaScript, HTML, and CSS and executed strictly on the client-side, at the user-interface level. Since our mobile application engine is implemented as a hybrid web application using PhoneGap [27], all widgets are both working on Android, as well as iOS smart-phones. By using PhoneGap, we can provide an embedded web browser (e.g. *Android WebView*) as an integral part of the client application engine such that JavaScript interpreter is always available and enabled.

4.5. Performance considerations

Implementing interpreters with solid execution performance is challenging; it took us four developer-years to develop the proposed framework and obtain a satisfying user experience from a performance perspective (e.g. application load-time, GUI rendering-time, and communication-time). We encountered issues with the limited computing power of smartphones, resulting in slow UI rendering and slow execution of defined *Process* nodes. We compared AGM-modeled applications with previous generation non-modeled applications and noticed severe performance degradation. We tracked the main performance bottlenecks in the preparation process for executing the code contained in *Process* nodes and added *caching* mechanisms that stored the state machine specifications for each interpretation cycle as well as generated user interface code. After introducing such mechanisms, the interpreted model exploits performance benefits previously available only to generative approaches; the *cached* data represents *compiled* fragments of AGM models. *Caching* introduced drastic performance improvements to the initial prototype and reduced the performance penalty to only 15–20% compared to classically developed non-modeled applications. Our approach to the AGM model interpretation can be considered as a form of *just-in-time* (JIT) compilation.

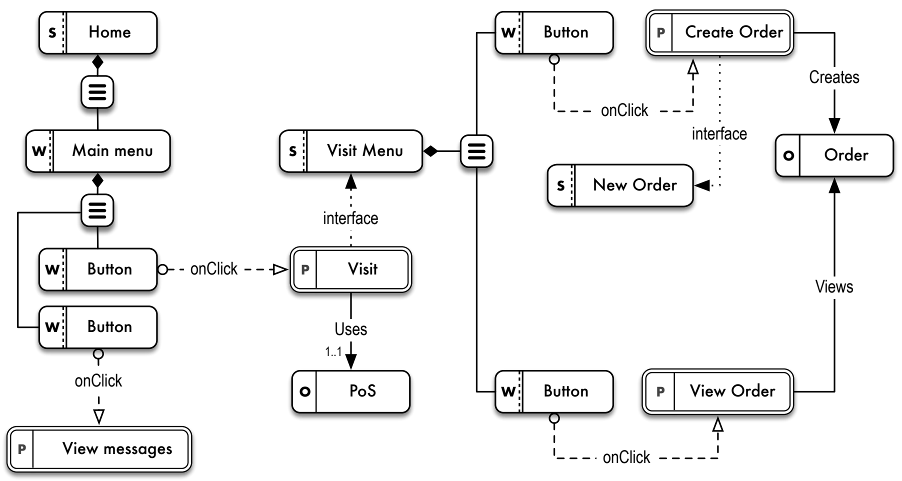
4.6. Reference application

In order to better illustrate the applicability of the AGM framework, we will disseminate an AGM model example and the resulting mobile application derived from it. The application presented is a subset of the typical application in the domain of supply chain management. The application is used by the field operatives from the distribution company in the supply chain process that are visiting end distributors (noted *PoS* - Point of Sales) and collecting orders for products that should be distributed. Orders that are collected are sent through the Internet and stored in the central database where the integration modules are used to transfer those orders into existing systems. The integration components are currently not provided by the AGM system due to the vast differences between different ERP vendors.

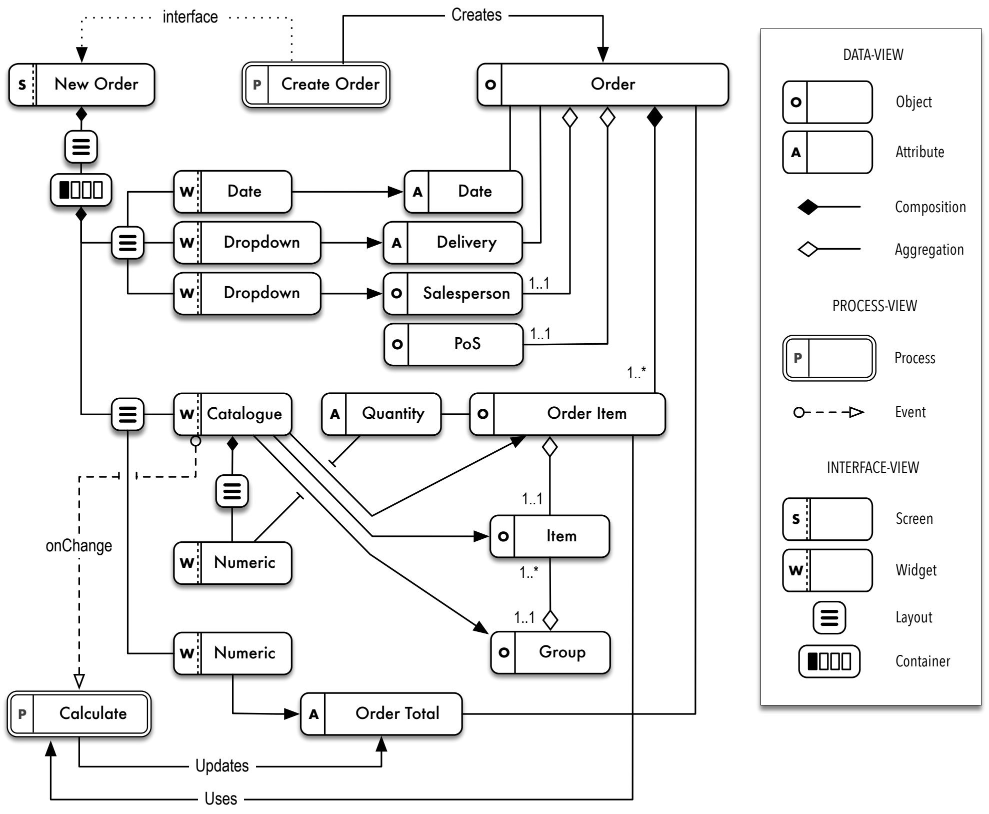
Figure 13a displays the main part of the AGM application model from the *process-view* and *interface-view* perspective. We can observe that the application consists of four main processes: (a) *Visit* process - where the users select a Point of Sales and can create a new Order, (b) *Create Order* process for the actual user-input of a new order, (c) *View Order* process for displaying previous orders, and (d) *Messages* for internal communication between organization members (this process is not further disseminated for brevity). Figure 13b displays the details of the *Create Order* process including the elements of the user interface and their links to the objects and attributes of the order object. We can observe that the *PoS* object required no widget since it is automatically extracted from the context of the *Visit* process which requires *PoS* to be defined.

Figure 14 displays the resulting user interface screens:

(a) a *Home screen* which is rendered from the specification of the *Home* Screen element that contains the *Menu* widget,
(b) *PoS selection screen* which is not explicitly modeled but inferred from the fact that the *Visit* process that is selected requires a *PoS* object to be selected. The user interface for selecting a PoS instance is not specified so it is automatically derived.
(c) *Visit screen* which is specified with *Visit Menu* screen from the model,
(d) and (e) - *Order insertion* screens which are specified with the *Create Order* nodes and their connections with different *user-interface* and *data-view* nodes.

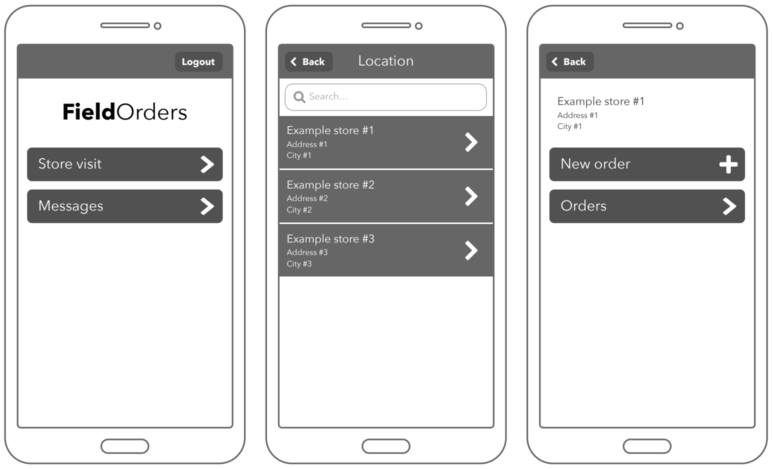


(a) The process and interface for navigating through application

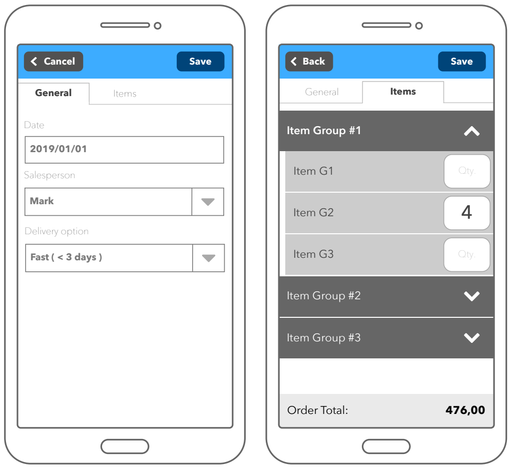


(b) The process for creating order

Fig. 13. The AGM source model for the example application



(a) Application home screen **(b)** Selecting a store to visit **(c)** The menu with activities when a store is selected



Date: 2019/01/01
Salesperson: Mark
Delivery option: Fast (< 3 days)

Item G2: 4

Order Total: 476,00

(d) Inserting new order **(e)** Selecting the items for the new order

Fig. 14. Example mobile cloud application in the supply chain domain.

4.7. Evaluation

We applied the AGM approach in implementing eleven projects for Southern European customers in the retail, supply chain management, and merchandising domains. The projects included building mobile information systems integrated with existing customer information systems. Combined, the projects involved over 300 end-application users with Android-based smartphones. In the retail domain, application functionality included collecting product orders and inspecting current stock levels in retail shops. The merchandising domain applications included conducting various surveys at points of sale to gain input on product quality, shelf placement, exposure metrics, and retail prices from competitive products. The sizes of the projects varied from 80 to 170 modeled entities, and from 8 to 27 modeled business processes. The largest project stored approximately 2–3 million transactions each month.

Table 1. A list of software development process improvements introduced by AGM

	Before AGM	After introducing AGM
Requirements Definition	UML semantics were hard for our customers to understand	Visual application feedback on a graphical user-interface level
Implementation	Applications were implemented according to defined UML models. Software evolution meant additional effort for keeping the UML models in sync	The model itself is the implementation; implementation process leaves on implementing the specific JS-DSL action scripts which are integral to model itself
Verification and Validation	Each functionality point required a set of unit tests across all architecture layers (storing and representing information)	Unit tests are executed on a per predefined component level. New tests are required only when introducing new user-interface or data-processing components.
Distribution	Each new functionality point required the repackaging and redistribution of whole system.	New functionality points are introduced with new application model versions which are synchronized automatically upon application load. Repackaging and redistribution was required only when new modeling artifacts are introduced.

By using AGM, our application development team achieved noticeable time-savings. Although we have not yet reached a point where our customers have been able to develop their own information system applications, we have enabled our software analysts to define nearly complete end-applications. The only point at which software engineers were required was when using the JS-DSL to fine-tune the models.

The main benefit was accrued from the quicker development cycle, which enabled visual exploration and negotiation while gathering requirements with our customers. Model interpretation enabled rapid visual feedback of the end system; thus, it allowed quicker convergence to the end-stage requirements. A more detailed set of improvements grouped by software development process phases is given in Table 1.

AGM also made the installation and distribution of end-applications application users easier because the users all shared the same mobile interpreter. After installation on the users' smartphones, the client interpreters loaded their AGM models from the associated cloud environment. This facilitated the *configuration management* process in the sense that we were able to reduce the number of client application versions and releases.

Software evolution also became easier because the interpreters always pull the latest model changes at run-time. The previous classically-built client applications binaries were over 10 MB in size which made remote transport and installation on each smartphone difficult because some users did not have solid, stable cellular network connections. This is a common issue in mobile cloud computing [28]. Introducing the AGM solution required synchronizing only new model versions, which was considerably faster. The AGM mobile interpreter occupies slightly more than 1 MB and is redistributed to users only upon new AGM meta-model element releases or when new reusable components are added. In comparison, the generative approach typically generates source code compiled the same or similar to classical methods, meaning that every model update requires a complete re-installation.

The downside of the AGM approach is in the large investment required to implement the distributed interpreter. A single software defect in the interpreter is usually manifested among all end-applications and solving such problems involves redistributing the interpreter to all end-application users. The greatest challenge was implementing an interpreter that was both fast and energy efficient enough for use in the mobile cloud domain. It was essential for end-application users to be able to use the application throughout an 8–10-hour business day without having to recharge their smart-phones or tablets. As mentioned earlier, just-in-time interpretation and caching are essential in achieving that goal.

5. Limitations

Our research demonstrates an end-user developer friendly framework for building information system applications in the cloud. There are some limitations that need to be considered both in terms of the system itself, and the way we have conducted the evaluation. We present these separately together with our current belief and future plans on how these limitations could be circumvented.

5.1. AGM limitations

There are two drawbacks to the current implementation of the AGM. The first drawback is using the textual AGM-DSL language in defining the data, process and interface structure. The users are required to learn the syntax of the language which is achieved by using existing examples of the language constructs and the way these are mapped to resulting applications. We are working on a visual representation of the language which follows the AGM meta-model. Visually, the language will resemble the visual representation used in Figure 13. This will also enable even faster visual feedback where the resulting application can be rendered side-by-side to the model itself.

The second challenge for end-users is using the action language embedded in Javascript which requires a basic knowledge of the JavaScript itself. This is a serious limitation and a significant learning step for an end-user developer. Although this language only uses a small subset of vanilla JavaScript in the form of basic control flow statements and data-structures (e.g. we require no regular expressions, higher-order functions, Document Object Model, modules&packages, callbacks, or closures), end-users have a significant problem to understand the basic concepts of programming. This design choice results in the fact that these scripts were needed to be additionally composed by software developers. While we can still report significant time savings in the development of the end applications, the development process cannot be 100% offloaded to end-user developers. We plan to build a visual representation language in order to specify the behavior needed. A good example we are considering is the *Blueprints* language[7] which should be adapted so that it fits our AGM meta-model.

5.2. Research methods limitations

There are also some threats to the validity of this research that also should be considered [29].

Construct validity There are many forms of mobile applications that can be built. Our approach currently targets the data-collection applications which complement the existing information systems. Currently, the AGM framework has limited support for the data transformation, analysis, and reporting of the higher-level data aggregations.

Internal and external validity Our study did not perform controlled experiments on the degree of usability in designing the applications compared to the classical methods. We plan to conduct these experiments once we complete the framework with visual modeling and action script programming parts.

Conclusion validity Based on that controlled experiments were not conducted, a more general conclusion on the applicability of our results to the general case of mobile software development cannot be reported with significant confidence. A full scale controlled experiment on a convincing number of different application domains is required.

6. Related Work

Information systems modeling has been well researched within the generative MDD field. Table 2 lists some of the well-established general-usage MDD tools, the majority of which follow a generative MDD approach. For building information systems, Milicev proposed an approach using an executable UML profile called *Object-Oriented Information Systems* (OOIS) [2]. After the models are compiled, they can be used in a special runtime Java-based environment. *SOLOist* is a tool based on OOIS that uses Java code for application customization.

Unlike OOIS, which uses Java, Popovic et al. [9] developed a DSL to specify application customization code at a *platform independent model* (PIM) level. Similar to AGM, PIM is targeted at information systems, and its approach is also generative but uses a pre-generated application interface and database. Dimitrieski et al. [30] also took a generative approach in their Multi-Paradigm Information System Modeling Tool (MIST) for building information systems through the simultaneous use of three different approaches.

[7] Blueprints language is used in the *Unreal Engine 4* engine, which is available at `https://www.unrealengine.com/en-US/`

The selected approach can thus depend on the problem domain and on the knowledge and personal preferences of an IS designer. MIST translates models to a relational data model or a class model.

MIDAS is a model-driven generative methodology proposed by Cáceres et al. [31] for developing web-based information systems. MIDAS is a specific application of Model-Driven Architecture (MDA) for Web platforms that uses XML and object-relational methodology. Currently, however, MIDAS provides only structural modeling of information systems.

Boyd and McBrien [32] also used graph structures for model representation. They proposed a *hypergraph data model (HDM)* structure for data model representation. HDM concentrates only on the data model of application and alleviates the need for model-to-model transformations used in previous generative approaches. AGM also uses a directed property graph, but unlike HDM, it is interpreted at runtime.

Many studies have emphasized end-user involvement in application development. Cappiello et al. [33] developed a UI-centric model that enables end-user developers to create *mashup* applications by applying WYSIWYG (what-you-see-is-what-you-get) specifications of data integration and service orchestration. They argued that user interfaces function as the medium most easily understandable by end-users. Vera [34] suggested that MDD methodology can be simplified by using a set of user interface components configured to define system behavior. Francese et al. [35] proposed an approach for model-driven development of portable applications based on a finite-state machine for specifying GUIs, transitions, and data-flow. Rivero et al. [36] proposed an MDD approach to capture requirements from end-users faster by using user interface prototypes that end-users completely understand. Garzotto [37] also promoted end-user development by proposing an approach that combined Model-Driven and End-user Development paradigms in modeling web applications in *cultural heritage* and *cultural tourism* domains.

There are few tools intended to perform model interpretation. Mendix, a commercial MDD tool, exploits runtime model interpretation for modeling web applications [38]; however, we are unaware of the internal details of the Mendix interpreter's operation because it is a closed-source commercial product.

The idea of directly executing UML models was introduced by Riehle et al. [11], who proposed a UML virtual machine; the biggest issue with this approach was that UML is too abstract to specify the behavioral aspects of applications. Following the work of Shlaer and Mellor [39], the OMG issued two important standards: an executable subset of the UML language called Foundational UML (fUML) [40] and the Action Language for Foundational UML (Alf) [41]. These standards enabled designers to create UML models with detailed behavioral specifications that could be effectively transformed into executable programs. This capability enables graphical specification of UML models supplemented by textual semantically related Alf code. Because fUML and Alf are novel specifications, few tools support them yet, especially tools targeted toward the IS domain. There has been some research proposing Alf transformation [10], but to best of our knowledge, no tools for interpretation of these standards yet exist, especially in the domain of modeling mobile cloud applications. However, because these standards are aligned with our proposal, we are exploring ways to integrate fUML and Alf when specifying AGM models.

Table 2. Some of available MDD Tools

Product	Url
Generative approach	
WebRatio	`http://www.webratio.com`
WebML	`http://www.webml.org`
EMF	`http://www.eclipse.org/modeling/emf`
AndroMDA	`http://www.andromda.org`
IBM Rational Rhapsody	`http:/ibm.com/software/awdtools/rhapsody`
OpenMDX	`http://www.openmdx.org`
MetaEdit+	`http://www.metacase.com`
Cloudfier	`https://cloudfier.com/`
SOLOist	`http://www.soloist4uml.com/`
Interpretative approach	
Mendix	`http://www.mendix.com`
Hybrid approach	
System Vision	`http://www.mentor.com/products/sm`
OOA Tool	`http://ooatool.com/OOATool.html`

7. Concluding Remarks

In this paper, we proposed an approach that enables faster development of information system applications. The developed models are then interpreted directly at runtime.

We presented an architectural framework for an Application Graph Model (AGM), which is used to model IS applications using generic components and an action scripting language contained directly within the model. Through model interpretation, we enabled run-time adaptations of modeled systems, resulting in faster prototyping and rapid software delivery.

Implementing the AGM framework in concrete industrial projects resulted in several improvements. We enabled software analysts and developers to cooperate in implementing information systems, which drastically improved requirements negotiation and reduced the team size to a single analyst and engineer. However, we did measure a 15–20% performance penalty, which is especially noticeable in smartphone execution environments. This performance penalty is due to the overhead associated with querying the model for interpretation and run-time interface generation.

We are also working on building a graphical modeling environment for end-user developers [42] that will increase their productivity and reduce errors. Spreadsheet-like software has amply demonstrated that the *what-you-see-is-what-you-get* concept is highly appealing; having a runtime interpretive model is the foundation for a similar solution when designing IS systems. Our future work will also include efforts to implement a graph analysis algorithm that could be used to propose optimally efficient cloud deployment strategies [43] based on operational data inspections. Using this approach, a cloud executor could save costs by dynamically reassigning computation tasks among heterogeneous cloud resources according to workload demands.

Acknowledgments. This work has been supported in part by Croatian Science Foundation's funding of the project UIP-2014-09-7945.

References

1. Li, J., Rong, W., Yin, C., Xiong, Z.: Goal-oriented dependency analysis for service identification. Computer Science & Information Systems **16**(2) (2019)
2. Milicev, D.: Model-Driven Development with Executable UML. John Wiley & Sons (2009)
3. Harel, D., Marron, A.: The quest for runware: On compositional, executable and intuitive models. Software and Systems Modeling **11**(4) (2012) 599–608
4. Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. Journal of Internet Services and Applications **1**(1) (apr 2010) 7–18
5. Samad, J., Loke, S.W., Reed, K.: Mobile Cloud Computing. Cloud Services, Networking, and Management (2015) 153–190
6. Tankovic, N., Vukotic, D., Zagar, M.: Rethinking Model Driven Development: analysis and opportunities. In Luzar-Stiffler, V., Jarec, I., Bekic, Z., eds.: Information Technology Interfaces (ITI), Proceedings of the ITI 2012 34th International Conference on, SRCE (2012) 505–510
7. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal **45**(3) (2006) 451–461
8. Stahl, T., Völter, M., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology, Engineering, Management. (2006)
9. Popovic, A., Lukovic, I., Dimitrieski, V., Djukic, V.: A DSL for modeling application-specific functionalities of business applications. Computer Languages, Systems & Structures **43** (2015) 69–95
10. Ciccozzi, F., Cicchetti, A., Sjodin, M.: Towards Translational Execution of Action Language for Foundational UML. 2013 39th Euromicro Conference on Software Engineering and Advanced Applications (SEPTEMBER) (2013) 153–160
11. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The architecture of a UML virtual machine. Environment **36**(11) (2001) 327–341
12. Tanković, N., Vukotić, D., Žagar, M.: Executable graph model for building data-centric applications. Proceedings of the International Conference on Information Technology Interfaces, ITI (2011) 577–582
13. Dori, D.: Object-Process Methodology: A Holistic Systems Paradigm; with CD-ROM. Volume 1. Springer Science & Business Media (2002)
14. Ma, Q., Kelsen, P., Glodt, C.: A generic model decomposition technique and its application to the Eclipse modeling framework. Software & Systems Modeling (2013) 1–32
15. OMG: Meta Object Facility^{TM} (MOF^{TM}) Version 2.5 Specification (2015)
16. Reinhartz-Berger, I., Dori, D.: A Reflective Meta-Model of Object-Process Methodology: The System Modeling Building Blocks. Business Systems Analysis with Ontologies (2005) 130–173
17. OMG: OMG Unified Modeling Language (OMG UML) Superstructure (2010)
18. Nicolaescu, P., Klamma, R.: A Methodology and Tool Support for Widget-Based Web Application Development. In Cimiano, P., Frasincar, F., Houben, G.J., Schwabe, D., eds.: Engineering the Web in the Big Data Era. Volume 9114 of Lecture Notes in Computer Science. Springer, Cham (2015) 515–532
19. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
20. Brdjanin, D., Banjac, D., Banjac, G., Maric, S.: Automated two-phase business model-driven synthesis of conceptual database models. Computer Science & Information Systems **16**(2) (2019)
21. Miller, J.: Graph Database Applications and Concepts with Neo4j. Proceedings of the 2013 Southern Association for Information Systems (2013) 141–147
22. Charland, A., Leroux, B.: Mobile application development. Communications of the ACM **54**(5) (may 2011) 49

23. Elmore, A.J., Das, S., Agrawal, D., El Abbadi, A.: Towards an elastic and autonomic multitenant database. In: Proc. of NetDB Workshop. (2011)
24. Freeman, S., Pryce, N.: Evolving an embedded domain-specific language in Java. Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (2006) 855–865
25. Kossakowski, G., Amin, N., Rompf, T., Odersky, M.: JavaScript as an embedded DSL. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **7313 LNCS** (2012) 409–434
26. Crnkovic, I., Sentilles, S., Vulgarakis, a., Chaudron, M.R.V.: A Classification Framework for Software Component Models. IEEE Transactions on Software Engineering **37**(5) (2011) 593–615
27. Wargo, J.M.: PhoneGap essentials: Building cross-platform mobile apps. Addison-Wesley (2012)
28. Dinh, H.T., Lee, C., Niyato, D., Wang, P.: A survey of mobile cloud computing: architecture, applications, and approaches. Wireless Communications and Mobile Computing **13**(18) (dec 2013) 1587–1611
29. Jedlitschka, A., Ciolkowski, M., Pfahl, D.: Reporting experiments in software engineering. Guide to Advanced Empirical Software Engineering (2008) 201–228
30. Dimitrieski, V., Čeliković, M., Aleksić, S., Ristić, S., Alargt, A., Luković, I.: Concepts and evaluation of the extended entity-relationship approach to database design in a multi-paradigm information system modeling tool. Computer Languages, Systems & Structures **44** (2015) 299–318
31. Cáceres, P., Marcos, E., Vela, B., Juan, R.: A MDA-Based Approach for Web Information System Development. Methodology
32. Boyd, M., McBrien, P.: Comparing and Transforming Between Data Models via an Intermediate Hypergraph Data Model. Journal on Data Semantics IV **4** (2005) 69–109
33. Cappiello, C., Matera, M., Picozzi, M.: A UI-Centric Approach for the End-User Development of Multidevice Mashups. ACM Transactions on the Web **9**(3) (2015) 1–40
34. Vera, P.M.: Component Based Model Driven Development:. International Journal of Information Technologies and Systems Approach **8**(2) (jun 2015) 80–100
35. Francese, R., Risi, M., Scanniello, G., Tortora, G.: Model-Driven Development for Multiplatform Mobile Applications. In Abrahamsson, P., Corral, L., Oivo, M., Russo, B., eds.: Product-Focused Software Process Improvement. Volume 9459 of Lecture Notes in Computer Science. Springer International Publishing, Cham (2015) 61–67
36. Rivero, J.M., Luna, E.R., Grigera, J., Rossi, G.: Improving user involvement through a model-driven requirements approach. In: 2013 3rd International Workshop on Model-Driven Requirements Engineering (MoDRE), IEEE (jul 2013) 20–29
37. Garzotto, F.: Enterprise Frameworks for Data Intensive Web Applications: An End-User Development, Model Based Approach. Journal of Web Engineering **10**(January) (2011) 87–108
38. Henkel, M., Stirna, J.: Pondering on the key functionality of model driven development tools: The case of mendix. Perspectives in Business Informatics Research **BIR 2010,** (2010) 146–160
39. Shlaer, S., Mellor, S.J.: The Shlaer-Mellor Method. (1996) 1–13
40. OMG: Semantics of a Foundational Subset for Executable UML Models (FUML) Version 1.1 Specification (2013)
41. OMG: Action Language For Foundational UML (ALF) 1.0.1 Specification (2013)
42. Tankovic, N., Galinac Grbac, T., Zagar, M.: Experiences from building a EUD business portal. In: 2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), IEEE (may 2014) 551–556
43. Tanković, N., Galinac Grbac, T., Truong, H.l., Dustdar, S.: Transforming Vertical Web Applications Into Elastic Cloud Applications. In: International Conference on Cloud Engineering (IC2E 2015), IEEE (mar 2015) 135–144

Nikola Tanković is a postdoctoral researcher at the Juraj Dobrila University of Pula. His main research interests are directed to model-driven development of information systems, quality optimisation of distributed systems, and generally black-box model optimisation using soft computing and simulation. He is involved in several industry projects in developing predictive machine-learning models and web services in cloud.

Tihana Galinac Grbac is a full professor of computer science and the head Software Engineering and Information Processing Laboratory (SEIP Lab) at the Juraj Dobrila University of Pula. Her main research interests are related to large scale and complex software systems that are evolutionary developed. In a broader sense, she is also interested in a variety of large scale complex systems including smart cities, telecommunication networks and others. She is actively involved as the leader, management committee member and researcher in a number of research projects funded by European Union, Croatian government or industry partners. The results of her work are continously published in international scientific journals and conferences.

Received: October 11, 2018; Accepted: September 17, 2019.