IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received August 2, 2019, accepted August 27, 2019, date of publication September 30, 2019, date of current version October 16, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2944807

Restful State Machines and SQL Database

JOSEF KUFNER[1] AND RADEK MAŘÍK[2]
[1]Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague, 121 35 Prague, Czech Republic
[2]Department of Telecommunication Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 166 27 Prague, Czech Republic

Corresponding author: Josef Kufner (kufnejos@fel.cvut.cz)

This work was supported in part by the Knowledge-Based and Software Systems Group, Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic, and in part by the Grant Agency of the Czech Technical University in Prague, under Grant SGS14/193/OHK3/3T/13.

ABSTRACT State machines and a relational database may look like completely unrelated tools, yet they form an interesting couple. By supporting them with well-established architectural patterns and principles, we built a model layer of a web application which utilizes the formal aspects of the state machines to aid the development of the application while standing on traditional technologies. The layered approach fits well with existing frameworks and the Command-Query Separation pattern provides a horizontal separation and compatibility with various conceptually distinct storages, while the overall architecture respects RESTful principles and the features of the underlying SQL database. The integration of the explicitly specified state machines as first-class citizens provides a reliable connection between the well-separated formal model and the implementation; it enables us to use visual comprehensible formal models in a practical and effective way, and it opens new possibilities of using formal methods in application development and business process modeling.

INDEX TERMS State machine, web application, REST, MVC, multi-tier architecture, CQS, CQRS, ORM, SQL.

I. INTRODUCTION

How to build a model layer of a web application? Modern web applications are usually built using an MVC framework [1] (or similar, e.g., MVP). While the roles and responsibilities of the view and the controller are well established, the exact scope of the model remains somewhat vague. The remaining question is where to draw a line between the model and the controller.

A multi-tier (n-tier) architectures extend the MVC approach by inserting a business logic layer between the model and the controller (and sometimes a few more layers through the application). The business layer allows us to separate a low-level data access layer, a high-level business logic, and a presentation/application logic (the controller, and the view). Therefore, the rather complicated description of the behavior of model entities will not mix with SQL queries on the one end nor with a user interface and an API logic on the other end.

A command-query separation (CQS) pattern [2] provides another approach. Instead of adding an extra layer, it separates the command and the query execution paths which lead orthogonally through the layers. One path to query the data and retrieve the response, another to execute commands. Such a separation reflects the distinct requirements and responsibilities of each path, respecting SOLID principles [3]. The pleasant detail is that the CQS pattern is not in conflict with the multi-tier architecture or with the MVC pattern. In fact, these patterns fit together rather well as we will show later.

A common issue of the model and business logic is insufficient encapsulation. The multi-tier architecture expects that any tier communicates only with a layer directly above or below. Unfortunately, deeper layers often leak through into higher layers. For example, ORM (object-relational mapping) frameworks operate at the low-level data access tier and provide data objects which are tracked for modifications. The ORM framework then forwards these modifications directly into a database. If such an object gets passed into the presentation layer, the modifications may entirely bypass the business logic layer. Proper decoupling of the layers is difficult to do and easy to break; the price is high maintenance cost.

Another issue of today applications is a lack of formal models. The likely reason is the initial cost of the related infrastructure and the consequence of the misused YAGNI principle ("You Aren't Gonna Need It") [4]: In the beginning, the application is simple, and the formal tool would

The associate editor coordinating the review of this manuscript and approving it for publication was Zhangbing Zhou.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

INQUISIENT 2002
ServiceNow v. InQuisient
IPR2023-01071

bring too much overhead, and then, when the things get complicated, it is already too late. However, even a simple formal model gives us an explicit idea of how the application behaves, especially when security is our concern (regarding both reliability and access control). Web applications are typically built on top of two formal concepts: the relational algebra on which SQL databases stand, and the rather trivial REST & CRUD to define universal HTTP API. Unfortunately, the business logic is out of the scope of both of these formal concepts.

Many information systems already use the concept of "state" to manage workflows of their entities. Typical examples are bug tracking systems (open/close issue) and order management in e-shops (new/confirmed/delivered order). Many of these systems already use explicit state machines (finite automata) to visualize and manage the states. However, the scope of such a model usually dwells within the single entity with little to no connection to the rest of the application. Often a developer implements such a state machine using the formal model only as a specification with no permanent link between the model and the implementation. Such an approach is error-prone and requires additional work when updating the original model.

The use of a formal model is usually supported by theoretical arguments, like to provide proofs of certain aspects of the software, or to allow optimizations based on the model. With no doubt, such features are important. However, there is also another question to answer: Can we use the formal models to aid the software development itself?

This paper presents an architectural pattern based on Smalldb state machines [5], which we applied in the further proposed Smalldb framework. The core idea is to represent every entity in the model layer using a Smalldb state machine, a persistent nondeterministic finite automaton, which consists of a declarative formal definition, an implementation of transitions, and persistent data repository. The model layer then consists of two tiers: a low-level data access tier, and a high-level business logic encapsulated in Smalldb state machines – see Fig. 1. The upper tiers of the web application then read the state of the state machines and invoke their transitions. The state machines also provide access control and various additional metadata, for example, user-friendly icons and labels for generated navigation and menus.

The main reason why Smalldb uses simple state machines instead of more expressive formal tools, like workflows based on Petri Nets, is the simplicity and understandability. A non-technical customer can usually understand a statechart with only a little explanation. This provides a common graphical language to programmers and their customers so that they can discuss the business logic together. Moreover, the state machines are conceptually very close to REST [6], and it is easy to create a REST API for a state machine.

II. REST API FOR A STATE MACHINE

Before we get to explore the architecture of Smalldb framework, let us take a better look at the REST resource [6] and its

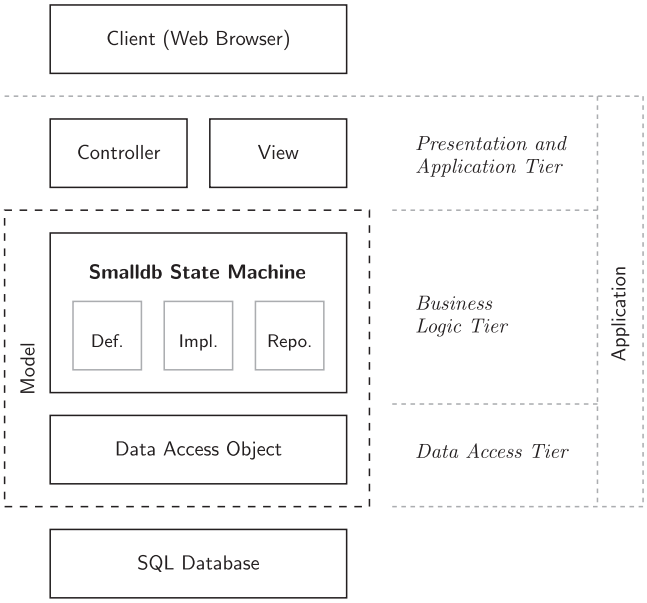

FIGURE 1. Application architecture with Smalldb.

behavior. In short, a REST resource provides a representation of an application entity. A globally unique URI identifies it, and a client manipulates it typically via HTTP requests using a predefined set of HTTP methods. The communication with the resource is meant to be stateless (usually with an exception on authentication), so that the communication is made of simple request-response cycles.

The HTTP/1.1 standard defines the following methods: Options, Get, Head, Post, Put, Delete, Trace, and Connect. Of these methods, only the methods Post, Put, and Delete are designed to modify a resource on a web server. A life cycle of the resource starts with its creation, usually via a Post request, then the resource exists and can be modified using Put requests. And finally, it can be deleted with a Delete request. We can represent such a life cycle as a state machine where the HTTP methods are transitions, and the resource has two states – "exists" and "not exists" – see Fig. 2. (The both black dots in Smalldb notation represent the "Not Exists"

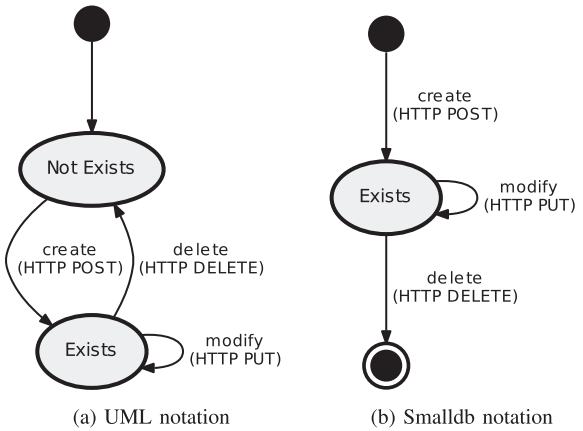

FIGURE 2. State diagram of a generic REST resource in two notations and mapping of their transitions to HTTP methods.

state, while in the UML notation the black dot only points to the initial state; the Section V will provide the details.) As we can see, the REST resources and state machines are closely related technologies.

Now, let us take the state machine a step further. Regardless of the HTTP methods, we can generalize the resource as a generic state machine and use as many states and transitions as desired (and practical). To do so, we need to change API so that it is possible to invoke any transition, not only those available via standardized HTTP methods. Nevertheless, an API for a generic state machine requires two operations: one to read its state, and another to invoke its transition. The suitable candidates are the HTTP methods Get and Post, respectively, as HTML forms use them for the very same purpose. Moreover, this way, we do not need to implement a specialized client. However, we need to specify which transition of the state machine the HTTP Post method should invoke. One possibility is to add the transition name as a parameter to the request, and another is to suffix URI with the transition name – RFC 3986 defines exclamation mark as an unused delimiter in URI which seems to fit this purpose. Having a URI assigned to each transition of each state machine (a REST resource) is useful as it allows us to query metadata about the transition, or an HTML form to invoke the transition by a subsequent HTTP Post request.

III. ARCHITECTURAL AND MODEL CONSTRAINTS

REST [6] provides us with a reasonable and well-tested set of architectural constraints which includes client-server architecture, stateless communication, layered architecture, and uniform interface with universal resource identifiers. Our goal is to extend the REST architecture by introducing state machines as a formal model of the business logic tier. To do so, we shall meet the following architectural and model constraints while preserving all RESTful properties of the application, so-called Smalldb Pattern:

A. DECLARATIVE FORMAL MODEL

If we wish to validate the behavior of the application, we need a useful description of the behavior. The source code is difficult to analyze (e.g., halting problem); therefore, a less expressive model is required. Such a model must be executable (in a sense it can be directly interpreted by a computer) so that programmers have no opportunity to introduce bugs during manual implementation. However, the model does not have to address all the details, and thus, it can stay simple, practical, and understandable. Smalldb state machines (and finite automata in general) provide such a formal model. A generic REST resource can be considered as a state machine of a fixed predefined structure.

B. ENCAPSULATED INTERFACE

The *formal model* defines not only the behavior of the entities but also their interface, and this interface should be the only API to access and manipulate the data. In our case, the state machine provides information about its state (name of the state and a persistent representation, e.g., a value object) and allows us to invoke a transition (to call a method). Additionally, it may provide us with various metadata (along with a reflection API). That's all. Nothing else matters nor should be visible to the higher tiers of the application; no data should be accessible nor manipulated any other way than via the interface of the business logic provided by the *formal model*, i.e., the Smalldb state machines.

C. STATE MACHINE SPACE

Each state machine should have a unique identifier. Such an identifier allows us to refer to a given state machine, and also, it allows us to map it to a unique URI of the corresponding REST resource. Therefore, all instances of state machines form a space which we can map to a space of URI. This allows us to build a generic API to browse and search the state machine space by various constraints (e.g., list e-shop products of desired parameters). Note the state machine space may be theoretically infinite; practically limited only by the range of data types used for the unique identifier and the available storage space.

D. PERSISTENT STATE STORAGE

Each state machine represents a persistent entity independent of the application run-time, typically a record stored in a database. Modifications of the state (updates of the records) should be done directly in the persistent storage without unnecessary caching or delays. This provides a single synchronization point (the database) when the application runs in many instances.

E. THE INITIAL "NOT EXISTS" STATE

Every state machine has the same one initial state – the ''Not Exists'' state. There are two reasons for this state: First, the ''Not Exists'' state introduces the concept of existence, instances, and constructors/destructors into the realm of finite automata. Second, the ''Not Exists'' state allows us to store the possibly infinite state machine space in a finite (and preferably small) database because there is no need to store the state machines in such a state.

F. ADDRESSABLE TRANSITIONS

Each transition of each state machine should have a unique identifier. Such an identifier should be composed of the ID of the state machine (a primary key) and the name of the transition. Then we can assign a URI to each transition and invoke it with an HTTP Post request or read the transition metadata (or an HTML Form) using an HTTP Get request. The combination of the *state machine space* and the *addressable transitions* allows us to build a RESTful *unified interface* (an HTTP API) to read and manipulate the state machines, as well as retrieve metadata from the formal model.

G. EXECUTABLE TRANSITIONS

The state machines are active during the transitions only; the states represent merely a passive waiting for the next

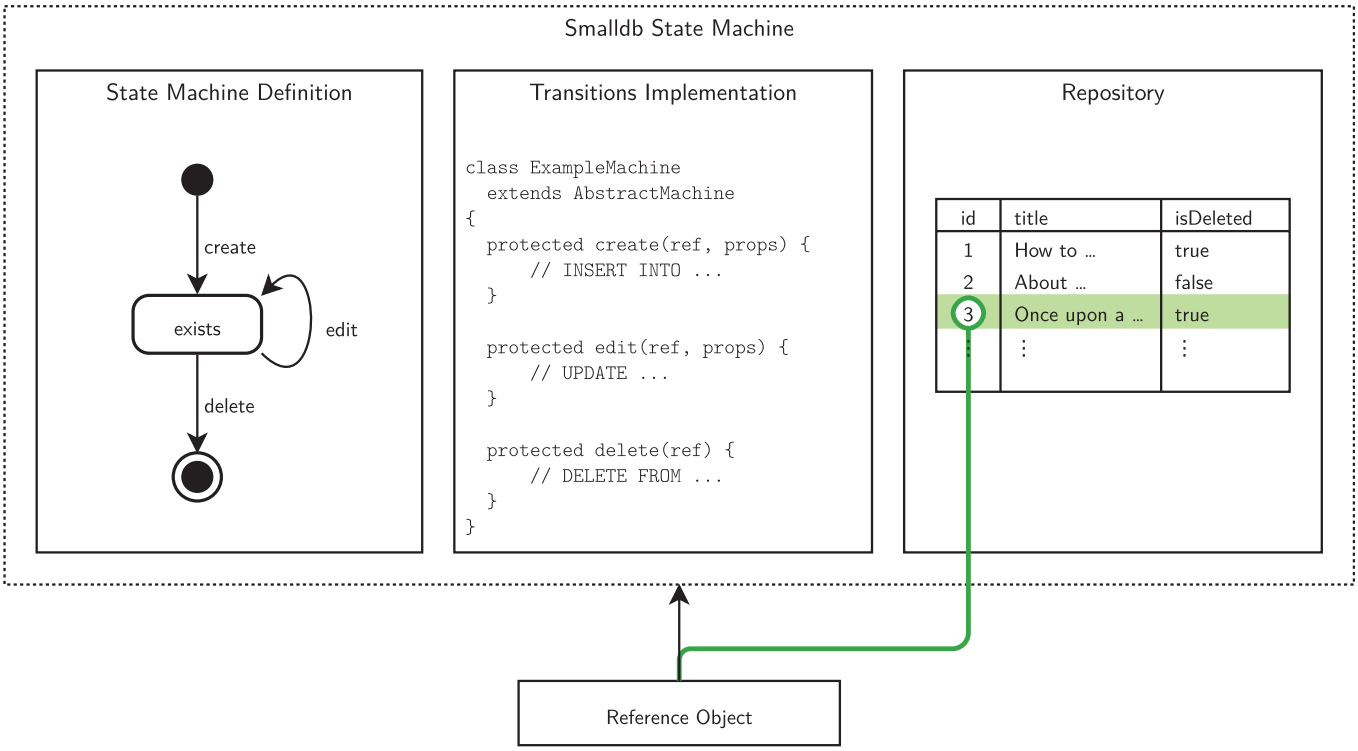

FIGURE 3. The three parts of Smalldb state machine.

transition. The transitions are expected to modify the persistent state and perform desired side effects. The transitions may accept parameters (like ordinary method calls), but otherwise, the invocation of a transition should be stateless, i.e., independent of the application state.

H. THE ABSTRACT ENTITY WITH A BORROWED RUN-TIME

The state machines are abstract entities which all exist since a developer defines the *state machine space*. Since then, all state machines passively wait in the initial "Not Exists" state with no run-time assigned to them. When the application invokes a transition, it also provides its run-time to the state machine so it can perform the transition. Once the transition is complete, the state machine returns the run-time to the application and passively waits for the next transition with its state stored in *the persistent state storage*. Because of this concept, the application does not instantiate the state machine; instead, the application communicates with the state machine via a reference object which provides the machine-related part of *the encapsulated interface*. This approach plays well with traditional SQL databases and does not impose additional limits on the horizontal scaling of the application as it decouples the model entities from the application run-time. Moreover, we can formally reason about the entity behavior without the need for a run-time.

I. CONCLUSION

These constraints give us a hint on how to enhance REST resources with the formal model. Note that these constraints are not in conflict with the REST constraints; therefore, the application should maintain all the RESTful properties. However, there is a conceptual shift in modifying the resources. The REST stands on representations, where the application translates a request for a representation modification into an action. With the state machines, the HTTP Post requests only invoke the transitions of the state machines explicitly, similarly to remote procedure calls. In this case, the representations provide the state information only, which can be retrieved by HTTP Get requests.

IV. THE IDEA

The constraints in the previous section gave us a rough idea of how to design the business logic tier of a web application and what we expect from the formal model which the business logic is supposed to use. In the following sections, we present the architecture of Smalldb framework, which combines the REST approach with the formal model based on nondeterministic finite automata.

The core idea of the Smalldb state machines is based on the three separated components – see Fig. 3: a state machine definition, a transitions implementation, and a repository. These components are relatively independent; however, a certain consistency is required to form a Smalldb state machine successfully. The state machine definition connects the circles and the arrows, the transition implementation implements the arrows, and the repository provides the circles. Such a decoupling of the components allows us to reuse the same definition for multiple state machines, e.g., the framework offers a prefabricated definition of CRUD state machine and allows the components to use different tools to fulfill their purpose.

The fourth component in Fig. 3 is a reference object, which has access to the three components and points to a particular state machine identified by a primary key (ID) in the repository (the ID number three in this case). The application uses the reference object to communicate with the state machine. The application does not instantiate the state machine itself like a traditional object in object-oriented programming as the state machine is an abstract construct mostly independent of the application run-time. Instead, the application instantiates only a reference object which provides an interface to invoke transitions and read the state of the state machine via the Smalldb framework.

A. THE STATE MACHINE DEFINITION

The state machine definition is a declarative formal definition of the state machine, typically in the form of a statechart or a state diagram. A programmer may draw such a statechart using a visual editor and store it as a GraphML file, which Smalldb framework can load directly, or he can enumerate the states and transitions in a state machine configuration file. Also, it is possible to combine these approaches – draw the statechart and then specify additional metadata in the configuration file. An additional exciting option is to generate the state diagram from BPMN process diagrams, where the generated state machine implements one of the participants (the application) in the process.

Due to the declarative nature of the state machine definition and use of the relatively simple model, it is possible to reason about the behavior of the state machine without running the application. For example, we can model users of the application as state machines too and then simulate the entire business process as a network of interacting state machines long before we implement the application. This way, we can formally verify the correctness of the application specification during the early design phase of the development.

The state machine definition is not only about states and transitions, but it also can provide various metadata. One of the highly desired extensions is access control: With each transition, we can specify who can invoke the transition. If a user is not allowed to invoke a transition, then the Smalldb state machine will reject the request. This provides us with simple and reliable access control, which we can formally verify along with the rest of the state machine definition.

The purpose of the state machine definition is to be the Single Source of the Truth (SSOT or SPOT; a variant of DRY – Don't Repeat Yourself) [7]. The transitions implementation asks the state machine definition, whether it shall allow the user to invoke a given transition and whether it is a valid transition at all. Moreover, the definition may enumerate available transitions of an entity, so that a user interface can show a menu, for example.

B. THE TRANSITIONS IMPLEMENTATION

The transition implementation is expected to update the state machine state in the repository, and to perform some desired actions (side effects). Unlike the traditional formal concept of the finite automata where transitions are considered instant, the arrows of the Smalldb state machine represent pieces of code, the transitions implementation, which moves the state machine from one state to the next. Still, the fundamental feature of the transitions is atomicity, although, its execution takes some time.

It is not essential how precisely a transition is implemented as long as it begins and ends in the defined states. Most implementation details are left hidden behind a few simple arrows. Therefore, to maintain the specification accurate and useful, the omitted details must be of known properties and well contained within explicit boundaries; the features which do not contribute to the model behavior are stripped away. Thus, the transition between states defines a gap which is then treated as a specification for a programmer who implements the transition. The definition with well-defined gaps, a sparse approach to the formal specification, is the key trick of the Smalldb framework.

Smalldb state machines are nondeterministic finite automata. The nondeterminism represents multiple possible outcomes of the invoked transition. In some cases, the state machine may not have the prior knowledge necessary to choose the correct arrow. In other cases, the behavior is just too complicated to specify, and such a specification would not be practical. Either way, it always ends in one of the allowed states (and if not, the state machine detects an implementation error).

The purpose of the transitions implementation is to informally but practically fill the formally defined gaps in the state machine definition. The two components form a complementary pair; the one allows us to reason on the overall behavior, the other provides the unpleasant details in a properly contained package.

C. THE REPOSITORY

The repository provides persistence. It provides a uniquely identified persistent representation for each state machine. Such a representation is typically obtained from a row in a SQL table, but it can be a file, an LDAP entry, or even a remote device. There are only two mandatory features of such a representation: a state machine unique identifier and a ''state function''. The unique identifier locates the state machine within the state machine space, e.g., a primary key on the SQL table. The ''state function'' maps the representation to a state of the state machine without any side effects, e.g., a SQL expression (usable in select queries) which returns a state name for each row in the SQL table. The rest consists of application-specific implementation details.

From the architectural point of view, it is important to realize that the repository spans over two tiers of the application – the business logic tier and the data access tier. However, it must respect these tiers; otherwise, the application would become unmaintainable. The difference between the two tiers is that the data access tier operates directly with the raw data in the persistent storage (e.g., rows in the SQL table), while the upper business logic tier operates with an interpretation

of the raw data (e.g., hydrates a value object with a SQL row data). As the Repository pattern [8] (pg. 322) shows, it is a good idea to shield the application from the underlying storage using a facade [9].

Concerning the other two components, the transitions implementation uses the repository to manipulate the persistent representation, and thus the state of the state machine. The role of the state machine definition is limited to provide valid values of the state function. The state function serves as the glue between the state machine definition and the persistent representation – no matter how complex the representation is, the state function converts the representation into a single scalar value, the state of the state machine. Moreover, since the state function is the only point of contact, it decouples the formal model from the representation.

The purpose of the repository is to identify individual state machines, retrieve their state, and to provide the transitions implementation with a tool to manipulate the persistent storage and thus the state of the state machine.

V. KEY FEATURES OF SMALLDB STATE MACHINE

The architecture of Smalldb framework stands on the following few specific features of the underlying state machine. While Smalldb framework assumes the use of nondeterministic finite automata [5], the architecture is not limited to this kind of automata and can use more sophisticated machines as long as these specific features are preserved, for example, hierarchical automata or workflows based on Petri Nets.

The first feature specific to Smalldb is *the initial "Not Exists" state*. As stated before, this unified initial state allows Smalldb to deal with the theoretically infinite state machine space, because the lack of any other information represents this state. Since every formal automaton has an initial state, Smalldb merely defines the "Not Exists" state as the initial state. In comparison to the generic initial state, the "Not Exists" state introduces the concept of construction and destruction of the automata – the transitions from the "Not Exists" state represent constructors, transitions to the "Not Exists" state represent destructors.

Section II and Fig. 2 presented a simple state machine using the two notations – the UML notation (Fig. 2a) and the Smalldb notation (Fig. 2b). The only difference between the two notations is in the syntax of the initial and final state. Since the initial state of every Smalldb state machine is the "Not Exists" state, we used the black dot to directly represent the initial state instead of using it as a pointer (marker) to the initial state as it is in UML notation. Likewise, the circled dot representing a final state got repurposed to denote the "Not Exists" state. This way, the practical state diagrams, which are usually small, are more linear and better correspond with real-world workflows with the start and the end. Therefore, when the initial and final states are the same (the "Not Exists" state), we cut the cycle in the diagram to reflect the logical workflow of the entity. We can see this happen in Fig. 2 where the UML notation hides the begin and the end of the life cycle in a loop around the "Not Exists" state, but the Smalldb notation is clear about the begin and the end of the business workflow. The use of slightly modified notation may complicate the use of existing tools and editors; however, our experience shows that such tools usually do not follow UML standards strictly (if at all), and provide sufficient level of customization to deal with the difference.

The second feature is *the state function,* which maps the representation in the persistent repository to the state machine states, usually implemented using a simple SQL expression. The purpose of this function is to connect the arbitrary representation in the persistent storage to the formal state machine definition where a state is merely a named circle. The state function is a tool that maps the reality to virtually any formal construct which deals with states. Additionally, the definition of the state function also defines instances of the state machine as the domain of the state function is a subset of state machine space.

The *concept of nondeterminism* is an optional feature of the formal model; however, if used, its interpretation needs to be consistent with the interpretation of Smalldb state machine. In Smalldb, the nondeterministic transition represents insufficient knowledge at the moment of transition invocation – an unknown external influence, or behavior too complicated to model; and thus the transition has multiple possible outcomes for the same (incomplete) input. In other words, multiple arrows with the same label represent a single transition with multiple results; we invoke such a transition like any other, and then we wait what happens (within the constraints).

If we could predict the unknown external influence, we could replace the nondeterminism with guards (signals from other automata) and make the state machine deterministic. While the real world[1] prevents us from doing so, we can apply such an approach in simulations, and formally verify the business process. This introduces an interesting concept where a single state machine is non-deterministic, but a group of interacting (non-deterministic) state machines may form a deterministic model.

As we can see, the architecture of Smalldb framework imposes only a few easily satisfiable requirements on the formal model. Therefore, we can choose the best fitting automaton for each underlying state machine, or extend such automata with application-specific features without jeopardizing the overall architecture.

VI. ARCHITECTURE OF SMALLDB FRAMEWORK

Smalldb framework provides a practical working implementation of the idea presented in the previous sections. The core component of the framework is the Smalldb state machine, which we will examine in this section.

Fig. 4 presents the structure of the Smalldb state machine and the nearby components – the application & presentation logic above the state machine, and the three underlying storages below the state machine. The schematics still preserves

[1] Assuming the real world is a system responding to unmodelled external events.

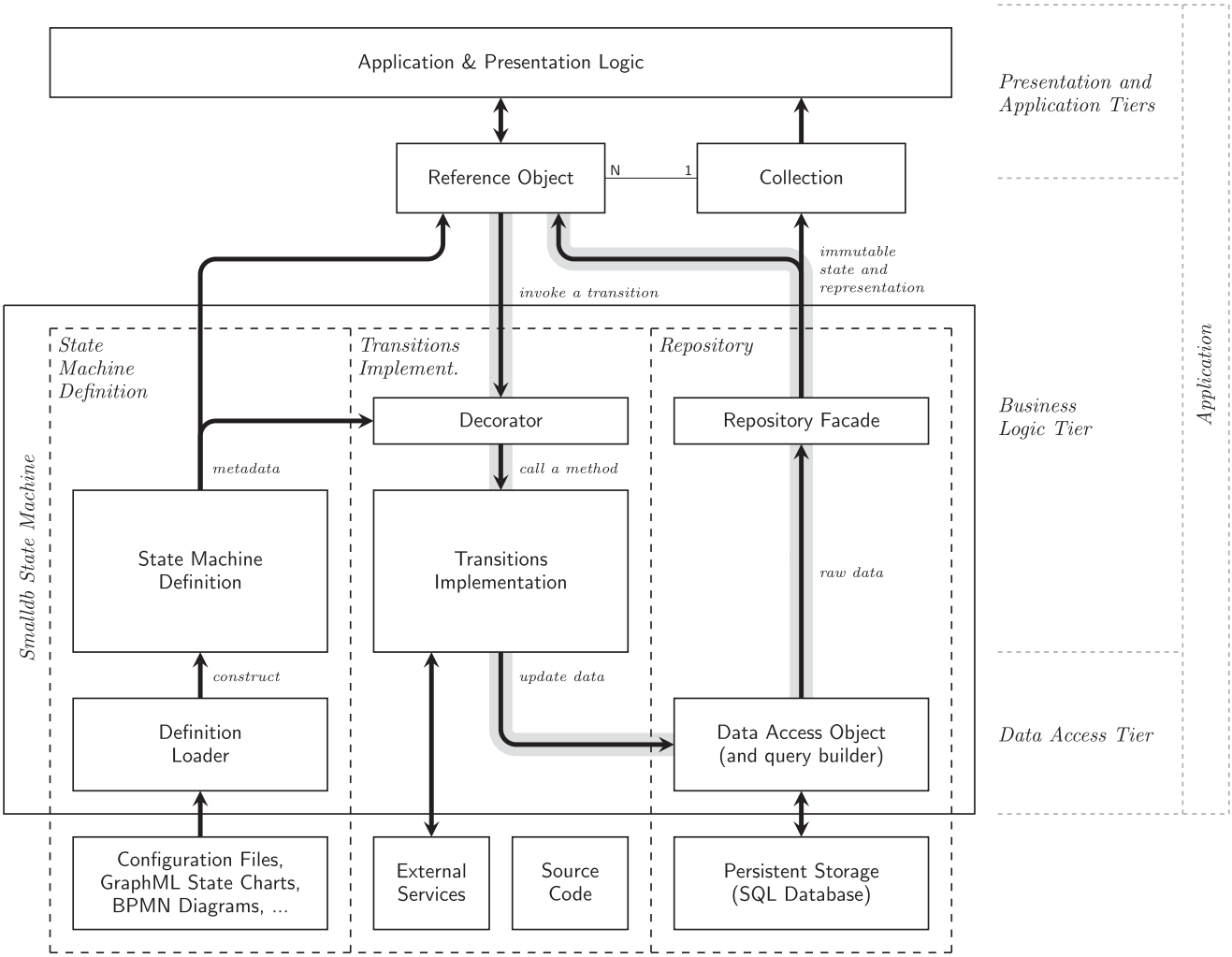

FIGURE 4. Architecture of a web application with Smalldb framework.

the separation into the three parts from Fig. 3, but since the underlying infrastructure under each of the three parts is specific to each part, we expanded the boundaries of the parts to contain lower tiers in addition to the business logic.

A. COMPONENTS SUMMARY

The following sections will describe in detail how the components from Fig. 4 work together, but before that, let us take a brief look at the individual components and their respective roles.

- *Reference Object* is a pointer pointing to the desired state machine. It provides API to read the state machine state, invoke its transitions, and inspect state machine definition.
- *Collection* is a collection of the *Reference objects*. It may provide some optimizations when retrieving state machines related to each state machine in the collection (e.g., retrieve all authors of all blog posts on a home page with a single SQL query).
- *Definition Loader* loads JSON files, GraphML statecharts, and BPMN business process diagrams. Then it combines these resources and infers the *State Machine Definition*. This processing may happen during compilation or during run-time initialization.
- *State Machine Definition* is a constant data structure describing the state machine; it is a run-time representation of the formal model.
- *Decorator* validates transition invocations against the *State Machine Definition* and rejects invalid or denied requests. It also verifies that the new state is one of the expected states once the transition is completed.
- *Transitions Implementation* is an executable code which implements the state machine transition. It manipulates the data within the *Persistent Storage* via *Data Access Object* to update the state machine state.
- *Repository Facade* translates raw data from the *Persistent Storage* into business logic entities in the form of *Reference Objects* or *Collections* of the *Reference Objects*. It provides an API to browse state machine space (e.g., faceted search [10]) and to retrieve *references* to state machines of specific IDs.
- *Data Access Object* provides tools to read and manipulate the data within the *Persistent Storage*. It may also provide a query builder to search the state machine space

effectively. Alternatively, an ORM framework may be used instead (see Sec. VII).

B. LOADING THE DEFINITIONS

Before the first interaction with a Smalldb state machine, the application must become aware of the state machine. The framework loads configuration of each Smalldb state machine and passes it to the *Definition Loader* – see Fig. 4, bottom left. The configuration may be provided as a combination of various forms – from source code annotations, configuration files, GraphML statecharts, to BPMN business process diagrams. Some forms may be usable as they are; others may require advanced analysis and inferring of the state machine [11]. The *Definition Loader* may process the configuration during application compilation, or it may do so during run-time startup and cache the result (e.g., within a compiled dependency injection container). In the end, the *Definition Loader* compiles each configuration into a state machine definition.

Once the definition is loaded, the *State Machine Definition* component provides it to the rest of the application. It answers questions like "Is this transition allowed in the given state?", or "Which transitions are available for this state machine?". At this point, the state machine definition is a static data structure with a library of helper methods. We would call it a reflection API in the object-oriented world. Note the definition is a data structure, and this whole process does not involve any generated code which a programmer should modify; the *Definition Loader* may generate a simple source code as an effective way of caching the static data structure representing the state machine definitions.

The loaded definitions are merely a beginning. There are two more components, the transitions implementation and the repository, which each must be initialized separately (if used[2]), but these three components need to be linked together to cooperate properly.

C. FRAMEWORK INITIALIZATION

Traditional ORM frameworks, for example, Doctrine [12], typically start their initialization from a repository class. Such an entry point requires an entity manager (or a similar component) which bootstraps the whole framework, while the repository registers its entity type in the entity manager.

Smalldb Framework, on the other hand, stands on three relatively independent components – the state machine definition, the transitions implementation, and the repository. The initialization of each component using a traditional dependency injection (DI) container is straightforward, and it is likely to be automatic; however, linking the correct three components together to form a single state machine is something with which the DI containers have difficulty to deal, at least if we try to utilize autowiring and other features which configure components automatically.

A typical DI container [13] stands on a configuration which describes how to instantiate registered services, i.e., which constructors or factories to call, and which other services each of them requires, e.g., as constructor arguments. Such a configuration forms a service dependency graph. When a service is requested from the DI container, the container recursively walks through the dependency graph and instantiates the requested service as well as the services on which it depends, effectively constructing a spanning tree of the service dependency graph. There are two ways to specify the dependencies of the services. First is to name each service and specify the name of each dependency of each service, which requires a tedious amount of manual work. The second approach, the autowiring, is to use the type system of the used language to tell us which services match with which constructor or method arguments so that the DI container can guess the dependencies automatically. Unfortunately, this approach is not viable when there are multiple services of the same type, e.g., multiple definitions of Smalldb state machines; such dependencies must be defined explicitly.

The consequence of the Smalldb framework architecture is that API entry point, the repository, does not match with an initialization entry point. For the DI container, this means there is no obvious spanning tree in the component dependency graph because we do not have a single root from which to start; in fact, we have three or four roots. The Smalldb framework overcomes this difficulty by extending state machine definitions with a name of related transitions implementation and repository. During the DI container compilation, the Smalldb framework collects the state machine definitions and generates lightweight provider objects, which each carry a quadruplet identifying the linked three components and a reference class. The provider objects are then registered in the DI container to establish the explicit connections, and to provide lazy loading of the linked components.

It seems that a state machine definition is a suitable place where to identify the other components related to the given state machine because it is the only component available at compile-time and thus we can use it to generate the configuration of the DI container.

D. APPLICATION INTERFACE

The *Application & Presentation Logic* communicates with the Smalldb state machine using *Reference Objects* (Fig. 4, top). Each reference object points to its state machine, provides the state of the state machine, and allows its user to invoke a transition of the machine. Additionally, the reference object mediates access to the relevant state machine definition. The reference object alone does nothing but forwards the requests to the responsible components, though.

The *Collection of the reference objects* (Fig. 4, top right) serves as a representation of a typical answer of the

[2]For example, there is no need to initialize the *Transitions Implementation* when we wish only to provide the information about the state of a state machine without invoking a transition. Similarly, we may not need to initialize the *Repository* until we need to load the state of a state machine.

Repository Facade when browsing the state machine space. The collection may be a simple list, but it may provide significant optimizations when retrieving state machines related to each state machine in the collection (e.g., retrieve all authors of all blog posts on a home page with a single SQL query).

E. SMALLDB STATE MACHINE WORKFLOW

The processing begins with the *Application Logic* receiving an HTTP request. Using the data within the request (e.g., an ID, or another unique identifier) the *Application Logic* obtains a reference object from the *Repository Facade* – see the arrows with the grey outline in Fig. 4. Such a *Reference Object* points to the desired Smalldb state machine and allows the application to retrieve the state of the machine, and the persistent representation from the *Persistent Storage* (e.g., the data stored in the SQL database).

If the HTTP request is only to retrieve the data, the workflow can end here, and the *Presentation Logic* replies with an HTTP response.

When the application logic needs to manipulate the state machine state, it invokes a transition using the *Reference Object*. The *Reference Object* passes the invocation to the *Decorator* inside the state machine. The *Decorator* decides whether the transition is valid given the state of the state machine and other facts, e.g., user's access rights. If the transition invocation is valid, the *Decorator* calls the corresponding *Transition Implementation*. The *Transition Implementation* uses the *Data Access Object* to manipulate the representation stored in the persistent storage. As a consequence, the state of the state machine is updated, and the cycle closes.

As we can see, the workflow forms a cycle where the only branching is whether the *application logic* retrieves a single *reference* or a *collection of the references*. This fact suggests that the architecture is as simple as possible.

F. COMMAND–QUERY SEPARATION

The general idea of a Command-Query Separation pattern (CQS) [2], [14] and a closely related Command-Query Responsibility Segregation pattern (CQRS) [15] is to strictly separate components which query data and which modify the data – see Fig. 5. The reason for such a separation lays in distinct responsibilities and requirements of such components so that they respect the SOLID principles [3] better. Both CQS and CQRS is mostly known from microservice applications, but the concept itself is much older [2].

The benefits of CQS pattern origin in the elegance with which it works in distributed environments. Because the commands are separate units which do not return data; they can either perform the actions directly or enqueue events for later processing. It is also possible to use different models to read and write the data.

The query component of the CQS pattern is closely related to the Repository pattern [8] (pg. 322), where the repository provides an interface to query the underlying data store. The

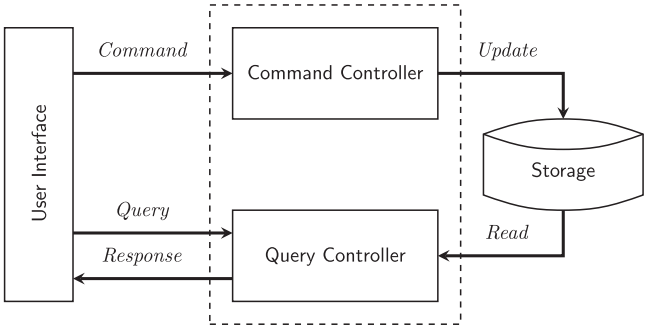

FIGURE 5. The core concept of CQS and CQRS.

repository encapsulates the query logic and, e.g., keeps all SQL queries in one place so that they are easier to manage. It also can use various caching techniques to scale the application for large loads. The Smalldb architecture specifies only the repository facade with no details because the underlying mechanism to query the database may vary. When using an SQL database, the repository facade is likely to utilize a query builder, and then it only wraps the result into the Reference objects. However, we may use a fundamentally different data store for some of the entities in the application (e.g., an aggregate root of event sourcing [16] implementation, or sensors connected to a local bus) and then the Smalldb state machine provides a unifying API.

The command component corresponds to the transitions implementation component in the Smalldb architecture. It executes the actions required to transition the state machine from one state to another. To do so, it requires various tools, e.g., to send an e-mail notification. Also, it must ensure data consistency and respect access control rules. Its API focuses on individual transitions (commands) rather than queries and collections. While the CQS pattern does not specify details on how to perform the commands, the Smalldb architecture defines the decorator to validate the requested actions against the formal model, the state machine definition, and ensure that such a transition is valid and allowed. Similarly to the repository facade, the decorator provides a unifying API to the underlying mechanism which performs the transitions. When using an SQL database, the underlying transition implementation will likely execute some update and insert SQL queries; moreover, it may also emit events or send commands to connected devices.

As we can see, such a separation makes each component easier to develop and maintain. While the overall architecture may look more complicated, the individual components become simpler and more manageable; thus, the application development is easier (and cheaper).

G. ACCESS CONTROL

When the application logic invokes a transition, the request always goes through the *Decorator* (see Fig. 4) which decides whether the request is valid according to the state machine definition and if so, the *Decorator* executes the *Transition*

Implementation. Since the *Decorator* processes all transition invocations, it is the perfect place where to implement an access control mechanism.

The access control mechanism requires typically two more things aside from the request: access control rules and security context.

The security context typically represents a currently logged-in user and his roles. The DI container of the application injects the context into the Decorator. The particular form of the context depends on the framework used in the rest of the application.

The access control rules are the interesting part. The state machine definition is the single source of the truth, and as such component, it provides not only the states and transitions but also various metadata relevant to these, including the access control rules. Therefore, each transition is labeled with a rule saying who and under which circumstances is allowed to invoke the transition.

Presence of the access control rules in the formal model allows us to visualize and reason about who can do what and when. State reachability is usually a very simple problem, but only until we apply access control rules and thus disallow some transitions. Then the state reachability algorithm can verify that the given group of users with a specific combination of permissions can reach the given goal.

H. CONSISTENCY

The presented architecture of Smalldb framework builds on three relatively separate components: the state machine definition, the transitions implementation, and the repository. For the proper function of the whole framework and the application above, we must ensure the consistency of the three components. As stated before, the state machine definition connects the circles and the arrows, the transition implementation implements the arrows, and the repository provides the circles. It all must fit together.

The question is how to ensure consistency and who is responsible. The easy answer is to leave it up to the programmer. While we do not have a better answer (yet), we can at least provide a tool to verify some of the consistency constraints automatically.

At the compile time, when the Smalldb framework loads the state machine definition, we can trivially verify that an implementation exists for each defined transition, i.e., that each arrow is backed by executable code.

Static analysis of the state function may tell us whether every possible representation in the repository maps to a valid state machine state and vice versa, i.e., all the used circles are defined properly. The static analysis may be difficult to automate in general; however, most practical applications suffice with a simple decision tree applied to the representation space, which is easy to analyze manually. At this point, we want to verify that the state function returns only the valid states defined within the respective state machine, and the state function returns such a valid state for every possible (not necessarily valid) representation.

Verification of the transitions implementations is feasible only in run-time as the implementation is usually a piece of code written in a Turing-complete language. The Smalldb framework retrieves the new state machine state (using the state function) after each transition and validates it against the state machine definition. In case the new state is not allowed by the definition, the Smalldb state machine yields an error. Moreover, if the invalid transition is limited to the scope of a SQL transaction, it can be safely rolled back to the previous valid state. This kind of validation is useful during development and testing of the application but does not enforce the correctness of the application.

While the separate formal definition of the state machine requires the developer to maintain consistency, it makes it much easier to collect the specification from the customer. Then, the rigid connection with the implementation ensures that the specification (the model), the implementation, and also documentation always stay up-to-date.

VII. ORM: OBJECT–RELATIONAL MAPPING

While many of the web applications today use ORM as their model layer, the Smalldb state machine builds on top of a data access object, as presented in Fig. 4. So, is it possible to use ORM within the Smalldb state machine?

Before we answer this question, let us take a look at how ORM frameworks work. The first and the most obvious concept of ORM is to map records in the SQL database to objects of object-oriented language. However, the core concept of ORM is the "Unit of Work" [8], [17] (pg. 184). In short, the ORM framework collects modifications done by the application as a "unit of work," and once the application finishes, the ORM framework applies the modifications to the SQL database. This allows the ORM framework to aggregate lots of calls of setter methods of the mapped objects into a single SQL update query.

The design of Smalldb state machine naturally bounds modifications of the SQL database into state machine transitions, and reference objects can provide caching, thus replacing the need for the Unit of Work as they both solve the very similar problems. However, there is no conflict between these two as long as each unit of work stays enclosed within a single state machine transition and the objects passed outside the Smalldb state machine are disconnected from their unit of work to avoid any unexpected behavior when the application logic processes the objects.

It is essential to realize that ORM frameworks do not replace the model layer of the application. The ORM is rather a low-level tool to access a SQL database. In this light, it may make sense to use an ORM framework in place of the data access object within Smalldb state machine. The ORM framework may be useful when dealing with relations and object hydration (mapping raw SQL records to business-domain objects).

On the other side, ORM frameworks add much complexity to the application, which is not always desirable, and many

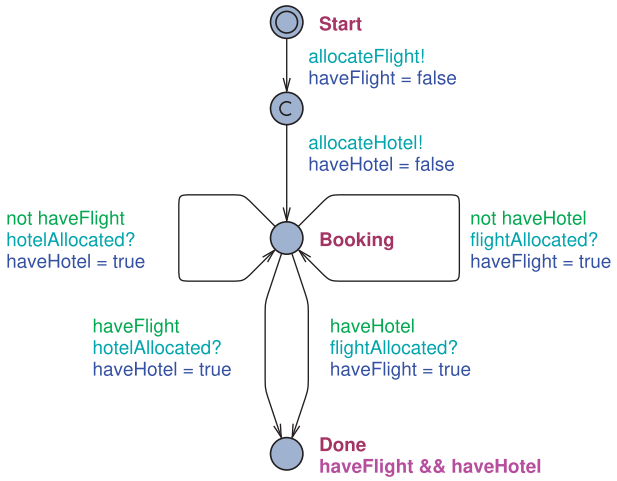

FIGURE 6. Example: Booking of a flight and a hotel using Uppaal [19].

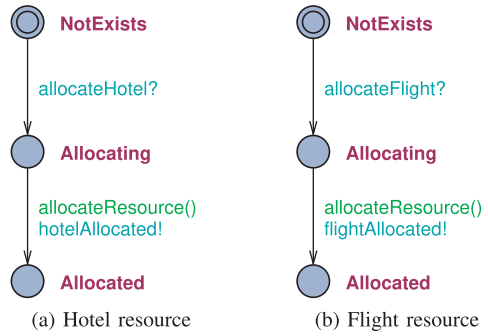

FIGURE 7. Flight and hotel resources for the example in Fig. 6.

developers prefer to use more straightforward and predictable tools. The Smalldb framework provides a choice as the repository facade shields the rest of the application from either of the low-level tool.

VIII. TRANSACTIONAL BEHAVIOR

The theory of finite automata defines a transition as an atomic operation [18]. A practical implementation of such atomicity is a challenging problem typically solved using locking mechanisms and transactions, which can be rolled back if something unexpected happens. Traditional SQL databases provide us with such mechanisms, and thus we can implement transitions of our state machines as atomic operations. The Smalldb state machine alone has no mechanism to enforce transactional consistency of the transitions, and it fully relays on a correct implementation of transactions within the transitions.

When we deal only with entities stored in an SQL database, we can ensure the transition atomicity by wrapping each state machine transition into an SQL transaction. A failure during a state machine transition then rolls back the transaction and the state machine returns into the previous state. The situation is the same as with any other database application.

The fun begins when dealing with external resources which may become unavailable before the transaction is committed, for example, when booking a flight and a hotel. In such situations, the use of a single state machine transition to allocate multiple resources may be impractical. We may add an intermediate state in which the state machine will wait for the allocation of the resources. The state function of Smalldb state machine (see Sec. V) allows us to asynchronously wait for multiple resources in a single state – the intermediate state may have a transition which records successful allocation of each resource, and when the last resource is allocated, the transition will advance to the next state instead of returning into the intermediate state.

Additionally, we may want to represent the resource allocation using an additional simple Smalldb state machine, so that we can track which resources we allocated from 3rd-party services. Then the resource allocation process becomes a network of interacting state machines, and we can use a formal tool, e.g., Uppaal [19], to ensure correctness in such a complicated situation as presented in Fig. 6 and 7 (where "allocate!" represents sending a signal to another state machine, and "allocate?" represents receiving of such a signal; failure recovery neglected for simplicity).

IX. CASE STUDY: APPLICATION DEVELOPMENT

The architecture of Smalldb framework is designed to ease development of web applications, so this section will highlight some of the interesting aspects of the use of the Smalldb framework and, more importantly, how it affects the development cycle of a web application.

The example application is an information system for a school to help it with organizing its training courses. The school lists the courses, trainees may enroll, or their employers may enroll them. The companies and individuals have different requirements on how to process payments. Additionally, some courses are finished with an exam so that successful participants earn a certificate.

While we do not wish to discuss particular details and complications of the business processes that this information system supports, we would like to provide a glimpse of the overall complexity of the application and its entities – it started with 18 state machines of which eight are core machines we designed on a single sheet, see Fig. 8. As we can see, many entities involve some nontrivial behavior which needs to be specified, modeled, implemented, and tested. To do so, the precise description of their behavior needs to be passed among various groups of people – from the training facility employees demanding the information system, to software architects, developers, and testers.

A. ANALYSIS

The initial analysis showed, that while the application is rather small, the behavior of the core entities is difficult to comprehend because the application deals with users with vastly distinct workflows due to their situation and

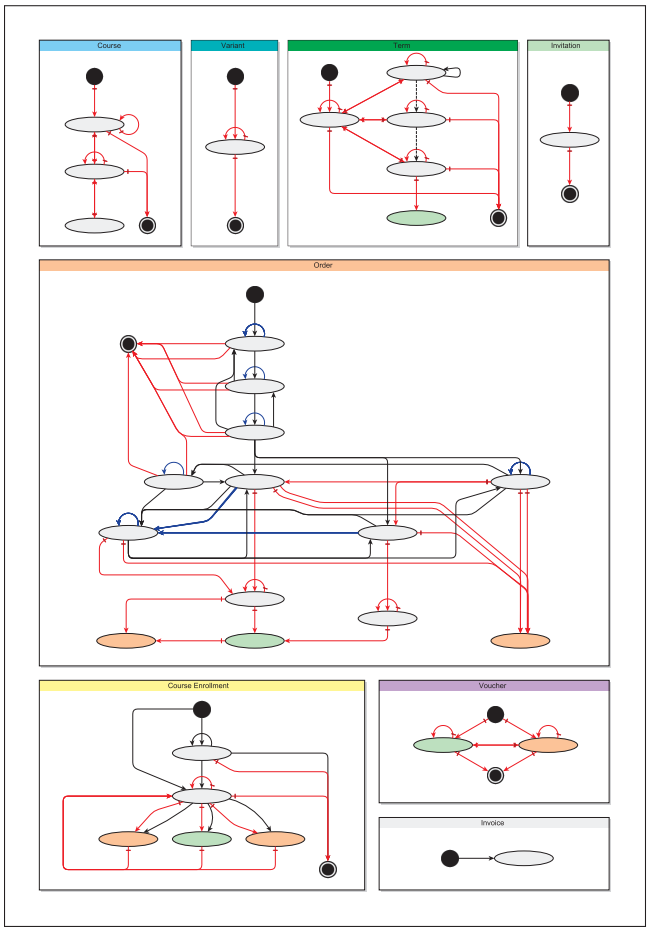

FIGURE 8. State diagram sheet preview to present the complexity of the application.

environment. These workflows required thorough discussions with the customer, and we needed to find common ground to understand what the users are expected to do. In particular, some of the users were people attending the courses and paying themselves in advance, while others were companies ordering courses for their employees in bulk, some of them even billed monthly with pre-approved limits instead of per course basis. The goal was to provide a unified ordering process which could satisfy the needs of all customers without unnecessary complications to either side.

Such a complex workflow requires one thing more than anything – thorough communication with the customer (or users) to get the process right. The state diagram sheet presented in Fig. 8 proved invaluable during the discussions over the workflows because it provided a representation of the application internals, which was understandable by both developers and the customer. Even non-technical customer can, with some basic explanation, understand the concept of a state diagram and what it represents in respect to the discussed business process. In fact, our experience showed that the customer was able to find mistakes in our models while discussing the diagram sheet.

Such a diagram sheet is not meant for the discussion only – it already is a part of the implementation because we shall use it later as part of the state machine definition component. Moreover, whenever the definition sheet changes, the framework automatically compiles it into new state machine definitions, and thus the implementation instantly reflects their changes. This way, the state diagram sheet become a substantial part of the source code.

B. PROTOTYPING

Initial prototyping consisted mostly of manual walkthroughs through the scenarios, carefully interpreting the state diagrams and inspecting what user can and will do at a given step. Once the workflows took shape, early prototypes of the user interface helped to clarify details. Also, because every transition of a state machine requires some user interface to invoke it, we knew that the designed user interface covers all the use cases. The early prototyping allowed by the state diagram sheet provided us with a useful model on which we could build the application. Moreover, we avoided significant redesigns in later stages of the development because we could (and we did) perform significant changes in the design as our understanding of the customer's workflows improved. Note that at this stage, the implementation did not exist yet, or it consisted only of user interfaces without any logic behind them.

To advance from the formal model and drafts to a minimal working implementation, each entity used in the application needs to be defined. The basic idea is to utilize the diagram sheet as much as possible to avoid duplicate efforts in the design and implementation. Such a definition consists primarily of two things: The first is a link to the state diagram in the state diagram sheet; in this case, it was an ID of the element in the GraphML file of the state diagram sheet created using yEd editor. The second item that is entered to the definition is the name of an SQL table, where the state machine stores its data. Remaining parts of the definition include mapping database columns to object properties, access control rules (see Sec. VI-G) and various metadata, e.g., labels and icons, to automatically render navigation menus and buttons – these parts are mostly indifferent to the state machine itself.

C. IMPLEMENTATION

Once we have all state machines (entities) defined, we need to implement their transitions. A state machine with a transition defined but not implemented is useful for reasoning (e.g., to enumerate available transitions to generate navigation menu) but it is not possible to invoke such a transition (because it would fail). This behavior allows us to define a complete model, use it in the (partially) working application, but because of the well-separated transition definition, we can implement the application one transition at a time. The separation also allows developers to work concurrently.

Watching the number of unimplemented transitions also gives us a good estimate of how much of the project remains. Unfortunately, each transition takes a different amount of

time to implement because with such a transition we need to implement tools (services) it uses and user interface as well; therefore, this metric is only partially useful for time estimates but useful nonetheless.

D. CHANGES DURING THE IMPLEMENTATION

A few times during the later stages of the development, we had to rearrange already implemented transitions while adding a new state because the workflow got more complicated than expected. Most of these changes did not require any changes in the source code because the transition preconditions and postconditions remained unchanged, and thus, the only required change in such cases was to update the original state diagram which is the only place where transitions are defined and which serves as a formal model, source code, and documentation.

More substantial changes in the transitions, e.g., where preconditions or postconditions change, required changes in the transitions implementations to preserve consistency between the definition and implementation (see Sec. VI-H). However, such changes were limited to the respective methods implementing the affected transitions, and there is no generated code which could interfere with the manual changes.

Thanks to the direct use of the state diagram in run-time and the limited scope of each transition implementation, we successfully avoided the duplicate effort when updating the business logic.

X. FUTURE WORK

A. INTERNET OF THINGS AND MICROSERVICES

When connecting various devices together and building the Internet of Things, we often want to implement a web interface or API to control such devices. Smalldb framework has been designed with this use case in mind. By replacing the data access object with an interface to control the device, we can use Smalldb state machine to represent the device with its state and behavior. In this case, we will read the state of the state machine directly from the device and invoked transitions will be forwarded as control commands to the device. Such a device then can be used as any other entity in the application. For example, an automatically generated administration interface and REST API will manage such devices in the same way as rows in the database without any modification or special configuration. This can be done without any changes to the framework itself because both the transitions implementation and the repository are components provided by an application programmer.

Similarly, we can migrate from the simple data access object to microservices, CQRS [14], [15], and Event Sourcing [16]. Because of the separation of transition implementation (the ''command'') and repository (the ''query'') are already separate in the Smalldb architecture, it should be simple to implement the CQRS pattern and send events via a message broker instead of updating an SQL database.

The purpose of Smalldb state machines is to provide a unified API and describe the behavior of the encapsulated entities, so that other components, services, or even humans can meaningfully interact with them and reason about their behavior.

B. FLUENT CALCULUS

Smalldb state machines use names to identify its states and the state function to map a representation from the repository to one of the state names. Such a mechanism is universal and should fit most practical scenarios; however, in its generality, it does not provide us with any semantic information about the state. If we wish to reason about the states and properties of the system in such states, it would be useful to extend the concept of state function with a more advanced formalism; e.g., Situation Calculus [20] and Fluent Calculus [21], or Event Calculus [22].

Fluent Calculus [21] represents states using fluents (predicates). Each state is a composition (a set) of such fluents. Depending on the valuations of the fluents, we can decide in which state we are. Therefore, we could define fluents to describe various aspects of the representation from the repository, and then use such fluents to construct the state function for the Smalldb framework. Such an approach would enable the use of Fluent Calculus to reason about the states and the identification of operations needed to transition between two given states. Since Fluent Calculus is used in planning [23] (robotics; including temporal planning [24]), and automated web service orchestration [25], we could gain access to many advanced formal tools while developing web applications.

C. TOWARDS FORMAL PROOFS AND AUTOMATED PLANNING IN WEB APPLICATIONS

At the highest level, we should be able to verify the correctness of the model in the form of state diagrams or business process diagrams [11] using simulations and various model checking techniques verifying networks of interacting finite automata [19], where some of the automata represent the users and others represent the application entities.

While the Smalldb framework provides us with the required infrastructure connecting the high-level model to low-level code (transition implementation), we still need to provide a formal bridge between these layers [26]. For example, we may utilize the earlier discussed Fluent Calculus (or a similar formal tool) to enhance the basic universal infrastructure with a powerful formal tool, so that we can model and reason about various aspects of the underlying implementation.

The model defines in which state the state machine is at any given moment, the properties of individual transitions can be specified using differences between such states. Using the Fluent Calculus, we merely identify which fluents must change during the transition. Thus, we have a set of fluents as preconditions of the transition and another set of fluents as postconditions. Depending on available knowledge and the used calculus, we may even infer a list of actions which the transition implementation should perform during the transition, and potentially utilize an automated planning method.

Finally, we need to implement the transition. Because of the use of Fluent Calculus, we can expect to have a much detailed specification of the transition – not only names of the surrounding states but also some basic predicates about the states. All we need to do is to infer the executable code or verify the existing implementation. In general, the verification task is impossible (e.g., because of the halting problem); however, the transition implementation in a practical application should be very simple and static analysis may provide useful results in the most cases.

The role of Smalldb framework here is to provide the chain of trust from the implementation to the model and to enable the use of formal tools along the way.

XI. CONCLUSION

The Smalldb architecture presented in this paper combines the number of well-established patterns into a single coherent unit so that we get the best of each pattern. The multi-tier architecture provides the layered approach to reduce vertical complexity. The CQS pattern improves horizontal scaling by separating concerns and responsibilities of the repository and the transitions implementations. The architecture respects REST principles and provides a RESTful API to interact with other applications. The data access object or ORM provide effective low-level access to the database. And finally, the formal model provided by the Smalldb state machines describes the overall behavior of each entity and thus the whole application, so that we can reason about it and prove its correctness in regards to the implemented business processes.

The rigid connection between the formal model and implementation ensures that both the model and documentation always stays up-to-date. Therefore, the maintenance and further development of the application becomes much easier. As demonstrated in the case study[3], the practical integration of the formal model improves the whole development process, because the model provides a precise overview of the application as well as the well-defined scope of individual transitions which provides developers with a better specification of their tasks.

The proper encapsulation of the Smalldb state machine enables us to enforce access control from a single place, and to build a universal API for each entity in the application. Moreover, the access control can be included in the formal model so that we can reason about who can do what and when.

The visual nature of the state machines (and business processes) provides a common language between customers, architects, and developers – the easier communication results in the better specification and thus it leads to the cheaper development process.

The Smalldb architecture is designed for better developer experience and provides tools to improve the development process. It enables effective and practical use of formal models without the undesired overhead and, hopefully, it is a further step towards full formalization and provability of web applications.

[3]The Smalldb framework source code and documentation available from https://smalldb.org/.

REFERENCES

[1] G. E. Krasner and S. T. Pope, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, Aug. 1988.
[2] B. Meyer, *English Object-Oriented Software Construction*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 1997.
[3] R. C. Martin. *Principles of Ood*. Accessed: Aug. 29. 2016. [Online]. Available: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.
[4] M. Fowler. (2015). *Yagni*. [Online]. Available: https://martinfowler.com/bliki/Yagni.html
[5] J. Kufner and R. Mařík, "State machine abstraction layer," in *Information and Communication Technology* (Lecture Notes in Computer Science), vol. 8407, M. M. S. Linawati, E. J. Neuhold, A. M. Tjoa, and I. You, Eds. Berlin, Germany: Springer, 2014, pp. 213–227. 10.1007/978-3-642-55032-4_21.
[6] R. Fielding, *Architectural Styles and the Design of Network-Based Software Architectures*. Irvine, CA, USA: Univ. of California, 2000.
[7] G. J. Holzmann, "Points of truth," *IEEE Softw.*, vol. 32, no. 4, pp. 18–21, Jul./Aug. 2015.
[8] M. Fowler, *Patterns of Enterprise Application Architecture* (A Martin Fowler signature). Reading, MA, USA: Addison-Wesley, 2003. [Online]. Available: https://books.google.cz/books?id=FyWZt5DdvFkC
[9] E. Gamma, *Design patterns: Elements Reusable Object-Oriented Software*. London, U.K.: Pearson, 1995.
[10] M. Hearst, "Design recommendations for hierarchical faceted search interfaces," in *Proc. ACM SIGIR Workshop Faceted Search*, Seattle, WA, USA, 2006, pp. 1–5.
[11] J. Kufner and R. Mařík, "From a BPMN black box to a smalldb state machine," *IEEE Access*, vol. 7, pp. 56276–56296, 2019.
[12] (2019). *Doctrine ORM*. [Online]. Available: https://www.doctrine-project.org/projects/orm.html
[13] D. R. Prasanna, *Dependency Injection*. Shelter Island, NY, USA: Manning Publications, 2009.
[14] M. Fowler. (2005). *Command Query Separation*. [Online]. Available: https://martinfowler.com/bliki/CommandQuerySeparation.html
[15] G. Young. (2010). *CQRS Documents by Greg Young*. [Online]. Available: http://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
[16] M. Fowler. (2005). *Event Sourcing*. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html
[17] S. W. Ambler. (2005). *The Design of a Robust Persistence Layer for Relational Databases*. [Online]. Available: http://www.ambysoft.com/downloads/persistenceLayer.pdf
[18] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell Syst. Tech. J.*, vol. 34, no. 5, pp. 1045–1079, Sep. 1995.
[19] G. Behrmann, A. David, and G. Kim Larsen, "A tutorial on uppaal," in *Formal Methods for the Design of Real-Time Systems* (Lecture Notes in Computer Science), vol. 3185, M. Bernardo and F. Corradini, Eds. Berlin, Germany: Springer-Verlag, Sep. 2004, pp. 200–236.
[20] H. Levesque, F. Pirri, and R. Reiter, "Foundations for the situation calculus," *Comput. Inf. Sci.*, vol. 3, no. 18, pp. 1–21, 1998.
[21] M. Thielscher, "Introduction to the fluent calculus," *Electron. News J. Reasoning About Actions Change*, vol. 3, pp. 12–20, Oct. 1998.
[22] M. Shanahan, "The event calculus explained," in *Artificial Intelligence Today*. Berlin, Germany: Springer, 1999, pp. 409–430.
[23] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. London, U.K.: Pearson, 2016.
[24] A. Bit-Monnot, "Temporal and hierarchical models for planning and acting in robotics," Ph.D. dissertation, Laboratoire d'Analyse et d'Architecture des Systèmes, Inst. Nat. Polytechnique de Toulouse, Université de Toulouse, Toulouse, France, 2016. http://oatao.univ-toulouse.fr/17704/
[25] P. N. Lumpoon, "Toward a framework for automated service composition and execution: E-tourism applications," Ph.D. dissertation, Laboratoire d'Informatique de Grenoble et de Ecole Doctorale Mathématiques, Sciences et technologies de l'information, Informatique (MSTII), 2015. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01164389
[26] Z. Diskin and T. Maibaum, "Category theory and model-driven engineering: From formal semantics to design patterns and beyond," in *Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice*. Oakville, ON, Canada: Apple Academic Press, 2014, p. 173.

JOSEF KUFNER received the M.S. degree in software engineering from Czech Technical University in Prague, Czech Republic, in 2013, where he is currently pursuing the Ph.D. degree in artificial intelligence. Over the last decade, he has worked on various projects in the field of web technologies. This experience has led him to focus on finding a better way to compose web applications in both his bachelor and diploma thesis. He is currently continuing these efforts during his Ph.D. studies. One of the practical results of his research is the Smalldb framework, which combines a formalism of finite automata with an SQL database and traditional source code into a model layer of a web application.

RADEK MAŘÍK received the M.S. and Ph.D. degrees in technical cybernetics from Czech Technical University in Prague, Czech Republic, in 1984 and 1992, respectively. For 15 years, he has worked mainly in the field of computer vision with Czech Technical University in Prague and the University of Surrey, U.K. Since 1997, he has been with Protys s.r.o., Czech Republic, working on research contracts with Rockwell Automation in the field of software diagnostics with a focus on metaprogramming and automated design of software tests. Since 2007, he has been working on technology trend identification, knowledge-based software comprehension, and semantically based product interoperability on research projects at CA Technologies. He joined the Czech Technical University Team, in 2011. He has coauthored approximately 80 articles and has filed two patents in complex networks analysis and machine learning in signal processing.