HAL
open science

State Machine Abstraction Layer

Josef Kufner, Radek Mařík

▶ **To cite this version:**

Josef Kufner, Radek Mařík. State Machine Abstraction Layer. 2nd Information and Communication Technology - EurAsia Conference (ICT-EurAsia), Apr 2014, Bali, Indonesia. pp.213-227, 10.1007/978-3-642-55032-4_21 . hal-01397199

HAL Id: hal-01397199

https://inria.hal.science/hal-01397199

Submitted on 15 Nov 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

INQUISIENT 2001
ServiceNow v. InQuisient
IPR2023-01071

State Machine Abstraction Layer

Josef Kufner and Radek Mařík

Department of Cybernetics, Faculty of Electrical Engineering,
Czech Technical University in Prague, Czech Republic
kufnejos@fel.cvut.cz, marikr@k333.felk.cvut.cz

Abstract Smalldb uses a non-deterministic parametric finite automaton combined with Kripke structures to describe lifetime of an entity, usually stored in a traditional SQL database. It allows to formally prove some interesting properties of resulting application, like access control of users, and provides primary source of metadata for various parts of the application, for example automatically generated user interface and documentation.

1 Introduction

The most common task for a web application is to present some entities to an user, and sometimes the user is allowed to modify these entities or to create a new one. Algorithms behind these actions are usually very simple, typically implemented using few SQL queries. The tricky part of web development is keeping track of the behavior and lifetime of all entities in application. As number and complexity of the entities are growing, it is getting harder for a programmer to orientate in the application, and situation is even worse when it comes to testing.

Smalldb brings a bit forgotten art of state machines into the web development, unifying specifications of all entities in an application, creating a single source of all important metadata about many aspects of each entity, and allowing to build formal proofs of application behavior.

The basic idea of Smalldb is to describe lifetime of each entity using state machine, and map all significant user actions to state transitions. To make the best of this approach, the state machine definition is extended with additional metadata, which are not essential for the state machine itself, but can be used by user interface, documentation generator, or any other part of the application related to the given entity.

Smalldb operates at two levels of abstraction within the application. At the lower level it handles database access, it acts as the model in MVC pattern. At higher level of abstraction it can describe API, URIs and behavior of large parts of the application, however, it does not directly implement these parts.

In the next two sections an example of typical entity in web application is presented. In section 4 Smalldb state machine is formally defined. And section 5 describes relation between state machine instances and underlaying database. A basic implementation with some interesting implications is roughly described in

section 6. Section 7 introduces Smalldb as a primary metadata source. Remaining sections are dedicated to application correctness.

2 REST resource as a state machine

Let's start with simple example of generic resource (entity) in RESTful application [1]. RESTful applications typically use HTTP API to manipulate resources. Since REST does not specify a structure of a resource nor exact form of the API (simply because it is out of REST's scope), it is impossible to use this API without additional understanding of application behind this API. However, HTTP defines only a limited set of usable methods, so a general example can be provided.

Figure 1 presents a generic state machine equivalent to a REST resource. The resource is created by HTTP POST request on a collection, where the resource is to be stored. Then it can be modified using HTTP PUT (or similar HTTP methods), and finally it can be removed using HTTP DELETE method.

Without further investigation of the resource an influence of the "modify" transition cannot be determined, but we can safely assume the resource is more complex than Figure 1 presents, otherwise the "modify" transition would make no sense.

Transitions from the initial state and to the final state represents creation and destruction of the resource and the machine itself. These two states are denoted as a separate features, but they both represent the same thing – resource does not exist. This semantics is one of the key ideas behind Smalldb.

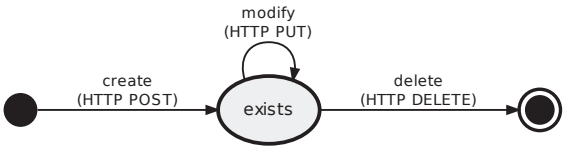

Figure 1. REST resource state machine

3 Real-world example

When building real web application, situation is rarely as simple as example in previous section. Typically there is more actions to perform on an entity, and the entity passes through a few states during its lifetime.

A very common application on the Web is a blog. Typical blog is based on publication of posts. Each post is edited for some time after its creation, and then it is published. Some posts are deleted, but they can also be undeleted (at least in this example).

A state machine representing lifetime of the post is in Figure 2. As we can see, there are three states and a few transitions between them. Note that there is no final state in this state machine. That is because the blog post is never completely destroyed.

There is one interesting feature in this state machine – the undelete action. In both HTTP specification and REST there is nothing like it. It is possible to implement it using an additional attribute of the blog post, but it does not fit well into RESTful API, for example there is no counterpart to HTTP DELETE method. Similar troubles occur when controlling nontrivial long-running asynchronous processes, since it is unnatural to express events and commands in REST.

There is also one big problem with both this and previous examples. If these state machines are interpreted as usual finite automata, the edit action has no effect. Invoking the edit action makes no difference, because it starts and ends in the same state. To justify this behavior, the state machine must use a concept very similar to Kripke structures. Each state represents a possibly infinite group of sub-states, which have common behavior described by the encapsulating state. Therefore, the edit transition is in fact a transition between different sub-states within the same state, i.e. sub-states belong to the same equivalency class. Omitting these sub-states from the state diagram is very practical since it allows easy comprehension. The sub-states are implemented using "properties" of the state machine instance, for example title, author and text of the blog post (this concept will be described in Section 4.4).

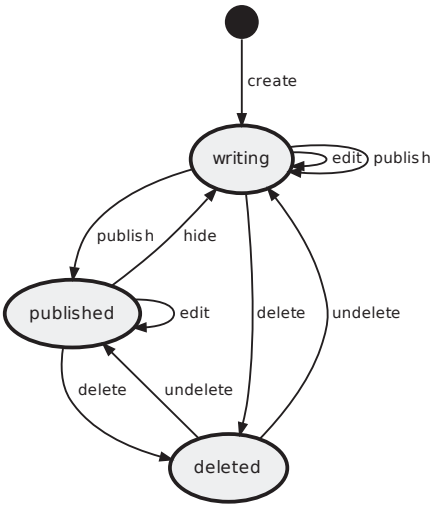

Figure 2. State diagram of blog post

4 State machine

As previous examples showed, it is necessary to modify and extend definition of finite automaton [2], to make any use of it. In this section a Smalldb state machine is formally defined and its features are explained. The definition is designed to follow an actual implementation as close as possible, so it can be used to formally infer properties of final applications.

4.1 Smalldb state machine definition

Smalldb state machine is modified non-deterministic parametric finite automaton, defined as a tuple (Q, P, s, P_0, Σ, Λ, M, α, δ), where:

- Q is finite set of states.
- P is set of named properties. P^* is (possibly infinite) set of all possible values of P. P_t is state of these properties in time t. $P_t \in P^*$.
- s is state function $s(P_t) \mapsto q$, where $q \in Q$, $P_t \in P^*$.
- P_0 is set of initial values of properties P, $P_0 \in P^*$.
- Σ is set of parametrized input events.
- Λ is set of parametrized output events (optional).
- M is finite set of methods: $m(P_t, e_{in}) \mapsto (P_{t+1}, e_{out})$, where $P_t, P_{t+1} \in P^*$, $m \in M$, $e_{in} \in \Sigma$, $e_{out} \in \Lambda$.
- α is assertion function: $\alpha(q_t, m) \mapsto Q_{t+1}$, where $q_t \in Q$, $Q_{t+1} \subset Q$, $e_{in} \in \Sigma$.

$$\forall m \in M : s(P_{t+1}) \in \alpha(s(P_t), m) \Leftrightarrow (\exists e_{in} : m(P_t, e_{in}) \mapsto (P_{t+1}, e_{out}))$$

- δ is transition function: $\delta(q_t, e_{in}, u) \mapsto m$, where $q_t \in Q$, $e_{in} \in \Sigma$, $m \in M$, and u represents current user's permissions and/or other session-related attributes.

4.2 Explanation of non-determinism

Non-determinism in the state machine has one specific purpose. It expresses possibility of a failure and uncertainty of the result of invoked action.

For example when the blog post (see Section 3) is undeleted, it is not known in advance in which state the blog post will end, because if user has no permission to publish, the result state will be "writing", even if the blog post was already published.

Similar situations occur when invoked action can fail. For example when the blog post cannot be published, because requested URI is already used by another post, or if some external material must be downloaded during publication and remote server is inaccessible.

In all these cases, a requested action is invoked, but which transition of state machine is used, is determined by result of invoked action.

4.3 Simplified deterministic definition

Because the complete definition described in section 4.1 is a bit too complex, here is a simplified deterministic definition with most of unimportant features thrown away. These features are present in the implementation, but they are not important for a basic understanding. This definition may also be useful for some formal proofs, where these two definition can be considered equivalent, if thrown away features are not significant for the proof.
Please keep in mind, that the rest of this paper always refers to the full definition in the section 4.1.

The simplified definition is: Smalldb state machine is defined as a tuple $(Q, q_0, \Sigma, \Lambda, \delta', m')$, where:

- Q is finite set of states.
- q_0 is starting state, $q_0 \in Q$.
- Σ is set of input events.
- Λ is set of output events (optional).
- δ' is transition function: $\delta'(q_t, e_{in}, w) \mapsto q_{t+1}$, where $q_t, q_{t+1} \in Q$, $e_{in} \in \Sigma$, and w is unpredictable influence of external entities.
- m' is output function: $m'(q_t, e_{in}, w) \mapsto e_{out}$, where $e_{in} \in \Sigma$, $e_{out} \in \Lambda$, $q_t \in Q$, and w is the same external influence as in δ'.

This is basically Mealy (or Moore[1]) machine [3,4], only difference is in introducing additional constraint w to handle possibility of failure. However, the w is not known in advance when transition is triggered (see Section 4.2).

Main simplification is made by chaining transition function δ and assertion function α into one transition function δ':

$$\forall q_t \forall e_{in} \forall w \ : \ \big(\delta'(q_t, e_{in}, w) \ = \ Q_{t+1}\big) \ \Leftrightarrow \ \big(\alpha(q_t, \delta(q_t, e_{in}, w)) \ = \ Q_{t+1}\big)$$

This simplification assumes, that the implementation of the transitions is flawless, which is way too optimistic for real applications.

4.4 Properties and state function

As came out in the blog post example (see Section 3), finite state automaton is not powerful enough to store all arbitrary data of an entity. To overcome this limitation, Smalldb state machine has properties. Each property is identified by name, and rest is up to the application. Properties can be implemented as a simple key–value store, columns in SQL table, member variables in OOP class, or anything like that.

Since properties are not explicitly limited in size, they can store very big, theoretically infinite, amount of data, data of high precision, or very complex

[1] Slight differences between Mealy and Moore machines are not important here, and e_{in} may or may not be used in m'.

structures. To handle these data effectively, the state function is used to determine state of the machine. The state function converts properties to single value, the state, which is easy to handle and understand.

Because applying the state function on different sets of properties can (and often will) result in the same state, the state represents entire equivalence class, rather than single value. This approach is very similar to Kripke structures [5].

The state function must be defined for every possible set of properties:

$$\forall P \in P^* : s(P) \in S$$

On the other side, an inverse function to s usually does not exist, so it is not possible to reconstruct properties from state. The only exception is a null state q_0, in which entity represented by state machine does not exist and properties are set to P_0, in short, $q_0 = s(P_0)$.

Typically the state function is very simple. In trivial case (like the first example in section 2) it only detects existence of a state machine. In more common cases (like the blog post example in section 3) it is equal to one of properties, or checks whether a property fits in a predefined range (for example, if date of publication is in future). Since the state function is key piece of the machine definition and it is used very often, it should be kept as simple and fast as possible.

The state is not explicitly stored and it is calculated every time it is requested. If both the state function and a property storage allow, the state may be cached to increase performance, but it is not possible to allow it in general. However, it is usually possible to store some precalculated data within properties to make state function calculations very fast.

4.5 Input events

The input events Σ can be understood as *actions* requested by user. The action is usually composed of method name $m \in M$ and its arguments. Input events are implementation specific and their whole purpose is to invoke one of expected transitions in a state machine.

4.6 Output events

The output events Λ are simply side effects of methods M, other than modifications of state machine's properties. These events usually include feedback to user and/or sending notification to an event bus interface, so other parts of application can be informed about change.

4.7 Methods

The methods M implement each transition of the state machine. They modify properties and perform all necessary tasks to complete the transition. These methods are ordinary machine-specific protected methods as known from object

oriented languages, invoked by universal implementation of the state machine. Since the methods cannot be invoked directly, access to them is controlled by state machine, and it is possible to implement advanced and universal access control mechanism to secure an entire application.

There is a few methods with special meaning in object oriented languages. If $\forall e_{in} \forall u : m_c = \delta\left(s(P_0), e_{in}, u\right)$, then m_c is known as constructor or factory method. If $\forall q \in Q : \alpha(q, m_d) = s(P_0)$, then m_d is known as destructor. However, in Smalldb both these methods are ordinary methods with no special meaning, and both can occur multiple times in single state machine.

4.8 Transitions and transition function

Main difference from classic non-deterministic finite automaton is in division of each transition into two steps. The transition function δ covers only the first step. The second step is performed by method $m \in M$, which was selected by the transition function δ. Point of this separation is to localize the source of non-determinism (see Section 4.2) and accurately describe a real implementation.

The complete transition process looks like this (explanation will follow):

$$\begin{aligned}(P_t, e_{in}) \xrightarrow{\delta(s(P_t), e_{in}, u)} (P_t, e_{in}, m) &\xrightarrow{m(P_t, e_{in})} P_{t+1} \\ &\xrightarrow{\alpha(s(P_t), m)} s(P_{t+1})\end{aligned}$$

Before a transition is invoked, only the properties P_t and the input event e_{in} are known. First, the transition function δ is evaluated, which results in the method m being identified. Then the m is invoked and the properties get updated. Finally, the assertion function is evaluated to check, whether the state machine ended in correct state.

The transition function δ also checks, if an user is authorized to invoke the requested transition. User's permissions are represented by u. This check can be used alone (without transition invocation) to determine, which parts of user interface should be presented to user.

4.9 Assertion function

A simple condition must be always valid:

$$s(m(P_t, e_{in})) \in \alpha(s(P_t), m)$$

Otherwise there is an error in the function m.

Purpose of the assertion function α is to describe expected behavior of m and validate its real behavior at run-time. Since m is piece of code written by humans, it is very likely to be wrong.

5 Space of state machines

Everything said so far was only about definition of a state machine. This definition is like a class in an object oriented language – it is useless until instances are created. In contrast with the class instances, the state machine instances are persistent. Definition is implemented in source code or written in configuration files, and properties of all state machine instances are stored in database.

But there is one more conceptual difference: The state machine instances are not created. All machines come to existence by defining a structure of a machine ID, which identifies machine instance in the space of all machines.

At the beginning, all machines are in null state q_0, which means "machine does not exist" (yes, it is slightly misleading). Since it is known, that properties of a machine in q_0 state are equal to P_0, there is no need to allocate storage for all these machines.

Machine ID is unique across entire application. There is no specification how such ID should look like, but pair of machine type and serial number is a good start. A string representation of the ID is URI, a world wide unique identifier. Conversion between string URI and application-specific ID should be simple and fast operation which does not require determining a state of given machine.

Once machine instance is identified, a transition can be invoked. Once machine enters state different than q_0, its properties are stored in database. This corresponds with calling a constructor in an object oriented language. When machine enters the q_0 state again, its properties are removed from database, like when destructor is called. But keep in mind that machine still exists, it only does not use any memory.

5.1 Smalldb and SQL database

An SQL database can be used to store machine properties. In that case, each row of the database table represents one state machine instance, and each column one property. The table name and primary key are used as the machine ID. Machines in q_0 state do not have their row in the table.

It is useful to implement the state function using an SQL statement, so it can be used as regular part of SQL query. That way it is easy and effective to obtain list of machines in given state.

Machine methods M typically call few SQL queries to perform state transitions. It is not very practical to implement the methods in SQL completely, since it is usually necessary to interact with other non-SQL components of the application.

5.2 RESTful HTTP API

URI as a string representation of the machine ID was chosen to introduce Smalldb HTTP API. This API respects the REST approach [1], and mapping to Smalldb state machine is very straightforward: HTTP GET request can be used

to read state machine status and properties, HTTP POST request to invoke a transition.

This may remind RPC[2] a little, where procedures on remote machine were invoked. Smalldb tries to pick the best of both REST and RPC, since these approaches are not in direct conflict. Entities are identified using URI, just like REST requires. Transitions are identified in RPC fashion, but structure of the machine behind this API is unified and data are retrieved in standard way, so close coupling does not happen, in contrast with RPC.

Question is, how to specify the transition to invoke. Probably the best approach is to append transition name to URI using query part (for example `http://example.org/post/123?action=edit`). This may not be as elegant as somebody could wish, but it is compatible with old plain HTML forms, because HTTP GET on such URI with the transition name can result in obtaining a form, which will be used to create HTTP POST request later. This makes it possible to use Smalldb without need to create a complex JavaScript frontend.

But if more interactive frontend is required, a HTTP header `Accept` can be used to specify other format than HTML page, and retrieve data in JSON or XML, just like any modern RESTful API offers. Also a HTTP GET on URI with a transition name specified can return transition definition, like HTTP OPTIONS does in REST API.

This approach was chosen pragmatically for the best compatibility with current stable (old) and widely available technologies.

6 Smalldb implementation

6.1 Prototype

A basic implementation is composed of two base abstract classes, `Backend` and `StateMachine`, and two helper classes, `Reference` and `Router`.

The `Backend` class manages `StateMachine` instances and takes care of stuff like shared database connection. It acts as both factory and container class. The `Backend` must be able to determine a type of requested machine from its ID, to prepare correct instance of `StateMachine` class, and delegate almost all requests to it. This way the `Backend` is responsible for entire state machine space without even touching it. Classes derived from the Backend class implement application specific way to access list of known state machines (descendants of `StateMachine` class).

The abstract `StateMachine` class and classes derived from it contain definition of the state machine and implementation of methods M (see Section 4.1). There is only one instance of `StateMachine` class per `Backend` and state machine type, which handles all state machines instances of given type. So the `StateMachine` instances are responsible for disjunctive subspaces of state machine space. When transition is to be invoked, machine ID and input event is passed to a `StateMachine::invokeTransition` method, which validates request

[2] RPC: Remote Procedure Call

using machine definition and executes appropriate protected method implementing the transition.

The Reference class is mostly only syntactic sugar to make application code prettier. It is created by `Backend`'s factory method, which takes state machine ID as an argument. It contains the ID and a reference to both the `Backend` and the corresponding `StateMachine` (obtained from the `Backend`). The `Reference` is used as proxy object to invoke state machine transitions and retrieve its state and properties. Its implementation is very specific to used language. Usage of `Reference` object is similar to Active Record pattern, however, the semantics is different.

Finally, the `Router` class is a little helper used to translate URI to state machine ID and back. Each application may require specific mapping of URIs, so the instance of `Router` class is injected into `Backend` during its initialization.

6.2 Metaprogramming

Since the `StateMachine` class is responsible for loading of a state machine configuration, it is possible to generate the configuration dynamically in `StateMachine` constructor. This allows to create more general state machines for similar entities using one `StateMachine` class initialized with different configuration.

It is also possible to determine state machine properties from structure of SQL tables, and load rest of the definition from the SQL database too. This way it is possible to define new state machines and entities without need to write a single line of code, using only an administration interface of the application.

6.3 Spontaneous transitions

When state function includes time or some third party data source, it may happen that state machine will change from one state to another without executing any code. Since this changes happen completely on their own and without any influence of Smalldb, it is not possible to perform any reaction when they happen.

There are two ways of dealing with this problem. The first way is to live with them and simply avoid any need of reaction. This approach can be useful in simple cases where an entity should be visible only after specified date. For example the blog post (see Section 3) can have "time of publication" property and state function defined like "if time of publication is in the future, post is in state writing, otherwise post is published".

Other way is to not include these variables into the state function and schedule transitions using cron or similar tool. This, however, usually require introduction of "enqueued" state. For example the blog post will have additional "is published" boolean property and there will be regular task executed every ten minutes, which will look for "enqueued" posts with "time of publication" in the past and will invoke their "publish" transition.

The spontaneous transitions can be useful tool, it is only necessary to be aware of their presence and handle them carefully. They also should be marked in state diagram in generated documentation.

7 State machine metadata

Role of the Smalldb state machine in an application is wider than it is typical for a model layer (as M in MVC), because Smalldb provides many useful metadata for the rest of the application. The state machine definition can be extended to cover the most of entity behavior, which allows Smalldb to be the primary and only source of metadata in the application.

Having this one central source makes the application simpler and more secure. Simpler because metadata are separated from application logic, so they do not have to be repeated everywhere, which also makes maintainability easier and development faster. More secure because metadata located at one place are easier to validate and manage.

Other important benefit of centralized metadata source is generated documentation. Since the metadata are used all over the application, it is practically guaranteed that they will be kept up to date, otherwise the application will get broken. And in addition, the metadata in the state machine definition are already collected and prepared for a further processing. All this makes it very valuable source for documentation generator.

For example, the Figures 1 and 2 used in examples (sections 2 and 3) were rendered automatically from a state machine definition in JSON using Graphviz [6] and a simple, 120 lines long, convertor script.

Additional use for these metadata is in generating user interface, determining which parts of it user can see and use, user input validation, access control, or API generating. And if metadata are stored in static configuration files or database, they can be modified using administration interface embedded in the application, which allows to easily alter many aspects of the application itself. Dynamically generated metadata then allows building of large and adaptive applications with very little effort.

8 Application correctness

A lot of research was done in model checking and finite automata, resulting in tools, like Uppaal [7], which allows to formally verify statements about given automaton. Since Smalldb is built on top of such automata, it is very convenient to use these tools to verify Smalldb state machines. And thanks to existence of formal definition of Smalldb state machine, it is possible to export state machine definition to these tools correctly.

8.1 Access control verification

Verification of basic properties, like state reachability[3], safety[4] and liveness[5], is nice to have in basic set of tests, however, these properties are not very useful on their own. Situation gets much more interesting, when user permissions are introduced.

User access is verified just before transition is invoked. Therefore, an user with limited access is allowed to use only subset of transitions in state diagram, and some states may become unreachable. If expected reachability of a state by given user is stated in the state machine definition, it is easy to use the earlier mentioned tools to verify it. And in the most cases, any allowed transition originating from unreachable state means security problem.

Similar situation is with liveness property, where unintentional dead ends, created by insufficient permissions, can be detected.

Because access control is enforced by general implementation of state machine (in abstract `StateMachine` class, see Section 6), which can be well tested and it is not modified often, probability of creating security issue is significantly reduced.

8.2 Optimizations vs. understandability

In the era of discrete logical circuits, a state reduction was very important task, because circuits were expensive and less states means less circuits.

In Smalldb, a state machine is used in very different fashion. The state machine is expected to express real behavior of a represented entity in a way, which can be understood and validated by non-technical user (customer). A connection between understandability of state diagram and automated generation of this diagram from the single source of truth (see Section 7) is important feature, since it eliminates an area, where errors and misunderstandings can occur – a gap between expectations and software specification.

From this point of view, any state diagram optimizations are undesirable.

8.3 Computational power of Smalldb machine

Classical finite state automaton is not Turing complete, because it has limited amount of memory, so it cannot be used, for example, to count sheeps before sleeping. But in Smalldb state machine this limitation was overcome by introducing properties and methods implemented in Turing-complete language (see Section 4.1), so the sheep counting can be done using one state, increment loop-back transition and sheep counter property.

Smalldb state machine is a hybrid of two worlds. On one side, there is nice non-deterministic finite automaton, which allows all the nice stuff described in

[3] State reachability: "Is there path to every state?"
[4] Safety property: "Something bad will never happen."
[5] Liveness property: "Machine will not get stuck."

this paper. On the other side, there are Turing-complete methods M, the barely controllable mighty beasts, which do the hard work. As long as these two parts are together, the computational power is the same as of the language used to implement the methods M.

By introducing properties and state function, the used automaton cannot be easily considered finite, since single state represents an equivalence class of property sets, which is not required to be finite. It is also possible to let methods M to modify state machine definition on the fly. And since both state function and transition function are also implemented using Turing-complete language, it is possible to define them in the way where the amount of the states is not finite at all. However, rest of this paper does not consider these possibilities and, for sake of clarity, expects reasonable definitions of all mentioned functions.

A practical example of self-modifying Smalldb state machine is a graphical editor of state machine definition which uses Smalldb to store modified configuration.

8.4 Troubles with methods M

Because methods M (see Section 4.7) are Turing complete, it is not possible to deduce their behavior automatically. This means it is not possible to predict, whether all transitions of the same name will be used by machine, and therefore some of the states considered reachable, when methods M were not took into account, may not be ever reached. This problem can be partially solved by careful testing and reviewing of the methods M.

Another problem is, when some of the methods are flawed and machine ends up in other state than transition allowed. This is detected by assertion function and it must be reported as a fatal error to a programmer.

Smalldb cannot solve these troubles completely, but it is designed to locate these kinds of errors as accurate as possible.

9 Workflow correctness

9.1 State machine cooperation

The workflow can be understood as cooperation of multiple entities with compatible goals. When these entities are specified as Smalldb state machines, it is relatively straightforward to involve tool like Uppaal (see Section 8), and let it calculate, what will happen, when these entities are put together.

Once state machine instances are required to cooperate, there is a danger that state machines will got stuck in deadlock. As Smalldb state machines represent entity lifetime, the cooperation troubles may mean there is something wrong with processes outside an application.

But the Smalldb state machine does not have to represent the entity within an application only. It also can be used to describe behavior of external entities, however, such entity should not be included in the application.

9.2 BPMN and BPEL

Entity lifetime is closely related to users' workflow and related processes. BPMN[6] and BPEL[7] were developed to describe them in some formal way. It should be possible to extract a formal model of each entity included in the process from BPMN and/or BPEL description, and convert them to Smalldb state machines. Then the state machine representing an application entity can be used as starting point of its implementation. And the other state machines, which represents humans and external applications, can be used to execute a simulation of complete process.

This approach should eliminate need for software specification when there is model of the entire process. Another benefit could be possibility of testing and formal proving of the application not against its specification, but rather against other entities in the process, removing the gap between what is expected and what is specified.

This area will require a lot of research and it is mostly out of the scope of Smalldb and this paper, however, it might be inspirational to put a bit wider context here.

10 Conclusion

Smalldb represents valuable source of metadata in an application, and allows to formally verify various aspects of the application, while maintaining practical usability and development effectivity.

From certain points of view it is similar to object oriented programming, where invoking of a transition is similar to method call in OOP, but with benefits of additional validation and better documentation of entity lifetime, which helps to manage complex and long-term behavior of the entities.

Smalldb also allows definition of simple RESTful HTTP API, which includes some aspects of RPC, to make the API more universal and easier to use. This API is also compatible with standard HTML forms, so it can be used on web sites without creating complex JavaScript clients.

Smalldb is meant as both as production-ready solution and as a building block for further research of software synthesis. However, there are areas left unexplored in integration of Smalldb with business process modeling (see section 9) and various aspects of verification.

[6] BPMN: Business Process Model and Notation
[7] BPEL: Business Process Execution Language

References

1. R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, 2000, aAI9980887.
2. A. Gill, *Introduction to the theory of finite-state machines*, ser. McGraw-Hill electronic sciences series. McGraw-Hill, 1962.
3. E. F. Moore, "Gedanken Experiments on Sequential Machines," in *Automata Studies*. Princeton U., 1956, pp. 129–153.
4. G. H. Mealy, "A Method for Synthesizing Sequential Circuits," *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.
5. K. Schneider, *Verification of Reactive Systems: Formal Methods and Algorithms*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
6. J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph – static and dynamic graph drawing tools," in *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 2003, pp. 127–148.
7. G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, ser. LNCS, M. Bernardo and F. Corradini, Eds., no. 3185. Springer–Verlag, September 2004, pp. 200–236.