Received February 26, 2019, accepted April 7, 2019, date of publication April 24, 2019, date of current version May 9, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2912567

From a BPMN Black Box to a Smalldb State Machine

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 by the Grant Agency of the Czech Technical University in Prague, under Grant SGS14/193/OHK3/3T/13.

ABSTRACT The state of a state machine is a path between two actions; however, it is the rest of the world who walks the path. The development of a web application, especially of an information system, starts with use cases, i.e., model scenarios of how users interact with the application and its entities. The goal of this paper is to turn the use cases into a useful specification and automatically convert them into a model layer of a web application, in our case using finite automata. Business Process Model and Notation (BPMN) provides a graphical syntax to capture the use cases, which is based on the theoretical background of Petri Nets. However, because BPMN does not capture the state of the modeled entities, it is impractical to use it as a specification of a persistent storage and model layers of the web application. To overcome this problem, we propose a new STS algorithm to infer a finite automaton that implements a chosen participant in a BPMN diagram that represents a given entity of the web application.

INDEX TERMS State machine, finite automata, state diagram, BPMN, business process model.

I. INTRODUCTION

A customer and a software architect are sitting at a table. The customer is describing what she wants and how it should fit into business processes in her company. The software architect is trying to figure out what she needs, making many notes and sketching various diagrams to capture the discussed use cases, key database features, and other vital details of a future web application. This is a usual beginning of a software development process.

During this design phase of software development, many diagrams are created. Most of them are informal sketches, but some become a part of the product specification, and a proper notation, e.g., UML or BPMN, is used. Unfortunately, these diagrams rarely survive the initial stages of the development, as they quickly become obsolete and forgotten.

Most of the diagrams are not formally useful because there is little to no connection between them and programming languages. Only few diagrams can be interpreted or compiled into executable code, for example, state charts into finite automata. Our goal is to introduce a connection between the use cases (scenarios) in the form of BPMN diagrams and a Smalldb state machine [1] that implements a participant of the BPMN process and provide a partial but automatic implementation of the modeled entities. We assume that the chosen participant is a machine, while the other participants are the users (the humans using the machine). To do so, we had to change our understanding of a "state" and figure out how to draw software systems in BPMN diagrams and then cut the BPMN diagram into pieces. Moreover, we had to realize that, while BPMN is based on the formalism of Petri Nets, it can also be seen as a group of interacting finite automata.

The BPMN diagrams contain plenty of data regarding the modeled software system, even when such a system is presented as a black box with which users interact. However, while programmers can retrieve the knowledge hidden across the diagrams intuitively, we lack a tool to do so automatically.

The result we present in this paper is the STS algorithm (the acronym is explained in Section IX-E), which inspects interactions of a chosen participant in a BPMN diagram (a trivial example of such an interaction is in Fig. 1) and then generates a state diagram of a Smalldb state machine that implements the modeled behavior of the participant. A programmer is then expected to provide a database schema and fill in some code implementing the state transitions. Effectively, the algorithm provides him with a skeleton or an outline with the blank spots to fill.

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

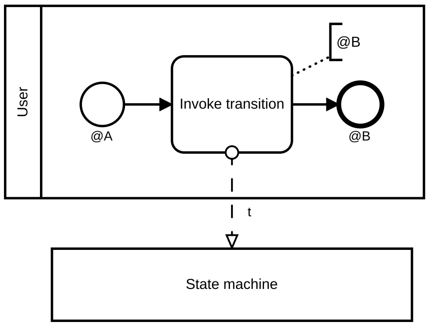


FIGURE 1. BPMN diagram (with state annotations) of a user invoking a state machine transition *t*.

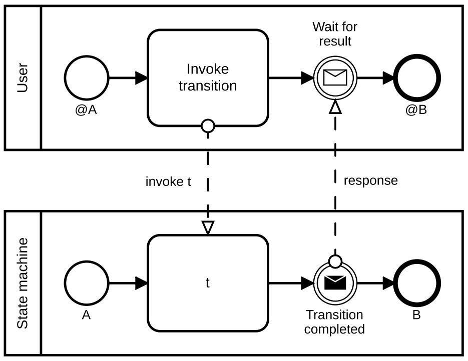


FIGURE 2. Transition invocation from Fig. 1 in detail.

The implementation of the STS algorithm provides us with not only a tool but also a better understanding of the BPMN itself. Modeling interactions with a software system using BPMN may be tricky. It is easy to draw a BPMN diagram that is too complicated or too vague. Therefore, before the algorithm itself, we will present our approach and interpretation of the BPMN notation [2].

Running the STS algorithm has shown a somewhat unexpected side effect. Various assertions and constraints built into the algorithm make it a valuable tool for performing a semantic check on the BPMN diagrams. The STS algorithm can detect conflicts in the BPMN diagrams as it inspects paths in the diagram. Additionally, it can detect gaps in the modeled workflow by producing a disconnected state machine (i.e., it has unreachable states). This helps us to draw semantically sound diagrams and reduces development time since the design mistakes are found early.

II. BPMN

A. THE NOTATION

BPMN 2.0, the business process modeling notation [2], defines a graphical syntax for describing a business process using an oriented graph; see Fig. 2. Each participant is represented by a "pool" ("User" and "State machine" in Fig. 2) with one or more "lanes" where his process is encapsulated. The process of the participant consists of "events" (circles), "activities" (rectangles with rounded corners), and "gateways" (squares; see the center of Fig. VI-B) connected by "sequence flows" (solid arrows). Communication and synchronization between participants (resp. their processes) are implemented using "message flows" (dashed arrows), which are the only edges crossing boundaries of the pools. There are a few additional features – "associations," "data objects," "messages," and "groups" – that are irrelevant to the STS algorithm, and it quietly ignores them. Finally, a "text annotation" is a comment node attached to a commented node or a text label within such a node. For a precise definition, please refer to the BPMN 2.0 specification.

As we will see later, because the algorithm is based on reachability inspection within the graph, it does not need to understand all features of the BPMN notation. It inspects only a few specific features in the diagram while interpreting the rest as a generic graph only to inspect the reachability. This approach makes the STS algorithm robust and lets users utilize all of the features of the BPMN, as the diagrams are meant to be used as documentation, not only as an input for the algorithm.

B. PROCESS AND PARTICIPANT

BPMN distinguishes "participants" and "processes." The "participant" is a real-world entity attending the overall business process. The "process" is a nested graph contained within a "pool." The "process" represents the behavior of the "participant." In more complicated scenarios, participants may be arranged hierarchically by splitting the pool into "lanes."

A simple example of a BPMN diagram with two interacting participants is in Fig. 2. Fig. 1 presents a similar diagram with one of the participants collapsed because we are not interested in its internals.

The "process" and "participant" terms are often interchanged and, in many cases, are incorrectly treated as synonyms. Many texts on the Web simply treat them both as a single feature. However, because there is a 1:1 relation between them in a BPMN diagram, this inaccuracy usually causes no harm; in fact, it can make the text easier to read. For the needs of the STS algorithm, we assume that the participant is part of its process, and in most cases, we do not differentiate between these two either.

C. WEB APPLICATION IN BPMN

Traditional web applications are based on HTML forms. Each form, when submitted by a user, invokes an HTTP (POST) request, which usually changes a state of a server-side resource, and the resulting HTTP response typically redirects the user to a page showing the new state of the resource. Modern web applications hide this request–response cycle from users, but the situation is the same from the server's perspective. As we showed earlier, such a resource can be modeled using a finite automaton; in our case, we represent the finite automaton explicitly using a Smalldb state machine [1]. Then, the state change of the resource is merely a transition of the state machine.

In a BPMN diagram presenting the basic use of an HTML form, see Fig. 2, there will be two participants, namely, the user utilizing a web browser and the server-side resource (the state machine). Commonly, we can draw the interaction of the user with the resource as a single message flow from the user to the resource, as presented in Fig. 1. This message flow then represents the entire cycle of an HTTP request and its response. Such a simplified representation is practical and usually sufficient.

In detail, (Fig. 2), we may draw the HTTP response separately, using a returning message flow from the resource to the user. Such representation is more accurate but too detailed and tiresome for common use. Therefore, the returning message flow is only required when the situation is ambiguous and usually can be omitted. The intermediate throw event (e.g., denoted as "Transition completed" in Fig. 2) may be omitted in any case; the returning message flow then originates from the preceding task node.

An important detail in Fig. 1 is that a collapsed pool represents the state machine. In this manner, the diagram can be drawn with no detailed knowledge of how the resource actually behaves and with no knowledge regarding state machine states or transitions. At this point, the state machine is a black box we want to identify. We know how users will use the state machine, but we do not know the state machine; however, as we will see, we have enough data to infer it. In fact, the primary goal of this paper is the synthesis of such a state machine from the available data.

D. BPMN, PETRI NETS, AND A GROUP OF FINITE AUTOMATA

A BPMN diagram forms a graph, which can be converted to a Petri Net [3], [4]. The conversion rewrites fragments of a BPMN diagram into places and transitions. However, this conversion is not lossless due to a conflict between the simplistic nature of the Petri Net and the verbosity of BPMN diagrams. Petri Nets have no concept of pools and lanes, nor do they distinguish between sequence flows and message flows. To preserve these, we would need to carry various metadata through the transformation, which is error-prone and unnecessarily complex [5]. However, such a transformation provides us with a valuable theoretical tool for interpreting BPMN diagrams and knowing which properties to expect from the model. The STS algorithm does not use the Petri Nets; it interprets BPMN diagrams directly, but it is based on the same theoretical background. The Petri Nets show us conditions under which a participant in a BPMN diagram can be transformed into a finite automaton.

If we assume single-threaded participants in a BPMN diagram, the resulting Petri Net has an intriguing property, i.e., each lane contains precisely one token (because of the assumption). A Petri Net of a single token can be trivially converted into a nondeterministic finite automaton (places become states). Therefore, we could convert a BPMN diagram into a Petri Net and then replace fragments of the Petri Net with finite automata, creating a network of interacting finite automata.

The assumption of the single-threaded processing may seem limiting, but it applies only to the participant we wish to implement using a finite automaton because the finite automaton can be seen as a special case of a Petri Net with a single token, and other participants may use a more powerful implementation. Moreover, we are modeling a lifecycle of an entity, not the execution of the program. From this point of view, it is better to keep things simple, and practical limitations are negligible.

The STS algorithm is based on reachability within the graph assuming single-threaded processing, which is unaffected by the transformation to a Petri Net; therefore, we can inspect reachability on a BPMN diagram directly and avoid the difficult conversion to a Petri Net altogether.

III. SMALLDB STATE MACHINE

Smalldb is a framework.[1] for implementing the model layer (i.e., M in MVC) of traditional server-side web applications. The basic idea of the framework is to use nondeterministic finite automata called Smalldb state machines [1] to describe lifecycles of entities within the application. The Smalldb state machine is an abstract construct that combines a formal state diagram with a transitions implementation (executable code) and persistent storage (typically an SQL database) to implement the desired model layer of the web application. In contrast to the usual use of finite automata, where the automata exists at run-time only, the Smalldb state machine is persistent and independent of application run-time. In comparison to ORM (object-relational mapping), the Smalldb state machine also encapsulates business logic, which is often implemented in higher layers of the application, and thus Smalldb is able to enforce access control at the model level.

For the purposes of this paper, we can interpret the Smalldb state machine solely as a persistent nondeterministic finite automaton. However, we need to understand two fundamental concepts of the Smalldb state machine, namely, the interpretation of nondeterminism and the persistent lifecycle of the state machine.

The nondeterminism of the Smalldb state machine represents various possible outcomes of an action (transition) invoked by a user. Theoretically, this approach is equivalent to the use of guards on deterministic finite automata, but practically, the information required to evaluate the guard is not available at the time of invocation of the transition because there may be unpredictable external influences or it may just be too complicated to model. A more accurate formal model would replace the nondeterminism with a microstep in which the state machine obtains additional information, and then the machine deterministically continues to one of the available next states.

The reason why the Smalldb state machine uses nondeterminism to capture nonmodeled external events, to avoid

[1] See https://smalldb.org/

complexity and to provide as much convenience as possible is because the target users are also customers who may discuss the state charts with software architects. Our experience shows that a nontechnical customer can understand such state charts with a short explanation quite well.

The other concept is how the state machine deals with persistence. The usual procedural use of finite automata is strictly limited to application run-time. Such an application instantiates the automaton and loads its state from the persistent storage. After the transitions are invoked and finished, the application stores the resulting state back in the persistent storage. The state machine ceases to exist when the application run-time terminates, and the state machine state is undefined until the application starts again.

The Smalldb state machine works the other way around. The Smalldb state machines exist as abstract constructs regardless of the application run-time. The application obtains only a reference object to access the otherwise-abstract state machine. When the application invokes a transition, it provides computational power (by executing the transition implementation) to update the persistent storage directly. Because the state is defined as a function of a representation in the persistent storage, the state of the state machine is known even when there is no application run-time active. This concept is the key feature that turns the Smalldb state machines into a model layer of the application.

For a detailed explanation of these and other concepts, please refer to our previous paper, "State Machine Abstraction Layer" [1].

IV. STATE IS A PATH

A state of a state machine is a path between two actions; however, it is the rest of the world who walks the path.

While this claim may sound philosophical, it reflects the basic idea of path searching through workflow graphs, which subgraphs are related to transition activities and activities of other participants performed during a given state.

Let us start with an example to shed some light on this claim, which is one of the cornerstones of the STS algorithm. Imagine a crossroad with traffic lights. A finite automaton and a timer control the traffic lights.[2] The timer provides input events to the finite automaton, and the finite automaton, depending on its state, turns green and red lights on and off in the prescribed order. In one state, there is a red light in one direction and a green light in another. Then, the automaton receives an event from the timer and switches to orange lights, and, with another timer event, to red and green in the other directions. As we can see, the automaton waits in one state and then performs an action to switch to another state.

Now let us take a look at the crossroad as a whole. While the automaton is waiting for the next timer event, cars are driving through the crossing, the timer is counting time, and pedestrians are walking around. There is much activity happening while the automaton is passively waiting in a state.

[2]Assuming an older model before computers were widely available.

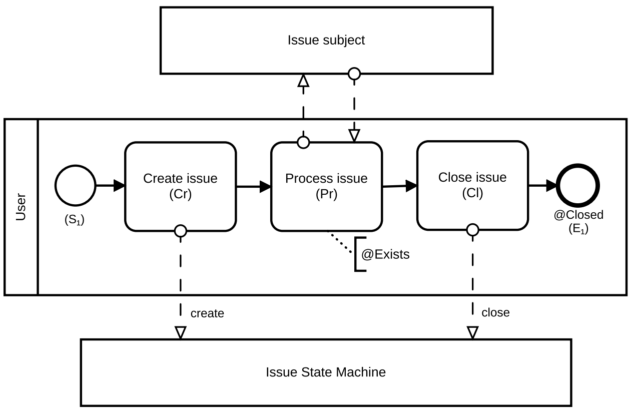


FIGURE 3. A simple issue tracking – invocation of transitions "create" and "close"; the state machine presented as a black box.

From the automaton's point of view, its transitions are times when something happens, and its states are seen as passive. For the rest of the world, the situation is opposite. The automaton's state provides data for external processes, and the processes then generate input events for the automaton. Therefore, the activities of the external entities form a path from one automaton's transition to another, connecting its output events to its input events.

V. ISOMORPHIC PROCESSES

The second cornerstone of the STS algorithm is the concept of isomorphic processes, which is about realizing that a user and the used tool must be synchronized to perform their task successfully: "When a user uses a tool, the tool is being used." In a BPMN diagram, the synchronization will appear as an isomorphism between the user's and the tool's processes (lanes) if we replace paths irrelevant to the tool with a simple sequence flow.

For example, Fig. 3 presents a user who gets an idea, creates a note in a To Do Application, and after implementing the idea, checks the note off. At this point, we know how the application is going to be used, but we do not know what the application will actually do. Thus far, the application is a black box to us. During a software development process, we would likely have similar diagrams after initial discussions with a customer.

The next step is to identify the black box so that a programmer can implement it. If we apply the concept of isomorphic processes to the diagram in Fig. 3, we can identify that the issue tracking application needs to create an issue and then close the created issue, as pictured in Fig. 4. Once we disregard the two nodes when the user implements the idea (replace the middle two nodes and the three connecting sequence flows with a single sequence flow), because they are irrelevant to the To Do Application, we can see that the user and the application processes form the isomorphic subgraphs.

The theoretical background on which this concept stands originates in Category Theory [6], particularly in the natural transformations between functors and commutative

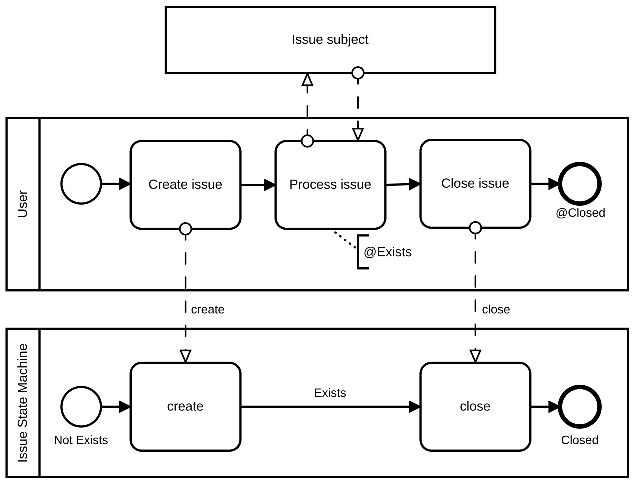


FIGURE 4. **Isomorphic processes in detail (compare with Fig. 3).**

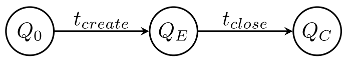


FIGURE 5. **The state machine inferred from Fig. 3.**

diagrams [7]. We may see the user's actions as functors changing the world from one state to another and the application code as functors changing the application state. The interaction between the user and the application then may be interpreted as a natural transformation between these two functors, forming a commutative diagram very similar to the BPMN diagrams. However, for the sake of clarity (and sanity), we skip the details.

The question is whether it is possible to apply the concept of isomorphic processes automatically, or do we need a human programmer?

VI. INTERPRETING BPMN DIAGRAMS

The real challenge when modeling interaction between users and an application in BPMN is to keep the diagrams comprehensive and straightforward while capturing all essential details. To do so, we identified the four key features occurring in the BPMN diagrams. The first is a simple interaction between a user and an application. The other two represent the control flow of the business process, differing in who decides what the next step is. The last feature is synchronization (or notifications) between the users via the application, where an interaction of one user with the application results in notification of another user. While this list may not be complete, we found it sufficient to cover all of the scenarios we have had to deal with thus far.

In this section, we present only the core ideas on which the STS algorithm stands and how the example BPMN diagrams should be transformed into Smalldb state machines. Without the STS algorithm, this transformation is done intuitively by developers, and for now, we shall rely on that intuition. The formal details are left to be refined later, in Section IX, along with the complete description of the algorithm.

A. SIMPLE TASK

The basic interaction with an application is a submission of a request (or a command). In a web application, this act closely follows HTTP communication; the user, via a web browser, sends an HTTP request to the application on a web server, and then the application replies with an HTTP response. We already introduced this scenario in Section II-C; now let us take a look at it from the STS algorithm's perspective.

In BPMN, we can model the request–response interaction as two message flows between the user and the application, or more precisely, between the user and an entity within the application (see Fig. 1). The first message flow, the "invoking" message flow originating in the "invoking node," represents the request. The second message flow, the "returning" message flow ending in the "receiving node," represents the response.

In simple cases such as this one, where the invoking node is also a receiving node, we may omit the returning message flow from the diagram because it is obvious that it should be there, but it remains a part of the model as an *implicit returning message flow*. However, the placement of the returning flow may significantly change the meaning of the diagram, as we will see later.

Because we are going to implement the entities of the application as Smalldb state machines, we can assume that each invoking message flow represents the invocation of a transition in the given state machine. The returning message flow then contains information about the new state of the entity. An implementation of the invoked transition may be as simple as a single SQL query, or it may include complex orchestration of remote services. From our point of view, it does not matter what is hidden behind the transition invocation. For now, it is just an invocation of an API with which the user communicates.

For example, Fig. 3 presents a situation in which the user utilizes an application entity, represented by the issue state machine, to perform two tasks, namely, to create an issue and then to close the issue. The sequence flows from the start event to the end event (the circles) determine the order of these two tasks. The message flows from the tasks to the state machine represent the two uses of the issue state machine. Only the invoking message flows need to be drawn because it is unambiguous that returning flows are antiparallel to the invoking message flows and return back to the same task.

Based on the BPMN diagram in Fig. 3, the STS algorithm can infer the "Issue state machine" as presented in Fig. 5. The state machine contains the two transitions, t_{create} and t_{close}, for the user's two tasks. Moreover, the state machine enforces the order of the transitions as specified by the sequence flows in the BPMN diagram. The states of the state machine do not have meaningful labels because there is no such information in the BPMN diagram (we will solve this later in this paper). Section XI will present the details of the STS algorithm execution for the example in Fig. 3 once we define the algorithm.

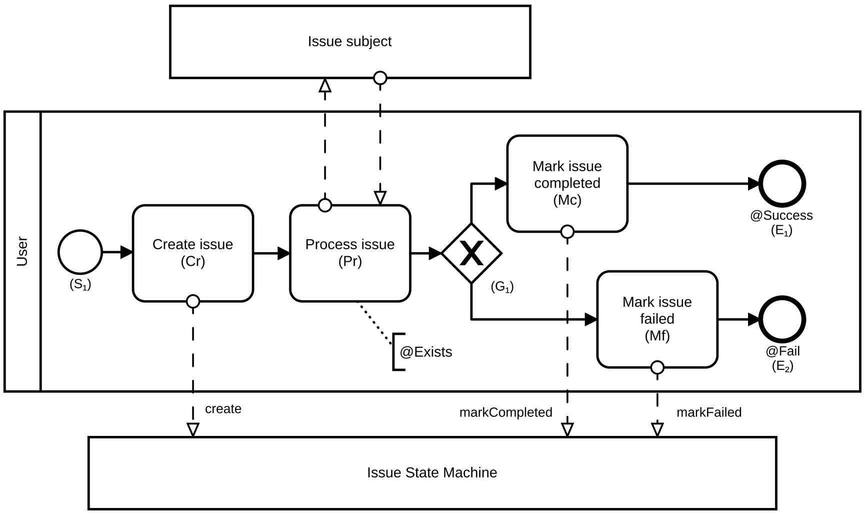
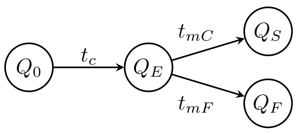


FIGURE 6. User decides scenario.



$$\begin{tikzcd} & & Q_S \\ Q_0 \arrow[r, "t_c"] & Q_E \arrow[ru, "t_{mC}"] \arrow[rd, "t_{mF}"'] & \\ & & Q_F \end{tikzcd}$$

FIGURE 7. The state machine inferred from Fig. 6.

B. THE USER DECIDES

What if a user has multiple options for which tasks to perform next in the business process? Such a scenario can be expressed in a BPMN diagram as a simple branching using an exclusive gateway. The valid order in which the user can perform the tasks is then determined by reachability between the tasks.

For example, Fig. 6 shows an extended version of the previous example. The user creates the issue as before, but then the user has two options as to how to resolve the issue, namely, to mark the issue as completed or as failed. Which of the options the user chooses is a result of the process issue task (in the middle).

A state machine inferred from this BPMN diagram will have three transitions, one for each of the invoking message flows from the user process to the issue state machine process. As in the previous example, there are no returning message flows explicitly drawn in the BPMN diagram, so the information regarding the new state is returned to the invoking tasks, and thus we can assume that the returning message flows return back to the origin of each invoking message flow.

The transitions of the inferred issue state machine are arranged so that the mark issue transitions follow after the create issue transition because the respective tasks in the BPMN diagram are reachable from the "create issue" task. A state diagram of the issue state machine is pictured in Fig. 7. The user invokes transition t_c followed by t_{mC} or t_{mF} depending on the desired result state Q_S (Success) or Q_F (Fail), respectively.

Note that the process issue task along with the preceding intermediate timer event, the issue subject process, and the exclusive gateway do not influence the inferred issue state machine.[3] However, this part of the process is essential for the user to decide between the success and the failure of the issue. To infer the issue state machine, we need only know that the "mark issue" tasks are reachable from the "create issue" task, but it is not important what is on the paths or how many participants are involved.[4] Section XII will present the details of the STS algorithm execution for the example in Fig. 6.

C. THE MACHINE DECIDEs

In the previous section, we looked into a case where a user had multiple options. However, the user is not the only entity capable of making decisions. What if the user only invokes an operation and then waits for what happens next? The operation may require complex calculations, involve remote services, or depend on data unavailable at the time of the invocation. One way or another, the decision is not up to the user.

As before, the scenario is still a simple branching, but additionally, it includes an information transfer from the machine to the invoking user. To express such branching in the BPMN diagram, we use an event-based gateway with explicit returning message flows from the state machine to the events following after the gateway. The returning message flows tell the user which branch of the BPMN diagram to use, i.e., what the next valid action is.

Because we are interpreting the BPMN in the scope of a user interacting with a web application for which individual interactions are mapped on rapid HTTP request–response cycles, we consider the transitions to be atomic operations,

[3] As long as the reachability property is preserved.

[4] As long as the path does not include anything within the inferred state machine. We will get to this detail later, in Section IX.

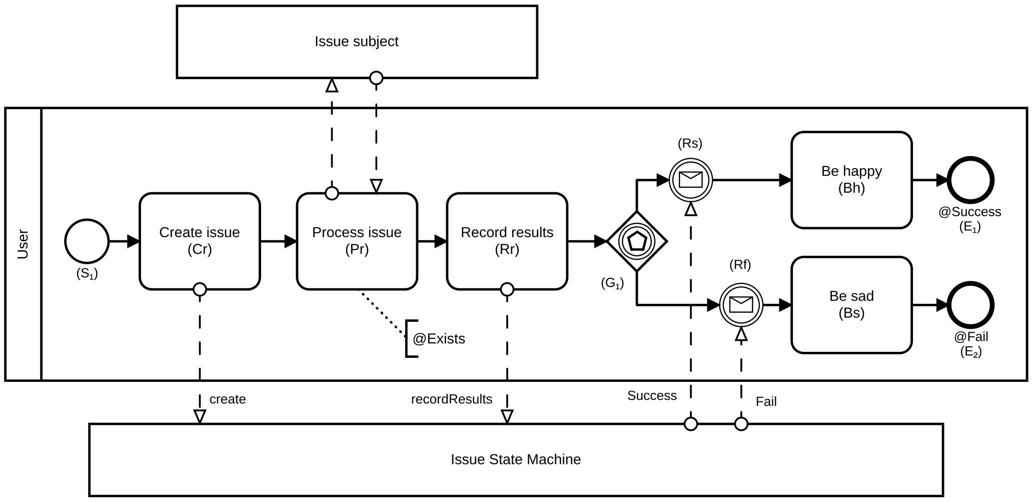


FIGURE 8. Machine decides scenario.

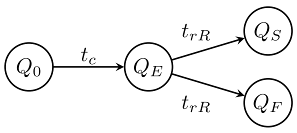


FIGURE 9. The state machine inferred from Fig. 8.

at least from the user's point of view. Therefore, no additional task or event nodes can be placed between an invoking and related receiving message flows in the BPMN diagram. This limitation makes the placement of the returning flows unambiguous.

For example (see Fig. 8), if we take the previous example and let the user record results from the process issue task, then the issue state machine will tell the user whether the issue is solved successfully or if it failed. Note the opposite direction of the last two message flow arrows in comparison to the previous example; the user invokes the record results transition and then receives one of the two possible answers rather than invoking one of the two possible transitions.

A state machine inferred from the example will have only two transitions; however, the latter transition will have two possible results – see Fig. 9. The user invokes transitions t_c followed by t_{rR}, but it is up to the state machine to decide whether the resulting state will be Q_S (Success) or Q_F (Fail). This makes the inferred state machine nondeterministic, as explained in Section III. Section XIII will present the details of the STS algorithm execution for this example.

D. SYNCHRONIZATION BETWEEN USERS VIA THE APPLICATION

Thus far, we have dealt with one user interacting with a state machine, but what if we have two users interacting with a single state machine? How to deal with such a scenario without opening the Pandora's box of parallel computing? To keep things simple, we shall deal with only a simple case

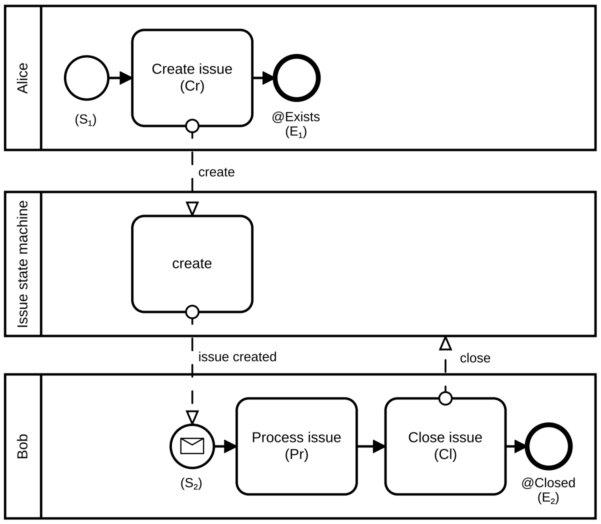


FIGURE 10. Synchronization between users.

where one user tells another user to continue. It is a common and useful scenario with a simple synchronization.

Let us change the first example (see Fig. 3) so that one user, Alice, creates an issue and another user, Bob, closes the issue. Fig. 10 presents this scenario in a BPMN diagram. There are two invoking message flows labeled "create" and "close," just as before, but there is also a third message flow that notifies Bob that Alice created the issue. Returning message flows are as simple as before; both implicitly return to the tasks where the respective invoking message flows originated (not pictured in the figure because they are implicit). The third message flow in the figure, labeled "issue created," is not a returning message flow or an invoking message flow. Such a message flow only propagates information about a new state of the state machine. Once the create transition finishes, the returning message flow will inform Alice about the result of the transition, i.e., the new state. Additionally, at the same moment, the third message flow will notify Bob so that he can react and deal with the situation.

The third message flow in our example and all message flows leaving a state machine process, in general, shall be referred to as potential returning message flows (where returning message flows are a subset of potential returning message flows). If a potential returning message flow has an invoking message flow assigned, then it is a returning message flow (it is not potential anymore). Thus, the returning message flows must always return to the same participant who invoked the transition (otherwise, it is only a potential returning message flow).

The state diagram of the state machine is precisely the same as in the first example (see Fig. 5) because the state machine does not care who invokes the transitions. However, if we would like to introduce access control to the state machine, the transitions might have different permissions assigned to reflect access rights of the respective invoking user.

The important thing to realize is the semantics of the returning message flows and potential returning message flows. While the invoking message flows represent a unidirectional command, the potentially returning message flows represent a notification, a propagation of information. A user invokes a transition using an invoking message flow, the returning message flow notifies the invoking user that the transition is complete (with the provided result), and the other potential returning message flows notify other participants that the transition has happened.

This distinction is not apparent from the BPMN diagrams, but it is essential for their interpretation. The notification about a new state represents equivalence of information; at the given moment, both the recipient and the sender of the message flow have the same information about the new state. In our example, it means that both Alice and Bob know the state of the Issue state machine once the "create" transition is completed. Such information may quickly become obsolete, however.

Section XIV will present the details of the STS algorithm execution for the example in Fig. 10.

VII. STATE LABELING

The BPMN diagrams describe business processes using tasks, gateways, and events connected by sequence flows. Therefore, the diagram can express when a participant is performing a certain task or is waiting for a certain event. What the diagram does not describe is in which state the participant is or what the participant knows about another participant's state. BPMN notation merely lacks the syntax to express such information.

When inferring a state machine from a BPMN diagram, the STS algorithm can number the states of the state machine or use some heuristic to guess reasonable labels based on nearby strings in the BPMN diagram. However, neither will produce useful results, and both programmers and users need proper labels to work with the state machine and refer to it while communicating with each other.

To provide custom state labels, we introduced annotations into the BPMN diagrams. Such an annotation is specifically formatted text placed in a node label or an attached comment node. To name a state, we use the state name prefixed by the "@" symbol – see Fig. 1, which illustrates the invocation of transition t leading from state A to state B with two possible annotations for state B.

The semantics of the annotation are as follows: "The latest information about the state of the state machine this participant received at this point is this state."

Note that the potential returning message flows propagate information about the state of the state machine. Therefore, the convention is to interpret the annotations after the message flows are received. Additionally, the information might have changed since it was initially received, but the participant will not know until another potential returning message flow arrives, so the state information defined by the annotation is valid on all nodes from the previous potential returning message flow to the next one or the next invoking message flow.

Because the scope of an annotation may span over multiple nodes or even a considerable part of the BPMN diagram, it is vital to avoid the specification of conflicting state information in the annotations. The STS algorithm detects such conflicts and reports them as errors because a single state cannot have more than one label.

Alternately, it is possible to specify the same state on multiple unconnected places in the BPMN diagram, or even in different diagrams. Such a situation will cause the detected states in the BPMN diagrams to be merged into one, effectively allowing various aspects of the behavior to be modeled in separate BPMN diagrams. Such shared state labels should be used only at start and end events; otherwise, the state machine may continue according to a different BPMN diagram than the user.

A Smalldb state machine uses the concept of a "Not Exists" state, which represents nonexistence of the modeled entity. This state usually occurs at the beginning and the end of the process. To avoid unnecessary clutter in BPMN diagrams, the STS algorithm provides an implicit state labeling; unless an annotation specifies otherwise, the start and end events define the inferred state to be the "Not Exists" state. The implicit state labeling is merely a syntactic sugar to ease the creation of the BPMN diagrams, and it can be easily omitted from the algorithm.

VIII. NOTATION AND OPERATORS

For easier manipulation with arrows and relations (binary and ternary) in the next sections, we define the following helper functions σ and τ to retrieve the domain and the range of a relation (sigma σ as the source node and tau τ as the target node of an arrow or a set of arrows); a helper function λ to retrieve the third item of the ternary relation (we use lambda λ as a label of otherwise binary relation); graph projection operators V and E to obtain nodes and edges of a graph, respectively; and a pathfinding operator $\multimap$ as follows:

- $\sigma(X) = \{x \mid \exists y : (x, y) \in X\}$
  $\sigma(X) = \{x \mid \exists y, z : (x, y, z) \in X\}$
- $\tau(X) = \{y \mid \exists x : (x, y) \in X\}$
  $\tau(X) = \{y \mid \exists x, z : (x, y, z) \in X\}$
- $\lambda(X) = \{z \mid \exists x, y : (x, y, z) \in X\}$
- Additionally, for a simple pair (x, y) or arrow $\overrightarrow{xy}$:
  $\sigma((x, y)) = x$, $\tau((x, y)) = y$
- For a ternary relation, a triplet (x, y, z):
  $\sigma((x, y, z)) = x$, $\tau((x, y, z)) = y$, $\lambda((x, y, z)) = z$
- For empty sets: $\sigma(\emptyset) = \emptyset$, $\tau(\emptyset) = \emptyset$, $\lambda(\emptyset) = \emptyset$
- $V(G') = V'$, $E(G') = E'$ for a graph $G' = (V', E')$.
- $a \twoheadrightarrow b$ is an oriented path (see [8]) through graph G' from $a \in V(G')$ to $b \in V(G')$; it is a subgraph (V', E') of G', as it includes both nodes $V(a \twoheadrightarrow b) \subseteq V(G')$ and arrows $E(a \twoheadrightarrow b) \subseteq E(G')$.

The STS algorithm is based on sets of pairs or triplets that represent various binary or ternary relations. We consider a function or a map as such a relation as well.

A binary relation represented by a set $F_2 = \{(x, y), \dots\}$ is always a function $y = F_2(x)$, and also a set of arrows $F_2 = \{(x \rightarrow y), \dots\}$. Additionally, for $f_2 \in F_2$, we can write the following:

$$f_2 = (x, y) = (x \rightarrow y) = (\sigma(f_2) \rightarrow \tau(f_2))$$

A ternary relation represented as a set $F_3 = \{(x, y, z), \dots\}$ is always a function $(y, z) = F_3(x)$, and also a set of labeled arrows $F_3 = \{(x \xrightarrow{z} y), \dots\}$. Additionally, for $f_3 \in F_3$, we can write the following:

$$f_3 = (x, y, z) = \left(x \xrightarrow{z} y\right) = \left(\sigma(f_3) \xrightarrow{\lambda(f_3)} \tau(f_3)\right)$$

IX. THE STS ALGORITHM

The STS algorithm generates a Smalldb state machine that implements the designated participant, the so-called *state machine participant*, in a provided BPMN diagram. A programmer draws use cases as to how users will use an application, and the algorithm generates the application to fit the use cases. Therefore, the input of the algorithm is a BPMN diagram with a chosen state machine participant, which the STS algorithm will implement using a state machine, and one or more user participants, which interact with the state machine participant.

The core idea behind the STS algorithm workflow is identification of transitions and inspection of the propagation of information regarding new states after the transitions. To do so, the algorithm locates state machine transitions within the BPMN diagram by inspecting reachability from places where users invoke state machine transitions to possible ends of each state machine transition, and then it inspects reachability between the transitions to detect states as they connect subsequent transitions. As we will see, the STS algorithm exhibits a certain symmetry in transition and state detection, but let us start from the beginning.

The algorithm generates the state machine in the following five stages:

1) Identification of invoking and receiving nodes
2) Transition detection
3) State detection
4) State labeling
5) State machine construction

First, the algorithm identifies invoking and potential receiving nodes by analyzing message flows to and from the state machine participant, i.e., the points where transitions start or end. Then, the algorithm inspects reachability from the invoking nodes to potential receiving nodes, forming a transition relation and identifying which of the potential receiving nodes are truly receiving nodes. The transition relation describes where a given transition is invoked in the BPMN diagram. The remaining nodes provide connections between individual transitions; therefore, the algorithm inspects reachability from the potential receiving nodes to invoking nodes, forming a state relation. These two relations are then put together to build a transition function (table) of the resulting state machine.

Because the STS algorithm is based mostly on inspecting reachability with constraints, it interprets only a few features of the BPMN notation. Namely, it distinguishes message flows and sequence flows, respects which participant nodes belong to, and interprets events as start/intermediate/end only. Everything else is just a path when inspecting the reachability. It is important to keep in mind that the BPMN diagrams are used on multiple occasions during software development, not only to synthesize the state machine.

However, the algorithm requires the states to be named because BPMN is oriented around tasks and actions, not the states. Therefore, custom annotations are added into the BPMN diagrams to provide human-friendly state labels. Theoretically, these annotations are not mandatory, but without them, the state names would be unpleasant to use later in the generated application. The secondary benefit of the annotations is in providing natural points through which multiple diagrams can be merged into a single state machine.

For a better understanding, the STS algorithm is explained using a simple example based on the "machine decides" situation from Section VI-C. The example input of the algorithm is presented in Fig. 11. The example covers two transition invocations. The first is from the node A, and the second is from the node C. In the case of the second transition, the user's next action (D_1 or D_2) depends on the result of the invoked transition (t_{C1} or t_{C2}).

After a mostly informal explanation of the entire algorithm in the following subsections, a compact formal summary will follow.

A. STAGE 1: INVOKING AND RECEIVING NODES

The first step is to identify invoking and potential receiving nodes, i.e., the sets I and R^+. An invoking node is a node where a participant invokes a transition of the state machine. A receiving node is where the invoking participant receives

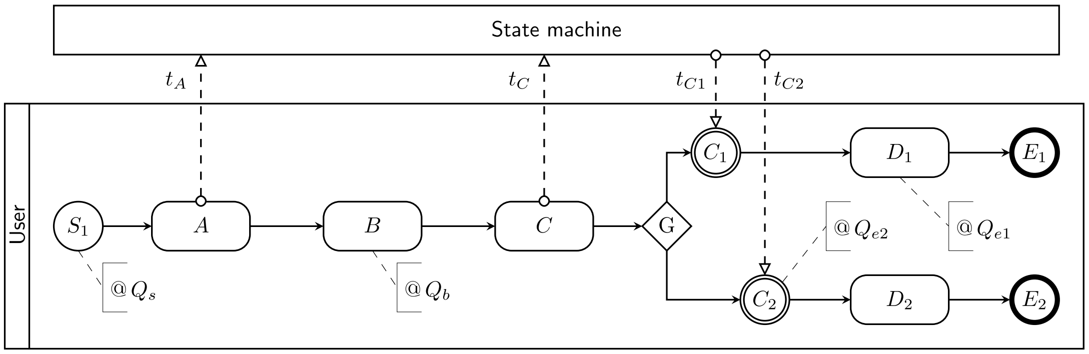


FIGURE 11. **Example – Source BPMN diagram.**

the result or confirmation of the invoked transition. A potential receiving node is where a situation suggests the node could be the receiving node, but further analysis (transition detection) is needed for a confirmation.

Both the invoking and the potential receiving nodes are identified by message flows to/from the state machine participant. Invoking nodes I are those with an outgoing message flow (to the state machine participant). Possibly receiving nodes R^+ are all nodes with incoming message flows outside of a state machine process or with an implicit returning message flow. The implicit returning message flow is an assumed message flow antiparallel to an invoking message flow when there is no other receiving node reachable from the given invoking node (as described in the next section). This means that some of the invoking nodes may also be potential receiving nodes.

The receiving nodes R are those nodes of R^+ that have an invoking node assigned (these will be identified later).

Because the STS algorithm will later need to know which transition is invoked by each invoking message flow, it collects labels of invoking message flows in a relation $L = \{(n_i, m)\}$, which assigns a method m (a name of a transition and also an input symbol of the state machine) to each invoking node n_i. In this paper, we assume at most one invoking message flow per node.

Example (Fig. 11): $I = \{A, C\}$ because of message flows t_A and t_C. $R^+ = \{A, C_1, C_2\}$ because of message flows t_{C1} and t_{C2}. Node A is considered a receiving node because it has an implicit returning message flow. Labels $L = \{(A, t_A), (C, t_C)\}$ are collected from the invoking message flow labels.

B. STAGE 2: TRANSITION DETECTION

1) TRANSITION RELATION

An invocation of a state machine transition starts from an invoking node, where a user invokes the transition, and ends in one of the receiving nodes, where the user receives one of the possible answers from the state machine.

To capture the transition invocations, we define a ternary transition relation $T = \{(n_i, N_r, m)\}$, which assigns a set of receiving nodes $N_r \subseteq R^+$ to an invoking node $n_i \in I$, along with a label $m \in M$ denoting which transition[5] is invoked. We can picture the transition relation T as a set of the following arrows:

$$\underset{\text{invoking}}{n_i} \xrightarrow[(T)]{m} \underset{\text{receiving}}{N_r}$$

An important secondary result of the transition detection consists of identification of receiving nodes $R \subseteq R^+$. The receiving nodes are such potential receiving nodes that are part of the transition relation T. The remaining potential receiving nodes $R^+ \setminus R$ may help with synchronization of the participants and with state propagation.

To build the transition relation, the STS algorithm needs to inspect reachability from the invoking nodes to the potential receiving nodes over sequence flows and nodes, which are not within the state machine participant and which are not event nodes. Invoking nodes and receiving nodes can occur only as endpoints. Therefore, if a path from an invoking node n_i to a receiving node n_r exists, then these nodes are connected by T, i.e., $(n_i, \{n_r, \dots\}, m) \in T$. Note the assumption of single-threaded processing, which allows us to identify the relation uniquely.

The method m, which labels the relation T, is determined by the label of an invoking message flow leaving the given invoking node n_i.

Luckily, the STS algorithm only needs to detect the existence of paths from n_i to N_r; it does not need to find any path in particular. This small detail means that the transition relation T can be built very quickly with linear time complexity.

Example – Fig. 12: The gray bold arrows represent the following transition relation:

$$T = \{(A, \{A\}, t_A), (C, \{C_1, C_2\}, t_C)\}$$

Note that t_A and t_C will become labels of transitions, the arrows, of the resulting state machine, although they are nodes in Fig. 12.

2) IMPLICIT TASKS AND RETURNING MESSAGE FLOWS

An invocation of a state machine transition is represented in the BPMN diagram by three elements: the invoking message

[5]In a Smalldb state machine, the transition is implemented by a method m.

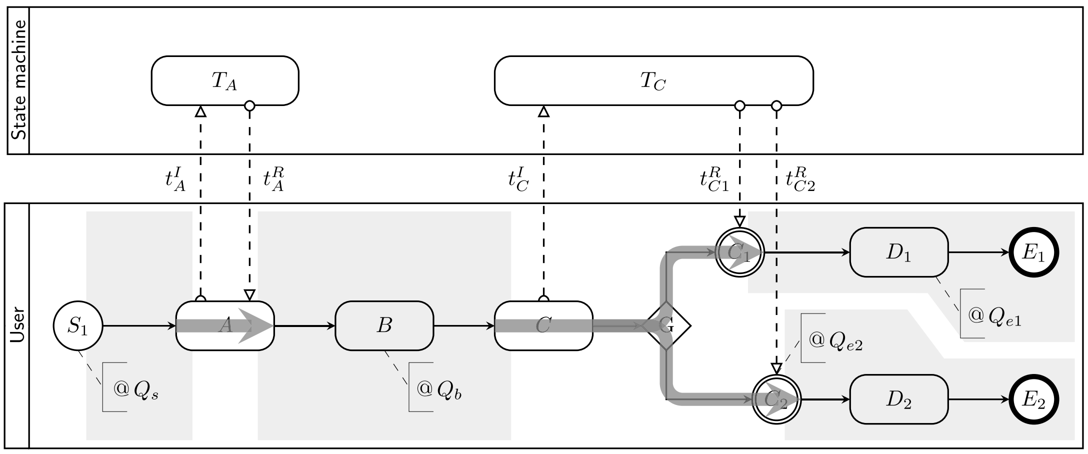


FIGURE 12. Example – Transition detection (grey bold arrows) and state detection (grey areas).

flow from the user's task to a state machine task, the state machine task representing the transition itself, and finally a returning message flow back to the user. Because of the process isomorphism (as explained in Section V), the state machine task and the returning flow may be omitted in simple cases[6] without losing any information from the diagram. As discussed in Section VI-A, such simplification is often used intuitively in practice and is also desirable because it makes diagrams easier to draw and comprehend.

While the STS algorithm does not need to reconstruct the implicit tasks and message flows, it is helpful for our example to do so anyway because it makes understanding the algorithm much easier.

The rule adding the implicit task node is quite simple: if the invoking message flow $t_A = \overrightarrow{A\,V_{SP}}$ from the user's task A ends on a border of the state machine participant V_{SP}, then a new task node T_A (the implicit task) is added to the state machine participant V_{SP}, and the original message flow t_A is replaced with a new message flow $t_A^I = \overrightarrow{A\,T_A}$.

The rule adding the implicit returning message flows is even simpler: if the invoking node is also a receiving node, then there should be a returning message flow to this node.

In the case of implicit task nodes and existing returning message flows, the reconstruction may occur during the transition detection. When an element $t = (n_i, N_r, m)$ of the transition relation T is found, the returning message flows from a task node to each node of N_r should exist. The returning message flows are already present but start from the participant V_{SP} rather than from a task node.

Example – Fig. 12: At this point, the invoking message flows t_A and t_C are replaced with nodes T_A and T_C and message flows $t_A^I = \overrightarrow{A\,T_A}$ and $t_C^I = \overrightarrow{C\,T_C}$. The message flow t_A^R is added because no receiving node is found as $C \in I$ separates A from $C_1, C_2 \in R$. t_{C1}^R and t_{C2}^R are assigned to t_C^I.

[6]Unless there is some more complex communication between multiple users.

C. STAGE 3: STATE DETECTION

1) STATE ANNOTATIONS

As explained in Section VII, state annotations represent knowledge of one participant about the state of another participant. The annotations are collected from the BPMN diagram and stored in a map $A = V \to Q$, which assigns states Q of the future state machine to nodes V of the BPMN diagrams.

It is important to keep in mind that the annotations do not reflect the current state of the inferred state machine but merely knowledge about the latest known state of the state machine, and such information arrives via a message flow only. Therefore, the scope of the annotation is a path from a previous potential receiving node to a next invoking node.

In other words, if there is a path from $n_r \in R^+$ to $n_i \in I$ with a node $n_a \in (n_r \twoheadrightarrow n_i)$, then annotation $A(n_a)$ attached to node n_a means that, in the node n_r, the participant received information that the state machine is in state $A(n_a) \in Q$.

2) STATE RELATION

A state is a path from one transition to another. Therefore, to identify states, we inspect reachability from each potential receiving node and start event[7] to all invoking nodes and end events. The paths may include sequence flows and any nodes except the invoking and potential receiving nodes since, in these nodes, information about the current state is updated. Inclusion of message flows between users in the paths leads to complicated state propagation; therefore, we consider only sequence flows when constructing the state relation and expect the use of additional annotations.

The result of the state detection is a state relation $S = \{(n_r, N_i, \{q_A\})\}$, which assigns a set of invoking nodes $N_i \subseteq I$ to a potential receiving node $n_r \in R^+$ along with a label q_A denoting in which state the state machine has been when the state was entered. We can picture the state relation

[7]As the start events are related to the "Not Exists" state in the Smalldb state machine [1].

S as a set of the following arrows:

$$\underset{\text{receiving}}{n_r} \xrightarrow[(S)]{\{q_A\}} \underset{\text{invoking}}{N_i}$$

To build the state relation, the STS algorithm needs to inspect reachability from the potential receiving nodes and start events to the invoking nodes and end events over the connected sequence flows and nodes that are not within the state machine participant and that are not invoking nodes or potential receiving nodes. Therefore, if a path from a potential receiving node n_r to an invoking node n_i exists, then these nodes are connected by S, i.e., $(n_r, \{n_i, \dots\}, \{q_A\}) \in S$.

The state annotation q_A must be the only annotation found on all the paths from the potential receiving node n_r to all invoking nodes N_i. In case there is more than one annotation found (and these annotations are not the same), the algorithm terminates with an error. If no annotations are found, implicit labeling is applied instead (we will return to this later).

A duality between the state relation S and the transition relation T is not accidental as there are no significant structural differences between the state regions[8] and the transition regions[9] of a BPMN diagram.

Example – Fig. 12: The gray areas mark regions of reachability. The state relation is the following:

$$S = \{(S_1, \{A\}, \{Q_s\}), (A, \{C\}, \{Q_b\}), (C_1, \{E_1\}, \{Q_{e1}\}), (C_2, \{E_2\}, \{Q_{e2}\})\}$$

D. STAGE 4: IMPLICIT STATE LABELING

Once the state relation is calculated, we may want to make a few minor adjustments for users' convenience. The following sections describe an implicit interpretation of two features occurring in BPMN diagrams, which can serve as a fallback behavior when no annotation specifies otherwise. In both cases, we modify the earlier computed state relation, and both operations are optional.

1) IMPLICIT "NOT EXISTS" STATE

Start and end events in the context of a Smalldb state machine represent the "Not Exists" state, unless defined otherwise. Therefore, if there is no annotation specifying the state, the "Not Exists" state is assumed if the state relation S starts in a start event node or ends in an end event node.

This rule implements a convention to begin and end the business process, a syntactic sugar, and thus it is completely optional. In case this rule is omitted, the affected states will be given random names similar to any other state with no annotations.

Example – Fig. 12, 13: Since there are no annotations missing in our example, the implicit "Not Exists" state will not apply. However, in case we would remove all the annotations, the states Q_s, Q_{e1}, and Q_{e2} would be merged into the single "Not exists" state, and Q_b would be named randomly.

[8]Paths from potential receiving nodes and start events to invoking nodes and end events.

[9]Paths from invoking to receiving nodes.

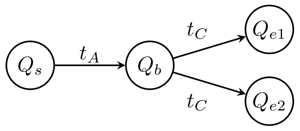


FIGURE 13. Example – Inferred state machine.

Example – Fig. 14, 15: Since there are no annotations on a path $S_1 \twoheadrightarrow R$, the "Not Exists" state will be assumed (Q_{S1}). Both paths $A \twoheadrightarrow E_1$ and $C \twoheadrightarrow E_2$ are annotated; therefore, the implicit state rule does not apply there. The path $S_2 \twoheadrightarrow M_2$ is ignored as it does not end in an invoking node.

2) IMPLICIT STATE PROPAGATION BETWEEN PARTICIPANTS

A state machine may serve as a communication channel to synchronize multiple users. One user invokes a transition, and as this user receives the response, other involved users receive the response in the form of a notification about the new state. Such a scenario is represented by multiple message flows leaving a transition task node in the state machine participant, some message flows leading to the invoking user, and the remaining message flows leading to the other involved users.

When there are multiple possible outcomes of the transition, and thus multiple returning message flows, we need to rely on annotations to mark matching message flows. However, if there is only one possible outcome of the transition, we can safely assume that all involved users will receive the same notification and implicitly propagate the state information to all of them.

For example, in Fig. 14, we have a tester and a developer. The tester reports a bug and assigns it to the developer, and then the state machine sends a notification to the developer, who then fixes and closes the bug. An annotation connected to the node A tells us that the transition ends in the "Assigned" state. Because there is a message flow $\overrightarrow{T_A M_2}$ connected to the same transition node as the annotated receiving node (via $\overrightarrow{T_A A}$), the state information is propagated to M_2, and therefore we know that the "close" transition is invoked from the "Assigned" state (because of the path $M_2 \twoheadrightarrow C$). As we can see, the path $S_2 \twoheadrightarrow M_2$ has no influence on the state machine; however, it provides a better understanding of the developer's workflow.

The implicit state propagation is also an optional feature and can be completely replaced by the explicit use of annotations in all cases. Similar to the implicit "Not Exists" state (discussed in the previous section), its purpose is to provide convenience and reduce clutter in the diagrams.

E. STAGE 5: STATE MACHINE CONSTRUCTION

At this point, all necessary computations are already done; both transition relation and state relation are completed:

- Transition relation T is a set of arrows:

$$\underset{\text{invoking}}{n_i} \xrightarrow[(T)]{m} \underset{\text{receiving}}{N_r}$$

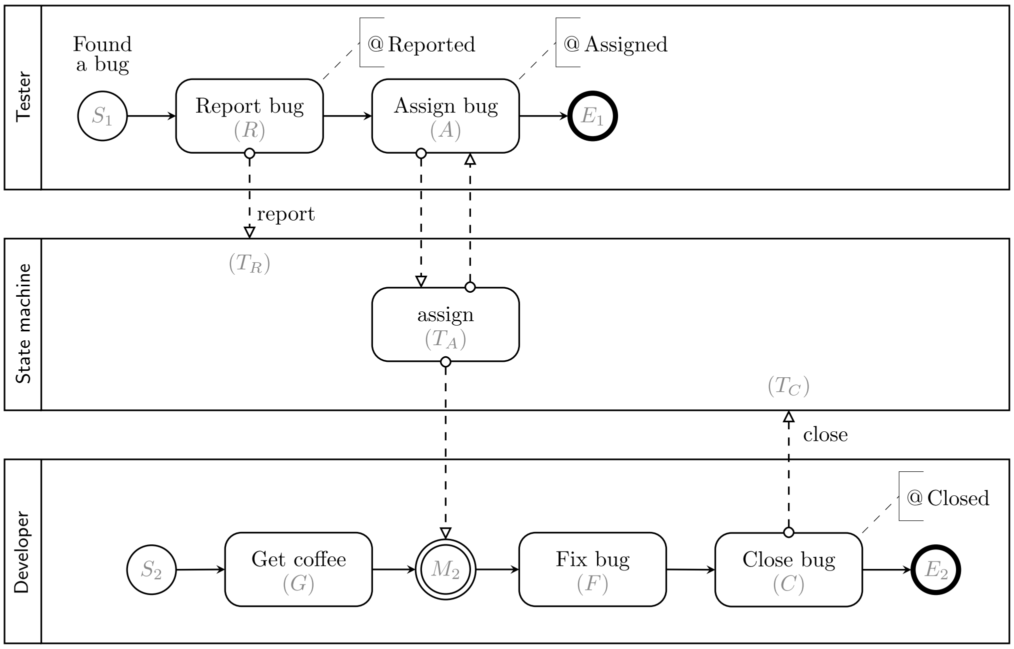


FIGURE 14. State propagation between multiple participants.

- State relation S is a set of arrows:

$$\underset{\text{receiving}}{n_r} \xrightarrow[(S)]{\{q_A\}} \underset{\text{invoking}}{N_i}$$

We only need to collect the results and construct the desired state machine:

- States Q of the state machine are $\lambda(S)$, the labels of the state relation S.
- Names of methods M as input symbols are $\lambda(T)$, the labels of the transition relation T.
- Transition function $\alpha : (q_t, m) \mapsto Q_{t+1}$ is a composition of the state relation S and the transition relation T using the $S \to T \to S$ schema:

$$\cdots \xrightarrow[(S)]{\{q_t\}} \underset{\text{invoking}}{N_i \ni n_i} \xrightarrow[(T)]{m} \underset{\text{receiving}}{N_r \ni n_r} \xrightarrow[(S)]{Q_{t+1}} \cdots$$

  Both relations are connected using invoking and receiving nodes. Then, the respective labels of the relations are collected. Finally, to each pair of a state $q_t \in Q$ and a name of a method m, the possible next states $Q_{t+1} \subseteq Q$ are assigned, forming a transition table of a nondeterministic finite automaton.

The $S \to T \to S$ schema, the final step of the algorithm, is what gives the STS algorithm its name.

Example – Fig. 13: From the previous stages, we know that the transition relation T and state relation S are:

$$\begin{aligned} T &= \{(A, \{A\}, t_A), (C, \{C_1, C_2\}, t_C)\} \\ S &= \{(S_1, \{A\}, \{Q_s\}), (A, \{C\}, \{Q_b\}), \\ &\quad (C_1, \{E_1\}, \{Q_{e1}\}), (C_2, \{E_2\}, \{Q_{e2}\})\} \end{aligned}$$

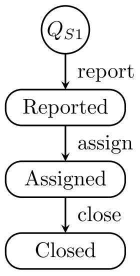


FIGURE 15. The state machine inferred from Fig. 14.

Therefore, the Smalldb state machine has states Q, names of methods M, and transition function α:

$$\begin{aligned} Q &= \{Q_s, Q_b, Q_{e1}, Q_{e2}\} \\ M &= \{t_A, t_C\} \\ \alpha &= \{((Q_s, t_A), \{Q_b\}), ((Q_b, t_C), \{Q_{e1}\}), \\ &\quad ((Q_b, t_C), \{Q_{e2}\})\} \end{aligned}$$

F. SUMMARY OF THE STS ALGORITHM

In the previous sections, we learned that the input of the STS algorithm is the BPMN diagram B (Fig. 11). There are two important intermediate results: the transition relation T, which connects invoking to receiving nodes (Fig. 12), and the state relation S, which connects the remaining fragments of the diagram not covered by T. The desired state machine (Fig. 13) is then created by combining the two relations together.

Note that the operators $V, E, \sigma, \tau, \lambda$, and $\rightarrowtriangle$ are defined in Section VIII.

The full-yet-compact description of the STS algorithm is as follows:

- The output of the STS algorithm is a nondeterministic finite automaton with a set of states Q, names of methods as input symbols M, and transition function[10] $\alpha(q_t, m) \mapsto Q_{t+1}$, where $q_t \in Q$, $Q_{t+1} \subseteq Q$, $m \in M$, and the "Not Exists" state $Q_0 \in Q$.
- The first input of the STS algorithm is a BPMN diagram, a directed graph $B = \{V, E, A, M, P\}$ of nodes V, arrows E, annotations $A : V \to Q$, message flow labels $L : E_m \to M$, participants $V_P \subset V$, and an affiliation of a node to a participant's processes $P : V \to V_P$ (where $\forall v_p \in V_P : P(v_p) = v_p$). The second input is a state machine participant $V_{SP} \in V_P$, for which the state machine should be constructed by the STS algorithm (example: see Fig. 11). Furthermore:
  - $V_S = \{v \mid v \in V \wedge P(v) = V_{SP}\}$ is the set of all nodes of the state machine process.
  - $V_e \subseteq V$ is the set of all events.
  - $V_{start} \subset V_e$ is the set of start events.
  - $V_{end} \subset V_e$ is the set of end events.
  - $E_m \subseteq E$ is the set of all message flows.
- Stage 1: Identify invoking and potential receiving nodes (Sec. IX-A)
  - Invoking nodes I (note $S \in V_S$):
    $I = \{\sigma(e) \mid \forall e \in E_m \wedge \tau(e) \in V_S\}$
  - Primary potential receiving nodes R_m:
    $R_m = \{\tau(e) \mid \forall e \in E_m \wedge \sigma(e) \in V_S\}$
  - Path constraint predicate[11] $P_T(p = a \twoheadrightarrow b)$
    $\equiv ((I \cup R_m \cup V_S \cup V_e) \setminus \{a, b\}) \cap p = \emptyset$
  - Potential receiving nodes R^+:
    $R^+ = R_m \cup \{n_i \mid n_i \in I, \forall n_i \neg\exists n_r \in R_m :$
    $(\exists p_t : p_t = n_i \twoheadrightarrow n_r \wedge P_T(p_t))\}$
    Note that the path p_t is the same as in T. Therefore, R^+ can be computed together with T in the next stage. Moreover, $I \cup R_m = I \cup R^+$.
- Stage 2: Transition detection (Sec. IX-B)
  - Transition relation T (ternary) from I to R^+ (by constructing N_r and finding m):
    $T = \{(n_i, N_r, m) \mid n_i \in I,\ N_r \subset R^+,\ N_r \neq \emptyset,$
    $\forall n_r \in N_r \exists p_t : p_t = n_i \twoheadrightarrow n_r \wedge P_T(p_t),$
    $\exists e_i \in E_m : \sigma(e_i) = n_i \wedge \tau(e_i) \in V_S),$
    $m = L(e_i)\}$
    (E.g., paths p are the gray arrows in Fig. 12.)
  - Receiving nodes:
    $R = R^+ \cap \bigcup \tau(T) = \bigcup \tau(T), R \subseteq R^+$
- Stage 3: State detection (Sec. IX-C)
  - Path constraint predicate $P_S(p = a \twoheadrightarrow b)$
    $\equiv \big((I \cup R^+ \cup V_S) \setminus \{a, b\}\big) \cap p = \emptyset$
  - State relation S (ternary) from $R^+ \cup V_{start}$ to $I \cup V_{end}$ (by constructing N_i and collecting annotations[12] Q_A found along the paths):
    $S = \{(n_r, N_i, Q_A) \mid n_r \in R^+ \cup V_{start},$
    $N_i \subseteq I \cup V_{end}, N_i \neq \emptyset,$
    $\forall n_i \in N_i \exists p_s : p_s = n_r \twoheadrightarrow n_i \wedge P_S(p_s),$
    $Q_A = \tau(A_S), |Q_A| \leq 1, A_S \subseteq A,$
    $\forall a \in A_S \exists p_s : p_s = n_r \twoheadrightarrow n_i \wedge P_S(p_s)$
    $\wedge\, \sigma(a) \in p_s \wedge \sigma(a) \notin N_i\}$
    (E.g., paths p_s are in the gray areas in Fig. 12.)
- Stage 4: Implicit State labeling (Sec. IX-D)
  - Implicit labeling (assign $\lambda(s)$; use the "Not Exists" state when there is no annotation but a start or end event is present):
    $\forall s \in S : \lambda(s) = \emptyset \wedge (\sigma(s) \in V_{start}$
    $\vee (\tau(s) \cap V_{end}) \neq \emptyset) \implies \lambda(s) = \{Q_0\}$
  - Implicit state propagation in transitions with a single receiving node (assign $\lambda(s_r)$):
    $\forall t_1 \in T, e_i \in E_m, E_r \subset E_m, S_r \subset S :$
    $|\tau(t_1)| = 1 \wedge \sigma(t_1) = \sigma(e_i)$
    $\wedge\, (\forall e_r \in E_r : \tau(e_i) = \sigma(e_r))$
    $\wedge \sigma(S_r) = \tau(E_r)$
    $\implies \forall s_r \in S_r \exists q_A \in Q : \lambda(s_r) = \{q_A\}$
- Stage 5: State machine construction using the $S \to T \to S$ schema (Sec. IX-E):

$$\cdots \xrightarrow[(S)]{\{q_t\}} \underset{\text{invoking}}{N_i \ni n_i} \xrightarrow[(T)]{m} \underset{\text{receiving}}{N_r \ni n_r} \xrightarrow[(S)]{Q_{t+1}} \cdots$$

  - States: $Q = \lambda(S)$
  - Names of methods as input symbols: $M = \lambda(T)$
  - Transition function $\alpha : (q_t, m) \mapsto Q_{t+1}$ by composing relations S and T using the $S \to T \to S$ schema:

$$\alpha = \{((q_t, m), Q_{t+1}) \mid \forall s_1 \in S, t \in T, s_2 \subset S :$$
$$\tau(s_1) \ni \sigma(t), \tau(t) = \sigma(s_2),$$
$$q_t = \lambda(s_1), m = \lambda(t), Q_{t+1} = \bigcup \lambda(s_2)\}$$

G. COMPUTATIONAL COMPLEXITY AND CONVERGENCE

The STS algorithm is surprisingly effective. The first stage of the algorithm is only a simple linear iteration over all message flows while collecting invoking and potential receiving nodes. The second stage, transition detection, involves reachability inspection using DFS (depth-first search) over a portion of the graph ($I \twoheadrightarrow R^+$), which has linear complexity $O(|E| + |V|)$. The third stage, state detection, uses DFS to inspect the reachability on the remaining portion of the graph, but DFS is run twice – forward ($R^+ \twoheadrightarrow I$) to inspect the reachability and then backwards from each reached invoking node ($R^+ \twoheadleftarrow I$) to collect state annotations on all found paths. Finally, the fourth stage, state machine construction, involves combining $\sigma(S)$ with $\tau(T)$.

In the worst case scenario, a large set of parallel task which are receiving nodes followed by a long chain of regular

[10]In the Smalldb state machine definition, [1] α has slightly different semantics, and the transition function involves a microstep, but for now we can ignore it.

[11]$P_T(p)$ valuates true iff path p does not intersect nodes from I, R^+, V_S, nor V_e. It may start or end in these nodes, though.

[12]There should only be one annotation present on the paths from n_r to N_i.

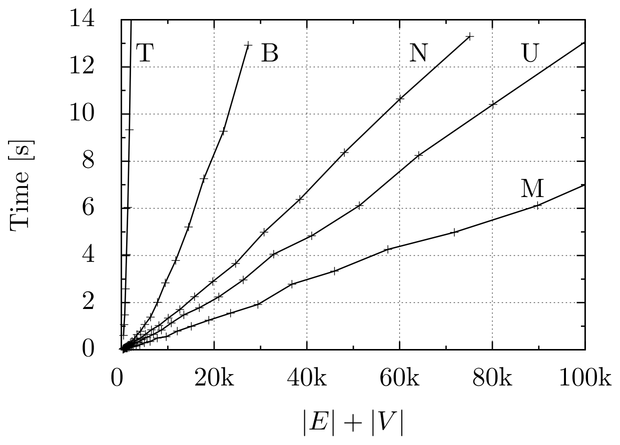


FIGURE 16. Processing speed of the tested scenarios: a chain of tasks (N), user decides (U), machine decides (M), a combination of user decides and machine decides (B), and a T-shape BPMN diagram (T).

nodes terminated with a receiving node (forming a T-shaped graph; see Fig. 17), such a component will have the total quadratic complexity $O(|V| \cdot (|E| + |V|))$ when inspecting reachability from each of the parallel tasks. However, in a typical scenario, the graph consists of small fragments, where only a small number n of receiving nodes are reachable from each invoking node and vice versa. Therefore, the practical complexity is roughly $O(n \cdot (|E| + |V|))$, which is linear complexity (since n is bounded in practice).

We tested the STS algorithm on large synthetic graphs up to 175 000 nodes plus edges using a laptop (Intel i7 2.8 GHz, 8GB RAM, using PHP 7.3) – see the plot in Fig. 16. The shape of the generated graphs followed the examples presented earlier: a chain of N simple tasks (N in Fig. 16), the "user decides" scenario with N parallel branches (U), and the "machine decides" scenario with N possible results (M). In all three cases, the algorithm complexity was linear with a processing speed of approx. 5 000 to 13 000 nodes plus edges per second, i.e., 50 000 to 130 000 nodes and edges in approx. 10 seconds.

To explore the limits of the algorithm in the worst case scenarios, we constructed the following two cases. The first case combines the "user decides" scenario and "machine decides" scenario, where each branch of user's decision was terminated with a machine decides fragment each leading to the same set of possible results, effectively forming a total bipartite graph. In this case, the algorithm performed with sublinear complexity because the overhead of a nonoptimized graph representation become apparent; see the B in Fig. 16.

The second, final, case was the mentioned T-shape graph (Fig. 17) with $N/2$ parallel task nodes followed by an $N/2$ nodes long sequence of nodes not in I or R^+. As expected, in this case, the STS algorithm performed with quadratic complexity, where the processing of 1 000 nodes and edges took about 4 seconds and 10 000 nodes and edges took about five minutes; see the T in Fig. 16. However, we can utilize the caching of reachable nodes from once visited nodes to exchange linear memory complexity for nearly linear time complexity.

This effectivity allows the use of the algorithm to generate live previews and to implement error detection during interactive edits of the BPMN diagrams.

Because the algorithm only iterates the BPMN graph using DFS and then combines collected data, the convergence of the algorithm is guaranteed in all cases.

X. A SIMPLE PRACTICAL EXAMPLE: CRUD

Let us show the basic operation of the STS algorithm by implementing CRUD (create, read, update, delete), the basic and most common pattern occurring in web applications. This pattern is a primary tool of modern ORM (object-relational mapping) frameworks, and it is also a base for RESTful API, which maps the operations to HTTP methods. While most frameworks expect model entities to be CRUD and nothing else, Smalldb does not enforce how an entity is expected to behave.[13] The Smalldb framework provides a prefabricated component to implement CRUD behavior easily; however, in this example, we take a look at a different approach to achieve the CRUD behavior. This approach is not meant to be used as–is in practice, but it may serve as a starting point for more complex workflows.

Fig. 18 presents a BPMN diagram of how a user uses a CRUD entity (ignore the gray areas for now). First, the user creates an entity, then she may edit it multiple times, and finally, she deletes it. Each of the three message flows in the figure represents a state machine transition invocation, and all of the invocations are simple tasks as presented in Section VI-A.

Note that the read operation is missing in the diagram because the read operation may be performed at any time. Typically, the read operation precedes every invocation of a state machine transition as the user loads an HTML page that he then uses to invoke the transition.

Once we let the STS algorithm process the BPMN diagram, we obtain the inferred state machine presented in Fig. 19.

To obtain better insight into how the algorithm inferred the state machine, the gray areas in Fig. 18 present detected states as a partitioning of the BPMN diagram (similar to Fig. 12). First, the algorithm identifies the invoking nodes $I = \{C, E, D\}$, and the receiving nodes are the same $(R = R^+ = I)$.

Therefore, the transition relation T and state relation S are very simple:

$$\begin{aligned} T = \{&(C, \{C\}, \text{create}), (E, \{E\}, \text{edit}), \\ &(D, \{D\}, \text{delete})\} \\ S = \{&(S, \{C\}, \{\text{Not Exists}\}), (C, \{E, D\}, \{\text{Exists}\}), \\ &(E, \{E, D\}, \{\text{Exists}\}), (D, \{F\}, \{\text{Not Exists}\})\} \end{aligned}$$

Once S and T are combined together, the algorithm provides us with the state machine as presented in Fig. 19. Note how the gray regions in Fig. 18 become the states

[13] As long as it can be expressed using a nondeterministic state machine.

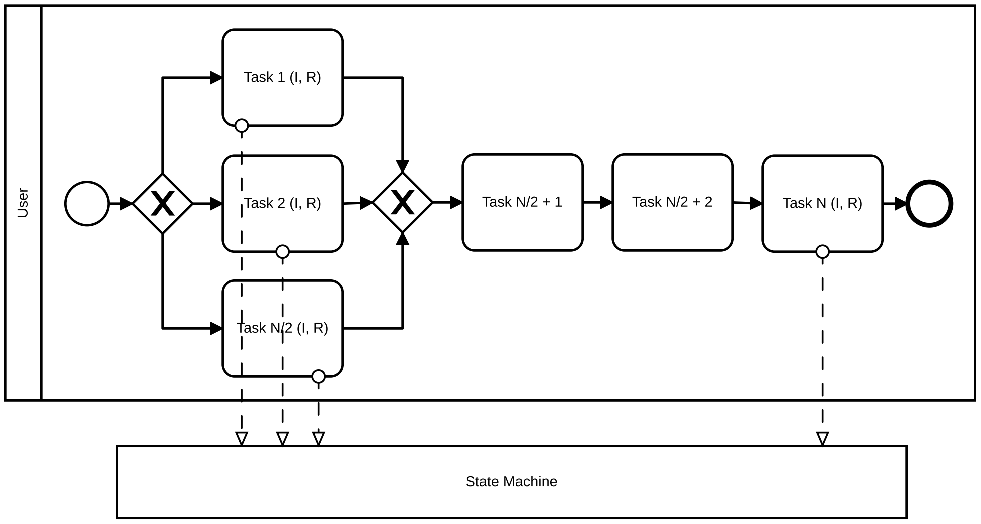


FIGURE 17. T-Shape BPMN Diagram.

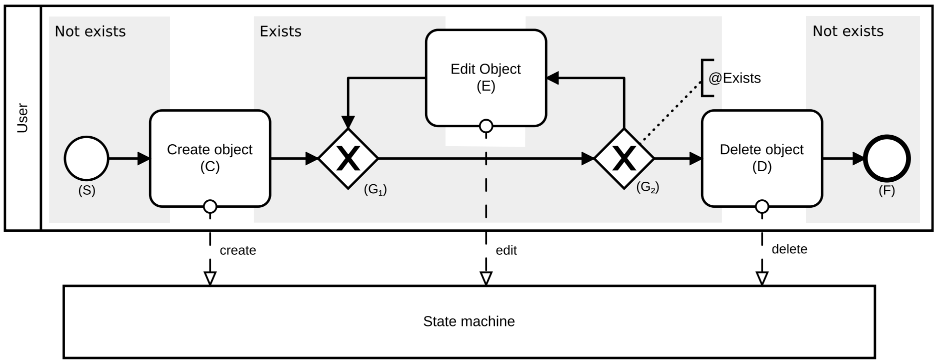


FIGURE 18. Crud entity in a BPMN diagram (The grey areas in the background are not present in the source diagram).

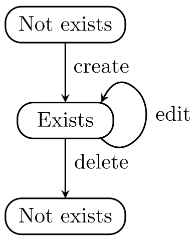


FIGURE 19. The state machine inferred from Fig. 18.

in Fig. 19 and the task nodes connecting the regions become the arrows.

The "Not Exists" state is both the initial and final state and is drawn twice for better readability; the state machine forms two loops since the entity may be created again once deleted. This state represents a state in which the entity does not exist [1]; there is no record of the entity. In the language of object-oriented programming, the "create" transition represents a constructor, and the "delete" transition represents a destructor.

As we can see, the edit loop is present in both the BPMN diagram and the inferred state machine. While the BPMN diagram represents the loop using sequence flows and the "Edit object" task, the state machine has a transition returning to the same state.

XI. EXAMPLE: SIMPLE TASK

In Section VI, we presented the four basic features found in BPMN diagrams and relevant to the STS algorithm. The first of these was the simple task scenario (Sec. VI-A), in which a user invokes two subsequent state machine transitions ("create" and "close") as presented in Fig. 3. The result is a state machine of two transitions and three states, as shown in Fig. 5 (Q_0 = Not Exists, Q_E = Exists, Q_C = Closed). Earlier, we assumed that a programmer infers the implementation of the Issue State Machine intuitively, but now we have the STS algorithm available, so let us take a look at how it will cope with the scenario.

The STS algorithm identifies the invoking nodes I and receiving nodes R in Fig. 3, and then it infers the transition relation T and state relation S as follows:

$$\begin{aligned} I &= R = R^{+} = \{Cr, Cl\} \\ T &= \{(Cr, \{Cr\}, \text{create}), (Cl, \{Cl\}, \text{close})\} \\ S &= \{(S_1, \{Cr\}, \{\text{Not Exists}\}), (Cr, \{Cl\}, \{\text{Exists}\}), \\ &\quad (Cl, \{E_1\}, \{\text{Closed}\})\} \end{aligned}$$

XII. EXAMPLE: THE USER DECIDES

The scenario discussed in Section VI-B presents a situation in which a user creates an issue and later decides whether the process has completed or failed – see Fig. 6. The result is a state machine of three distinct transitions, as presented in Fig. 7 (notice transitions t_{mC} and t_{mF}).

The STS algorithm identifies the invoking nodes I and receiving nodes R in Fig. 6, and then it infers the transition relation T and state relation S as follows:

$$\begin{aligned} I &= R = R^{+} = \{Cr, Mc, Mf\} \\ T &= \{(Cr, \{Cr\}, \text{create}), \\ &\quad (Mc, \{Mc\}, \text{markCompleted}), \\ &\quad (Mf, \{Mf\}, \{\text{markFailed}\})\} \\ S &= \{(S_1, \{Cr\}, \{\text{Not Exists}\}), \\ &\quad \big(Cr, \boldsymbol{\{Mc, Mf\}}, \{\text{Exists}\}\big), \\ &\quad (Mc, \{E_1\}, \{\text{Success}\}), (Mf, \{E_2\}, \{\text{Fail}\})\} \end{aligned}$$

XIII. EXAMPLE: THE MACHINE DECIDES

In contrast with the previous example, the scenario discussed in Section VI-C presents a situation in which a user creates an issue, but it is the machine that decides whether the process has completed or failed – see Fig. 8. The result is a state machine of one deterministic transition and one nondeterministic transition, as presented in Fig. 9 (notice the two arrows both labeled t_{rR}).

The STS algorithm identifies the invoking nodes I and receiving nodes R in Fig. 8, and then it infers the transition relation T and state relation S as follows:

$$\begin{aligned} I &= \{Cr, Rr\}, R = R^{+} = \{Cr, Rs, Rf\} \\ T &= \{(Cr, \{Cr\}, \text{create}), \\ &\quad \big(Rr, \boldsymbol{\{Rs, Rf\}}, \text{recordResults}\big)\} \\ S &= \{(S_1, \{Cr\}, \{\text{Not Exists}\}), \\ &\quad (Cr, \{Rr\}, \{\text{Exists}\}), \\ &\quad (Rs, \{E_1\}, \{\text{Success}\}), (Rs, \{E_2\}, \{\text{Fail}\})\} \end{aligned}$$

Note the difference between this and the previous example. When user decides about the process, it is the state relation S which represents the branching – Cr to Mc or Mf in this case. However, when machine decides, it is the transition relation T which represents the branching – Rr to Rs or Rf in this case.

XIV. EXAMPLE: SYNCHRONIZATION BETWEEN USERS

The last example from Section VI presents a situation when one user notifies another user via the state machine, see Section VI-D and Fig. 10. The result is the state machine same as in the Simple Task Example, see Fig. 5. The difference is in who invokes which transition.

The STS algorithm identifies the invoking nodes I and receiving nodes R in Fig. 10, and then it infers transition relation T and state relation S as follows:

$$\begin{aligned} I &= \{Cr, Cl\}, R^{+} = \{Cr, S_2, Cl\} \\ R &= I = \{Cr, Cl\} \\ T &= \{(Cr, \{Cr\}, \text{create}), (Cl, \{Cl\}, \text{close})\} \\ S &= \{(S_1, \{Cr\}, \{\text{Not Exists}\}), (Cr, \{E_1\}, \{\text{Exists}\}), 5 \\ &\quad (S_2, \{Cl\}, \{\text{Exists}\}), (Cl, \{E_2\}, \{\text{Closed}\})\} \end{aligned}$$

XV. LARGE PRACTICAL EXAMPLE: PIZZA DELIVERY

How to order pizza delivery online? The following example presents the entire process starting with a hungry customer ordering a pizza via a web application, continuing with a chef baking the pizza, and ending with a delivery boy bringing the pizza to the customer. While the example attempts to be as realistic as possible, the limited space of this paper allows us to explore only the important aspects of the process.

A. THE SCENARIO

The BPMN diagram presented in Fig. 20 describes the interaction between a web application and three people, namely, a customer, a chef, and a delivery boy. The source BPMN diagram does not contain the gray areas or the numbers and letters in brackets, as these are only added to help with the following explanation of the example.

The scenario starts with a hungry customer (see node 0 in Fig. 20). The customer visits a pizzeria website and selects a pizza (node 1). Then, she can set the delivery address (4) and change the size of the pizza (5). Additionally, the customer may choose to delete the order (6) and not have the pizza at all (8).

Once the order is ready, the customer submits it (7). At this point, the order is either accepted (9) and the chef is notified (18) or rejected (10) and returned to the customer for additional changes, i.e., the chef may be too busy with other orders, so the user may submit it again later or never (12, 14).

As soon as the pizza order is properly submitted, the chef confirms the order (19) and bakes the pizza (20). Then, a delivery boy delivers the pizza to the customer (21). After the payment, the delivery boy marks the order as delivered (23), gives the receipt to the customer, and returns to the pizzeria (25). In case the delivery boy fails to locate the customer (e.g., due to an incorrect address), he marks the delivery as failed (22) and enjoys the pizza (24).

While the diagram may look slightly verbose at first, it is important to realize that there are three people and a pizza to orchestrate, programmers likely have no idea how the business works in the background, and such a diagram may be the only attempt to describe the business workflow.

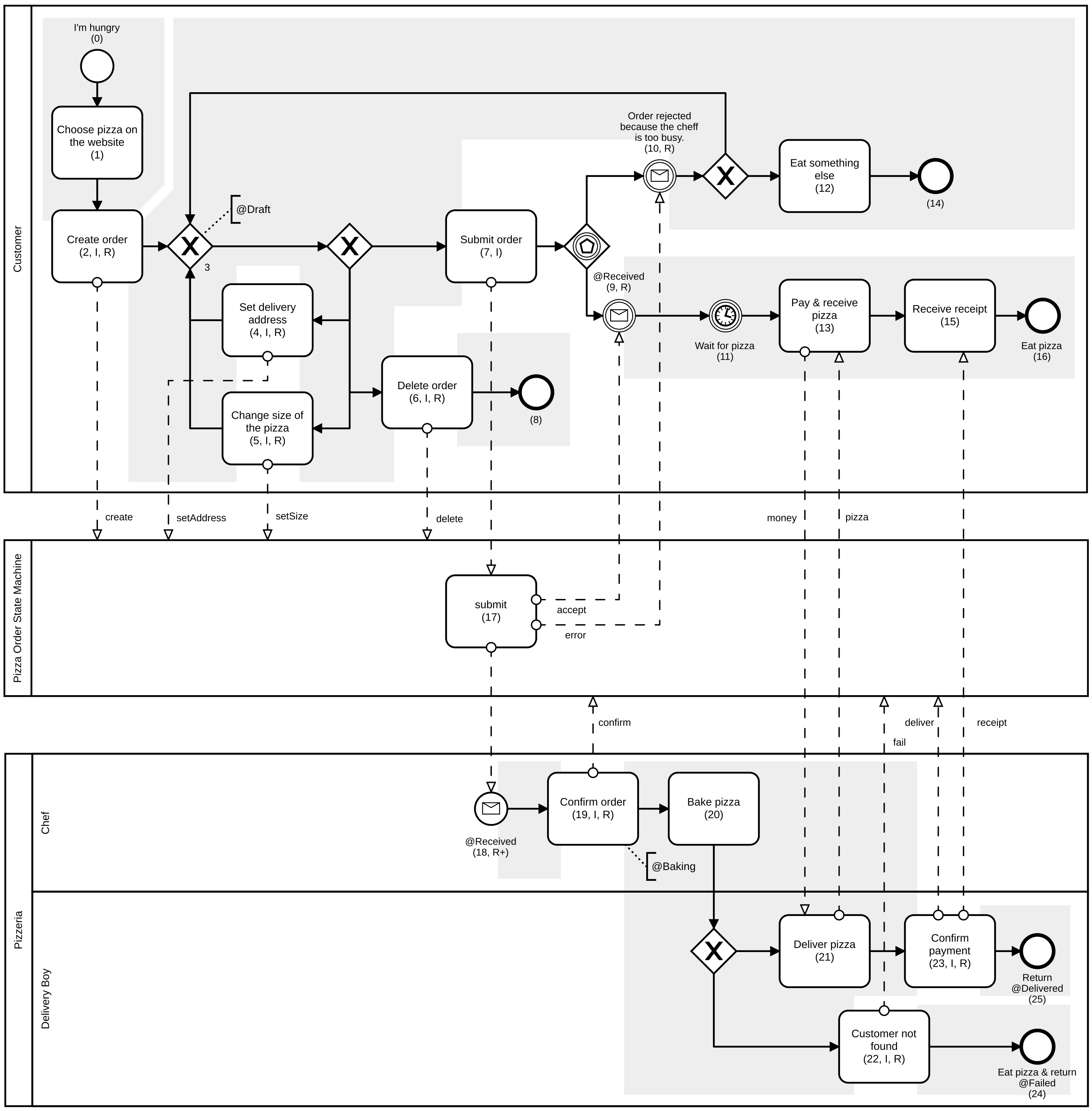


FIGURE 20. Pizza delivery example (The grey areas in the background are not present in the source diagram.).

B. THE FEATURES

Earlier, in Section VI, we described various features occurring in BPMN diagrams, and all of them occur in the pizza delivery example.

The first described feature was the "simple task" (Sec. VI-A), i.e., a simple invocation of a state machine transition. Such tasks are quite common in our example (nodes 2, 4, 5, 6, 19, 22, and 23). The user invokes a given transition, immediately receives the response and continues with another task. From each of these tasks, we see an invoking message flow from the task node to the state machine participant, and we do not see implicit returning message flows anti-parallel to the respective invoking message flows (the algorithm will infer these).

The "user decides" scenario (Sec. VI-B) occurs twice in the example. The first occurrence is the CRUD loop between creating and submitting the order (nodes 2, 4, 5, and 6). The second occurrence is at the end when the delivery boy may not find the customer (nodes 19, 22, and 23).

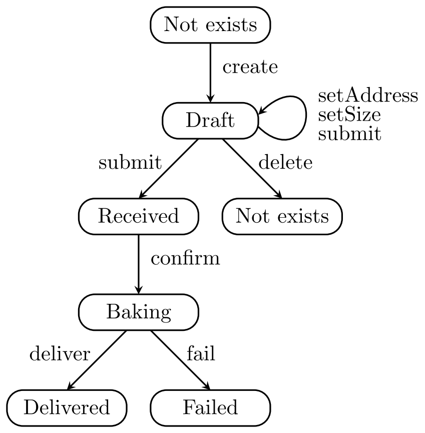


FIGURE 21. The Pizza Order State Machine (from Fig. 20).

The "machine decides" scenario (Sec. VI-C) only occurs once when the pizza order state machine decides whether or not to accept the order (nodes 7, 9, and 10). Moreover, this feature is combined with the synchronization between users (Sec. VI-D; nodes 7, 17, and 18).

C. HIGHLIGHTS FROM THE STS ALGORITHM EXECUTION

Once we let the STS algorithm process the BPMN diagram (Fig. 20), the algorithm will identify the invoking and the (potential) receiving nodes as marked using the letters I and R (R^+), respectively; $I = \{2, 4, 5, 6, 7, 19, 22, 23\}$, $R = \{2, 4, 5, 6, 9, 10, 19, 22, 23\}$, and $R^+ = R \cup \{18\}$. Note that nodes 13 and 21 are not invoking or receiving as the message flows do not involve the state machine. Transition relation T and state relation S are the following:

$$\begin{aligned}
T = \{&(2, \{2\}, \text{create}), (4, \{4\}, \text{setAddress}),\\
&(5, \{5\}, \text{setSize}), (6, \{6\}, \text{delete}),\\
&(7, \{9, 10\}, \text{submit}), (19, \{19\}, \text{confirm}),\\
&(22, \{22\}, \text{fail}), (23, \{23\}, \text{deliver})\}\\
S = \{&(0, \{2\}, \{\text{Not Exists}\}), (2, \{4, 5, 6, 7\}, \{\text{Draft}\}),\\
&(4, \{4, 5, 6, 7\}, \{\text{Draft}\}), (5, \{4, 5, 6, 7\}, \{\text{Draft}\}),\\
&(6, \{8\}, \{\text{Not Exists}\}), (9, \{16\}, \{\text{Received}\}),\\
&(10, \{4, 5, 6, 7, 14\}, \{\text{Draft}\}),\\
&(18, \{19\}, \{\text{Received}\}), (19, \{21, 22\}, \{\text{Baking}\}),\\
&(22, \{24\}, \{\text{Fail}\}), (23, \{25\}, \{\text{Delivered}\})\}
\end{aligned}$$

D. THE RESULT

The result is the Smalldb state machine pictured in Fig. 21. As before, the "Not Exists" state is drawn twice, and the three loops in the "Draft" state are drawn using a single arrow for better readability.

The "user decides" feature is transformed into a simple branching from the "Draft" and "Baking" states. The "machine decides" feature is represented using the nondeterministic "submit" transition.

As we can see, the state machine is getting complicated even for this relatively simple business process. Without a thorough understanding of the business process, it would be tough to draw this state diagram. It would likely involve many discussions with a customer, and many errors would be found only when the first prototypes of the application were released for review. The BPMN diagrams help to share the needed knowledge regarding the business process and design the application correctly from the beginning.

XVI. RELATED WORK

A. UML SEQUENCE DIAGRAMS

UML sequence diagrams [9] are designed to describe interactions between multiple entities, while internals of the entities are omitted; unlike BPMN diagrams which also model internal workflows of its participants. A typical use of a UML sequence diagram is to picture a scenario of nontrivial method calls or network communication. Because the sequence diagrams are not very practical in regard to branching, they are mostly used to represent a single scenario of possible variants with a separate diagram for each case.

Hypothetically, we could apply the STS algorithm to UML sequence diagrams in a similar manner to what we do with BPMN diagrams (the state annotations would have to be added too). These two types of diagrams are not very different at a conceptual level, and the STS algorithm would likely provide us a meaningful result. However, the question is whether the syntax of UML sequence diagrams fits our needs.

For example, Fig. 22 presents the simple CRUD entity described in Section X. In comparison with Fig. 18, the BPMN representation of the same entity, we can see that the diagrams are quite similar. Both present the same two participants with the same communication (the returning message flows are implicit in the BPMN diagram), and both contain the same loop.

While both UML sequence diagrams and BPMN diagrams provide us with a very similar description of a given business process, the UML sequence diagrams are not as easy for nontechnical customers to understand when designing web applications. Therefore, we address the BPMN diagrams in this paper, while the UML option remains open. Alternately, the STS algorithm is not limited to web applications with the Smalldb framework; for example, technical applications, such as test synthesis or protocol verification, may not require details on participant behavior such that the simpler syntax may be more practical, and the STS algorithm can operate as long as the syntax preserves the reachability properties.

B. BPMN EXECUTION ENGINES

Efforts concerning the direct interpretation and execution of BPMN diagrams [2] (and related technologies, e.g., BPEL) have led to the creation of many enterprise tools. Most of them, such as, for example, Camunda BPMN Engine [10], [11], iterate BPMN diagrams as if they were Petri Nets and perform specified actions via (micro)service [12] orchestration or human task flow control. While this approach seems to be reasonable and may help

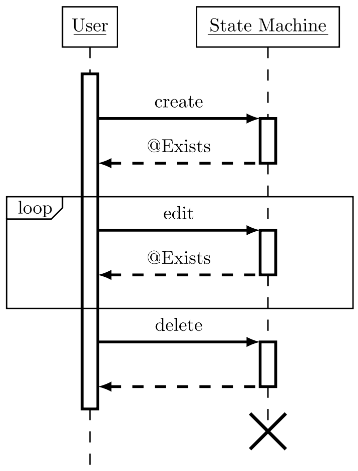


FIGURE 22. **CRUD entity in a UML sequence diagram.**

to manage complex business processes, it has significant drawbacks that have prevented the broad adoption of these tools over recent years.

The obvious issue is that such tools require the relatively high technological complexity of the modeled system. Even a simple use of a microservice requires a nontrivial amount of code, libraries, and infrastructure to get started. Most information systems are technologically (vertically) simple applications, often just a nice facade over an SQL database. The complexity of such systems lies in the wide (horizontal) number of features and covered use cases. Therefore, adding technological complexity to such a shallow but wide application has a multiplicative effect on the overall complexity (and price) of the system.

To identify a less obvious issue, we need to find to which level of abstraction our model belongs. BPMN is generally a high-level notation suited for conceptual models of a business process. Alternately, the execution requires many low-level details available only in the source code of the application and the orchestrated microservices. Both high-level and low-level models are necessary during development of the application, but fundamental properties of the models are in contradiction. We cannot add execution details to a conceptual model because it would make the model incredibly complex, which is the opposite of what we expect it to be. Stated another way, the high-level concepts are useless in a low-level execution model. As a result, we need to have both models and establish a connection between them, i.e., identify and name low-level features in the high-level model, but we do not specify them there.

The STS algorithm does not try to execute a process diagram, it only extracts the described business logic and provides a specialized model for use on lower levels of abstraction. In particular, from a high-level BPMN diagram, the algorithm infers a low-level state machine, which can be directly interpreted by, for example, the Smalldb framework. Clearly, the high-level model does not provide all of the details. Therefore, the inferred state machine is only a skeleton to be filled by a programmer rather than a complete implementation, but it still provides a valuable automated connection between the two levels of abstraction.

C. WS-BPEL AND BPMN

BPMN and WS-BPEL (commonly known as BPEL) are two technologies trying to establish a symbiotic relation, combining the graphical language of BPMN with the executability of BPEL.

WS-BPEL, Web Services Business Process Execution Language [13], is a web services orchestration language based on XML and WSDL. It defines how individual services are connected together and how they should interact with each other. It does not define how the service should be implemented; the point of BPEL is to orchestrate heterogeneous services into a cooperating system.

The scopes of the STS algorithm and WS-BPEL are distinct but not completely unrelated. Combination of WS-BPEL with the STS algorithm might provide programmers with a complete specification of inputs and outputs of the transitions in the inferred state machine, while BPMN diagrams provide us only with the existence of the transition.

In this paper, we have assumed interaction between humans and machines via a simple web application; however, a combination of WS-BPEL, BPMN, and the STS algorithm in the service-oriented architecture (SOA) could provide us with useful models of both the orchestration and the orchestrated services, where WS-BPEL describes interactions between the services and the STS algorithm infers logic of the services from the BPMN diagrams.

D. INDUCTION OF REGULAR LANGUAGES

Various approaches to finite automata synthesis, such as grammatical inference, induction of regular languages, automata learning [14]–[16], and machine learning, are generally all based on a common idea: induce compact rules from a provided set of input sequences. These techniques expect little to no other contextual knowledge, and because of that, they cannot benefit from this knowledge, which often leads to exponential complexity.

It may be possible to find all possible paths through a BPMN diagram and then feed the encountered message flows as input symbols to these algorithms and receive a reasonable result. However, if the diagram contains loops, then the number of paths is infinite; therefore, we cannot be sure that we found all states and did not miss a hidden state, which would appear if a loop was iterated one more time.

The fundamental difference of the STS algorithm is that, instead of analyzing possible input sequences, the STS algorithm identifies basic substructures, paths that connect interactions between the automaton and its surroundings, utilizing a specification which does not describe the automaton explicitly.

The description of the desired automaton is already present in the source BPMN diagrams; we just need to find it. This unique approach efficiently deals with cycles and provides reliable results, even for large scenarios thanks to its linear complexity.

XVII. CONCLUSION

The presented STS algorithm provides a tool to extract a state machine from a BPMN diagram. The purpose of the STS algorithm is to aid during the web application development process, specifically to automate transition from the high-level model, the BPMN diagram, to the low-level implementation, the Smalldb state machine. This is useful because it is easier to draw and understand scenarios in the form of BPMN diagrams rather than state machines or even source code.

The STS algorithm analyzes BPMN diagrams, validates their consistency and can detect various design flaws during discussions with a customer, long before a programmer is involved. Later, during the implementation phase of the project, the STS algorithm, together with the Smalldb framework, provides the programmer with a skeleton of the model layer of the web application. To complete the model layer, the programmer implements individual transitions of the inferred state machine, as these details are not modeled in the business process model (only their existence is defined).

The introduction of the STS algorithm changes our understanding of the concept of state. In the traditional theory of finite automata, a state is a static point between two transitions. The STS algorithm shows us that any state is a path between two transitions, but it is the rest of the world who walks the path. This relation is reflected by the duality of the state relation S and transition relation T used by the STS algorithm when constructing the state machine.

The concept of isomorphic processes is another concept the STS algorithm shows us. This rather trivial idea hidden in plain sight can be expressed using a pair of processes: "When a user uses a tool, the tool is being used." With this realization, we do not have to model the same process twice; once the user's point of view is modeled, the tool's behavior can be inferred. The question to answer is who controls the process, i.e., who decides the next action, either the user or the tool (machine). Based on this concept and the assumption of a single-threaded implementation of the state machine and its users, we have provided a few syntactical shortcuts to make the BPMN diagrams easier to draw and understand, though multithreaded models may be an intriguing topic of further research.

In the end, we presented a practical application of the STS algorithm for the Pizza Delivery Example, where the STS algorithm infers the order state machine, which then orchestrates a business process of three humans and a pizza.

REFERENCES

[1] J. Kufner and R. Mařík, "State machine abstraction layer," in *Information and Communication Technology* (Lecture Notes in Computer Science), vol. 8407, A. Linawati, M. Mahendra, E. Neuhold, A. Tjoa, and I. You, Eds. Berlin, Germany: Springer, 2014, pp. 213–227. doi: 10.1007/978-3-642-55032-4_21.

[2] OMG. (2018). *BPMN/2.0.* [Online]. Available: http://www.omg.org/spec

[3] H. Kubátová, K. Richta, and T. Richta, "Petri nets versus UML state machines," in *Proc. SDOT*, Nov. 2013, pp. 53–59.

[4] R. M. Dijkman, M. Dumas, and C. Ouyang, "Semantics and analysis of business process models in BPMN," *Inf. Softw. Technol.*, vol. 50, no. 12, pp. 1281–1294, Nov. 2008.

[5] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML sequence diagrams and statecharts to analysable petri net models," in *Proc. 3rd Int. Workshop Softw. Perform.*, Jul. 2002, pp. 35–45.

[6] M. Barr and C. Wells, *Category Theory for Computing Science*. New York, NY, USA: Prentice Hall, 1990, vol. 1.

[7] H. Simmons and A. Schalk, *Category Theory in Four Easy Movements*. Manchester, U.K.: Online Book, Univ., 2003.

[8] R. Diestel, "Graph theory," *Graduate Texts Mathematics*, vol. 173. New York, NY, USA: Springer-Verlag, 2005.

[9] *Superstructure Specification*, document formal/2017-12-05, Object Management Group, Version 2.5.1, Needham, MA, USA, 2011. [Online]. Available: http://www.omg.org/spec/UML/

[10] *Camunda Modeller*. Accessed: May 3, 2018. [Online]. Available: https://camunda.com/products/modeler/

[11] *Camunda BPMN Engine*. Accessed: May 3, 2018. [Online]. Available: https://camunda.com/products/bpmn-engine/

[12] L. Chen, "Microservices: Architecting for continuous delivery and DevOps," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, May 2018, pp. 339–379.

[13] O. Standard. (2017). *Web Services Business Process Execution Language Version 2.0.* [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[14] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Aug. 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0890540187900526

[15] A. Khalili and A. Tacchella, "Learning nondeterministic mealy machines," in *Proc. Int. Conf. Grammatical Inference*, 2014, pp. 109–123.

[16] A. L. Oliveira and J. P. M. Silva, "Efficient search techniques for the inference of minimum size finite automata," in *Proc. String Process. Inf. Retr. A, South Amer. Symp.*, Sep. 1998, pp. 81–89.

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 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 worked on technology trend identification, knowledge-based software comprehension, and semantically based product interoperability on research projects with CA Technologies. He joined the Czech Technical University teams, in 2011. He is a coauthor of approximately 80 papers and has filed two patents in complex networks analysis and machine learning in signal processing.

• • •