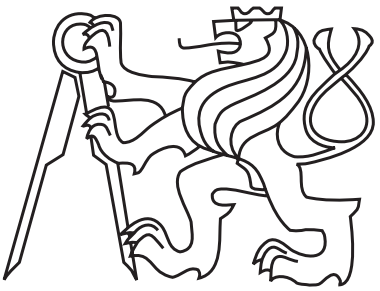
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science



Specifications for Intelligent Software Synthesis

Doctoral Thesis

Ing. Josef Kufner

Prague, January 2020

Ph.D. Programme: Electrical Engineering and Information Technology

Branch of study: Artificial Intelligence and Biocybernetics

Supervisor: Ing. Radek Mařík, CSc.

Abstract

To relieve programmers of repetitive and tiring work on simple, yet too diverse, entities in web applications, this thesis searches for an assistive framework, where machines aid the programmers with implementing such entities. The first question to answer is how to tell the computer what we want without specifying all the details; otherwise, we could just implement the application instead. The second question is how to effectively reason about the software so that we can analyze what we have and infer what we miss. The proposed solution introduces Smalldb state machines as a formal model that describes the behavior of the entities in a web application. Such a model is not designed to cover every aspect of the application; instead, it leaves well-defined gaps for the details that are impractical to specify formally. The formal model then becomes part of the application architecture and runtime, where it provides a secure API between business logic and presentation layer; moreover, it verifies that the implementation corresponds to the model. To answer the first question, we return to the very beginning of the software development process, where we identify software specifications that programmers already use and discuss whether such specifications are sufficiently machine-friendly for automated processing. The identified BPMN business process diagrams become an input for a novel STS algorithm — a software architect draws how users will use a web application, and the STS algorithm infers the implementation in the form of a Smalldb state machine. The thesis concludes with presenting a Cascade, a machine-friendly approach to software composition based on the utilization of the formal models.

Keywords: Software specification, finite automata, web application, business process modeling, software architecture, REST, MVC, SQL, CQS, BPMN.

Abstrakt

Implementace jednoduchých, avšak příliš různorodých, entit ve webových aplikacích je únavná úloha, jejíž řešení se pokusíme automatizovat. Tato práce směřuje k vytvoření asistivní technologie, kdy počítače programátorům s implementací pomáhají, ale nesnaží se je plně nahradit. První otázkou je, jak počítači sdělit, co po něm chceme tak, aby taková specifikace byla jednodušší, než kdybychom to sami naprogramovali. Druhá otázka cílí na možnosti uvažování o programech – jak analyzovat to, co máme a jak odvozovat, co nám chybí. V rámci navrhovaného řešení představíme Smalldb, které formálně popisuje chování entit webové aplikace pomocí konečných automatů, avšak nesnaží se zachytit každý aspekt takové aplikace. Pro věci, které je nepraktické formálně modelovat, se ponechají dobře definované mezery a ty se následně zaplní pomocí vhodnějšího nástroje. Formální model se tak začlení jak do architektury aplikace, tak do jejího běhového prostředí, kde tvoří API mezi business logikou a prezentační vrstvou aplikace. Takto integrovaný model navíc hlídá, zda implementované chování odpovídá tomu modelovanému. Abychom zodpověděli onu první otázku, je potřeba se podívat na samotný začátek procesu vývoje aplikace a identifikovat, jaké specifikace programátoři obvykle používají při návrhu a zda jsou tyto specifikace vhodné ke strojovému zpracování. Jednou z takových specifikací jsou procesní diagramy kreslené pomocí BPMN notace. Ty použijeme jako vstup nově vytvořeného STS algoritmu, kterému architekt nakreslí, jak budou uživatelé používat danou aplikaci, a STS algoritmus odvodí implementaci v podobě Smalldb automatu. Na závěr si představíme kaskádu, která je nástrojem pro automatizované sestavování programů využívající formální modely.

Klíčová slova: Softwarové specifikace, konečné automaty, webové aplikace, modelování business procesů, softwarová architektura, REST, MVC, SQL, CQS, BPMN.

Contents

List of Figures

Part I

Introduction

Chapter 1

Introduction

Once upon a time, there was a programmer. He was building various web applications; each was different, yet very similar. Create, Read, Update, Delete — the four operations he implemented for each entity. It looked like he could create a generic implementation shared by all the entities, but there were numerous small differences between them. Some entities missed one of the four operations, and some had a few additional operations. The generic implementation quickly became too complicated to maintain, polluted by many conditions and options. There must be a better way, he thought.

Would it be possible to generate or compose the application somehow so that we avoid both the overly complicated generic implementation and implementing everything manually again and again? Each entity in the web application follows a common pattern: It has some data stored in a database, a predefined set of operations, and few views present the entity to the user. Such a pattern is relatively simple and surely provides some space for automation, as there is plenty of repetitive pieces. Moreover, there are well-established frameworks that provide us with solid grounds on which we can build and a range of building blocks we can use. The question is how to describe such an application — thoroughly enough to generate it, but briefly enough so that it is simpler than the manual implementation?

1.1 Specification — Model — Implementation

The question of the software specification has two sides. The first is to describe the desired behavior of a program so that someone or something can implement it. The second is to reason about the implementation so that we can analyze it and verify its correctness against the specification.

We could ignore the second side and focus only on specifying what to synthesize but that would reduce software synthesis to a planning problem. While it is, undoubtedly, a valid approach suitable for many use cases, it seems to be somewhat limiting in the broader context of the software development process. Unfortunately, the verification of source code against a specification is a challenging open problem with a dedicated branch of studies, and a small web application would be hardly enough justification for such a tremendous amount of effort. So, what if we could simplify this problem? Instead of analyzing a Turing-complete language, it would be much easier to analyze, let say, finite automata; however, such an approach demands some trade-offs.

Because the finite automata (and other simple models) are much less expressive than typically used Turing-complete languages, we cannot simply convert our codebase into automata. At least not all of it. We can incorporate such a formal model into a business logic layer so that

it implements the behavior of the application entities. But even then, we will have to leave out some aspects and implement them using traditional programming; otherwise, the model would become too complex and incomprehensible. To keep the model sound, we will have to declare some fundamental properties of these left-out aspects, and the implementation will have to maintain these properties.

In other words, we intentionally leave some well-defined gaps in the formal model to simplify it, and then, we let programmers fill in these gaps using traditional informal programming techniques. As we will see in the following chapters, it is a rather small price to pay for a more practical formal model. During the development process, such a model will guide the problem decomposition. Moreover, it will identify and guard the scope of individual manually implemented pieces, reducing the total complexity, and improving code testability.

Note that in this approach, the model is intended as part of the implementation rather than a standalone description. This shift in the paradigm is an important detail. The close integration ensures that the implementation and the model are always in perfect sync because they are one.

Having a simple formal model integrated tightly with actual implementation means we can effectively reason about the implementation. The other way around, we can infer such a model and let it define the implementation. Suddenly, the effort to simplify software verification provided us with a way of creating software.

By this little detour, we obtain an interesting chain composed of the specification, the model, and the implementation. This chain reflects the software development process — first, we create a specification of the application, then we infer a formal model from the specification, and finally, we implement the gaps in the formal model and complete the application.

Part II of this thesis will explain components of this chain in detail, starting with the exact shape and properties of the formal model in Chapter 6, followed by a software architecture able to utilize this model in Chapter 7. Once we have the model defined and the properties of the required infrastructure identified, Chapter 8 will present an algorithm to infer the model from a specification, in this case, from a BPMN diagram. In the end, Chapter 9 will tackle another approach with a network of interacting state machines.

1.2 How to tell what we want?

In the previous section, we connected the specification with the implementation using a formal model; however, we entirely avoided the question of how the specification should look like and what it should tell. This question is the fundamental difficulty of software synthesis. A too abstract and vague specification is useless, but too detailed specification becomes overwhelming and incomprehensible. So, what is a good specification?

In general, a good software specification should make our lives better — programmers should better understand what to implement, their customers should understand and confirm what they ask from programmers, and users should receive a consistent product that makes sense as a whole.

When we look at what diagrams and schematics programmers use, we see many visual syntaxes, each specialized to describe a few specific aspects of the designed program. We have syntaxes specialized in representing class hierarchies, message sequences, database relations, use cases, data flows, and many more. None of them describe every detail and every aspect of a program — and that is a good thing. When programmers need to describe more than a single diagram can handle, they simply draw another diagram using whatever syntax they require — they create several partial specifications instead of a single complete specification. It is a process of creating projections of the desired product from many angles, rather than decomposing it,

not unlike tomography, MRI, or other imaging techniques, where slices, projections, and cross-sections contribute to the complete image.

In this thesis, we will follow this approach. We will use partial specifications to describe certain aspects of the application we wish to synthesize. Ideally, we will use the specifications programmers already use for software development so that we can significantly reduce efforts (and costs) required to utilize software synthesis. Chapter 2 will take a closer look at what inputs we have available, and Chapter 8 will present how to use one of these inputs in an automated way.

1.3 Software Synthesis as an Assistive Tool

The usual expectation from software synthesis is that it replaces programmers and saves us lots of resources. Thus we usually look for better ways to describe the behavior of a program, and unfortunately, we quickly reach a point where the specification is as complex as the desired program itself. Often the situation is even worse due to the additional abstractions.

Instead, the use of partial specifications (see Sec. 1.1) reduces possibilities of what we can synthesize automatically, but we can benefit from straightforward and comprehensible specifications. Thus, we need to adjust our approach to software synthesis. Instead of replacing the programmers, let us think of software synthesis as an assistive tool and let it help the programmers to do their job better.

In this light, we have two additional requirements for the good software specification (see Sec. 1.2). Aside from being human-friendly, it must be machine-friendly as well because we want machines to interpret it, reason about it, and finally, infer something useful from it. This implies the second requirement. The reasoning about the specification requires the specification to be formally sound — it must be consistent, without conflicts and ambiguities.

The concept of software synthesis as an assistive tool also allows us to limit the scope of the specification, both horizontally (features) and vertically (levels of abstraction), because we still can use the programmer as a backup plan for the cases where automated synthesis would be impractical. However, even in these cases, the models and specifications we use for software synthesis may help programmers to better identify what they need to implement or where potential bugs are likely to occur. For example, the algorithm presented in Chapter 8 verifies some semantic properties of the modeled business processes, and thus it can reveal mistakes long before implementation.

The formal model embedded in the application can also provide various metadata for other components, for example, navigation, access control, or API generators. These minor components are typically built around some kind of configuration rather than generated code, but it is often useful to generate such configuration from the formal model. Such an approach requires us to add the necessary metadata to the model, but then we have everything important in one place, and we eliminate the possibility of forgetting to update all the related components when the application changes.

As we can see, the approach to software synthesis presented in this thesis is designed to utilize established tools and frameworks as much as possible. The inclusion of formal techniques aims to enhance possibilities and open new options in software development.

1.4 Goals

The goal of this thesis is to open the door for intelligent and automated software synthesis; to establish a connection between the fields of Artificial Intelligence and Software Engineering.

The very first question we have to answer is how to tell the computer what we want to synthesize. The difficulty of this question lies in the level of detail — if we specify every aspect of the behavior, the specification becomes as complicated as the implementation itself but probably even worse. Instead of developing higher and more abstract languages, we shall explore possibilities of partial specifications to describe the applications we desire to synthesize. Naturally, such a specification will not cover every aspect of the application, but it still should spare the programmers a significant amount of work and improve the quality of the software products.

The second goal is to incorporate formal models into real-world applications so that we can effectively use the answers inferred from the partial specifications. Once we bridge the gap between the formal world and the real world, we can explore potential ways for further automation of software development and how to utilize existing planners and solvers.

This thesis aims for, from a human perspective, simple tasks. The goal is not to replace programmers but rather to provide them a tool that assists them with the boring and repetitive parts of their work so that they can focus on more interesting problems. Such a tool is expected to provide a useful, but possibly suboptimal, solution from provided inputs and identify missing pieces to be implemented by a programmer.

Because low-level code, like database access, can be sufficiently generic, we intend to work at higher levels of abstraction, using the low-level code as basic building blocks and connecting them to larger components.

1.5 Contributions

The main contributions of this thesis were published in peer-reviewed publications and form the following chapters:

- Chapter 6, State Machine Abstraction Layer (page 49, published in "Lecture Notes in Computer Science" [1]), introduces non-deterministic parametric finite automata as a formal model suitable for implementing business logic in a web application.
- Chapter 7, RESTful Architecture with Smalldb State Machines (page 59, published in "Lecture Notes in Computer Science" [2]), proposes application architecture and identifies its essential properties and features. The architecture combines the layered approach, command–query separation pattern, and RESTful principles to build a cohesive whole that smoothly incorporates the formal model defined earlier.
- Chapter 8, From a BPMN Black Box to a Smalldb State Machine (page 81, published in IEEE Access [3]), presents a novel algorithm to infer the implementation of a business process participant from a BPMN diagram. The expected use case is that a software architect and his customer draws a BPMN diagram of how users will use the designed application, and the presented algorithm infers implementation that fits the business process.
- Chapter 10, Cascade of the Blocks (page 121, published in IEEE Access [4]), defines a growing executable structure, loosely based on function blocks, which grows during the execution when each block can append additional blocks into the cascade — the executed program effectively writes itself as it runs. The cascade is intended as a machine-friendly alternative to software generators.

An additional contribution was the inclusion of the formal business process modeling in the exercises of the "Software Testing and Verification" course, which introduced our students to this area and helped us validate expectations about the understandability of the approach, as presented in Chapter 9.

1.6 Technologies for Web Applications

Traditional web applications based on server-side generated pages with HTML forms sent over HTTP may seem to be a dated technology, but it still is the foundation on which modern web stands. However, the very same HTTP requests are what powers the modern interactive facade of web applications, and when we open a web page, we still receive an HTML document from the server. So, to keep things simple, let us leave the bells and whistles aside for now and focus on the core technology.

In the following chapters, we will explore various techniques and tools to specify, model, infer, and synthesize a traditional web application. We assume that such an application consists of the following core features:

- An HTTP server that generates HTML web pages in response to HTTP requests.
- A web browser (a client) that requests the web pages from the server and presents the responses to a user.
- Links in the web pages for the user to navigate between the web pages, i.e., to request the next web page from the server.
- HTML forms in the web pages for the user to fill in additional parameters and submit them to the server, invoking an action that modifies the application state on the server.

Modern web applications often replace the traditional HTML forms with a sophisticated, highly interactive user interface implemented in JavaScript and using RESTful HTTP API [5] to communicate with the server. While it may look like a completely different thing, the underlying technology is still the same. The development of such a modern web application differs from the traditional one mostly because of the advances in software engineering over the last two decades. The architectural changes are rather subtle and not of much importance for our needs.

The applications we expect to implement are all kinds of information systems. Such an application consists of various persistent entities that typically represent real-world objects and their properties, for example, goods, customers, invoices, comments, and many more. The business logic of these entities is often trivial, mostly just to store data in an underlying database, but a few of the entities are overly complicated as they represent and control the core business processes of the users. It is these few entities that are the reason why we implement the application.

While the chosen technologies and applications certainly influenced many design decisions, the concepts and techniques we are going to introduce in this thesis are more general and surely suitable for other than the expected scenarios.

1.7 Thesis Structure

The thesis consists of four parts. The remaining chapters of this introduction investigate what resources we have available for software synthesis and what is the current state of the art. After the introduction, the second part presents the core of this thesis — the use of nondeterministic finite automata as a formal model suitable for web applications and how it can enhance the architecture and the development process by extracting the complexity out of the implementation, and also, how it can automate some of the development steps in combination with business process diagrams. The third part of this thesis will briefly present possibilities available thanks to the integration of the formal model in web applications. Finally, the fourth part concludes this thesis with some practical experience and remarks.

Chapter 2

Programmer's Inputs

Before programmers can create a program, they need to know what the program should do, what is its purpose, what tools are already available. They also need to know what their customer wants and needs, which usually is not the same. The programmers need inputs to produce the outputs, the program. So, what inputs are available to the programmers?

2.1 Use Cases and Requirements Management

A use case represents an intention of a user; it describes a flow of events and tasks towards fulfilling the intention. Use cases are often represented as high-level scenarios with actors, tasks, and goals. These scenarios are written in natural language with no ambition of formal specification. The purpose of the use cases is to identify and communicate users' goals and requirements.

The requirements introduce constraints on various aspects of the designed software. These constraints are also mostly informal; however, some of them we can express as tests. Then, we can measure and evaluate that an implementation meets the constraints. Such an approach is called Test Driven Development. The tests tell us whether the software is good enough, but unfortunately, they do not tell us how to implement the software.

The requirements always accompany software projects in vast numbers, especially in the early stages of a project. Keeping track of them and maintaining at least some order is a challenging task. Requirements management provides us with many methodologies and techniques to deal with such a task [6]. However, the focus of this thesis is in automated implementation rather than requirements management; therefore, let us assume that we already have our requirements sorted and analyzed. What useful data does a programmer receive?

We traditionally divide the requirements into the following (nonexclusive) groups:

1. Functional requirements
2. Nonfunctional requirements
3. User interface requirements
4. Business requirements

Such a taxonomy is rather administrative and useful more for project management than for a programmer. From the programmer's point of view, there are two important aspects of a requirement: How the requirement is defined, and how to verify the requirement satisfaction. These two closely related aspects then rearrange the requirements into the following groups:

1. Formal definition: The requirement is formally defined, and thus we can formally reason about it and prove or verify the correctness of the implementation. Such a formal definition may also be in the form of a process diagram, a statechart, or written in a domain-specific language, not only mathematical expressions. If the formal definition is in a machine-friendly format, we should be able to infer a skeleton of the implementation or a model usable at run-time automatically. Unfortunately, the required tools are not always available, or they do not even exist. We will introduce some in Chapter 8.

2. Defined by example: The expected behavior is defined informally and supplemented with examples of the program's valid inputs and outputs. In fact, we assume the intended behavior is a generalization of the provided examples. In some cases, the examples are available from other sources than the requirement itself (e.g., an API specification) or they are easy to infer. We can create automated unit tests using the provided examples to validate some of the program's behavior.

3. Metric constraint: The requirement is defined using a metric and a constraint on its value (e.g., response time). We can measure the metric and check that the constraint is satisfied.

4. User interface design: Requirements on user interface usually overlap with the groups above but do not fit well into either of them. Programmers use wireframes and graphical designs of the user interface to develop the user interface itself, but the quality of such source data is rather poor to be considered a formal specification. However, with proper tools, we could extract some useful metadata from the wireframes, e.g., data structures required and provided by HTML forms or dialogue windows.

5. Vague but testable: Such requirements are typically nonfunctional, often regarding usability and accessibility to the users. There is usually no potential for automated testing; however, we can validate such requirements on living subjects or using user behavior models, e.g., using usability testing methodologies. Sometimes we can measure some loosely related metric (e.g., time spent on a page) to determine whether we fulfilled such a requirement.

6. Vague and not testable: Such requirements provide context and represent wishes and expectations of our customer. Some of these requirements affect the architecture of the program or the chosen technologies. We usually say "Yes, of course." and carry on.

These six groups provide us with an insight into what kind of (meta)data a requirement provides and how we can process such a requirement. A precise requirement leaves less space for misunderstandings but requires more effort to specify. A vague requirement is easy to make, but difficult to verify whether it is fulfilled. To find the balance, we often rely on intuition and previous experience. The requirements then stand on an unspoken, implicit context, which users, developers, and their customers may not share entirely, and which computers lack completely.

If we want even to consider automated processing of the requirements, we need to represent them in a way that is not only human-friendly but also machine-friendly. That means the representation of a requirement must have a predefined structure and semantics known to both humans and machines. The software developers' need to understand each other gave rise to standardized visual languages (e.g., UML [7], BPMN [8], IEC 61131 [9]). Such a visual language has well-defined syntax and semantics; however, to make it machine-friendly, we need one final piece — a parsable representation. To do so, we need to use editors [10, 11] that use semantic elements, like nodes and edges or relations, rather than graphical primitives, like lines and circles. In case we wish to use words rather than diagrams and schematics, we may use (or

create) a domain-specific language (DSL), which allows us to define the behavior or properties of an entity, but in a way comprehensible by users instead of programmers.

If we look from this perspective onto the six groups mentioned earlier, we may wonder whether the requirements in each group are machine-friendly or not, and what kind of data we can automatically extract from them.

The formally defined requirements are the most promising group as long as we use a machine-friendly representation of the requirements. Aside from obviously useful and formal ways, like using a DSL or statecharts, we may also extract a formal specification from a seemingly informal requirement. An attempt to formally define such a requirement will fail due to the overwhelming complexity of the definition; however, if we formalize only some aspects of the requirement, we may obtain useful machine-friendly data while the requirement stays without any significant complications. Often we just need to use a proper editor and notation for diagrams. We will return to this topic in Section 2.4 and Chapter 8.

The requirements from the "defined by example" group have the potential for automated testing (e.g., using continuous integration [12] and test driven development [13]). The question is how to represent the examples so that we can test them. In the case of HTTP API or algorithms, we can simply provide examples of input and expected output data, and then create tests that will use these examples. The documentation then may provide executable or at least tested examples.

Another approach is to use simplified sentences in a natural language to define a test case, then map these sentences to function calls. These behavioral definitions [14] are written in a business language and then converted to a programming language using regular expressions or similar pattern matching technique as we will explore more in Section 3.14. This approach is no different from writing a formal definition using a DSL, but it shows that a DSL can be user-friendly.

The user interface requirements are a rather diverse group where we can combine various approaches to obtaining machine-friendly data. Also, these requirements are often part of other requirements; for example, business processes add requirements on navigation and transitions between screens of the application. It may be useful to enrich user interface designs (e.g., wireframes [15]) with additional metadata or extend the semantics of already existing elements. Then we can collect various data from the user interface designs and crossreference them with other requirements or use them to define entities required for interaction with the user. We will discuss the possibilities in Section 2.2.

As we can see, the software specifications and requirements provide not only inputs for the programmers, but also a potential for automated software synthesis hidden in small, machine-friendly pieces.

2.2 User Interface and API

User interface (graphical GUI, or command line CLI) is usually the most apparent input a programmer gets. Wireframes or similar drafts of the future user interface are common tools programmers use when starting a new project or feature, and the design of the user interface and its usability form a whole branch of the software development industry. The question is, what such a wireframe tells us about the application aside from its looks?

Each button of the application represents an action the user can do, and thus a piece of code that implements the action. Each input field represents data the program needs to accept, process, and likely to store. Many other widgets represent data the program needs to display, and to do so, it needs such data to load and process as well. By collecting these fragments, the user interface provides us with hints on how the internal data structures will look like and what

actions the application will perform.

Application interfaces (API) provide us with similar data in a more direct way — the requests and responses of the API are input and output data structures, which often reflect the internal data structures.

Of course, the data presented or obtained by the user interface and API typically do not match internal representations of the application entirely, and thus various transformations are required. The most common transformation is conversion to and from the text, typically when presenting numbers and dates. However, such a transformation does not change the semantics of the data structure, and therefore, the metadata obtained from the user interface are still useful. Additionally, the kind of GUI widget used for a particular field tells us what kind of data we expect there.

The user interface as whole also provides us with certain logical constraints between the data fields. For example, if we ask the user to provide departure and arrival dates, we can safely assume the departure date must be earlier than the arrival date. Unfortunately, it is not possible to express such constraints with most visual GUI editors, as well as attaching and exporting metadata from wireframes is a rather unexpected feature.

Web applications and especially information systems usually consist of a number of trivial entities with little to no business logic attached to them. The persistent representation (e.g., in an SQL database) is then trivially mapped to their visual representation in the user interface, and the application provides the user with the basic CRUD operations (Create, Read, Update, Delete). Such entities are good targets for automation because of their repetitive nature, and, in fact, many frameworks provide generic tools for this task(e.g., [16, 17]).

2.3 Data Structures

Every (useful) program has some inputs and outputs. Every nontrivial input and output requires a description, a data structure, how the program represents the data on the input or output. Many programs also persist some data, and the structure of such data must be defined as well, often in the form of a database schema. Once the programmer collects and defines these data structures, a large portion of the future implementation becomes obvious. Additionally, such data structures may contain additional metadata, e.g., annotations, which can specify what kind of form fields the application should present to the user, and what are some of the constraints on the data the user enters.

In case of simpler information systems, there is usually not much left to implement. Such an application provides a user interface to perform CRUD operations, as we already mentioned in the previous section. It converts the persistent data structures to output data structures for a user to see them, and then it converts the input data structures to the persistent ones shall the user decide to modify the data. The conversion between the input/output and the persistent data structures is usually trivial as well as the implementation of the CRUD operations, and modern frameworks are very helpful in this area.

If we take a look at the data flow between the layers of the application, from the presentation layer down to the business logic layer and the model layer, we will notice that each boundary between the layers requires some data structures to exchange the data. Therefore, outputs of one layer become inputs of the next and vice versa. In other words, if we know that one component expects a given data structure, then there must be another component that will provide the expected data structure.

2.4 Business Process Diagrams and State Machines

Business process diagrams describe how people act and interact to achieve a given goal. Such a diagram does not have to include people only; it can include robots, software systems, and other automated entities as well. The diagram can describe interactions between users and the designed application so that a software architect understands what user needs, and developers understand the context of what they implement.

While the business process diagrams are a useful tool for software design and architecture, they are not very practical to describe the implementation itself. The usual approach is to draw the diagrams to clarify what the software should do and then implement the desired behavior manually. There are some technologies to make process diagrams executable (e.g., BPEL), but their use in small applications is practically nonexistent.

Another syntax, which focuses on run-time, is a state diagram. It represents a state machine (a finite automata), which describes the behavior of a single entity, and as such, it is straightforward to implement it. When done properly, we can have both a formal description of the behavior and a practical implementation. More sophisticated approaches include Petri Nets and workflows, which can describe multithreaded behavior in a similar fashion as state machines.

As we will see in Chapter 8, it is possible to automatically extract state machines from business process diagrams [3]. Such a state machine then describes the behavior of a given participant in the business process. The programmer then "only" implements transitions of the inferred state machine.

2.5 Algorithms

When speaking about algorithms in the context of application development, we usually mean a piece of complicated code wrapped into a reusable library. Such a library provides tools that have inputs and outputs of a predefined structure and a well-defined relation between them. The programmer's knowledge of the libraries (supported by reference manuals) is what allows him to choose the components and then compose them into the application.

Human programmers also can extrapolate the knowledge of the libraries and describe a library required in a particular situation when any of the known libraries does fit. While the composition of the known libraries is a more-or-less solved planning problem, the ability to extrapolate a missing piece is out of reach of the modern automated tools.

Unfortunately, the application of the planning approach is not as easy as it theoretically looks — the planning in general stands on two concepts: a state space and operators that alter state. We describe the initial state and the desired state, and then we use the operators to navigate the state space, searching a path from the initial to the desired state. The operators are pieces of code we reuse from a library or write. The state is represented by data structures (variables) and other resources (e.g., open files) available at the moment, where some of these become the output of the program. The task is to find a sequence of the operators to convert the initial state to the desired state. The exponential complexity of such a task is not the main issue here; the main issue is with the consistency maintenance between the formal operators and their real implementation. The human programmers perform this planning task intuitively as they (hopefully) understand the semantics of the algorithm, and IDE or compiler helps to fit the inputs and outputs together using type systems and API definitions (i.e., method prototypes). The automated planner does not have the intuition; instead, it requires a formal description of the operators, that is difficult to infer from a Turing-complete code that implements the operators.

We should be able to describe the state using various conditions and constraints without too

much effort, as we already do, for example, using various type systems or in the form of foreign keys in an SQL database. The question is, how to describe the semantics of the operators in a machine-friendly way without writing the code twice.

2.6 Programmer's Experience

The least obvious input to the programmer's work is his own experience and education (the experience of others). This experience includes solutions to various problems and tasks solved in the past, as well as unsuccessful attempts to solve them. The successful solutions are available to us in the form of existing source code but without any reference to the problems they solve. The unsuccessful attempts are nothing more than unpleasant memories with little to no possibility of formal analysis.

The reuse of the experience for the new application involves generalization of the old solution and then its adaptation to the new problem. The generalized experience is sometimes captured in the form of design patterns [18]; however, such a design pattern is a very vague representation of the former experience, and it requires notable context to be useful.

The description and formalization of the programmer's experience alone is no small feat; the difficulty of the automated generalization and reuse of such experience is far beyond that.

2.7 Conclusion

As we identified the programmer's inputs, we also identified potential input data for automated software synthesis. Most of the programmer's inputs are informal, vague, and depend on the unspoken context. However, we may find useful fragments of the specification in a machine-friendly format, that we can use for automated processing. Such an approach will not allow a complete specification of the desired application; however, it should save a significant amount of work and reduce the surface for possible bugs and mistakes.

An artful combination of these machine-friendly partial specifications may provide us with a framework that speeds up application development. Even if we do not have the entire application formally specified, which would be likely as complex as its implementation, the machine-friendly partial specifications may cover a large enough portion of the application but in limited detail (which programmers can provide manually) and for a substantially lesser effort. The following chapters will explore some of the possibilities in this area and present a practically useful approach.

Chapter 3

Software Specification and Synthesis: State of the Art

The idea of converting one software representation into another is as old as the software itself. The need to write programs in a comprehensible way combined with the laziness of programmers leads to the creation of many tools that compile, convert, or generate code from more "friendly" representations. Software synthesis emphasizes the code generation approach, rather than conversion of one code into another, but the boundaries are blurry, and possibilities are vast. However, before we can synthesize software, we need to specify what do we want, and to do so, we need to model the behavior and reason about it. In this chapter, we briefly survey various approaches related to software specification and synthesis — some of them were an inspiration for this thesis; some will be used in later chapters.

3.1 Generics and Templates

A relatively easy way to create multiple similar programs is using generics or templates. A well-written algorithm may work on various data types. The basic structure of a user interface may be used for various entities (as long as entity-specific views are provided). This approach is well established in most modern programming languages, like C++, Java, or Rust, and it is often used to create versatile and powerful libraries effectively.

However, the template-based approach requires a programmer to create the templates in advance with a noticeable investment of effort. This requirement is not surprising, but it is essential to realize that all the work is still up to the programmer — templates and generics only provide a more effective way of writing code.

The templates are also used in some frameworks, like Ruby on Rails or Symfony (using Maker Bundle), to prepare the initial implementation of individual components, and then the programmer modifies the pre-generated code to his needs. In this case, the templates are, in fact, code generators, but the principle is the same.

3.2 Domain-Specific Languages

The opposite approach to the generics is in creating a Domain-Specific Language (DSL) [19]. Instead of writing generic code, a programmer provides tools that users (or other programmers) can use and combine as needed. A DSL may use a specialized (and often simplified) imperative programming language (e.g., PostScript, XSLT), or a declarative description of the desired state

(e.g., Alloy [20]), query (e.g., SQL, regular expressions), or a process (e.g., Make). As we mentioned in Section 2.1, such a DSL may be also a visual language utilizing diagrams and schematics, for example statecharts [21], UML [7], BPMN [8], or IEC 61131 [9].

From this thesis point of view, the imperative DSL is merely another way of writing a program, but the declarative DSL is a formal machine-friendly model we can use as input for the software synthesis. Either way, the both are useful in reducing the effort required to create and maintain an application.

3.3 Macros, Decorators and Annotations

One of the possible ways to implement a domain-specific language (DSL) is to extend the language we are already using in the application. A trivial way is to define an API in a way it lets us define the domain knowledge briefly and elegantly. Typical examples of such an approach are builder objects [18]. When the API is not enough, we need to extend the syntax. Some languages provide us with macros, either as part of the language (e.g., Rust [22]) or via a preprocessor (e.g., C). Other languages are based on macros entirely (e.g., TEX or M4), and we create DSLs in them without even realizing it.

Annotations are metadata attached to the code. Some programming languages support them directly (e.g., Java, C#), some support only the features required for their implementation (e.g., PHP via comments and reflection), some implement them via macros (e.g., Rust). A typical application for annotations includes ORM, where objects are annotated with their respective database representation so that the mapping can be done automatically. Routing and HTTP API definitions are yet another typical applications, in which the underlying framework can map HTTP requests to the annotated method calls. Static code analysis also uses annotations to denote intentional features in the code, e.g., thrown exceptions or coding style violations, to suppress false warnings.

Some languages support decorators at syntactic level (e.g., Python), where an annotation defines the use of a decorator and the decorator wraps the original method to extend or modify its behavior. A similar effect may be achieved using macros (e.g., Rust), which may replace or modify the decorated function using AST transformations during compile time.

Both macros and annotations are useful tools used in metaprogramming. While the annotations provide only the configuration and input data that compiler or framework can interpret, macros can also provide a mechanism to incorporate the generated code into the hand-written code base.

3.4 Moldable Tools

Domain-specific languages, visual representations, and high-level abstractions are something that modern debuggers and development tools, in general, do not handle very well, simply because authors of such tools cannot predict every application-specific visualization or representation a programmer would like to use. The idea of moldable tools [23] expects programmers to create extensions of the development tools so that they provide the application-specific representation instead of raw low-level data.

An alternative approach is to embed the necessary visualizations into the application itself, rather than into other tools effectively creating new development tools. While such a tool is not integrated with, e.g., a debugger, it is possible to use such a tool in a production environment or to visualize logged data and events. In other words, the development tool integrated into the application may be useful to others than the developers only.

Fortunately, these two approaches are not mutually exclusive because it may be possible to reuse a custom visualization in both the development tools and the application. Therefore, when development tools adopt the moldable tools approach, we may refactor the custom tools into plugins and integrate them into development tools.

In web development, there is a middle ground between extending the developer tools and creating the standalone custom tool. Most web frameworks have some way to report problems and present debugging data directly in the application. Such a tool is usually in the form of a toolbar that the framework adds to the web page. A nice example is the Symfony Profiler[1] that, in addition to the toolbar, provides an advanced browser to a large amount of debugging data and logs it collects, and, most importantly, it provides a relatively easy way to collect and present custom data in a custom way.

3.5 Components and Web Services

In the latest decade programming paradigm shifted from object-oriented approach to component-based design. The object-oriented languages are great for creating components, but not that good when components are to be connected together. In order to achieve component-based design an additional infrastructure must be added to plain objects.

Generative programming [24] then extends the component design with auto-configuration, where prepared components are fitted into prepared places in the skeleton of the resulting product, effectively dealing with large configuration space in the process. This approach brings large variability of the generated programs, but requires careful and domain-specific preparation.

Automated web service composition [25] is a similar approach, where each service is a standalone component that provides a well-known API, and we connect the services to achieve our goal. With a growing number of the services, a question of whether we can automate their orchestration appeared. After two decades of research [26], various solutions [27] have been proposed based on technologies like WSDL, WS-BPEL, and RDF — the technologies that provide standardized API, executable business process description, and semantic data, respectively.

The difficulty with web services lies in the complex infrastructure required for their use. Furthermore, the heterogeneous nature of the web service architecture introduces numerous small incompatibilities between the services with which we have to deal so that we can connect the services successfully. On the other hand, web service architecture enables us to scale and separate services to provide otherwise unreachable solutions, and thus, the price of web services is often worth paying.

Luckily, the technical details of how to connect the web services (i.e., web service orchestration) are not relevant for the needs of this thesis, and we gladly let this task to other projects. When we overlook the technical details and focus on the high-level composition of web services, we find that the most web services frameworks provide tools to implement and connect web services but not to decide what services and how should be connected. The most techniques used for web service composition can be used for traditional programming languages because a web service invocation is conceptually similar to a method call.

The task of choosing and connecting the services is usually left on the programmer, leaving space for automation. Thus, the problems like QoS-aware[2] Web Service Composition and various approaches to their solutions, e.g., [28, 29], provided us with inspiration for this thesis.

[1] https://symfony.com/doc/current/profiler.html
[2] QoS: Quality of Service

3.6 Dependency Injection Containers

A consequence of object encapsulation is that the object does not care about its surroundings. Such an object receives its dependencies via constructor arguments or setter methods and uses them as it needs. The instantiation of the object, as well as its dependencies, is not a concern of these objects because they receive ready-to-use instances. Therefore, we need a dedicated tool to manage these dependencies and their instantiation — a dependency injection container [30].

A dependency injection container is typically composed of two components: a container builder and the container itself. The builder loads configuration of what services (objects) the container should provide and what are their dependencies, then it compiles a container. The compiled container then provides services in run time as needed. Typically, such a container is a generated class with many factory methods that instantiate the desired service and inject its dependencies into it.

As programs grew in complexity, the configuration of simple containers become too hard to maintain, and automated solutions were developed. Simple type-matching algorithms are used in most frameworks to determine which dependencies should be injected where; however, these algorithms typically do not handle well situations, where more candidates are available [31]. On the other side, too complex logic is not desirable here, because a programmer must be able to predict which component is going to be used easily.

3.7 Specification Morphisms

Mathematically-based techniques, algebraic specifications, and their morphisms can provide a compelling way of programming [32], using theorem-proving techniques and allowing the creation of formally correct programs.

Such techniques usually do not focus on the connection between formal specification and its real-world representation; therefore, appropriate domain-specific language must be used.

From a practical point of view, the algebraic approach requires very exact specifications and in-depth knowledge of both formal methods and the used domain-specific language. Unfortunately, the requirement of the exact specification is in direct conflict with rapid prototyping and a client who does not know what he wants.

3.8 Graph Rewriting

When a program or its specification is represented as a graph, various techniques of manipulating the graph are available [33, 34, 35]. However, these techniques have much broader application as the data structures do not have to be explicitly represented as graphs. Various data transformations may be described as graph rewriting, and the graph rewriting techniques may provide inspiration and theoretical framework for many nontrivial operations.

Unfortunately, graph rewriting does not provide practically useful tools because of highly diverse graph representations, where each program or library uses different graph representation, and thus programmers end up implementing graph rewriting algorithms for the particular task again and again.

In Section 8, we will analyze business process diagrams to synthesize state machine. Such a task may be seen as graph rewriting; however, the described approach will analyze the existing graph, and then generate a new graph from collected constraints rather than applying graph rewriting rules. As we found out, it is rather challenging to preserve various semantic information during such rewrites when the input and output graphs are too distinct. In such cases, the separate steps of analysis and synthesis seem to be a more comprehensible approach.

3.9 Functional Block Programming

Functional Blocks is a paradigm widely used in practical implementations, especially in industrial automation. It was standardized as IEC 611 31 [9], and it is one of the available techniques to program PLCs (Programmable Logical Controllers).

The block programming approach [9] was a great inspiration when creating the cascade that Chapter 10 will present. The main difference between the cascade and functional blocks is that functional blocks are of static structure with dynamic data, but the cascade is growing during the evaluation, and the data are static.

There are many similar approaches, like Unix pipes or publish–subscribe architecture, that are mostly the same in the basic principle. However, some other approaches, which are visually very similar, have significantly different semantics, like behavior trees (see the next section).

3.10 Tensor Flow

Tensor flow [36] is a machine learning framework that utilizes a flow graph of operators on tensors (n-dimensional arrays). The flow graph is prepared in advance, optimized, and then executed on data. It supports parallel concurrent execution on overlapping subgraphs, and each operator may have multiple implementations for various devices (e.g., CPU, GPU, or a cluster of TPUs — Tensor Processing Units). Such a design enables powerful optimizations and parallel computation on various hardware without any code modifications.

The overall concept is similar to the Functional Blocks (see Sec. 3.9), but to effectively manipulate large datasets, Tensor Flow supports mutable variables so that the operations can be done in-place. Additionally, Tensor Flow supports various queues, so that programmers can arrange machine training process according to their needs, rather than using a predefined structure.

The flow graph is compiled from source code and optimized in advance of the execution and can be cached for later use. The compilation may convert language constructs (the source code) into a flow graph executed by Tensor Flow, or the executed program can prepare the flow graph using library calls, and then later execute it on the data; this approach is similar to the prepared statements in SQL.

The Tensor Flows approach is exciting and useful for data manipulation, like in the machine learning applications for which it was designed. In Chapter 10, we will present a similar approach designed for application composition that focuses on run-time flexibility instead of raw computational power on large datasets.

3.11 Finite Automata

Finite automata [37, 21], also called state machines, are a simple yet powerful construct that represents states and transitions between them. The concept dates back to 1955, and thanks to its simplicity, it has been in use ever since. “Recently,” the state machines got standardized as part of UML specification [7].

The classical use of finite automata is in lexical analysis, where the state machine accepts characters and produces tokens for further processing. However, in this thesis, we use state machines to represent application behavior. Business processes typically require stateful entities, but the behavior of such entities is often rather simple. For example, if we send a package via our favorite delivery service, the package tracking application will show us various stages of delivery, from being accepted to awaiting a courier to being delivered. The state machines are a natural and convenient tool for such cases as we will explore in chapters 6 and 8.

The user interface is another practical application for finite automata. For example, Qt State Machine[3] (part of Qt framework; based on State Chart XML [38]) utilizes hierarchical state machines [39] to control state and animations of user interface widgets. The hierarchical features provide an effective tool to mitigate problems with the combinatorial state explosion. Grouping similar states together allows us to simplify various conditions regarding widgets shared between multiple states, e.g., to show the correct step of a wizard or to enable the correct group of widgets.

From a formal point of view, we can see a finite automaton from two different perspectives: as a model (input) or as a result (output). When we use finite automata as a model, we can reason about its properties and what could or could not happen [40]. And then, when we are happy with the model properties, we can use the automata as an executable program (see Chapter 7).

The other way around is to see the finite automaton as a result. We specify constraints of our model, and then we let a solver, e.g., Alloy [20], to find a fitting model. Alternatively, we may process another kind of input to infer a state machine from it — for example, to extract a model of a participant from a business process, as we will see in Chapter 8.

3.12 Petri Nets and Workflows

If we swap states and transitions in a finite automaton, we receive an activity diagram, where nodes represent activities (tasks) and arrows represent states [41]. Depending on the context of our model and what we need to represent, activity diagrams may be as useful as the finite automata.

Petri Nets [42, 43] provide a formalism that allows parallel evaluation in addition to finite automata [41]. Instead of a simple state, the Petri Nets use the concept of places and multiple tokens that mark active places. The second type of nodes (Petri net is a bipartite graph) represents transitions that allow synchronization and splitting the tokens to multiple paths.

"Workflows" are typically executable practical implementations of Petri Nets. One of the frameworks providing such functionality is Symfony Workflow[4]. It provides a model that guards transitions and generates events that can be further processed by other components. However, Symfony Workflow fails to enforce the model constraints because it does not encapsulate entities and the ORM-based persistence layer properly. Chapter 7 will address this issue and provide architecture that expects the use of such a model from the beginning.

3.13 Behavior Trees

The use of finite automata to model complex behavior results in rapidly growing number of states and transitions because the basic finite automata do not provide any features to group similar behavior together, and the maintenance of such a large graphs soon becomes unsustainable. Hierarchical state machines [39] provide a solution to this problem by grouping states into super-states (like the inheritance in the object-oriented world), and by replacing several transitions with a single arrow between the super-states.

Behavior trees [44] combine the concept of hierarchical automata with the concept of decision trees and add some run-time semantics. The leaf nodes of a behavior tree represent actions, and intermediate nodes decide which action should be currently active, and the root node provides ticks to control the execution. The intermediate node may select a child node to execute (selector nodes), or they may select all their child nodes in sequence, one with each tick (sequence node), or they may tick all child nodes at once (parallel node). Each node returns its state (success, failure, running) so that the parent node can keep track of where to distribute the next tick.

[3] https://doc.qt.io/qt-5/statemachine-api.html
[4] https://symfony.com/doc/current/workflow.html

Behavior trees are widely used in the game industry to implement artificial intelligence for NPCs[5] because of the run-time semantics (the ticks). Their application in business process modeling seems to be rather impractical as this more event-based use case favors workflows and state machines.

3.14 Behavior-Driven Development

Behavior-Driven Development (BDD) [14] is a software development technique that encourages communication between developers and non-technical participants in a project (customers). To do so, it facilitates simple natural language constructs, i.e., simple English sentences, as a domain-specific language to communicate the behavior between all participants of the software development process.

BDD is similar to Test-Driven Development (TDD) [13], that teaches us to write tests first, and then create the implementation to make the tests pass. In BDD, the tests are implemented using the domain-specific language, so that the test is verifiable by non-technical participants. Developers then define a mapping from the DSL to a traditional method call, e.g., using regular expressions, making it compatible with usual unit testing tools.

The use of simple English sentences as a domain-specific language is an ingenious way to implement the machine-friendly specification in a human-friendly way, as we discussed in Chapter 2. The use of natural language only seems to be an arbitrary limitation, and it might be interesting to extend this approach with plots, schematics, and other features.

3.15 Literate programming

Literate programming [45] is a concept of multiplexing documentation and source code in a single file. A designated tool then demultiplexes the file into separate documentation file and source code file for further processing, to build the documentation, and to compile the program.

The advantage of this approach is in keeping the code and the documentation together to ease the maintenance. However, the location is the only relation these components share, and thus, maintaining consistency is an issue, similarly as keeping comments in the code up to date.

While the concept is simple, the use is surprisingly complicated because such a single file contains at least three different syntaxes — one for the documentation, another for the source code, and the third for the multiplexing itself. An editor for such a file then needs to understand the multiplexing mechanism and be able to switch between the interleaved files to provide correct syntax highlighting and code analysis. Back in the 1980s, when this concept was introduced, advanced editors (and IDE) were to be seen, and thus, this complexity was not an issue.

Another approach to literate programming is to integrate the documenting support directly into a programming language. For example, Haskell [46] allows us to store the source code as a LaTeX document with code blocks inside and with `*.lhs` file extension instead of the usual `*.hs` ([46], pg. 134). This avoids the complications with the multiplexing because the file is a valid document as well as the source code, but it requires support in the programming language itself.

A successful application of literate programming is the notebook interface first introduced by Wolfram Mathematica in 1988 [47] and widely used in many mathematical applications up today, and it found its way into other areas as well, e.g., Jupyter Notebook for Python. It consists of a rich text editor that includes executable blocks of code and optionally shows the results of the blocks, numerically or graphically. Mathematical calculations and proofs are, unlike traditional

[5] NPC: Non-Player Character.

programming, very compact and complex expressions that require context and substantiation. The results then need a proper presentation in the form of visualizations and plots. Such a computation is more of a document than a program, and therefore the literate programming via the notebook interface is a fitting approach.

In the context of this thesis, we see a use case for literate programming in precise specification of small but complex fragments of a program that can be used without further manual intervention.

3.16 Intentional Programming

Similarly to the literate programming, intentional programming [48] encourages the programmer to use multiple notations, but instead of multiplexing multiple files, intentional programming mixes the notation on the level of abstract syntax tree (AST). This approach enables the programmer to use the best suitable notation for every piece of code, for example, to use a mathematical formula that is precise both semantically and typographically, or to use tables and visualizations directly in the source code.

Intentional programming also experiments with an idea to mix various programming languages for the best fit with each problem. However, this would make such code bases much more difficult to understand, and with the modern languages, it is not feasible anymore. Intentional programming was proposed in 1995 [49] when most languages were simple imperative or functional with little differences, and object-oriented programming was still very young. Since then, programming languages explored event-based parallel programming (e.g., async/await constructs and promises in JavaScript), automated memory management (e.g., various garbage collector), and thread safety (e.g., variable ownership in Rust) to the point that the semantics of similar constructs differs too much to share a single module. Therefore, interoperation between the languages is usually defined using C-style function calls (run-time binary interface) and dynamic linking of the modules, or they are all compiled into the byte code for the same virtual machine (e.g., for Java Virtual Machine).

Nevertheless, the idea of using proper notation in proper places is worth implementing. As discussed in Chapter 2, we can convert various files in machine-friendly formats, and compile them into our applications. We may not include the diagrams and tables into our source code as seamlessly as intentional programming proposes, but we can still use them with acceptable comfort.

3.17 Aspect-Oriented Programming

Aspect-oriented Programming (AOP) pattern [50] allows us to extend application components in a way orthogonal to traditional modules, with a focus on cross-cutting concerns. The extending code typically implements an aspect of the application behavior using many small pieces of code scattered across many components and thus difficult to track and understand. The goal is to avoid any modifications of the extended component and to keep the extending code in one place to avoid the scattering.

Some AOP frameworks (e.g., AspectJ) implement the aspects at the method level, where the AOP framework allows us to add a method into a class or to call some code before or after a method call. This allows us, for example, to add logging or access control to certain components that do not expect nor implement such behavior.

Other frameworks (e.g., Symfony and HTML 5) use events and listeners to process requests (e.g., HTTP request or a click on the web page DOM element). An event is an object that holds data (both the request and response) and is passed to all registered listeners. Each listener can

modify the event data or stop the processing. To implement an aspect, we register our listeners to alter the events as necessary. Some frameworks (e.g., Laravel) achieve a similar effect by composing "middlewares" [51] that individual modules provide and which behave in a fashion similar to nested decorators or a pipeline.

From the software synthesis perspective, the aspect-oriented approach is rather difficult to grasp because it solves a problem orthogonal to software synthesis. When we generate software (see Section 3.19), we can easily include the scattered code to implement the aspects because there is no maintenance cost for the generated code — we simply generate the code again [52].

3.18 Planning

Planning [53, 54] is a process, where a planner looks for a path of operations leading from an initial state to a given target state. To do so, the planner requires a description of the world and operators to alter the state of the world. Possible sequences of the applied operators generate state space, and the planner's task is to effectively search this typically huge space.

There is exhaustive number of planning strategies and algorithms and there is not enough pages to describe them all. The basic approach on planning is to describe the world using predicates of the first-order logic, and let operators to change their valuation. The planner then searches for the right sequence of the operators.

For more complex tasks we can utilize hierarchical planning where a complex abstract task is recursively divided into smaller more concrete ones until we have only executable operators in the leafs of the tree of the Hierarchical Task Net [55]. This approach is, in certain aspects, similar to the behavior trees described in Section 3.13. Also, there is ongoing research of the hierarchical approach on automated web service composition [56].

GraphPlan [29] presented a breakthrough in planning. It constructs a planning graph in advance of searching for a plan. GraphPlan then navigates the planning graph towards the defined goal. Once it reaches the goal, it walks the planning graph backward to construct the plan. Such a planning graph encodes the planning problem and explicitly provides useful constraints to reduce the search space, and, unlike state-space graphs, it can be built in polynomial time and space [57]. The primary contribution of GraphPlan is that it enables the use of heuristics to direct the search, and thus it converges towards the solution instead of blindly trying all possibilities.

In Chapter 12, we will take a look at the relation between the existing planning techniques and software synthesis. The overall idea is to interpret methods and functions as operators for a planning algorithm and somehow describe what we want to achieve. Such an idea is not new; there are adaptations of planning algorithms for web service composition [58, 29]; therefore, this approach should be feasible for traditional local programs too. The remaining question is how to specify what we want to plan.

3.19 Generative Programming

Generative Programming [59] builds on top of object-oriented programming (OOP) and aims to address some issues the OOP does not solve. While OOP provides encapsulation via classes and objects, allows generalization via inheritance and polymorphism, it does not deal well with reusability, adaptability, and horizontal scaling, which includes separation of concerns and code tangling — especially in later stages of development, the originally well designed hierarchical code deteriorates and becomes "tangled" because of the changes and unexpected added features. Moreover, traditional programming techniques fail to extract concepts from the programmer's

head into source code, because general-purpose programming languages are designed to express how stuff should be done, not why.

To solve these issues, generative programming utilizes concepts like generic programming (see Sec. 3.1), domain-specific languages (see Sec. 3.2), and aspect-oriented programming (see Sec. 3.17). In addition to this mix, generative programming separates problem space from solution space using configuration knowledge. The separation of these two spaces is an important concept we will use in this thesis as well. It allows us to describe domain-level features and intentions without cluttering them with implementation details. From the other end, the implementation typically does not represent the domain model — it merely implements some of its aspects.

The implementation of generative programming consists of the configuration of what we want, the generator, and the generated output. The generator can be implemented using programming language constructs, e.g., C++ templates, or using a separate program. The output is typically source code that is compiled along with the hand-written code, but we can generate various models and configurations that are impractical for manual processing.

Generative Programming is a practical and universal approach suitable for many applications. In this thesis, we do not use it explicitly, but we still utilize various aspects of generative programming. In contrast with traditional code generators that require complete and precise specification of what they should generate, our approach focuses on synthesizing applications from incomplete specifications. However, due to the versatility of generative programming, these two approaches are not in conflict.

3.20 Evolutionary Programming

Evolutionary Programming or Genetic Programming [60] is a relatively simple concept: generate a (random) "population" of possible solutions, evaluate the individuals of the population according to their fitness (quality), select the best individuals, and breed the selected individuals via crossover and mutation to get a new population for the next evaluation; repeat until satisfying individual appears. In the end, the evolutionary algorithms, in general, are optimization algorithms. The difficulty lies in how to represent the individuals (which implies how to breed the next generations), and how to measure the fitness of the individuals [61, 62, 63, 64].

There are many approaches on what is subject to change between generations and what parameters we optimize. The individuals may represent a predefined set of parameters that may or may not vary in structure. It is also possible to encode source code into the individuals, e.g., in the form of an abstract syntax tree [65], or an executable binary [66].

From the perspective of this thesis, the major disadvantage of the evolutionary approach is the unpredictability of the result and the lack of confidence why the found solution should be correct.

3.21 Machine Learning

The point of machine learning [67] is in finding parameters or rules for a generic model to solve the current problem. The learning process is typically a statistical analysis of the classified instances to determine rules that will, hopefully, successfully classify the testing instances. Then we assume the trained model is usable for the real-world instance population. Some models may require identification of only a few rules (e.g., decision trees [68]), others may require to determine millions of parameters (e.g., neural networks [69]).

Machine learning techniques provide powerful tools to recognize and match patterns in large data sets. Although the machine learning provides significantly better results than the evolu-

tionary programming (Sec. 3.20), neither the machine learning algorithms can guarantee that the found solution should be correct because the practical usability of the trained model is determined by testing, not by proofs. This is not an issue in applications that can tolerate a reasonable amount of mistakes, i.e., when such a mistake leads only to a suboptimal behavior instead of a failure. We may improve the situation using proper testing techniques and methodologies, but fully explainable machine learning [70] is still a very young area.

The primary disadvantage of machine learning is the necessity of huge training datasets. For every parameter a model has, we need some training data to determine it; a bigger neural network can provide better results, but also require more training. This flaw is fatal for software synthesis applications because practically all source code is hand-made and thus relatively scarce. Moreover, programming errors can be extremely costly.

In this thesis, we shall focus on deterministic techniques that provide reliable feedback and error detection. Since we see the software synthesis as an assistive technology for a programmer, it is better to ask for more input data than to provide an answer of unknown quality. On the other side, when the software synthesis fails due to a missing component, we may use machine learning techniques to suggest what kind of component is missing and provide a partial solution with a reasonable gap for a programmer to fill.

3.22 Type-Driven Software Synthesis

Simple types alone are rarely sufficient for useful synthesis. For example, if we have a function that accepts two integers and returns another integer, we have no way of knowing whether the function adds or subtracts the given numbers. To do so, we need more input data. Type-driven Software Synthesis overcomes this issue by specifying additional conditions and utilizing refinement types [71]. It also may take into account the resource usage of the synthesized algorithm [72].

The type-driven synthesis is based on Type Transition Nets similar to Petri Nets, where transitions represent functions that map one type into another, and places represent variables of a given type. The marking of the network provides us with known variables, while the paths through the network represent a program. Therefore, the software synthesis problem is then converted into a pathfinding problem [73]. However, when the type transition net includes generic types or collections, it quickly becomes intractably vast. To overcome this problem, the solvers need to use some approximations over the type system introducing abstract type nets and further refine these abstractions so that they achieve viable results.

Because types are a low-level tool, the type-driven software synthesis is also a low-level tool. We still need to specify what should be synthesized and what its properties are. The high-level abstractions are simply out of the scope. In this thesis, we consider to delegate the technical details of the synthesis onto such a tool and focus on the high-level (business-level) description of our application to extract a low-level specification so that the type-driven synthesis solver has enough input data to deal with the details for us.

3.23 Conclusion

In this chapter, we explored various tools and methods related to software synthesis. Finite automata (Sec. 3.11) and Petri Nets (Sec. 3.12) provided us with formal frameworks to describe behavior in both machine-friendly and human-friendly way, and on which notations like BPMN are based. Domain-specific languages (Sec. 3.2) enabled us to describe what we want to synthesize and, when combined with literate programming, we can include formal specifications in

human-friendly specifications. We will use these in Part II, where we will explore some practical approaches on how to specify business logic and how to use such a specification.

Functional block programming (Sec. 3.9), Tensor Flow (Sec. 3.10), and behavior trees (Sec. 3.13) served as a base for an original executable structure we will introduce in Chapter 10.

In the remaining sections, we explored various concepts; some inspired us; others showed us what we do not want to do — and both are as important.

Chapter 4

Computing with the Unknown

– "How many dumplings do you want?"
– "I don't know."

If we try to implement an algorithm to deal with this tricky scenario, one of the essential questions is how to represent the answer. The number of the dumplings seems to be a natural number fitting the integer type; however, our answer does not seem to fit well with such an expectation. The physics and many engineering fields traditionally use uncertainty to represent values, e.g., 4 dumplings ±25 %, but a value with uncertainty is still a known value. The absence of a value is a different concept.

In this chapter, we will briefly survey possible representations and semantics of the unknown, starting from three-valued logic and continuing towards more expressive constructs. Some of the representations are used in the later chapters; others are stepping stones for a better understanding of not knowing. Reasoning with a good representation of the unknown can provide us with answers that more traditional methods are unable to reach. The understanding of these concepts changes our approach to how we solve the problems and how we interpret data we have (or do not have), including the context. For example, the answer in the introductory example suggests that there are no unusual circumstances known; therefore, the usual number of dumplings will do.

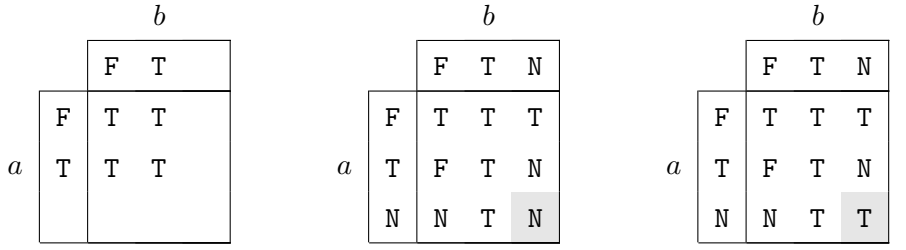
4.1 Three-valued Logic

When an algorithm is not provided with complete input, and it is not expected to provide a complete result, it must compute with the possibility that the value of a variable is not known. A simple way to handle such a situation is to use three-valued logic, where a third value (`N` as "null" or "nil") is added to traditional true (`T`) and false (`F`).

Since the exact interpretation of the third value is not universally defined, many three-valued logics (and multi-valued logics) were created. Among the most popular are Kleene logic [74] and Łukasiewicz logic , which differ in how implication is defined (see Figure 4.1.1).

Kleene logic is often criticized for not having tautologies [75], since any formula may not be true when one of its variables is assigned the third value. Łukasiewicz logic "solves" this issue by introducing tautology `N` = `N`, but such a tautology introduces unwanted relations between unknown variables — if $a =$ `N` and $b =$ `N`, then $a = b$, because of $a =$ `N` $=$ `N` $= b$.

On the other side, if `N` = `N` is not a tautology, then we cannot assign `N` to any variable because we do not know whether it is a valid operation until we know the value of the variable. So the



(a) Bool logic (b) Kleene logic (c) Łukasiewicz logic

Figure 4.1.1: Truth tables of implication $a \implies b$.

only thing we can do is to keep the variable in place until its value is known, which is not very practical when we need to solve a problem with incomplete inputs.

However, Łukasiewicz logic can be generalized to multi-valued logic and further to use real numbers between -1 and $+1$, or probabilistic 0 and 1 [76], towards introducing fuzzy logic.

4.2 Epistemic Modal Logic

Another approach is introduced by *epistemic modal logic* [77, 78]. An operator of knowledge (K) is used instead of the third value. For example, $\mathrm{K}_\alpha\varphi$ means that agent α knows φ. So instead of $a = \mathtt{N}$ from previous sections, we can write $\neg\mathrm{K}_\alpha a$.

The use of the operator avoids the problem of introducing relations between unknown variables (as described in Section 4.1) and generally works well with other tools based on traditional propositional calculus.

Epistemic modal logic has great use in multi-agent systems, but in this thesis, we explore the means of computing with knowledge within *a single creative agent*. From such point of view benefits of epistemic modal logic are limited, since it is focused on specifying whether a fact is known and who knows it, rather than on properties of an unknown fact. However, there is an intriguing idea tightly bound with modal logic: Kripke semantics.

4.3 Set Theory and Kripke Semantics

Kripke semantics [79] models knowledge of agents using Kripke structure — a graph, where nodes represent possible worlds (possible combinations of facts the agents believe) and edges (labeled by sets of agents' names) connect, from agent's point of view, indistinguishable states. The agent does not have enough knowledge to distinguish states within a single connected component of the graph using only edges with the agent's name.

An alternative way to express possible worlds is using Set Theory [80]. In a simplified way, when we assume only one agent, a set of possible values can be assigned to each variable in the model. So instead of writing $a = \mathtt{N}$ in three-valued logic, a statement $a \in \mathtt{N}$, where $\mathtt{N} = \{\mathtt{F}, \mathtt{T}\}$, is used. The "unknown" $\mathtt{N}$ is understood as "cannot decide which value it is". The use of the set solves the little problem described in Section 4.1, since having $a \in \mathtt{N}$, $b \in \mathtt{N}$ means $a \in \mathtt{N} = \mathtt{N} \ni b$, which does not introduce any additional relation between a and b.

Using sets representation, the statement $\mathrm{K}a$ (see Section 4.2) means $a \in A$, where A is set of a single element; therefore, a is known because there is only one possible valuation of it. Similarly, the statement $\neg\mathrm{K}a$ means there is more than one element in set A, so it cannot be decided which one is the correct valuation of a. In case when A is empty, the solution of the modeled problem does not exist.

4.4 Generalizing the Sets

An interesting feature of the set approach is that the "knowing" is determined by the number of elements in a given set, not by the values itself. Therefore, there is no need to limit the values to boolean true and false only; instead, we can use sets of actual possibilities, as it significantly simplifies the specification of a model.

For example, when deciding apple of which color to choose, instead of having three formulae $green = \mathtt{N}$, $red = \mathtt{N}$, and $brown = \mathtt{N}$, we can use single set $Color = \{green, red, brown\}$. Then, after a more or less complex computation, we get $Color = \{brown\}$, so we know it may not taste great.

4.5 Relations as Sets

After exploring the dead end in Section 4.1 and finally understanding how to compute with the unknown in Section 4.4, we need to take a step further and combine relations with sets. Unifying the sets with the relations provides us with a tool that can both represent the real world and our knowledge about it.

The core idea is to represent a particular relation as a set, or rather a tuple, of entities in the relation. A set of such tuples will represent the relation between sets of the entities, or it can represent our knowledge about the relation. For example, if we have a set of apples and a set of colors, then we can assign colors to the apples using a set of apple–color pairs, i.e., a binary relation. If we allow multiple colors assigned to a single apple, then such a relation describes the apple has one of the assigned colors, but we do not know which one exactly.

Alloy [20] is a language and a solver that allows us to define models using relations and constraints, where the relations are represented using sets. Instead of providing all possibilities in a set, Allow generates possible instances of the model which fulfill the defined constraints; unfortunately, the found model instances are limited in size due to the exponential complexity of the solver. The main use-case for such a tool is in rapid prototyping and experimenting with the model before we implement our application.

We may also notice certain similarities between such a set-based representation and type systems of programming languages, but we prefer no to go deeper.

4.6 The Incomplete Solution

When solving a problem using sets (see Section 4.5), there are the following three possible results:

1. All variables are unambiguously determined – each variable is a single element set: The solution is successfully found.

2. Some variables are sets with more than one element: More than one solution is found. There may be not enough data to determine which one is correct or best if they are all correct.

3. Some variables are elements of empty sets: Given constraints have eliminated all possible solutions.

   (a) A required "tool" (operator) is missing to find a solution.

   (b) There are conflicting constraints; no solution exists.

In practical applications, answers like "the solution does not exist" or "the input is ambiguous" are not very practical. The point of a creative agent is to create something useful, not to provide excuses. Therefore, even partial and incomplete solution is desirable. The question is, what answers can be obtained and how useful these are going to be.

In the first case, the solution is provided, and the work is done. In the third case, the work is also done but with a significantly less satisfying result.

Interesting is the second case, where the solution is ambiguous. In such a case, there is a good chance that some variables are determined, and only a few remain with more than one possibility. The creative agent (a programmer or inferring automaton) may ask the user for additional constraints to further reduce the set of possible solutions. The issue in this case is that some of the found solutions may use operators matching the criteria but not matching the semantics of the task. Thanks to the knowledge of which variables are ambiguous and which are not the agent can well target the question for the user and ask highly relevant questions, or at least highlight which part of input needs to be specified better.

4.7 Opening the Closed-World Assumption

In the previous section, we neglected the third case, in which we find no solutions. It may look like there is nothing to do, and within the closed-world assumption, it is true. But, what if we relax this constraint?

Open-world assumption introduces an infinite world of possibilities. When something cannot be inferred from known facts, it is considered unknown instead of impossible as it is with the closed-world assumption. However, neither of the answers is satisfying. What if we could infer a missing piece we need to reach the solution?

The open-world assumption provides us a possibility of another way to achieve the solution. Therefore, we know that we are looking for something compatible with our model. Also, we may assume that such a missing piece will be somehow similar to other pieces we already have because the open world is infinite, and we cannot look everywhere. We can look at the other models, though. The solution to a similar problem may contain a piece that we need for the current problem. This approach might allow our solver to learn from its previous experience, as we will discuss in Part III.

4.8 Conclusion

We advanced from a simple missing value in three-valued logic through modal logic to the sets of possible relations. This path was behind the creation of the STS algorithm that Chapter 8 will present. This path also has shown us that to reason about missing values, we need a tool above the values rather than a special value, and Set Theory provides us with such a tool.

Set Theory also provides us with a hint of a possible direction in further development. Because the use of sets naturally comes with the closed-world assumption, we should try to relax this assumption and look for more options when a set of possibilities is empty.

Part II

Models & Specification

Chapter 5

Specifications & Synthesis Overview

Synthesis of a complex thing like a web application requires us to identify our input knowledge carefully, what it tells us about the application, which components we can synthesize, and how these components affects each other. Traditional web applications are of a relatively simple architecture with only a handful of components we need to implement. Today frameworks are mostly based on the MVC architecture (Model, View, Controller) [81] with various adjustments opinionated by their authors.

In the following sections, we look briefly at the architecture of the web application we wish to implement. Then we investigate what knowledge we require for such a task and what knowledge sources can be used. Finally, we envision which steps of the development can be consistently and effectively automated.

5.1 Application Architecture

For the needs of this thesis, let us start with a classical architecture pictured in Figure 5.1.1. The User creates an HTTP Request using a web browser. The HTTP request the server receives is processed by the Router, which collects input variables from the request and identifies which Controller should handle the request. Then the framework instantiates the requested Controller and passes it the input variables. The Controller uses the Model to retrieve or update data in the Database and passes the results to the View. The View uses Templates to convert the

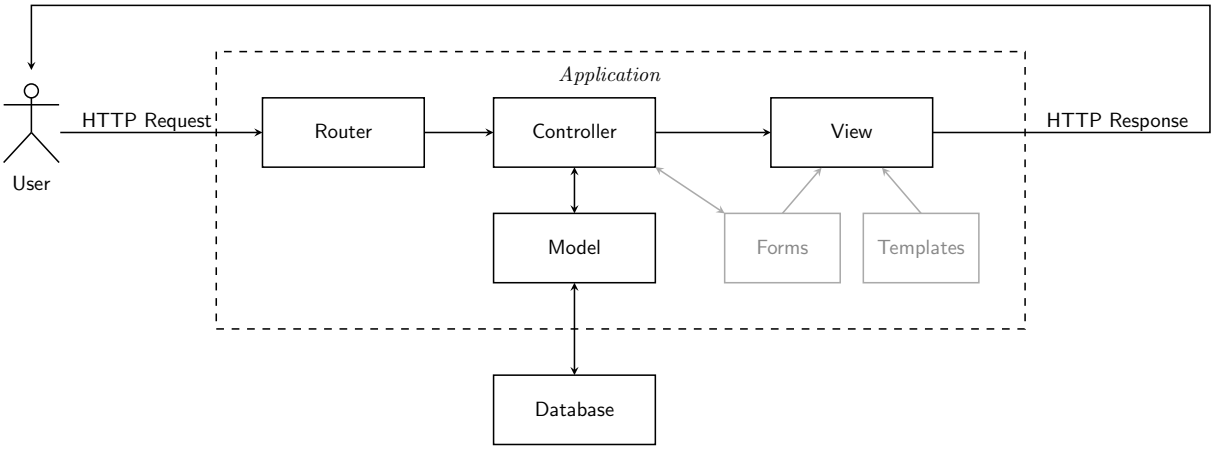


Figure 5.1.1: A classical web application architecture

results from the Controller into a final web page that it then sends back to the user as the HTTP Response. The View and Controller also use the Forms to both parse data from the HTTP Request and render the Form into the web page; this small anomaly is due to the specific concern of the Form component regarding the close relationship between the form data structure, its representation, and the user interface required to manipulate it.

A pleasant property of such an architecture is that individual controllers are rarely connected to each other, and views are also well-encapsulated components. Therefore, extending the application with a new feature usually adds a new controller and a view, but the overall application complexity (i.e., horizontal scaling) grows mostly linearly due to the limited dependencies and connections between the controllers and views. For the needs of software synthesis, we can benefit from the predefined structure of the application as we know the general role of each component, and thus, we have a sort of template to fill in.

5.2 Knowledge Sources for Development

Once we identified the individual components of the application, let us take a look at what knowledge we need to implement the components and where to get such knowledge. We already explored possible programmer's inputs in Chapter 2, so let us connect useful inputs with relevant components in the web application. For the sake of simplicity, we will focus on a few important inputs, namely: business processes, data structures, and user interface designs.

Figure 5.2.1 presents the influence of these knowledge sources on the web application described earlier (see Fig. 5.1.1).

The business processes influence the behavior of the application. Such behavior is implemented in the models and partly in the controllers. More complicated applications utilize an additional layer of business logic placed between the model and controllers so that the model turns into a data abstraction (storage) and controllers into a relatively thin application logic connecting the logic with the view. We will discuss this arrangement in Section 7.6.

The data structures at this point are properties of the real-world entities we need to represent in the application. For example, if we need to generate invoices, we know that such an invoice has, among other things, a serial number and a due date. Therefore, we know that the database will contain columns to store these two properties, the model will represent the properties somehow, and the forms will have a field for a user to specify the due date. We may not know the particular data types of the properties yet, and the application will likely require additional data, but it is a good start.

The GUI wireframes or sketches provide visual designs of the user interface. Aside from the looks, such designs help us with business processes and data structures. For example, if there is a button to navigate the user to the next step, we know that these two steps are somehow connected in the business processes. Moreover, if there is an input field to specify a due date, we know that some of the data structures will likely contain this due date. From the other side, if there is a property in a data structure, there should be a field to fill in the property or at least to display it.

At this point, we have collected all the knowledge we could, and now it is up to the programmers to implement the application. The question is, whether there is a better and more effective way to utilize the knowledge to help the programmers with the implementation?

5.3 Automation

In the previous sections, we presented the web application architecture and the knowledge we have. The programmers will now combine these into a working implementation. Unfortunately,

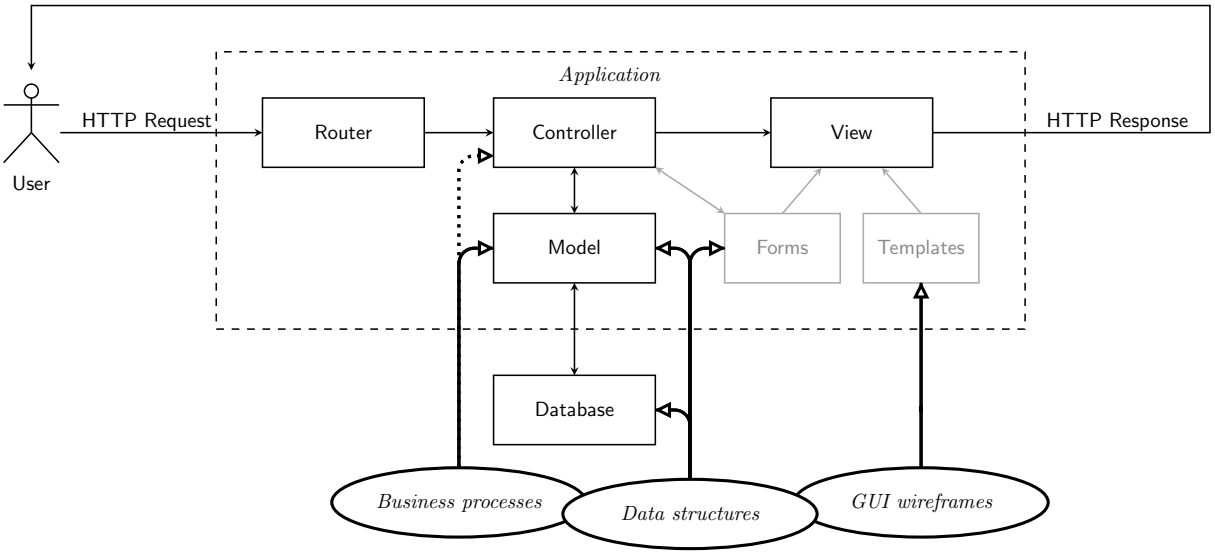


Figure 5.2.1: The primary influence of knowledge sources on a web application

the available knowledge is not a complete specification of the desired application, and thus the programmers will have to improvise, extrapolate, and use their previous experience to fill in the missing pieces. If the programmers understand their users' needs well, there is a good chance for success. If not, the programmers will have to consult and revise the implementation with the users until they get it right.

The full automation of the implementation process is practically impossible because of the incompleteness of the specification for two reasons. The first is that a complete specification of a behavior would be at least as complicated as the implementation itself. The second is a simple fact that the users do not know what they want. Therefore, we need to take smaller steps and ask what parts we can effectively automate using only an evolving partial specification and avoiding any interference with manually crafted components. We need to aim for a sweet spot where a simple specification allows us to automate a significant amount of work without raising the complexity of the application. So, what can we automate in the web application development?

Each arrow representing knowledge influence in Figure 5.2.1 has some potential for automation. However, as we discussed earlier in Chapter 2, automation requires machine-friendly input.

The GUI wireframes are the most challenging yet the most straightforward because we could simply draw forms and screens of the user interface if we had proper tools available. Unfortunately, the current availability of such tools is more than disappointing — there are excellent GUI editors for desktop applications (e.g., Qt Creator), but barely any for web applications.

The data structures are somewhat challenging to grasp in the early stages of development. A reasonable approach is to roughly design a database schema (assuming an SQL database is used) with a small example data set, and then incrementally build models from it. Existing ORM frameworks (e.g., Doctrine) provide tools to synchronize database schema with model definitions or to generate one from the other. Application frameworks (e.g., Symfony, Ruby on Rails) then provide tools to generate forms from the model definitions or even entire administration interfaces. There is certainly some space for improvements, but overall, we can consider this question answered.

To model *the business processes,* we can use already existing editors [11]. Creating a comprehensible diagram may take some exercise, and unfortunately, most of the examples we encountered do not involve interaction between software systems and humans — they mostly capture

scenarios of multiple workers interacting with each other. Once we figure out how to draw a business process, we need to figure out how to use it in the application. Many tried to execute such a business process directly [82] with a certain level of success; however, the required tools and frameworks are complex and difficult to deploy. We need something simple, something suitable for small applications with few complicated entities. To do so, we propose to incorporate some kind of a formal model into the web application, and then find a connection from business processes to this formal model.

In the following chapters, we will define a formal model based on nondeterministic finite automata (Chapter 6). It will provide us with basic but practical modeling abilities, and enables us to describe the behavior of a web application formally. Then we will take a look at a RESTful architecture that can effectively incorporate such a formal model (Chapter 7). And finally, we will find a way to infer such a formal model from business process diagrams (Chapter 8). As a result, we will be able to draw a business process diagram of how the user will use the application, and then automatically infer implementation that fits the given business process.

Chapter 6

State Machine Abstraction Layer

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

6.1 Introduction

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

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

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

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

The next two sections present an example of a typical entity in a web application. Section 6.4 formally defines the Smalldb state machine, and Section 6.5 describes the relationship between state machine instances and the underlying database. Later on, we investigate some intriguing implications of this design, like introducing Smalldb as a primary metadata source in Section 6.7 or analyzing application correctness.

6.2 REST Resource as a State Machine

Let us start with a simple example of a generic resource (entity) in a RESTful application [5]. RESTful applications typically use HTTP API to manipulate resources. Since REST does not specify a structure of a resource nor exact form of the API (simply because it is out of REST's scope), it is impossible to use this API without an additional understanding of the application behind this API. However, HTTP defines only a limited set of usable methods of predetermined semantics, and thus, we can interpret it as a generic state machine.

Figure 6.2.1 presents such a generic state machine equivalent to a REST resource. The resource is created by HTTP POST request on a collection, where the resource is to be stored. Then it can be modified using HTTP PUT (or similar HTTP methods), and finally, it can be removed using HTTP DELETE method.

Without further investigation of the resource, the influence of the "modify" transition cannot be determined, but we can safely assume the resource is more complicated than Figure 6.2.1 presents; otherwise, the "modify" transition would make no sense.

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

We will return to this concept in the next chapter, Section 7.2, to design a RESTful API for such state machines and to identify the fundamental architectural and model constraints, so that we can determine application architecture that lets us utilize the state machines. For now, however, we shall focus on the state machine itself.

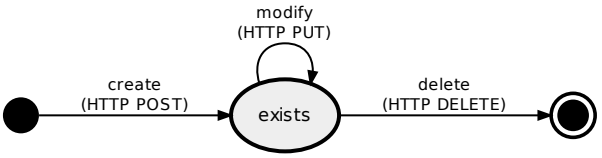


Figure 6.2.1: REST resource state machine

6.3 Real-world Example

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

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

A state machine representing the lifetime of the post is in Figure 6.3.1. As we can see, there are three states and a few transitions between them. Note that there is no final state in this state machine. That is because the blog post is never destroyed.

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

There is also one big problem with both this and previous examples. If these state machines are interpreted as usual finite automata, the edit action has no effect. Invoking the edit action makes no difference because it starts and ends in the same state. To justify this behavior, the state machine must use a concept very similar to Kripke structures. Each state represents a possibly infinite group of sub-states, which have common behavior described by the encapsulating state. Therefore, the edit transition is, in fact, a transition between different sub-states within the same state, i.e., sub-states belong to the same equivalency class. Omitting these sub-states from the state diagram is very practical since it allows easy comprehension. The sub-states are implemented using "properties" of the state machine instance, for example, title, author, and text of the blog post (this concept will be described in Section 6.4.4).

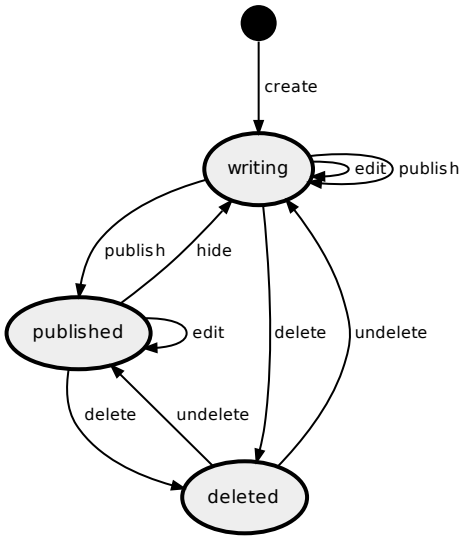


Figure 6.3.1: State diagram of blog post

6.4 State Machine

As previous examples showed, it is necessary to modify and extend the definition of finite automaton [83], to make any use of it. This section formally defines a Smalldb state machine and explains its features. The definition is designed to follow an actual implementation as close as possible, so it can be used to infer properties of final applications formally.

6.4.1 Smalldb State Machine Definition

Smalldb state machine is modified nondeterministic parametric finite automaton, defined as a tuple $(Q, P, s, P_0, \Sigma, \Lambda, M, \alpha, \delta)$, where:

- Q is a finite set of states.
- P is a set of named properties. P^* is a (possibly infinite) set of all possible values of P. P_t is the state of these properties in time t. $P_t \in P^*$.
- s is state function $s(P_t) \mapsto q$, where $q \in Q$, $P_t \in P^*$.
- P_0 is a set of initial values of properties P, $P_0 \in P^*$.
- Σ is a set of parameterized input events.

- Λ is a set of parameterized output events (optional).
- M is a finite set of methods: $m(P_t, e_{in}) \mapsto (P_{t+1}, e_{out})$, where $P_t, P_{t+1} \in P^*$, $m \in M$, $e_{in} \in \Sigma$, $e_{out} \in \Lambda$.
- α is assertion function: $\alpha(q_t, m) \mapsto Q_{t+1}$, where $q_t \in Q$, $Q_{t+1} \subset Q$, $e_{in} \in \Sigma$.

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

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

6.4.2 Explanation of Nondeterminism

Nondeterminism in the Smalldb state machine has one specific purpose. It expresses the possibility of failure and uncertainty of the result of invoked action.

For example, when the blog post (see Section 6.3) is undeleted, it is not known in advance in which state the blog post will end, because if a user has no permission to publish, the resulting state will be "writing", even if the blog post was already published.

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

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

6.4.3 Simplified Deterministic Definition

Because the complete definition described in Section 6.4.1 is a bit too complicated, here is a simplified deterministic definition with most of the unimportant features thrown away. These features are present in the implementation, but they are not crucial for a basic understanding. This definition may also be useful for some formal proofs, where these two definitions can be considered equivalent if the thrown away features are not significant for the proof.
Please keep in mind that the rest of this thesis always refers to the full definition in Section 6.4.1.

The simplified definition is: Smalldb state machine is defined as a tuple (Q, q_0, Σ, Λ, δ', m'), where:

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

This is basically the Mealy (or Moore[1]) machine [37, 21]; the only difference is in introducing additional constraint w to handle the possibility of failure. However, the w is not known in advance when a transition is triggered (see Section 6.4.2).

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

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

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

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

6.4.4 Properties and State Function

As came out in the blog post example (see Section 6.3), the finite automaton is not powerful enough to store all arbitrary data of an entity. To overcome this limitation, the Smalldb state machine has properties. Each property is identified by name, and the rest is up to the application. Properties can be implemented as a simple key–value store, columns in an SQL table, member variables in OOP class, or anything like that.

Since properties are not explicitly limited in size, they can store huge, theoretically infinite amounts of data, data of high precision, or very complex structures. To handle these data effectively, the state function is used to determine the state of the machine. The state function converts properties to a single value, the state, which is easy to handle and understand.

Because applying the state function on different sets of properties can (and often will) result in the same state, the state represents the entire equivalence class rather than a single value. This approach is very similar to Kripke structures [84].

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

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

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

Typically the state function is very simple. In trivial case (like the first example in section 6.2), it only detects the existence of a state machine. In more common cases (like the blog post example in section 6.3) it is equal to one of properties or checks whether a property fits in a predefined range (for example, if the date of publication is in future). Since the state function is a key piece of the machine definition, and it is used very often, it should be kept as simple and fast as possible.

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

6.4.5 Input Events

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

6.4.6 Output Events

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

6.4.7 Methods

The methods M implement each transition of the state machine. They modify properties and perform all necessary tasks to complete the transition. These methods are ordinary machine-specific protected methods as known from object-oriented languages, invoked by the universal implementation of the state machine. Since the methods cannot be invoked directly, access to them is controlled by the state machine, and it is possible to implement advanced and universal access control mechanism to secure an entire application.

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

6.4.8 Transitions and Transition Function

The main difference from classic nondeterministic finite automaton is in a division of each transition into two steps. The transition function δ covers only the first step. The second step is performed by method $m \in M$, which was selected by the transition function δ. The point of this separation is to localize the source of nondeterminism (see Section 6.4.2) and accurately describe a real implementation.

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

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

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

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

6.4.9 Assertion Function

A simple condition must always be valid:

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

Otherwise, there is an error in the function m.

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

6.5 Space of State Machines

Everything said so far was only about the definition of the Smalldb state machine. This definition is like a class in an object-oriented language – it is useless until instances are created. In contrast with the class instances, state machine instances are persistent. The definition is implemented

in source code or written in configuration files, and properties of all state machine instances are stored in a database.

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

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

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

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

6.5.1 Smalldb and SQL Database

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

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

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

6.6 Spontaneous Transitions

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

There are two ways of dealing with this problem. The first way is to live with them and simply avoid any need for reaction. This approach can be useful in simple cases where an entity should be visible only after a specified date. For example, the blog post (see Section 6.3) can have "time of publication" property and state function defined like "if the time of publication is in the future, then the post is in the writing state; otherwise, the post is in the published state".

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

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

6.7 State Machine Metadata

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

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

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

For example, the Figures 6.2.1 and 6.3.1 used in examples (sections 6.2 and 6.3) were rendered automatically from a state machine definition in JSON using Graphviz [85] and a simple, 120 lines long, convertor script.

An additional use for such metadata is in generating the user interface, determining which parts of it a user can see and use, user input validation, access control, or API generating. Furthermore, if metadata are stored in static configuration files or a database, they can be modified using an administration interface embedded in the application, which allows altering many aspects of the application itself easily. Dynamically generated metadata then allow us to build large and adaptive applications with very little effort.

6.8 Application Correctness

Much research was done in model checking and finite automata, resulting in tools like Uppaal [40], which allows us to verify statements about given automaton formally. Since Smalldb is built on top of such automata, it is very convenient to use these tools to verify Smalldb state machines. Moreover, thanks to the existence of the formal definition of the Smalldb state machine, it is possible to export state machine definition to these tools correctly.

6.8.1 Access Control Verification

Verification of essential properties, like state reachability[2], safety[3], and liveness[4], is nice to have in a basic set of tests; however, these properties are not very useful on their own. The situation gets much more interesting when user permissions are introduced.

User access is verified just before a transition is invoked. Therefore, a user with limited access is allowed to use only a subset of transitions in the state diagram, and some states may become unreachable. If the expected reachability of a state by a given user is stated in the state

[2]State reachability: “Is there a path to every state?”
[3]Safety property: “Something bad will never happen.”
[4]Liveness property: “The machine will not get stuck.”

machine definition, it is easy to use the earlier mentioned tools to verify it. Furthermore, in most cases, any allowed transition originating from an unreachable state represents a security problem.

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

Because access control is enforced by the generic implementation of the Smalldb state machine (in the decorator, see Sec. 7.6), which can be well tested, and it is not modified often, the probability of creating security issues is significantly reduced.

6.8.2 Optimizations vs. Understandability

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

In Smalldb, a state machine is used in a very different fashion. The state machine is expected to express the real behavior of a represented entity in a way, which can be understood and validated by a non-technical user (customer). A connection between understandability of state diagram and automated generation of this diagram from the single source of truth (see Section 6.7) is an important feature because it eliminates the area where errors and misunderstandings can occur – a gap between expectations and software specification.

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

6.8.3 Computational Power of Smalldb Machine

The classical finite automaton is not Turing complete, because it has a limited amount of memory, so it cannot be used, for example, to count sheep before sleeping. Nevertheless, the Smalldb state machine this limitation overcomes by introducing properties and methods implemented in Turing-complete language (see Section 6.4.1), so the sheep counting can be done using one state, increment loopback transition, and a sheep counter property.

Smalldb state machine is a hybrid of two worlds. On one side, there is a lovely nondeterministic finite automaton, which allows all the nice stuff described in this paper. On the other side, there are Turing-complete methods M, the barely controllable mighty beasts, which do the hard work. As long as these two parts are together, the computational power is the same as of the language used to implement the methods M.

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

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

6.8.4 Troubles with Methods M

Because methods M (see Section 6.4.7) are Turing complete, it is not possible to deduce their behavior automatically. This means it is not possible to predict whether the machine will use all transitions of the same name, and therefore some of the states considered reachable, when methods M did not take into account, may not be ever reached. This problem can be partially solved by careful testing and reviewing the methods M.

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

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

6.9 Workflow Correctness

6.9.1 State Machine Cooperation

The workflow can be understood as a cooperation of multiple entities with compatible goals. When these entities are specified as Smalldb state machines, it is relatively straightforward to involve tools like Uppaal (see Section 6.8) and let them calculate, what will happen, when these entities are put together.

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

The Smalldb state machine does not have to represent the entity within an application only. It also can be used to describe the behavior of external entities; however, such an entity should not be included in the application. We will take a brief look at this approach in Chapter 9, where we use Uppaal to model and verify such a network of interacting automata.

6.9.2 BPMN Diagrams

Entity lifetime is closely related to users' workflow and related processes. BPMN [8] was developed to describe them in some formal way. As Chapter 8 will show, it is possible to infer a Smalldb state machine as a formal model of each entity included in the process from a BPMN diagram. Then the state machine representing an application entity can be used as a skeleton of its implementation. Moreover, the other state machines which represent humans and external applications can be used to execute a simulation of the complete process.

This approach eliminates the need for a duplicate software specification when there is a model of the entire business process, and it introduces the possibility of testing and formal proving of the application not against its specification but rather against other entities in the process, removing the gap between what is expected and what is specified.

6.10 Conclusion

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

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

Smalldb state machines are meant as both a production-ready solution and a building block for further research of software synthesis.

Chapter 7

RESTful Architecture with Smalldb State Machines

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

7.1 Introduction

How to build a model layer of a web application? Modern web applications are usually built using an MVC framework [81] (or similar, e.g., MVP). While the roles and responsibilities of the view and the controller are well established, the exact scope of the model remains somewhat vague. The remaining question is where to draw a line between the model and the controller.

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

A command–query separation (CQS) pattern [86] provides another approach. Instead of adding an extra layer, it separates the command and the query execution paths which lead orthogonally through the layers. One path to query the data and retrieve the response, another to execute commands. Such a separation reflects the distinct requirements and responsibilities of each path, respecting SOLID principles [87]. The pleasant detail is that the CQS pattern is not in conflict with the multi-tier architecture or with the MVC pattern. In fact, these patterns fit together rather well as we will show later.

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

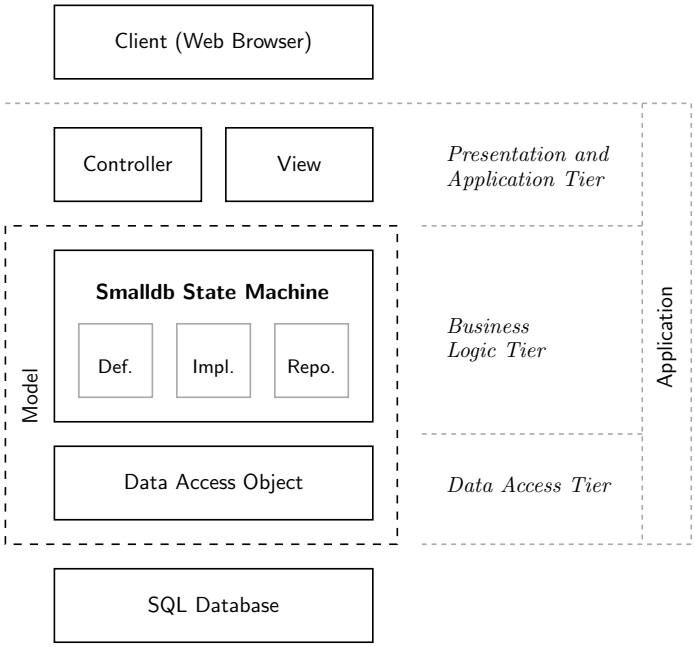


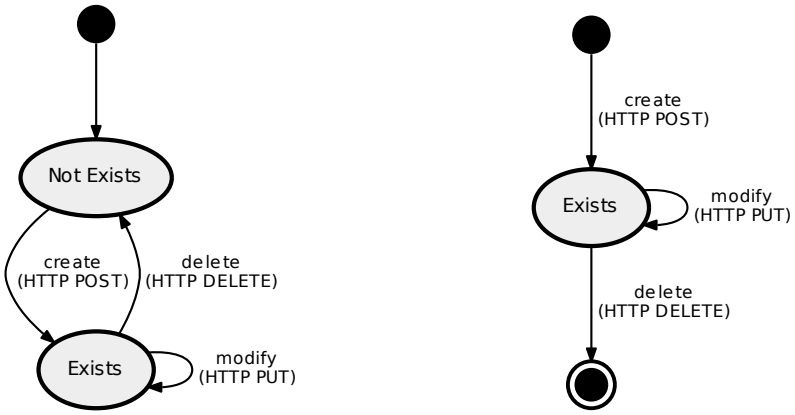
Figure 7.1.1: Application architecture with Smalldb

Another issue of today applications is a lack of formal models. The likely reason is the initial cost of the related infrastructure and the consequence of the misused YAGNI principle ("You Aren't Gonna Need It") [88]: In the beginning, the application is simple, and the formal tool would bring too much overhead, and then, when the things get complicated, it is already too late. However, even a simple formal model gives us an explicit idea of how the application behaves, especially when security is our concern (regarding both reliability and access control). Web applications are typically built on top of two formal concepts: the relational algebra on which SQL databases stand, and the rather trivial REST & CRUD to define universal HTTP API. Unfortunately, the business logic is out of the scope of both of these formal concepts.

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

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

This chapter presents an architectural pattern based on Smalldb state machines [1], which we applied in the further proposed Smalldb framework. The core idea is to represent every entity in the model layer using a Smalldb state machine, a persistent nondeterministic finite automaton, which consists of a declarative formal definition, an implementation of transitions, and persistent data repository. The model layer then consists of two tiers: a low-level data access tier, and a high-level business logic encapsulated in Smalldb state machines — see Fig. 7.1.1. The upper tiers of the web application then read the state of the state machines and invoke their



(a) UML notation (b) Smalldb notation

Figure 7.2.1: State diagram of a generic REST resource in two notations and mapping of their transitions to HTTP methods

transitions. The state machines also provide access control and various additional metadata, for example, user-friendly icons and labels for generated navigation and menus.

The main reason why Smalldb uses simple state machines instead of more expressive formal tools, like workflows based on Petri Nets, is the simplicity and understandability. A nontechnical customer can usually understand a statechart with only a little explanation. This provides a common graphical language to programmers and their customers so that they can discuss the business logic together. Moreover, the state machines are conceptually very close to REST [5], and it is easy to create a REST API for a state machine.

7.2 REST API for a State Machine

Before we get to explore the architecture of Smalldb framework, let us take a better look at the REST resource [5] and its behavior. In short, a REST resource provides a representation of an application entity. A globally unique URI identifies it, and a client manipulates it typically via HTTP requests using a predefined set of HTTP methods. The communication with the resource is meant to be stateless (usually with an exception on authentication), so that the communication is made of simple request–response cycles.

The HTTP/1.1 standard defines the following methods: Options, Get, Head, Post, Put, Delete, Trace, and Connect. Of these methods, only the methods Post, Put, and Delete are designed to modify a resource on a web server. A life cycle of the resource starts with its creation, usually via a Post request, then the resource exists and can be modified using Put requests. And finally, it can be deleted with a Delete request. We can represent such a life cycle as a state machine where the HTTP methods are transitions, and the resource has two states — "exists" and "not exists" — see Fig. 7.2.1. (The both black dots in Smalldb notation represent the "Not Exists" state, while in the UML notation the black dot only points to the initial state; the Section 7.5 will provide the details.) As we can see, the REST resources and state machines are closely related technologies.

Now, let us take the state machine a step further. Regardless of the HTTP methods, we can generalize the resource as a generic state machine and use as many states and transitions as desired (and practical). To do so, we need to change API so that it is possible to invoke any transition, not only those available via standardized HTTP methods. Nevertheless, an API for

a generic state machine requires two operations: one to read its state, and another to invoke its transition. The suitable candidates are the HTTP methods Get and Post, respectively, as HTML forms use them for the very same purpose. Moreover, using the HTML forms, we do not need to implement a specialized client. However, we need to specify which transition of the state machine the HTTP Post method should invoke. One possibility is to add the transition name as a parameter to the request, and another is to suffix URI with the transition name — RFC 3986 defines exclamation mark as an unused delimiter in URI which seems to fit this purpose. Having a URI assigned to each transition of each state machine (a REST resource) is useful as it allows us to query metadata about the transition, or an HTML form to invoke the transition by a subsequent HTTP Post request.

7.3 Architectural and Model Constraints

REST [5] provides us with a reasonable and well-tested set of architectural constraints which includes client–server architecture, stateless communication, layered architecture, and uniform interface with universal resource identifiers. Our goal is to extend the REST architecture by introducing state machines as a formal model of the business logic tier. To do so, we shall meet the following architectural and model constraints while preserving all RESTful properties of the application, so-called Smalldb Pattern:

7.3.1 Declarative formal model

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

7.3.2 Encapsulated interface

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

7.3.3 State machine space

Each state machine should have a unique identifier. Such an identifier allows us to refer to a given state machine, and also, it allows us to map it to a unique URI of the corresponding REST resource. Therefore, all instances of state machines form a space which we can map to a space of URI. This allows us to build a generic API to browse and search the state machine space by various constraints (e.g., list e-shop products of desired parameters). Note the state machine

space may be theoretically infinite; practically limited only by the range of data types used for the unique identifier and the available storage space.

7.3.4 Persistent state storage

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

7.3.5 The initial "Not Exists" state

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

7.3.6 Addressable transitions

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

7.3.7 Executable transitions

The state machines are active during the transitions only; the states represent merely a passive waiting for the next transition. The transitions are expected to modify the persistent state and perform desired side effects. The transitions may accept parameters (like ordinary method calls), but otherwise, the invocation of a transition should be stateless, i.e., independent of the application state.

7.3.8 The abstract entity with a borrowed run-time

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

Conclusion

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

7.4 The Idea

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

The core idea of the Smalldb state machines is based on the three separated components — see Fig. 7.4.1: a state machine definition, a transitions implementation, and a repository. These components are relatively independent; however, a certain consistency is required to form a Smalldb state machine successfully. The state machine definition connects the circles and the arrows, the transition implementation implements the arrows, and the repository provides the circles. Such a decoupling of the components allows us to reuse the same definition for multiple state machines, e.g., the framework offers a prefabricated definition of CRUD state machine and allows the components to use different tools to fulfill their purpose.

The fourth component in Fig. 7.4.1 is a reference object, which has access to the three components and points to a particular state machine identified by a primary key (ID) in the repository (the ID number three in this case). The application uses the reference object to communicate with the state machine. The application does not instantiate the state machine itself like a traditional object in object-oriented programming as the state machine is an abstract construct mostly independent of the application run-time. Instead, the application instantiates only a reference object which provides an interface to invoke transitions and read the state of the state machine via the Smalldb framework.

7.4.1 The State Machine Definition

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

Due to the declarative nature of the state machine definition and use of the relatively simple model, it is possible to reason about the behavior of the state machine without running the application. For example, we can model users of the application as state machines too and then simulate the entire business process as a network of interacting state machines long before we

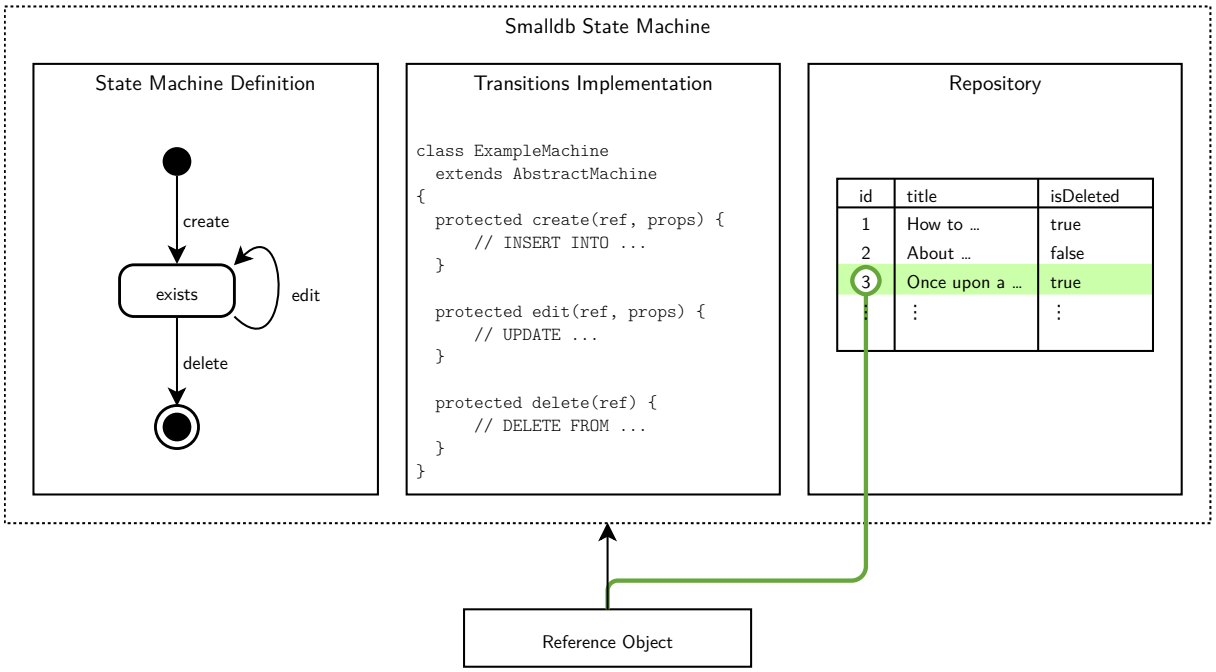


Figure 7.4.1: The three parts of Smalldb state machine

implement the application. This way, we can formally verify the correctness of the application specification during the early design phase of the development.

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

The purpose of the state machine definition is to be the Single Source of the Truth (SSOT or SPOT; a variant of DRY — Don't Repeat Yourself) [89]. The transitions implementation asks the state machine definition, whether it shall allow the user to invoke a given transition and whether it is a valid transition at all. Moreover, the definition may enumerate available transitions of an entity, so that a user interface can show a menu, for example.

7.4.2 The Transitions Implementation

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

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

trick of the Smalldb framework.

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

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

7.4.3 The Repository

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

From the architectural point of view, it is important to realize that the repository spans over two tiers of the application — the business logic tier and the data access tier. However, it must respect these tiers; otherwise, the application would become unmaintainable. The difference between the two tiers is that the data access tier operates directly with the raw data in the persistent storage (e.g., rows in the SQL table), while the upper business logic tier operates with an interpretation of the raw data (e.g., hydrates a value object with a SQL row data). As the Repository pattern [90] (pg. 322) shows, it is a good idea to shield the application from the underlying storage using a facade [91].

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

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

7.5 Key features of Smalldb State Machine

The architecture of Smalldb framework stands on the following few specific features of the underlying state machine. While Smalldb framework assumes the use of nondeterministic finite automata presented in Chapter 6, the architecture is not limited to this kind of automata and can use more sophisticated machines as long as these specific features are preserved, for example, hierarchical automata or workflows based on Petri Nets.

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

Section 7.2 and Fig. 7.2.1 presented a simple state machine using the two notations — the UML notation (Fig. 7.2.1a) and the Smalldb notation (Fig. 7.2.1b). The only difference between the two notations is in the syntax of the initial and final state. Since the initial state of every Smalldb state machine is the "Not Exists" state, we used the black dot to directly represent the initial state instead of using it as a pointer (marker) to the initial state as it is in UML notation. Likewise, the circled dot representing a final state got repurposed to denote the "Not Exists" state. This way, the practical state diagrams, which are usually small, are more linear and better correspond with real-world workflows with the start and the end. Therefore, when the initial and final states are the same (the "Not Exists" state), we cut the cycle in the diagram to reflect the logical workflow of the entity. We can see this happen in Fig. 7.2.1 where the UML notation hides the begin and the end of the life cycle in a loop around the "Not Exists" state, but the Smalldb notation is clear about the begin and the end of the business workflow. The use of slightly modified notation may complicate the use of existing tools and editors; however, our experience shows that such tools usually do not follow UML standards strictly (if at all), and provide sufficient level of customization to deal with the difference.

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

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

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

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

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

7.6 Architecture of Smalldb Framework

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

Fig. 7.6.1 presents the structure of the Smalldb state machine and the nearby components — the application & presentation logic above the state machine, and the three underlying storages below the state machine. The schematics still preserves the separation into the three parts from Fig. 7.4.1, but since the underlying infrastructure under each of the three parts is specific to each part, we expanded the boundaries of the parts to contain lower tiers in addition to the business logic.

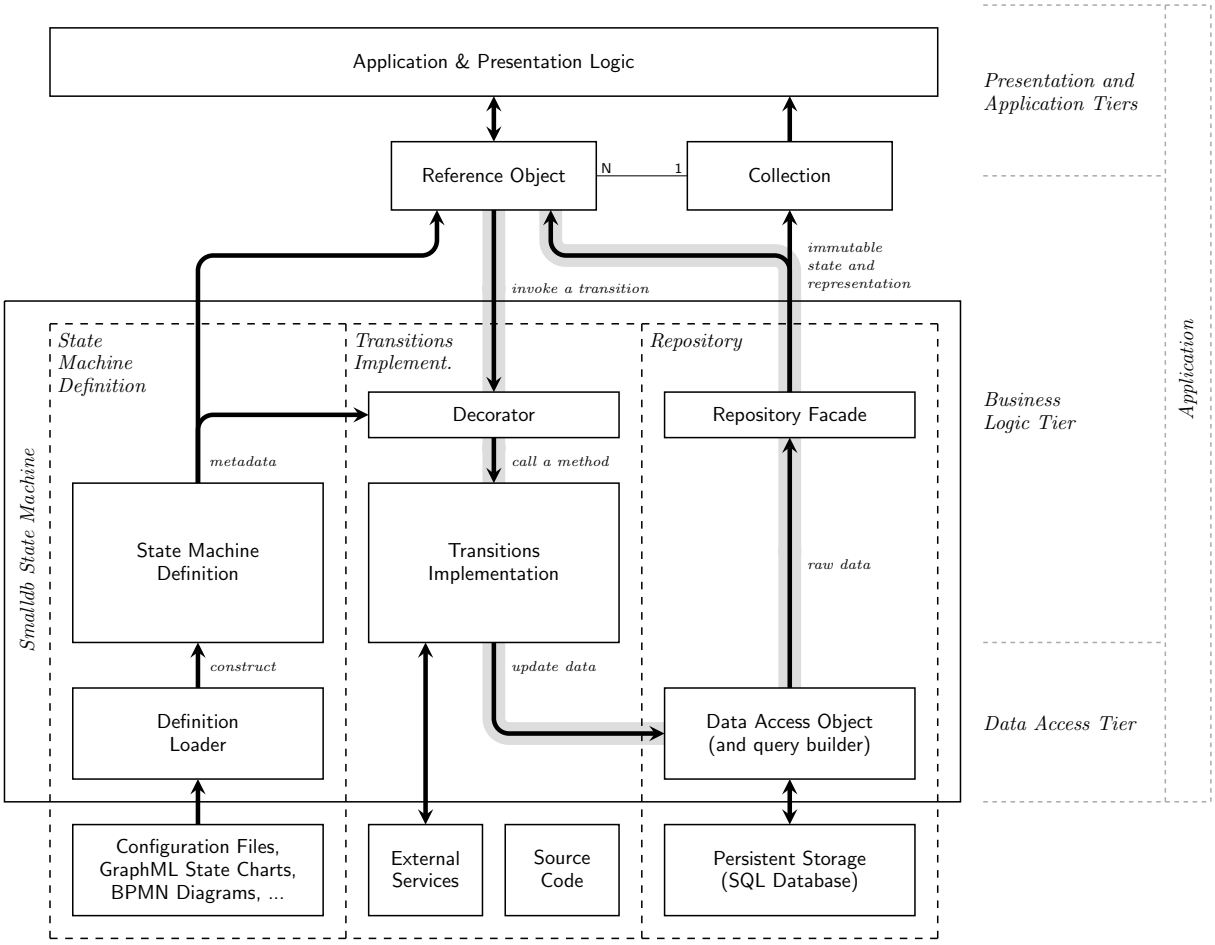


Figure 7.6.1: Architecture of a web application with Smalldb framework

7.6.1 Components Summary

The following sections will describe in detail how the components from Fig. 7.6.1 work together, but before that, let us take a brief look at the individual components and their respective roles.

- *Reference Object* is a pointer pointing to the desired state machine. It provides API to read the state machine state, invoke its transitions, and inspect state machine definition.
- *Collection* is a collection of the *Reference objects*. It may provide some optimizations when

retrieving state machines related to each state machine in the collection (e.g., retrieve all authors of all blog posts on a home page with a single SQL query).

- *Definition Loader* loads JSON files, GraphML statecharts, and BPMN business process diagrams. Then it combines these resources and infers the *State Machine Definition*. This processing may happen during compilation or during run-time initialization.
- *State Machine Definition* is a constant data structure describing the state machine; it is a run-time representation of the formal model.
- *Decorator* validates transition invocations against the *State Machine Definition* and rejects invalid or denied requests. It also verifies that the new state is one of the expected states once the transition is completed.
- *Transitions Implementation* is an executable code which implements the state machine transition. It manipulates the data within the *Persistent Storage* via *Data Access Object* to update the state machine state.
- *Repository Facade* translates raw data from the *Persistent Storage* into business logic entities in the form of *Reference Objects* or *Collections* of the *Reference Objects*. It provides an API to browse state machine space (e.g., faceted search [92]) and to retrieve *references* to state machines of specific IDs.
- *Data Access Object* provides tools to read and manipulate the data within the *Persistent Storage*. It may also provide a query builder to search the state machine space effectively. Alternatively, an ORM framework may be used instead (see Sec. 7.7).

7.6.2 Loading the Definitions

Before the first interaction with a Smalldb state machine, the application must become aware of the state machine. The framework loads configuration of each Smalldb state machine and passes it to the *Definition Loader* — see Fig. 7.6.1, bottom left. The configuration may be provided as a combination of various forms — from source code annotations, configuration files, GraphML statecharts, to BPMN business process diagrams. Some forms may be usable as they are; others may require advanced analysis and inferring of the state machine [3]. The *Definition Loader* may process the configuration during application compilation, or it may do so during run-time startup and cache the result (e.g., within a compiled dependency injection container). In the end, the *Definition Loader* compiles each configuration into a state machine definition.

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

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

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

7.6.3 Framework Initialization

Traditional ORM frameworks, for example, Doctrine [93], typically start their initialization from a repository class. Such an entry point requires an entity manager (or a similar component) which bootstraps the whole framework, while the repository registers its entity type in the entity manager.

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

A typical DI container [94] stands on a configuration which describes how to instantiate registered services, i.e., which constructors or factories to call, and which other services each of them requires, e.g., as constructor arguments. Such a configuration forms a service dependency graph. When a service is requested from the DI container, the container recursively walks through the dependency graph and instantiates the requested service as well as the services on which it depends, effectively constructing a spanning tree of the service dependency graph. There are two ways to specify the dependencies of the services. First is to name each service and specify the name of each dependency of each service, which requires a tedious amount of manual work. The second approach, the autowiring, is to use the type system of the used language to tell us which services match with which constructor or method arguments so that the DI container can guess the dependencies automatically. Unfortunately, this approach is not viable when there are multiple services of the same type, e.g., multiple definitions of Smalldb state machines; such dependencies must be defined explicitly.

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

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

7.6.4 Application Interface

The *Application & Presentation Logic* communicates with the Smalldb state machine using *Reference Objects* (Fig. 7.6.1, top). Each reference object points to its state machine, provides the state of the state machine, and allows its user to invoke a transition of the machine. Additionally, the reference object mediates access to the relevant state machine definition. The reference object alone does nothing but forwards the requests to the responsible components, though.

The *Collection of the reference objects* (Fig. 7.6.1, top right) serves as a representation of a typical answer of the *Repository Facade* when browsing the state machine space. The collection may be a simple list, but it may provide significant optimizations when retrieving state machines

related to each state machine in the collection (e.g., retrieve all authors of all blog posts on a home page with a single SQL query).

7.6.5 Smalldb State Machine Workflow

The processing begins with the *Application Logic* receiving an HTTP request. Using the data within the request (e.g., an ID, or another unique identifier) the *Application Logic* obtains a reference object from the *Repository Facade* — see the arrows with the grey outline in Fig. 7.6.1. Such a *Reference Object* points to the desired Smalldb state machine and allows the application to retrieve the state of the machine, and the persistent representation from the *Persistent Storage* (e.g., the data stored in the SQL database).

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

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

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

7.6.6 Command–Query Separation

The general idea of a Command–Query Separation pattern (CQS) [95, 86] and a closely related Command–Query Responsibility Segregation pattern (CQRS) [96] is to strictly separate components which query data and which modify the data — see Fig. 7.6.2. The reason for such a separation lays in distinct responsibilities and requirements of such components so that they respect the SOLID principles [87] better. Both CQS and CQRS is mostly known from microservice applications, but the concept itself is much older [86].

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

The query component of the CQS pattern is closely related to the Repository pattern [90] (pg. 322), where the repository provides an interface to query the underlying data store. The repository encapsulates the query logic and, e.g., keeps all SQL queries in one place so that they are easier to manage. It also can use various caching techniques to scale the application for large loads. The Smalldb architecture specifies only the repository facade with no details because the underlying mechanism to query the database may vary. When using an SQL database, the repository facade is likely to utilize a query builder, and then it only wraps the result into the Reference objects. However, we may use a fundamentally different data store for some of the entities in the application (e.g., an aggregate root of event sourcing [97] implementation, or sensors connected to a local bus) and then the Smalldb state machine provides a unifying API.

The command component corresponds to the transitions implementation component in the Smalldb architecture. It executes the actions required to transition the state machine from one state to another. To do so, it requires various tools, e.g., to send an e-mail notification. Also, it must ensure data consistency and respect access control rules. Its API focuses on

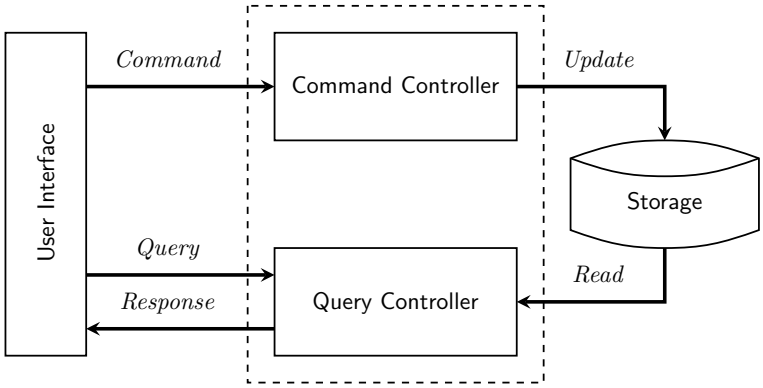


Figure 7.6.2: The core concept of CQS and CQRS

individual transitions (commands) rather than queries and collections. While the CQS pattern does not specify details on how to perform the commands, the Smalldb architecture defines the decorator to validate the requested actions against the formal model, the state machine definition, and ensure that such a transition is valid and allowed. Similarly to the repository facade, the decorator provides a unifying API to the underlying mechanism which performs the transitions. When using an SQL database, the underlying transition implementation will likely execute some update and insert SQL queries; moreover, it may also emit events or send commands to connected devices.

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

7.6.7 Access Control

When the application logic invokes a transition, the request always goes through the *Decorator* (see Fig. 7.6.1) which decides whether the request is valid according to the state machine definition and if so, the *Decorator* executes the *Transition Implementation*. Since the *Decorator* processes all transition invocations, it is the perfect place where to implement an access control mechanism.

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

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

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

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

7.6.8 Consistency

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

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

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

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

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

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

7.7 ORM: Object–Relational Mapping

While many of the web applications today use ORM as their model layer, the Smalldb state machine builds on top of a data access object, as presented in Fig. 7.6.1. So, is it possible to use ORM within the Smalldb state machine?

Before we answer this question, let us take a look at how ORM frameworks work. The first and the most obvious concept of ORM is to map records in the SQL database to objects of object-oriented language. However, the core concept of ORM is the “Unit of Work” [98], [90] (pg. 184). In short, the ORM framework collects modifications done by the application as a “unit of work,” and once the application finishes, the ORM framework applies the modifications to the SQL database. This allows the ORM framework to aggregate lots of calls of setter methods of the mapped objects into a single SQL update query.

The design of Smalldb state machine naturally bounds modifications of the SQL database into state machine transitions, and reference objects can provide caching, thus replacing the need for the Unit of Work as they both solve the very similar problems. However, there is

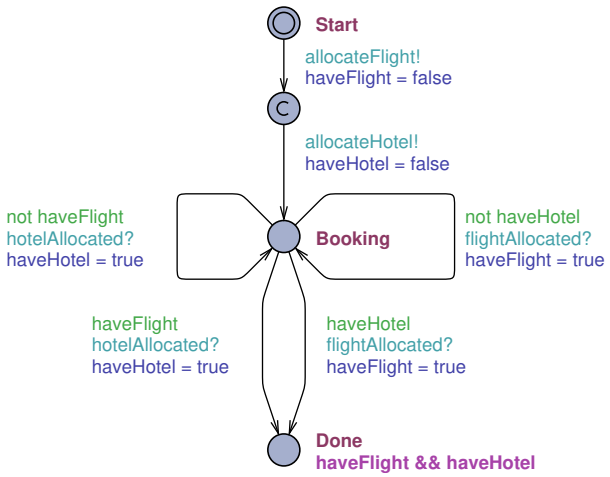


Figure 7.8.1: Example: Booking of a flight and a hotel using Uppaal

no conflict between these two as long as each unit of work stays enclosed within a single state machine transition and the objects passed outside the Smalldb state machine are disconnected from their unit of work to avoid any unexpected behavior when the application logic processes the objects.

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

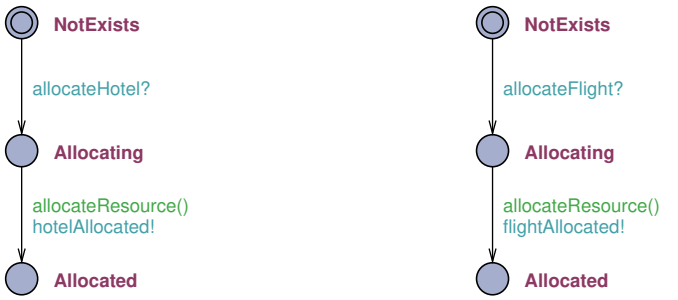
On the other side, ORM frameworks add much complexity to the application, which is not always desirable, and many developers prefer to use more straightforward and predictable tools. The Smalldb framework provides a choice as the repository facade shields the rest of the application from either of the low-level tool.

7.8 Transactional Behavior

The theory of finite automata defines a transition as an atomic operation [21]. A practical implementation of such atomicity is a challenging problem typically solved using locking mechanisms and transactions, which can be rolled back if something unexpected happens. Traditional SQL databases provide us with such mechanisms, and thus we can implement transitions of our state machines as atomic operations. The Smalldb state machine alone has no mechanism to enforce transactional consistency of the transitions, and it fully relays on a correct implementation of transactions within the transitions.

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

The fun begins when dealing with external resources which may become unavailable before the transaction is committed, for example, when booking a flight and a hotel. In such situations, the use of a single state machine transition to allocate multiple resources may be impractical. We may add an intermediate state in which the state machine will wait for the allocation of the resources. The state function of Smalldb state machine (see Sec. 7.5) allows us to asynchronously wait for multiple resources in a single state — the intermediate state may have a transition



(a) Hotel resource (b) Flight resource

Figure 7.8.2: Flight and hotel resources for the example in Fig. 7.8.1

which records successful allocation of each resource, and when the last resource is allocated, the transition will advance to the next state instead of returning into the intermediate state.

Additionally, we may want to represent the resource allocation using an additional simple Smalldb state machine, so that we can track which resources we allocated from 3rd-party services. Then the resource allocation process becomes a network of interacting state machines, and we can use a formal tool, e.g., Uppaal [40], to ensure correctness in such a complicated situation as presented in Fig. 7.8.1 and 7.8.2 (where "allocate!" represents sending a signal to another state machine, and "allocate?" represents receiving of such a signal; failure recovery neglected for simplicity).

7.9 Case Study: Application Development

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

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

While we do not wish to discuss particular details and complications of the business processes that this information system supports, we would like to provide a glimpse of the overall complexity of the application and its entities — it started with 18 state machines of which eight are core machines we designed on a single sheet, see Fig. 7.9.1. As we can see, many entities involve some nontrivial behavior which needs to be specified, modeled, implemented, and tested. To do so, the precise description of their behavior needs to be passed among various groups of people — from the training facility employees demanding the information system, to software architects, developers, and testers.

7.9.1 Analysis

The initial analysis showed, that while the application is rather small, the behavior of the core entities is difficult to comprehend because the application deals with users with vastly distinct workflows due to their situation and environment. These workflows required thorough discussions with the customer, and we needed to find common ground to understand what the users are expected to do. In particular, some of the users were people attending the courses and

paying themselves in advance, while others were companies ordering courses for their employees in bulk, some of them even billed monthly with pre-approved limits instead of per course basis. The goal was to provide a unified ordering process which could satisfy the needs of all customers without unnecessary complications to either side.

Such a complex workflow requires one thing more than anything — thorough communication with the customer (or users) to get the process right. The state diagram sheet presented in Fig. 7.9.1 proved invaluable during the discussions over the workflows because it provided a representation of the application internals, which was understandable by both developers and the customer. Even non-technical customer can, with some basic explanation, understand the concept of a state diagram and what it represents in respect to the discussed business process. In fact, our experience showed that the customer was able to find mistakes in our models while discussing the diagram sheet.

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

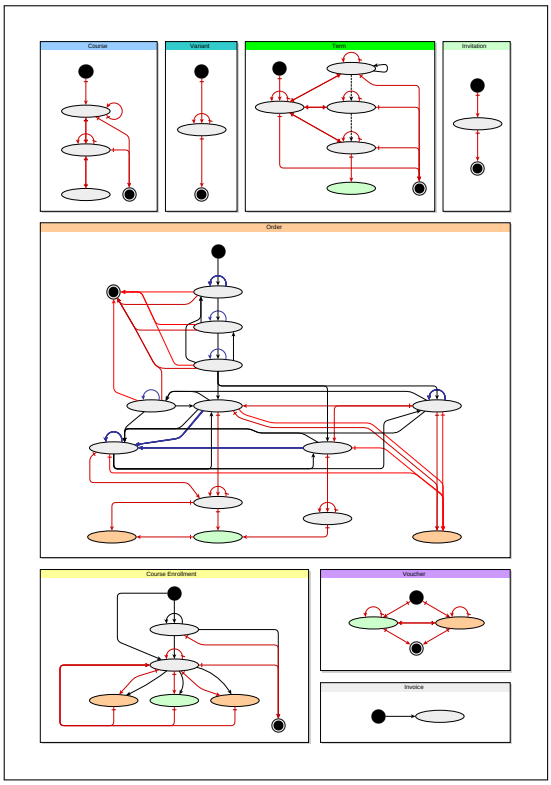


Figure 7.9.1: State diagram sheet preview to present the complexity of the application

7.9.2 Prototyping

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

To advance from the formal model and drafts to a minimal working implementation, each entity used in the application needs to be defined. The basic idea is to utilize the diagram sheet as much as possible to avoid duplicate efforts in the design and implementation. Such a definition consists primarily of two things: The first is a link to the state diagram in the state diagram sheet; in this case, it was an ID of the element in the GraphML file of the state diagram sheet created using yEd editor. The second item that is entered to the definition is the name of an SQL table, where the state machine stores its data. Remaining parts of the definition include mapping database columns to object properties, access control rules (see Sec. 7.6.7) and

various metadata, e.g., labels and icons, to automatically render navigation menus and buttons — these parts are mostly indifferent to the state machine itself.

7.9.3 Implementation

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

Watching the number of unimplemented transitions also gives us a good estimate of how much of the project remains. Unfortunately, each transition takes a different amount of time to implement because with such a transition we need to implement tools (services) it uses and user interface as well; therefore, this metric is only partially useful for time estimates but useful nonetheless.

7.9.4 Changes During the Implementation

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

More substantial changes in the transitions, e.g., where preconditions or postconditions change, required changes in the transitions implementations to preserve consistency between the definition and implementation (see Sec. 7.6.8). However, such changes were limited to the respective methods implementing the affected transitions, and there is no generated code which could interfere with the manual changes.

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

7.10 Future Work

7.10.1 Internet of Things and Microservices

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

Similarly, we can migrate from the simple data access object to microservices, CQRS [95, 96], and Event Sourcing [97]. Because of the separation of transition implementation (the "command") and repository (the "query") are already separate in the Smalldb architecture, it should be simple to implement the CQRS pattern and send events via a message broker instead of updating an SQL database.

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

7.10.2 Fluent Calculus

Smalldb state machines use names to identify its states and the state function to map a representation from the repository to one of the state names. Such a mechanism is universal and should fit most practical scenarios; however, in its generality, it does not provide us with any semantic information about the state. If we wish to reason about the states and properties of the system in such states, it would be useful to extend the concept of state function with a more advanced formalism; e.g., Situation Calculus [99] and Fluent Calculus [100], or Event Calculus [101].

Fluent Calculus [100] represents states using fluents (predicates). Each state is a composition (a set) of such fluents. Depending on the valuations of the fluents, we can decide in which state we are. Therefore, we could define fluents to describe various aspects of the representation from the repository, and then use such fluents to construct the state function for the Smalldb framework. Such an approach would enable the use of Fluent Calculus to reason about the states and the identification of operations needed to transition between two given states. Since Fluent Calculus is used in planning [53] (robotics; including temporal planning [102]), and automated web service orchestration [103], we could gain access to many advanced formal tools while developing web applications.

7.10.3 Towards Formal Proofs and Automated Planning in Web Applications

At the highest level, we should be able to verify the correctness of the model in the form of state diagrams or business process diagrams [3] using simulations and various model checking techniques verifying networks of interacting finite automata [40], where some of the automata represent the users and others represent the application entities.

While the Smalldb framework provides us with the required infrastructure connecting the high-level model to low-level code (transition implementation), we still need to provide a formal bridge between these layers [104]. For example, we may utilize the earlier discussed Fluent Calculus (or a similar formal tool) to enhance the basic universal infrastructure with a powerful formal tool, so that we can model and reason about various aspects of the underlying implementation.

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

Finally, we need to implement the transition. Because of the use of Fluent Calculus, we can expect to have a much detailed specification of the transition — not only names of the surrounding states but also some basic predicates about the states. All we need to do is to infer the executable code or verify the existing implementation. In general, the verification task is

impossible (e.g., because of the halting problem); however, the transition implementation in a practical application should be very simple and static analysis may provide useful results in the most cases.

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

7.11 Conclusion

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

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

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

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

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

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

Chapter 8

From a BPMN Black Box to a Smalldb State Machine

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. BPMN (Business Process Model and Notation) 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.

8.1 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, statecharts 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 8.9.5), which inspects interactions of a chosen participant in a BPMN diagram (a trivial example of such an interaction is in Fig. 8.1.1) 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 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 [8].

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.

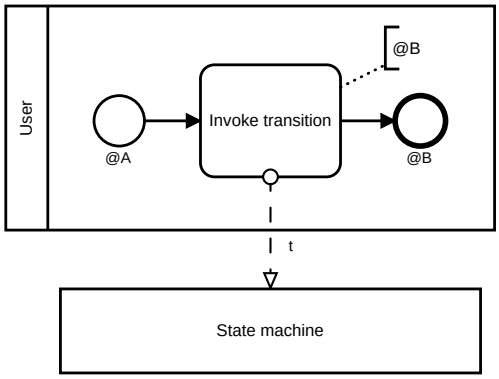


Figure 8.1.1: BPMN diagram (with state annotations) of a user invoking a state machine transition t

8.2 BPMN

8.2.1 The Notation

BPMN 2.0, the business process modeling notation [8], defines a graphical syntax for describing a business process using an oriented graph; see Fig. 8.2.1. Each participant is represented by a "pool" ("User" and "State machine" in Fig. 8.2.1) 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. 8.6.2) 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.

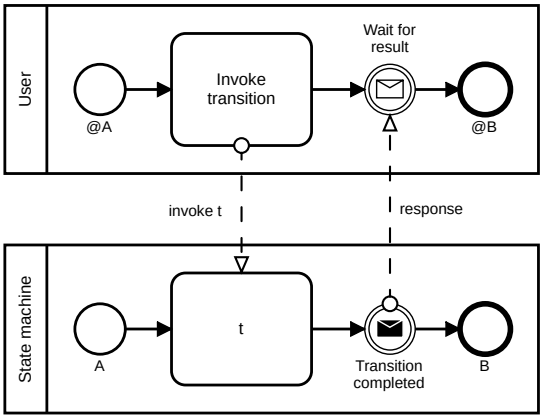


Figure 8.2.1: Transition invocation from Fig. 8.1.1 in detail

8.2.2 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. 8.2.1. Fig. 8.1.1 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.

8.2.3 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. 8.2.1, 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. 8.1.1. 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. 8.2.1), 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. 8.2.1) may be omitted in any case; the returning message flow then originates from the preceding task node.

An important detail in Fig. 8.1.1 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.

8.2.4 BPMN, Petri Nets, and a Group of Finite Automata

A BPMN diagram forms a graph, which can be converted to a Petri Net [41, 105]. 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 [43]. 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.

8.3 Smalldb State Machine

Chapters 6 and 7 already presented the Smalldb state machines in detail. While the STS algorithm is designed to work with Smalldb state machines, it requires only a few of their fundamental properties. Thus, let us recapitulate what we will need in the following sections.

For the purposes of the STS algorithm, we can interpret the Smalldb state machine [1] solely as a persistent nondeterministic finite automaton, that implements the model layer (i.e., M in MVC) of traditional server-side web applications. 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 complexity and to provide as much convenience as possible is because the target users are also customers who may discuss the statecharts with software architects. Our experience shows that a nontechnical customer can understand such statecharts 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.

8.4 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[1]. 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.

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.

8.5 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. 8.5.1 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. 8.5.1, we can identify that the issue tracking application needs to create an issue and then close the created issue, as pictured in Fig. 8.5.2. 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 [106], particularly in the natural transformations between functors and commutative diagrams [107]. 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?

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

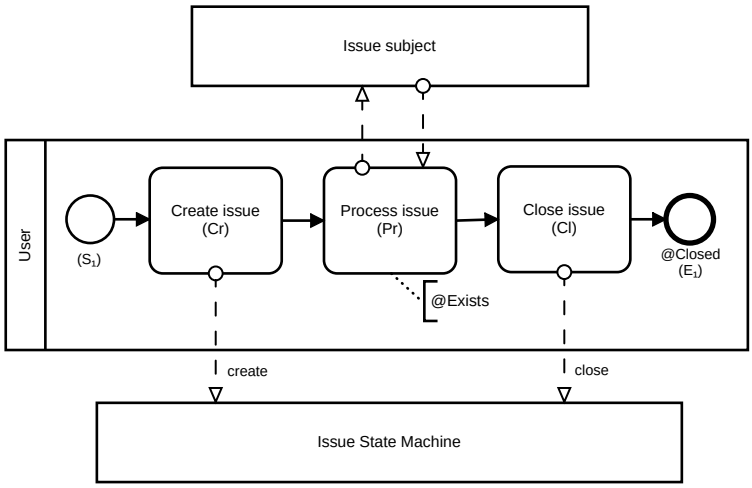


Figure 8.5.1: A simple issue tracking — invocation of transitions “create” and “close”; the state machine presented as a black box

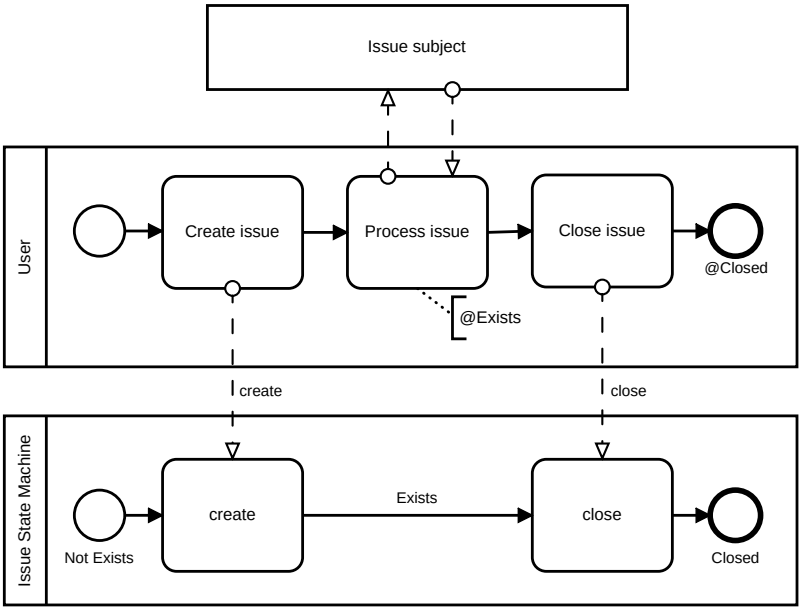


Figure 8.5.2: Isomorphic processes in detail (compare with Fig. 8.5.1)

8.6 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 8.9, along with the complete description of the algorithm.

8.6.1 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 8.2.3; 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. 8.1.1). 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.

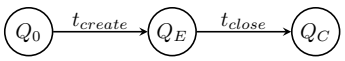


Figure 8.6.1: The state machine inferred from Fig. 8.5.1

For example, Fig. 8.5.1 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. 8.5.1, the STS algorithm can infer the "Issue state machine" as presented in Fig. 8.6.1. 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 8.11 will present the details of the STS algorithm execution for the example in Fig. 8.5.1 once we define the algorithm.

8.6.2 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. 8.6.2 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).

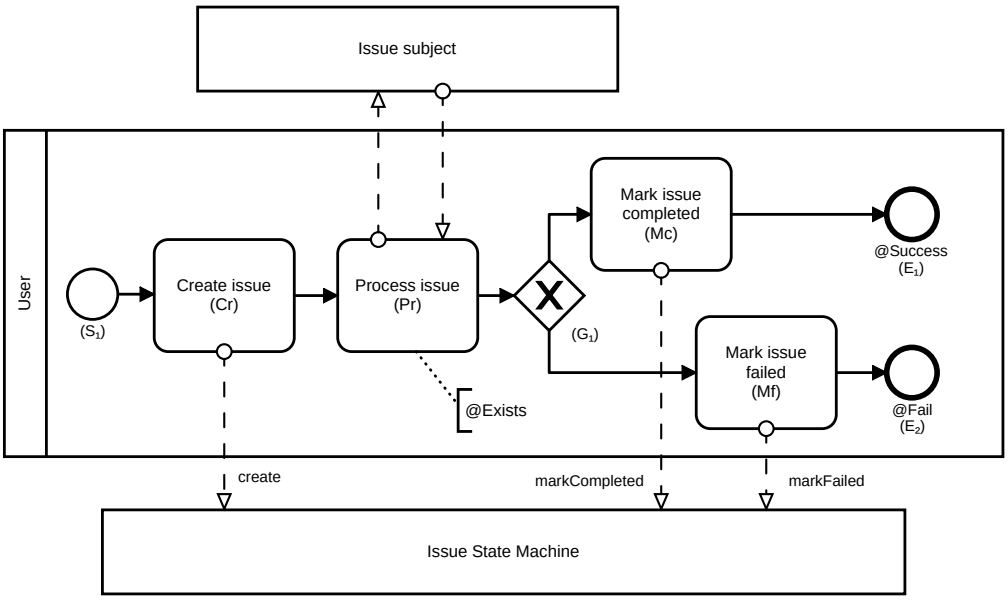


Figure 8.6.2: User decides scenario

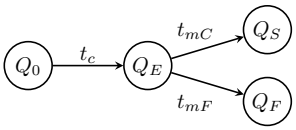


Figure 8.6.3: The state machine inferred from Fig. 8.6.2

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. 8.6.3. 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[2].

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

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[3]. Section 8.12 will present the details of the STS algorithm execution for the example in Fig. 8.6.2.

8.6.3 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, 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. 8.6.4), 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. 8.6.5. 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 8.3. Section 8.13 will present the details of the STS algorithm execution for this example.

8.6.4 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 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. 8.5.1) so that one user, Alice, creates an issue and another user, Bob, closes the issue. Fig. 8.6.6 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

[3] As long as the path does not include anything within the inferred state machine. We will get to this detail later, in Section 8.9.

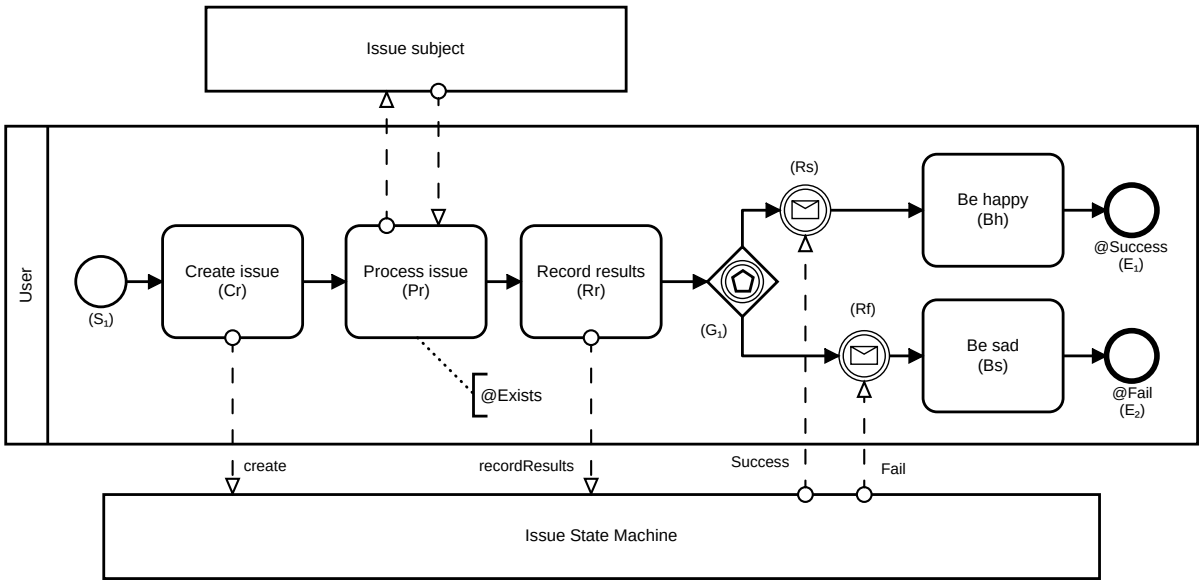


Figure 8.6.4: Machine decides scenario

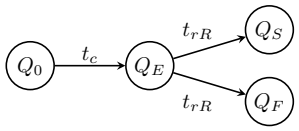


Figure 8.6.5: The state machine inferred from Fig. 8.6.4

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. 8.6.1) 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 8.14 will present the details of the STS algorithm execution for the example in Fig. 8.6.6.

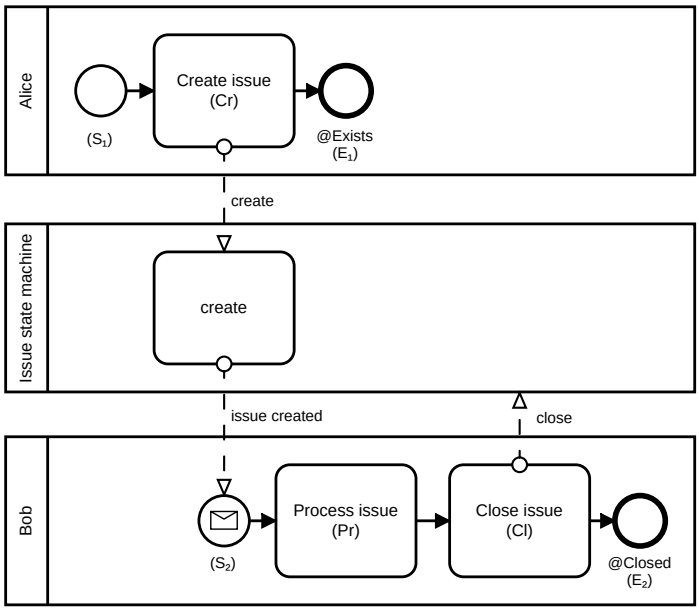


Figure 8.6.6: Synchronization between users

8.7 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. 8.1.1, 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.

8.8 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 $\twoheadrightarrow$ 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 [108]) 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 $F_2 = \{(x \to y), \dots\}$. Additionally, for $f_2 \in F_2$, we can write the following:

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

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:

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

8.9 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 h 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 8.6.3. The example input of the algorithm is presented in Fig. 8.9.1. 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.

8.9.1 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 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. 8.9.1): $I = \{A, C\}$ 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.

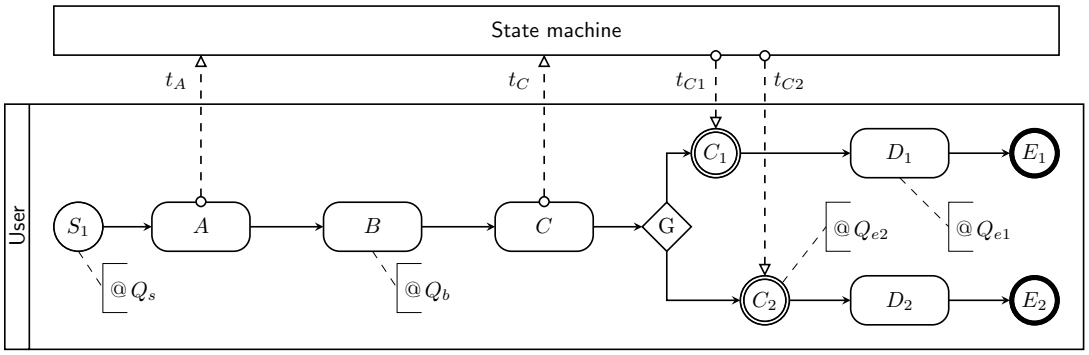


Figure 8.9.1: Example — Source BPMN diagram

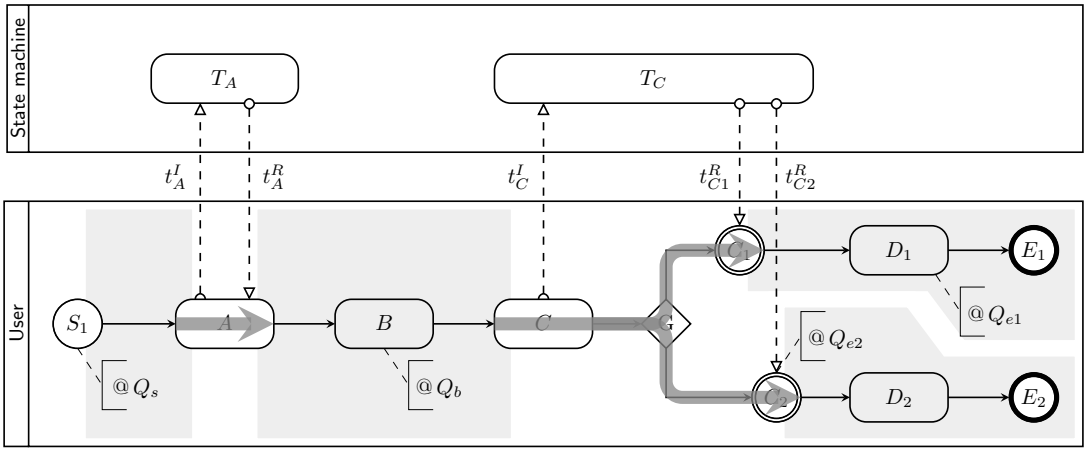


Figure 8.9.2: Example — Transition detection (grey bold arrows) and state detection (grey areas)

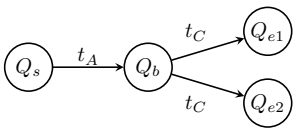


Figure 8.9.3: Example — Inferred state machine

8.9.2 Stage 2: Transition Detection

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[4] 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. 8.9.2: 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. 8.9.2.

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 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 8.5), the state machine task and the returning flow may be omitted in simple cases[5] without losing any information from the diagram. As discussed in Section 8.6.1, 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

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

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

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. 8.9.2: 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.

8.9.3 Stage 3: State Detection

State Annotations

As explained in Section 8.7, 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 $A(n_a) \in Q$.

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[6] 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 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

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

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[7] and the transition regions[8] of a BPMN diagram.

Example — Fig. 8.9.2: The gray areas mark regions of reachability. The state relation is the following:

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

8.9.4 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.

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. 8.9.2, 8.9.3: 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.

Example — Fig. 8.9.4, 8.9.5: Since there are no annotations on a path $S_1 \twoheadrightarrow R$, the "Not Exists" states 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.

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

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

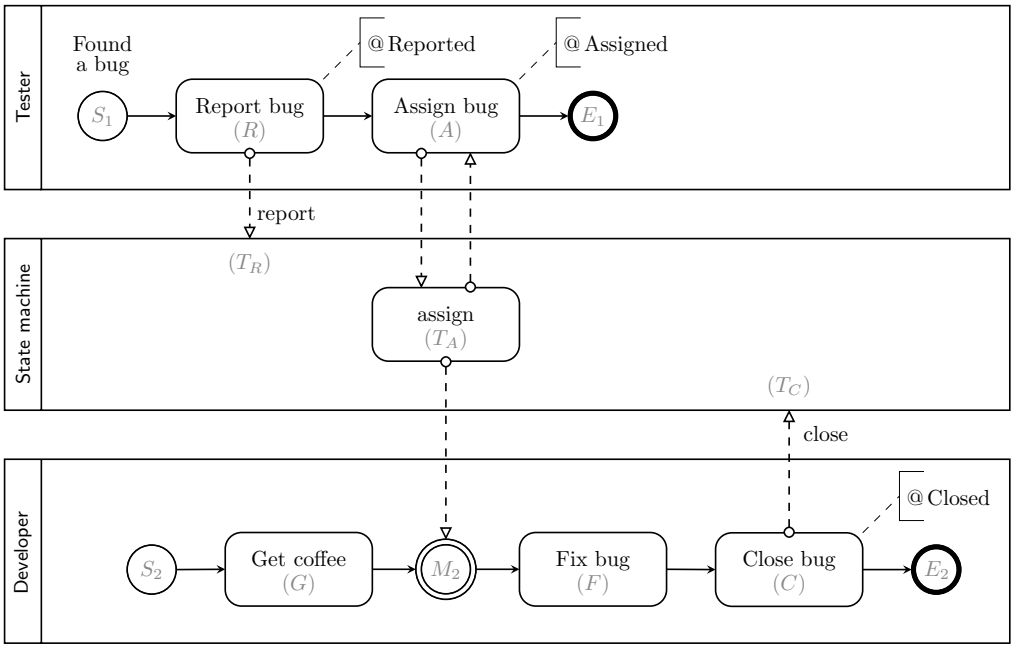


Figure 8.9.4: State propagation between multiple participants

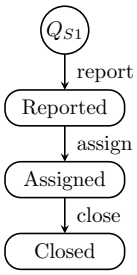


Figure 8.9.5: The state machine inferred from Fig. 8.9.4

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. 8.9.4, 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 $M_2 \rightarrowtriangle C$). As we can see, the path $S_2 \rightarrowtriangle M_2$ 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.

8.9.5 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:

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

- 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. 8.9.3: From the previous stages, we know that the transition relation T and state relation S are:

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

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

$$Q = \{Q_s, Q_b, Q_{e1}, Q_{e2}\}$$
$$M = \{t_A, t_C\}$$
$$\alpha = \{((Q_s, t_A), \{Q_b\}), ((Q_b, t_C), \{Q_{e1}\}), ((Q_b, t_C), \{Q_{e2}\})\}$$

8.9.6 Summary of the STS Algorithm

In the previous sections, we learned that the input of the STS algorithm is the BPMN diagram B (Fig. 8.9.1). There are two important intermediate results: the transition relation T, which connects invoking to receiving nodes (Fig. 8.9.2), and the state relation S, which connects the remaining fragments of the diagram not covered by T. The desired state machine (Fig. 8.9.3) is then created by combining the two relations together.

Note that the operators V, E, σ, τ, λ, and $\twoheadrightarrow$ are defined in Section 8.8.

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[9] $\alpha(q_t, m) \mapsto Q_{t+1}$, 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. 8.9.1). 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. 8.9.1)
  - 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[10] P_T:
    $P_T(p = a \twoheadrightarrow b) \equiv ((I \cup R_m \cup V_S \cup V_e) \setminus \{a, b\}) \cap p = \emptyset$
  - 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. 8.9.2)
  - 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. 8.9.2.)
  - Receiving nodes:
    $R = R^+ \cap \bigcup \tau(T) = \bigcup \tau(T),\ R \subseteq R^+$

[9] 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.

[10] $P_T(p)$ 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.

- Stage 3: State detection (Sec. 8.9.3)
  - Path constraint predicate P_S:
    $P_S(p = a \twoheadrightarrow b) \equiv ((I \cup R^+ \cup V_S) \setminus \{a, b\}) \cap p = \emptyset$
  - State relation S (ternary) from $R^+ \cup V_{start}$ to $I \cup V_{end}$ (by constructing N_i and collecting annotations[11] 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. 8.9.2.)
- Stage 4: Implicit state labeling (Sec. 8.9.4)
  - 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. 8.9.5):

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

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

8.9.7 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 nodes terminated with a receiving node (forming a T-shaped graph; see Fig. 8.9.6), 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

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

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. 8.9.7. The shape of the generated graphs followed the examples presented earlier: a chain of N simple tasks (N in Fig. 8.9.7), 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. 8.9.7.

The second, final, case was the mentioned T-shape graph (Fig. 8.9.6) 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. 8.9.7. 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.

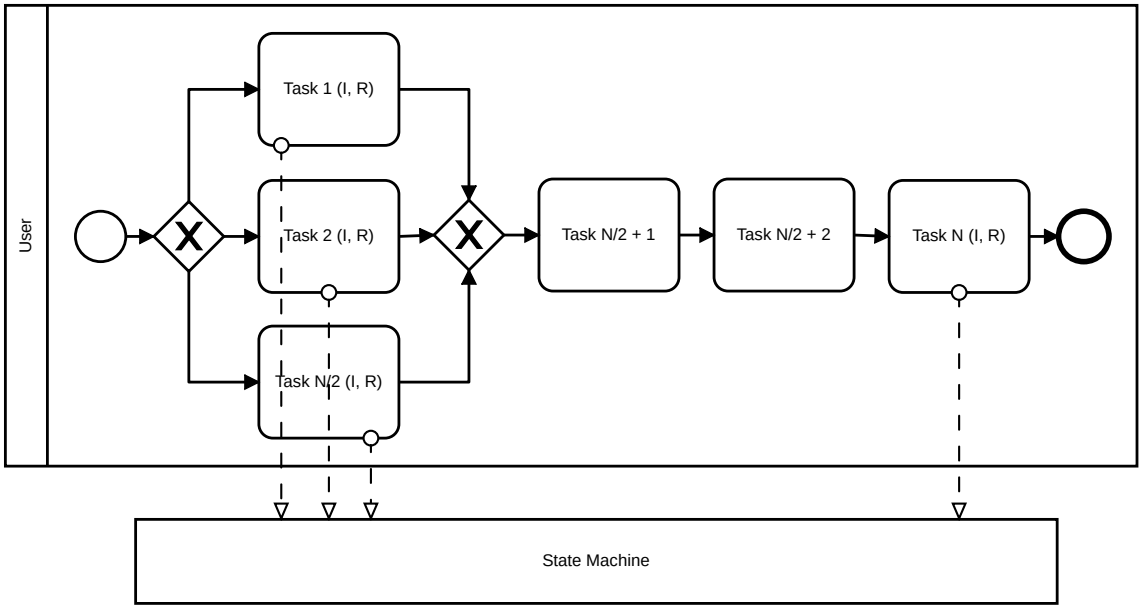


Figure 8.9.6: T-Shape BPMN Diagram

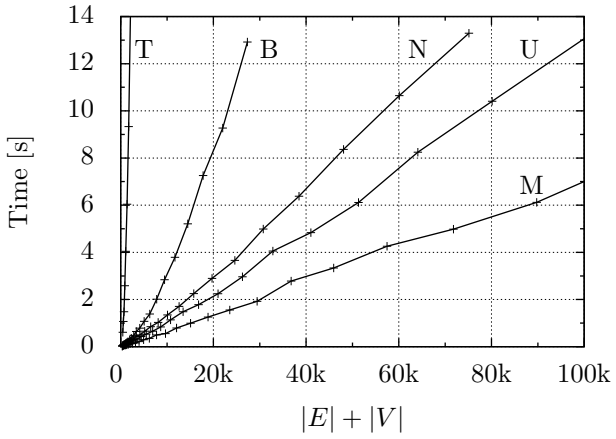


Figure 8.9.7: Processing speed of the tested scenarios: a chain of tasks (N), user decides (U), machine decides (M), a combination of user decides and machine decides (B), and a T-shape BPMN diagram (T),

8.10 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[12]. 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. 8.10.1 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 8.6.1.

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. 8.10.2.

To obtain better insight into how the algorithm inferred the state machine, the gray areas in Fig. 8.10.1 present detected states as a partitioning of the BPMN diagram (similar to Fig. 8.9.2). 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:

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

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

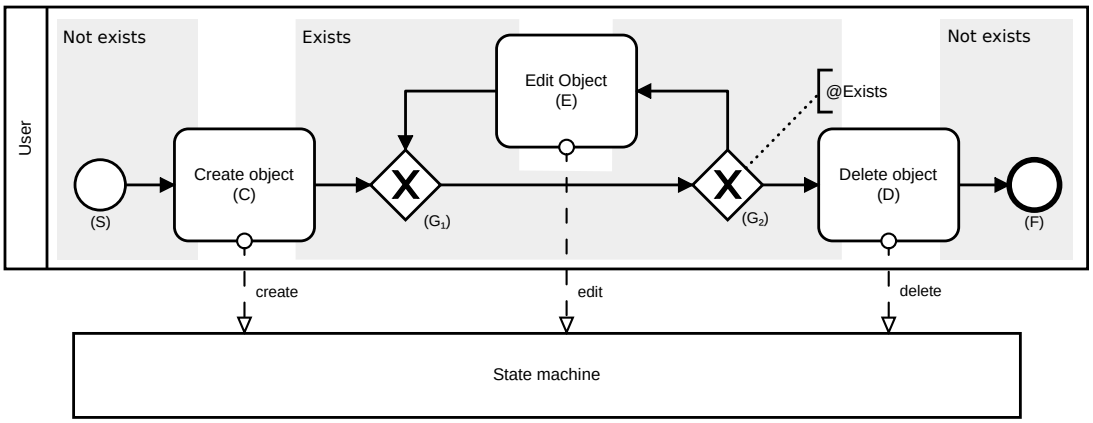


Figure 8.10.1: Crud entity in a BPMN diagram (The grey areas in the background are not present in the source diagram.)

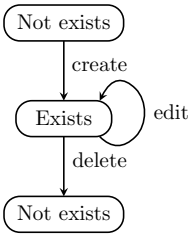


Figure 8.10.2: The state machine inferred from Fig. 8.10.1

Once S and T are combined together, the algorithm provides us with the state machine as presented in Fig. 8.10.2. Note how the gray regions in Fig. 8.10.1 become the states in Fig. 8.10.2 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.

8.11 Example: Simple Task

In Section 8.6, 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. 8.6.1), in which a user invokes two subsequent state machine transitions ("create" and "close") as presented in Fig. 8.5.1. The result is a state machine of two transitions and three states, as shown in Fig. 8.6.1 (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. 8.5.1, and then it infers the transition relation T and state relation S as follows:

$$I = R = R^+ = \{Cr, Cl\}$$

$T = \{(Cr, \{Cr\}, \text{create}), (Cl, \{Cl\}, \text{close})\}$
$S = \{(S_1, \{Cr\}, \{\text{Not Exists}\}), (Cr, \{Cl\}, \{\text{Exists}\}), (Cl, \{E_1\}, \{\text{Closed}\})\}$

8.12 Example: The User Decides

The scenario discussed in Section 8.6.2 presents a situation in which a user creates an issue and later decides whether the process has completed or failed — see Fig. 8.6.2. The result is a state machine of three distinct transitions, as presented in Fig. 8.6.3 (notice transitions t_{mC} and t_{mF}).

The STS algorithm identifies the invoking nodes I and receiving nodes R in Fig. 8.6.2, and then it infers the transition relation T and state relation S as follows:

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

8.13 Example: The Machine Decides

In contrast with the previous example, the scenario discussed in Section 8.6.3 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. 8.6.4. The result is a state machine of one deterministic transition and one nondeterministic transition, as presented in Fig. 8.6.5 (notice the two arrows both labeled t_{rR}).

The STS algorithm identifies the invoking nodes I and receiving nodes R in Fig. 8.6.4, and then it infers the transition relation T and state relation S as follows:

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

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.

8.14 Example: Synchronization between Users

The last example from Section 8.6 presents a situation when one user notifies another user via the state machine, see Section 8.6.4 and Fig. 8.6.6. The result is the state machine same as in the Simple Task Example, see Fig. 8.6.1. The difference is in who invokes which transition.

The STS algorithm identifies the invoking nodes I and receiving nodes R in Fig. 8.6.6, and then it infers transition relation T and state relation S as follows:

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

8.15 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.

8.15.1 The Scenario

The BPMN diagram presented in Fig. 8.15.1 describes the interaction between a web application and three people, namely, a customer, a chef, and a delivery boy. The source BPMN diagram

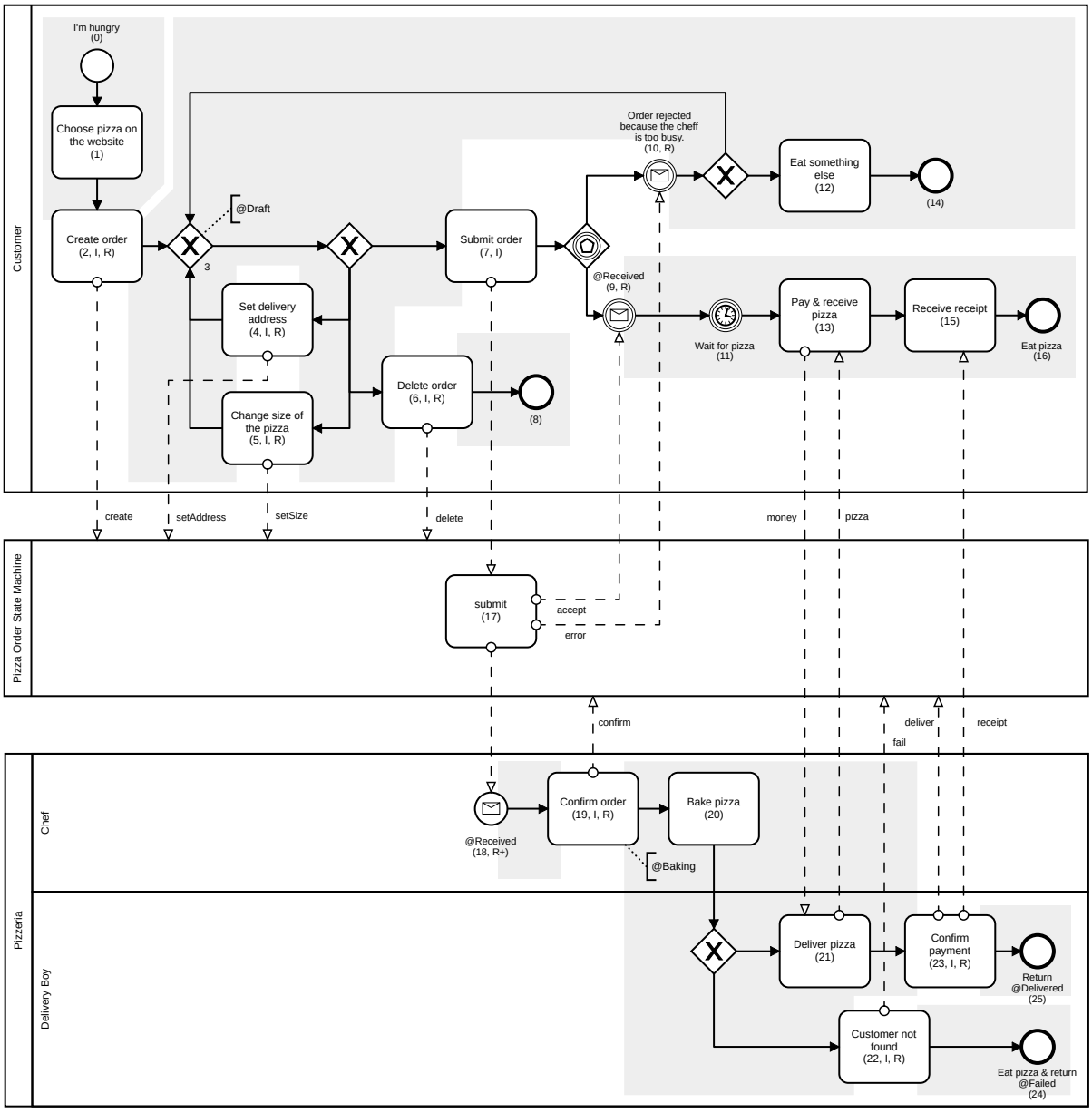


Figure 8.15.1: Pizza delivery example (The grey areas in the background are not present in the source diagram.)

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. 8.15.1). 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.

8.15.2 The Features

Earlier, in Section 8.6, 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. 8.6.1), 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. 8.6.2) 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).

The "machine decides" scenario (Sec. 8.6.3) 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. 8.6.4; nodes 7, 17, and 18).

8.15.3 Highlights from the STS Algorithm Execution

Once we let the STS algorithm process the BPMN diagram (Fig. 8.15.1), 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:

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

8.15.4 The Result

The result is the Smalldb state machine pictured in Fig. 8.15.2. 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.

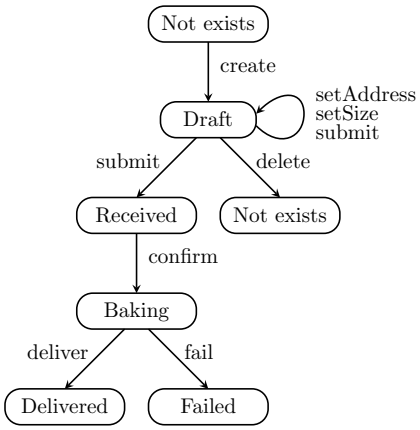


Figure 8.15.2: The Pizza Order State Machine (from Fig. 8.15.1)

8.16 Related Work

8.16.1 UML Sequence Diagrams

UML sequence diagrams [7] 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. 8.16.1 presents the simple CRUD entity described in Section 8.10. In comparison with Fig. 8.10.1, 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 non-technical 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.

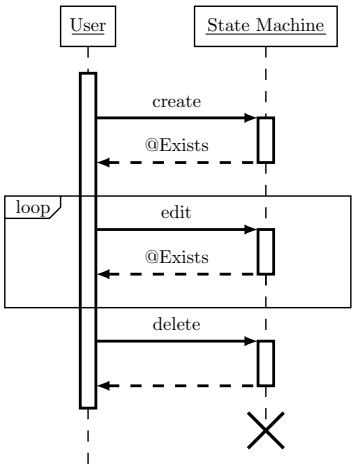


Figure 8.16.1: CRUD entity in a UML sequence diagram

8.16.2 BPMN Execution Engines

Efforts concerning the direct interpretation and execution of BPMN diagrams [8] (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 [11, 109], iterate BPMN diagrams as if they were Petri Nets and perform specified actions via (micro)service [110] orchestration or human task flow control. While this approach seems to be reasonable and may help 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.

8.16.3 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 [82], 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.

8.16.4 Induction of Regular Languages

Various approaches to finite automata synthesis, such as grammatical inference, induction of regular languages, automata learning [111, 112, 113], 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.

8.17 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.

Chapter 9

Business Process Simulation and Verification

Having the application built around the formal models enables us to reason about the application via reasoning about the models on which it stands. We may reason about the individual state machines, how they cooperate, or we may include external entities in our simulations and analyze how the business process functions as a whole.

Section 6.8 already explored some possibilities in state machine verification, for example, state reachability that can point out obvious errors in the state machine definition. In this chapter, we will take a look at how we can verify the cooperation of multiple Smalldb state machines.

9.1 The Network of Interacting State Machines

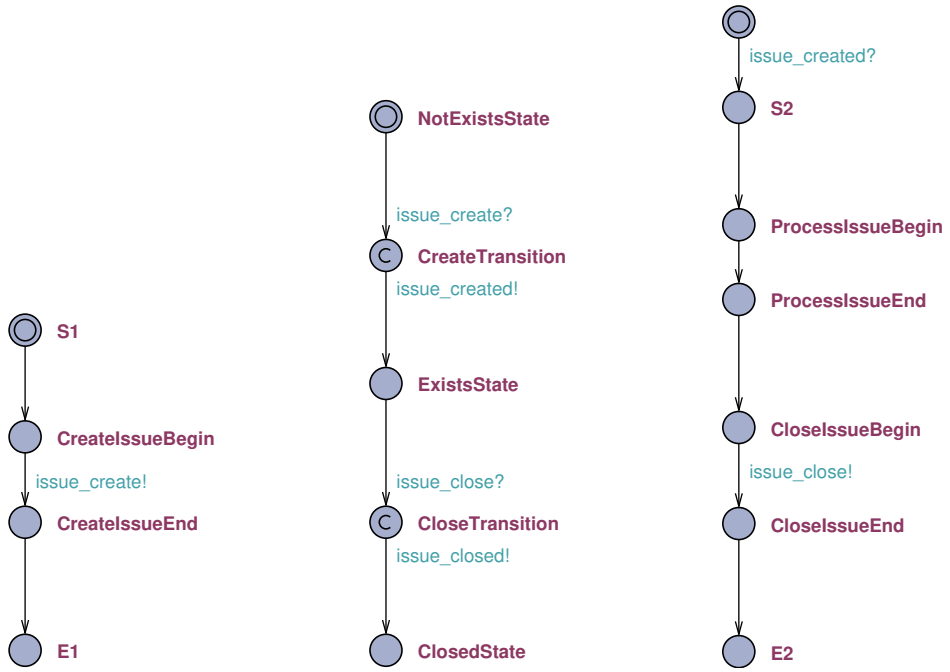
The STS algorithm shows us that we can represent a BPMN participant with a Smalldb state machine (with some preconditions; see Sec. 8.2.4). If we model all participants as a state machine, we can use tools like Uppaal [40] to simulate the business process and verify some of its properties.

Uppaal uses "channels" for the synchronization with an exclamation mark to send a notification and a question mark to wait for notification — for example, the transition labeled "issue_create!" triggers transitions labeled "issue_create?". In BPMN, this concept is represented using message flows.

For example, we can redraw the synchronization example from Section 8.6.4 to Uppaal, as presented in Figure 9.1.1. In this rather trivial scenario, we have a Uppaal system of three automata, where Alice creates an issue, then Bob gets notified about the new issue, deals with it, and finally, he closes the issue.

Smalldb state machines, like the Issue state machine, can be converted to Uppaal automatically by adding a microstep (in the form of a state) into each transition. The transition to the microstep will wait for a notification on a channel, and transitions from the microstep will broadcast notifications that the transition had happened. The Smalldb state machine in Uppaal will be represented as a bipartite graph where one group of nodes represents the states, and the other group represents the transitions, as we can see in Figure 9.1.1b. Nondeterministic transitions will have multiple edges coming from the microstep.

Other participants may be converted to Uppaal in various ways. As long as there are no parallel executions within a single participant, we can redraw BPMN participants into Uppaal



(a) Alice (b) Issue State Machine (c) Bob

Figure 9.1.1: Synchronization example in Uppaal (see Sec. 8.6.4)

using two states for each task node so that the intermediate transition can notify other state machines. For example, see Fig. 9.1.1a, Alice notifies the Issue state machine when creating the issue. In case we do not use BPMN, we may export state machines from Smalldb automatically and then model other participants manually in Uppaal.

9.2 Verifying the Model

Once we have a Uppaal system created, we can define requirements using modal logic to ask whether good things can eventually happen, and a bad thing will never happen. For example, we may ask if whenever Alice opens an issue, Bob will eventually close the issue (see Fig. 9.1.1):

```
E[] Alice.CreateIssueBegin imply Bob.CloseIssueEnd
```

In more practical scenarios, we may, for example, want to ask whether a customer who already paid always receives the ordered goods or gets the money back. We may also ask whether the process does not get stuck somewhere. Since Uppaal uses timed automata, we may also ask how long a particular process take, and whether a specific task can be done in the available time — this opens possibilities of validating QoS (Quality of Service) requirements.

9.3 Building and Extending the Model

The Uppaal provides much more features than a simple synchronization. It also implements variables, selections, guards, and time. Moreover, we can have multiple instances of each automaton. Unfortunately, Smalldb state machine definitions are unlikely to express most of these features because they describe internal properties of individual transitions that Smalldb outsources to their respective nonmodeled implementations.

On the other hand, we can still convert the state machines between Smalldb and Uppaal automatically, meaning we can export a state machine from Smalldb to use it in an Uppaal system, or we can load an automaton from an Uppaal system and use it in Smalldb. Therefore, we can use Uppaal as a prototyping tool, which will help us determine the essential properties of each transition and state invariants, even though we will have to reimplement these manually in the real application.

The technical details of synchronization between Smalldb and Uppaal remain an open question. In case we infer state machines from BPMN diagrams, we would need to improve nondestructive updates of Uppaal systems. In case we use Uppaal to draw Smalldb state machines, we would need to detect the microsteps correctly and declare some conventions for custom metadata and guards used for access control.

9.4 Tested on Students

The concept of modeling a business process around a Smalldb state machine was tested on master-degree students in the form of semestral assignments in the "Software Testing and Verification" course. The students received a Smalldb state machine definition (a statechart), and they were told to define other participants to create a plausible business process. Then they should use the verification features of Uppaal to prove some nontrivial property of the business process.

The assigned state machines reflected practical applications like a taxi ride, an article in a redaction, or a lifecycle of a product in an e-shop. Some assignments were directly taken from real-world applications, particularly, issue tracker and a process lifecycle in Supervisord (a service supervisor daemon). Each state machine had five to seven states and was of complexity comparable to real-world applications.

The initial reaction of the students was typically somewhat surprised that they are supposed to implement users rather than the software. Clearly, it was one of the less traditional assignments.

The students were mostly successful, and they understood the assignment quite well. Even though the majority of them had little to no practical experience with software development outside the school, they managed to produce meaningful scenarios. In practice, similar scenarios could help with the application design and with verifying the intended business process.

This experience showed us that such use of formal models is a viable approach and does not require deep knowledge in the formal methods as the students succeeded after only a few lectures on this topic.

The second thing this experiment showed was that the quality and availability of the tools is a crucial obstacle for practical applications. Also, even though the assigned state machines were available in a machine-friendly form (JSON and Graphviz DOT formats), none of about 90 students wrote a few lines of code to import the state machine into the Uppaal. It is a chicken and egg problem, where the formal tools are not used in practice because the research prototypes are not very practical, and due to the low demand for such tools, the tools do not get improved. We hope that just like a lizard predates both the chicken and egg, the solution to this problem lies in the use of simpler formal models that are sufficiently supported by existing tools, and thus they will be easier to adopt in the general practice not only in specialized applications.

Part III

Software Synthesis

Chapter 10

Cascade of the Blocks

When building complex applications, the only way not to get lost is to split the application into simpler components. Modern programming languages, including object-oriented ones, offer very good utilities to create such components. However, when the components are created, they need to be connected together. Unluckily, these languages are not a very suitable tool for that. To help with the composition of the components, we introduce *cascade* – a dynamic acyclic structure built from blocks, inspired by the Function Block approach. The cascade generates itself on-the-fly during its evaluation to match requirements specified by input data and automatically orders the execution of the individual blocks. Thus the structure of a given cascade does not need to be predefined entirely during its composing/implementation and fixed during its execution as most approaches usually assume it. It also provides a real-time and fully automatic visualization of all blocks and their connections to ease debugging and an inspection of the application.

10.1 Introduction

In the last 30 years, object-oriented languages have developed to state, where they are a pretty good tool for creating components, but when it comes to composing these components together, the situation is far from perfect.

There are not many successful and widely used tools or design patterns to compose applications. Probably the most known approach, which is well established in the field of programmable logic controllers, is *Function Blocks* [9], and its simplified variant, used by Unix shells, known as *Pipes and Filters*. Despite these approaches are decades-old [114], they are used in only a few specific areas, and there is very low development activity in this direction.

A basic idea of both Function Blocks and Pipes and Filters is to split a complex application to simpler blocks and then connect them together using well defined and simple interfaces. This adds one level of abstraction into the application and simplifies all involved components significantly. Simpler components are easier to develop. Well defined interfaces improve the reusability of the components. In total, it means faster and more effective development.

From another point of view, the connections between blocks can be easily visualized in a very intuitive way, and conversely, these connections can be specified using a graphical editor. This significantly lowers programming skill requirements and allows non-programmers to build or modify applications if a suitable GUI tool is provided.

The main limitation of Function Blocks is that blocks are often connected together in advance by a programmer, and this structure remains static for the rest of its lifetime. Therefore, an application cannot easily adapt itself to changing requirements and environment; it can usually change only few parameters. The next few sections will present how to introduce dynamics into

these static structures and what possibilities it brings.

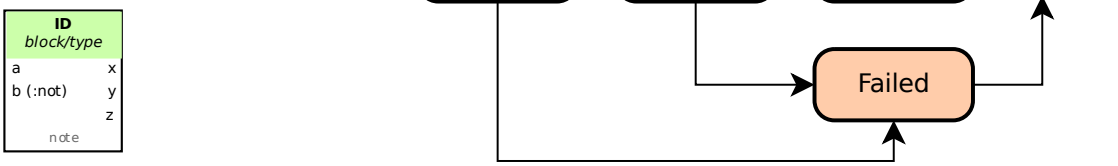
This chapter introduces a *cascade*, a dynamic acyclic structure built of blocks. The next two sections (10.2, 10.3) describe the blocks and how they are composed into a cascade, including a description of their essential properties. Then, we explain the most interesting features of the cascade in Section 10.4. Finally, in the last two sections (10.6, 10.5), practical use of cascade is described, and it is also compared to existing tools and approaches.

10.2 Creating the Blocks

As mentioned above, object-oriented languages are very good tools to create components. So it is convenient to use them to create blocks.

In cascade, a block is an atomic entity of a given type (class), which has named inputs and outputs, each block instance is identified by a unique ID, and can be executed. During the execution, the block typically reads its inputs, performs an action on received data, and puts results to its outputs.

The symbol used in this paper to represent a block is in Figure 10.2.1a. There are an ID and a block type in the header, named inputs on the left side, and outputs on the right. The color of the header represents the current state of the block. At the bottom, a short note may be added, for example, a reason of failure.



(a) The block symbol

(b) Lifetime of the block

Figure 10.2.1: The block

The block is implemented as a class, which inherits from the abstract block class. This abstract class implements a required infrastructure to manage inputs, outputs, and execution of the block. The execution itself consists of calling the `main` method of the block class.

Each block instance is executed only once. During its lifetime, the block goes through a few states, as presented in Fig. 10.2.1b. It starts in the state *queued*, where waits for execution. Then it enters state *running*, and when execution is completed, one of the final states *zombie* (success), *disabled*, or *failed* is entered. In the final state, the block exists only to maintain its outputs for other blocks[1]. This means that the block can process only one set of input data. To process another data set, a new block instance must be created. However, because the lifetime of the block ends before another input data arrive, it causes no trouble (this will be explained later).

[1]Therefore the successful state is called *zombie*, like terminated Unix process whose return value has not been picked up by the parent process.

10.3 Connecting the Blocks

To create something interesting out of the blocks, we need to connect them. These connections are established by attaching an input of the second block to the output of the first block – the connections are always specified at inputs, never at outputs. So the input knows where its value came from, but the output does not know whether it is connected somewhere at all. The inputs also initiate transfers of values over the earlier established connections. That means the outputs are passive publishers only. By connecting the blocks a *cascade* is being built.

The cascade is a directed acyclic graph composed of blocks, where edges are connections between outputs and inputs of the blocks.

When a block is inserted into the cascade, its inputs are already entirely defined. It means that a connection to another block or a constant is assigned to each of its inputs. At this moment, connections are specified using block ID and output name. Later, when the block is being executed, actual block instances assigned to these block IDs are resolved, so the specified connections can be established and values transferred from outputs to inputs. Thanks to that, it does not matter in which order the blocks are inserted into the cascade, as long as there are all required blocks present before they need to be executed.

10.3.1 Evaluation

Evaluation of the cascade is a process, in which the blocks are executed in the correct order, and data from the outputs of executed blocks are passed to the inputs of the blocks waiting for execution.

By creating a connection between two blocks, a dependency (precedence constraint) is defined, and these dependencies define the partial order in which blocks need to be executed. For single-threaded evaluation, a simple depth-first-search algorithm with cycle detection can be used to calculate topological order compatible with a given partial order [115].

Since the DFS algorithm requires a starting point to be defined, selected blocks (typically output generators[2]) are enqueued to a *queue*, when inserted into a cascade. Then the DFS is executed for each block in the queue. If a block is not enqueued, it will not be executed, unless some other block is connected to its outputs. Such a lazy evaluation allows preparation of a set of often, but not always, used blocks and let them execute only when required. The evaluation of the cascade ends, when the execution of the last block finishes, and the queue is empty.

These features relieve the programmer from an explicit specification of execution order, which is required in traditional procedural languages.

10.3.2 Visualization

It is effortless to visualize connections between blocks automatically. Once cascade evaluation finishes, its content can be exported as a code for Graphviz [85], which will automatically arrange a given graph into a nice image, and this generated image can be displayed next to the results of the program with no effort. It is a very useful debugging tool.

Note that there is no need for a step-by-step tracing of cascade evaluation because the generated image represents the entire process, including errors and the presence of values on connections.

For example, Figure 10.3.1 shows a cascade used for editing an article on a simple web site. The block `load` loads an article, then passes it to the block `form`. The form is displayed to the user by the block `show_form`. Because the form has not been submitted yet, the block `update`,

[2]Output generator is a block which prepares data for a future HTTP response as a side-effect. The prepared data are passed to template engine when cascade evaluation is finished.

which will store changes in the article, is disabled (a grey arrow represents `false` or `null` value). This figure was rendered by Graphviz, and similar figures are automatically generated when creating web sites with a framework based on the cascade (see Section 10.6).

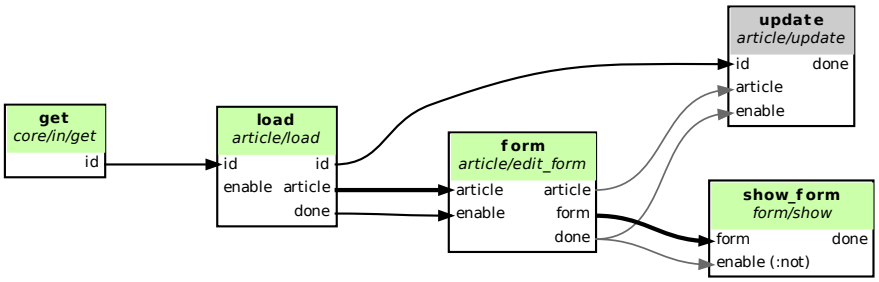


Figure 10.3.1: Cascade example

10.3.3 Basic Definitions

It is necessary to define a few basic concepts and a used notation before a behavior of cascade can be described in detail.

A block name in the text is written using a monospaced font, for example, `A`. The execution of block `A` starts with an event A (i.e., beginning of the execution) and ends with an event $\bar{A}$ (i.e., end of the execution).

During the event A, the `main` method of the block is called. Within the `main` method, the block reads its inputs, performs an operation on received data, and sets its outputs.

During the event $\bar{A}$, the cascade performs all requested output forwarding (see Section 10.4.4), and the block execution finishes. The block itself performs nothing at this point.

Because any execution of a block must begin before it ends, a trivial precedence constraint is required for each block:

$$A \prec \bar{A} \tag{10.3.1}$$

10.3.4 Automatic Parallelization

One of the first questions usually asked after a short look at the cascade is whether the blocks can be executed in parallel. A short answer is "yes, of course", and in this section, we try to explain how straightforward it is.

Let there are blocks `A`, `B`, and `C`, where `C` is connected to some outputs of blocks `A` and `B`, as displayed in Figure 10.3.2.

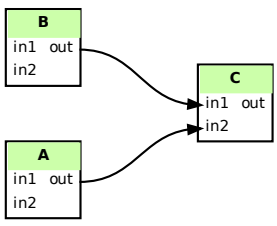


Figure 10.3.2: Automatic parallelization example

Because the begin of block execution precedes its end, trivial precedence constraints are defined:

$$A \prec \bar{A},\ B \prec \bar{B},\ C \prec \bar{C} \tag{10.3.2}$$

Due to the connections, execution of the blocks `A` and `B` must be finished before the execution of `C` begins:

$$\bar{A} \prec C,\ \bar{B} \prec C \tag{10.3.3}$$

When this cascade is evaluated in a single thread, blocks have to be executed in a topological order, which is compatible with partial order defined by precedence constraints (10.3.2) and (10.3.3). Assuming non-preemptive execution of the blocks, there are two compatible orders:

$$A \prec \bar{A} \prec B \prec \bar{B} \prec C \prec \bar{C} \tag{10.3.4}$$

$$B \prec \bar{B} \prec A \prec \bar{A} \prec C \prec \bar{C} \tag{10.3.5}$$

Both (10.3.4) and (10.3.5) will give exactly the same results because blocks `A` and `B` are completely independent. Therefore these two blocks can be executed in parallel with no trouble:

$$\left(\left(A \prec \bar{A}\right) \parallel \left(B \prec \bar{B}\right)\right) \prec C \prec \bar{C} \tag{10.3.6}$$

A naive implementation of parallel execution can be done by spawning a new thread for each block and using a simple locking mechanism to postpone the execution of blocks with unsolved dependencies. More efficient implementations may involve a thread pool and per-thread block queues for solving dependencies. Since cascade is being used to generate web pages, the block-level parallelization was not investigated any further because all web servers implement parallelization on a per-request basis.

10.4 Growing Cascade

So far, everything mentioned here is more or less in practical use by various tools, especially in data mining, image processing, and similar areas. What makes cascade unique is the ability to grow during the evaluation.

When a block is being executed, it can also insert new blocks into the cascade. Inputs of these blocks can be connected to the outputs of any other blocks (as long as circular dependencies are not created), and it can be enqueued to the queue for execution. The algorithm described in section 10.3 will handle these new blocks exactly the same way as previous blocks because it iterates over the queue, and the new blocks will be enqueued there before the enqueuing block is finished.

10.4.1 Namespaces

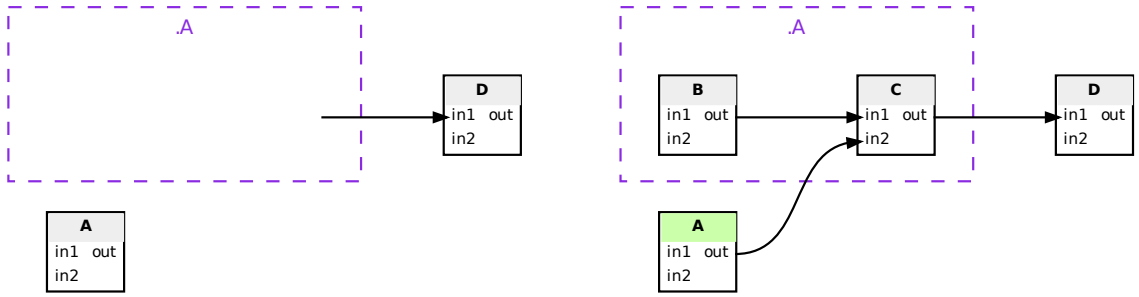
To avoid collisions between block IDs, each block inserts new blocks into its own namespace only. These namespaces are visualized using a dashed rectangle with an owner block ID under the top edge of the namespace rectangle.

In traditional languages like C, namespaces are used to manage visibility (scope) of local variables, where code located outside of a namespace cannot access a variable defined inside this namespace, but the inner code can reach global variables. However, the global variables can be hidden by local variables.

The same approach is used in the cascade with a small difference – it is possible to access the content of a namespace from outside *explicitly*, so connections across the namespaces can be made with no limitation.

Dot notation is used to identify a block in another namespace, similarly to the use of slashes in file systems. For example, we refer block `B` in the namespace of block `A` in the root namespace as `.A.B` (see Figure 10.4.1b). If there is no leading dot, the first block is searched in the namespace of the current block and all its parent namespaces up to the root. By specifying additional blocks, it is possible to enter namespaces of other and completely unrelated blocks.

Since the primary purpose of namespaces is to avoid collisions in IDs, there is no reason to deny connections from the blocks outside of the namespace (see block `D` in Figure 10.4.1b). This allows extending existing applications by attaching additional blocks without the need to change the application.



(a) Cascade *before* the execution of the block `A`

(b) Cascade *after* the execution of the block `A`

Figure 10.4.1: Growing cascade

10.4.2 Dependencies During the Growth

The secondary purpose of the namespaces is to help to handle dependencies on blocks that are not yet present in the cascade.

Take a look at Figure 10.4.1a. The block `D` is connected to a so-far nonexistent block `C` inside the namespace of the block `A`, and block `A` has not been executed yet. At this moment, the only known precedence constraint is $\bar{C} \prec D$, and because there is no connection between `A` and `D`, block `D` could be executed before `A`, but that would end up with a "block `C` not found" error.

Because the only block, which can insert blocks into the namespace of the block `A`, is the block `A`, additional precedence constraint can be introduced: Each block depends on its parent (creator). Therefore cascade in Figure 10.4.1a contains the following precedence constraints:

$$\bar{A} \prec C,\ \bar{C} \prec D \tag{10.4.1}$$

Thus, because block `C` is not present yet, the only compatible topological order is:

$$A \prec \bar{A} \prec D \prec \bar{D} \tag{10.4.2}$$

Block `A` inserts blocks `B` and `C` into the cascade during its execution and additional precedence constraints are created:

$$\bar{B} \prec C,\ \bar{A} \prec C,\ \bar{A} \prec B \tag{10.4.3}$$

Note that $\bar{A} \prec C$ is there for the second time, because of the connection between blocks `A` and `C`. The constraint $\bar{A} \prec B$ is already fulfilled because block `A` has been executed already.

Now, after block `A` is finished, the cascade contains new blocks and new precedence constraints, so a new topological order must be calculated before a next block is executed. The only topological order compatible with (10.4.1) and (10.4.3) is:

$$\underbrace{A \prec \bar{A}}_{\text{executed}} \prec \underbrace{B \prec \bar{B} \prec C \prec \bar{C} \prec D \prec \bar{D}}_{\text{queued}} \tag{10.4.4}$$

10.4.3 Safety of the Growth

It is not possible to break an already evaluated part of the cascade by adding new blocks. The reason is fairly simple – the new blocks are always appended after the evaluated blocks. That means the new precedence constraints are never in conflict with already existing constraints, and the new topological order is always compatible with the old one.

Each block is executed only after the execution of all blocks on which it depends. Furthermore, because connections are specified only when a block is inserted into the cascade, the already executed part of the cascade cannot be modified. It also means that all precedence constraints in the executed part of the cascade have been fulfilled.

When inserting a new block, there are two kinds of connections that can be created: a connection to an already executed block, and connection to enqueued or missing block. When connected to the already executed block, any new precedence constraint is already fulfilled. When connected to the enqueued or missing block, the order of these blocks can be easily arranged to match the new constraints – the situation is the same as before the execution of the first block.

For example, the only difference between topological orders (10.4.2) and (10.4.4) is in added blocks `B` and `C`, thus the relative order of all actions occurring in (10.4.2) is same as in (10.4.4).

10.4.4 Nesting of the Blocks and Output Forwarding

The basic idea of solving complex problems is to decompose them into simpler problems hierarchically. A function call and a return statement are the primary tools for this decomposition in λ-calculus and all derived languages (syntactical details are not important at this point). The function call is used to breakdown a problem and the return statement to collect partial results so that they can be put together to the final result.

Nevertheless, cascade evaluation is a one-way process. The entire concept of return value makes no sense here. Everything in the cascade goes forward and never looks back. Also, there is no stack in the cascade where return address could be stored, so even if something would want to return a result, it has no chance of knowing where to.

To allow a hierarchical problem decomposition in the cascade, we introduce a slightly different tool — *output forwarding*. When block solves some problem, it presents results on its outputs. When block delegates solving to some other blocks, the results are on their outputs. Then, the results must be transferred to the original outputs of the delegating block to achieve the same final state as before. This schema exhibits similar behavior as the return statement.

From another point of view, there is no need to transfer result values back, if all connections are redirected to the forwarded outputs. Both approaches are equivalent, and they both preserve all properties of the cascade mentioned earlier.

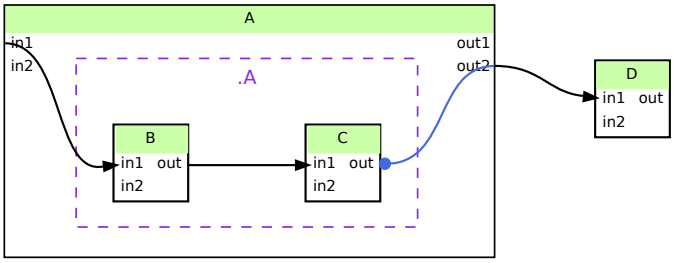


Figure 10.4.2: The idea of nested blocks

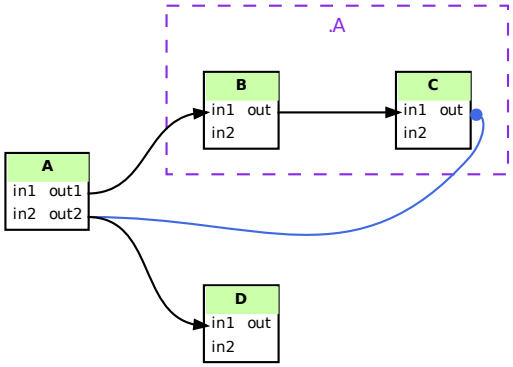


Figure 10.4.3: "Nested" blocks in a real cascade

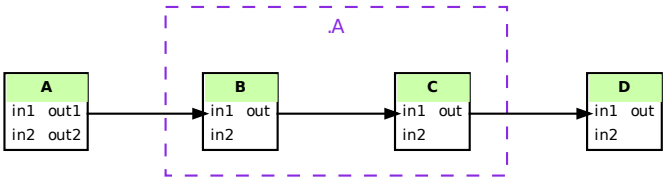


Figure 10.4.4: Equivalent cascade without output forwarding

For example, let block `A` perform a complex task and block `D` display result of this task – see Figure 10.4.2. Block `A` inserts blocks `B` and `C` into the cascade, and passes a parameter from its input to the input of the block `B` and then collects a result from block `C`, which received a partial result from the block `B`. The final result is then published on an output of the block `A`. Figure 10.4.2 presents this solution as it is usual in languages based on λ-calculus.

The cascade is a flat structure and does not allow nesting of the blocks. It emulates this hierarchy using namespaces, but all blocks are in the flat space. Therefore, blocks `B` and `C` are inserted next to the block `A`, as presented in Figure 10.4.3.

Note that output forwarding can be chained over multiple outputs. For example, some other blocks could request forwarding of the output of block `A` in Figure 10.4.3. In such cases, output forwarding is solved recursively with no trouble.

Output forwarding adds exactly the same precedence constraint like any other connection between the blocks. The situation here is almost the same as in Figure 10.4.1b. When the value copying approach is used, the output forwarding adds the following constraint:

$$\bar{C} \prec \bar{A} \tag{10.4.5}$$

This is because all connected blocks are tied to the $\bar{A}$ event, so it is easier to delay this event

until dependencies are solved. If the second (redirecting) approach is used, the situation would be exactly the same as presented in Figure 10.4.4, but an implementation of this transformation may be too complicated.

All constraints in the example cascade in Figure 10.4.3 are:

$$\bar{A} \prec B,\ \bar{A} \prec C,\ \bar{B} \prec C,\ \bar{A} \prec D,\ \bar{C} \prec \bar{A}$$

Thus, the compatible topological order is:

$$A \prec B \prec \bar{B} \prec C \prec \bar{C} \prec \bar{A} \prec D \prec \bar{D} \tag{10.4.6}$$

10.4.5 Semantics of the Output Forwarding

Using the output forwarding a block says to the cascade: "When I am done, set my output to a value which is on that output of that block." From a block's point of view, the output forwarding is exactly the same process as setting a value to an output. The only difference is in specifying what value will be set – output forwarding uses a reference instead of a value.

The namespaces and the output forwarding were both designed to allow a hierarchical decomposition of a problem, but they are completely independent tools in contrast to a function call and a return statement in languages based on λ-calculus. Therefore, because the output forwarding is not limited to any particular namespace, it enables very unusual use cases, where the "return values" can be picked up anywhere in the cascade.

10.5 Comparison with Other Approaches

Probably the most similar approach to the cascade is Function Block programming [9], which has a very long history in industrial automation to program PLCs[3]. The main difference from the cascade is that blocks and connections between them are static. The connections are left unchanged as long as the programmer does not upload a new version of software into PLC. The second difference is in data transmitted via inputs and outputs. In the cascade, once an output is set, it stays constant forever, but in Function Blocks, it is a stream of values.

Various data processing tools like Orange [116], Khoros Cantata [117], and Rapid Miner [118] adopted the function blocks approach with some modifications. Nonetheless, the fundamental difference still stands – all these tools use a static structure built from blocks by the programmer.

λ-calculus differs from the cascade in the semantics of a return statement and the heavy use of a stack. Because there is no stack in the cascade, the return statement is thrown away and replaced by the output forwarding mechanism, which has slightly different semantics, but it can be used to achieve the same goals.

Cascade exhibits some shared features with Hierarchical Task Networks (HTN) planning [119]. HTN planning provides a much more sophisticated approach, including backtracking and different constraint mechanisms, such as constraint refinement, constraint satisfaction, constraint propagation, and others. Cascade trades these advanced methods for execution speed by the elimination of decomposition alternatives.

The preference calculus based on properties of partial order relations forms a foundation of dependency trees heavily used in scheduling theory [120].

Dependency injection is a software design pattern used to ease dependencies in the code [121]. A basic idea is in the separation of tool users (code) from creators (factory) of the tool (object), so the creators can be easily replaced. In the cascade, this approach is very natural to use,

[3]PLC: Programmable Logic Controller

since it is easy to wrap the creator into a common block and let users connect to this block and retrieve the tool. Thanks to the embedded visualization of the cascade, it is very easy to track these dependencies in an application. However, the cascade does not aim to replace dependency injection containers because of a different approach to the configuration of such containers due to the different developer needs.

10.6 Real-world Application

10.6.1 Web Framework

The cascade serves as a core of a push-style web framework written in PHP, where the cascade takes the place of a controller (as in MVC pattern). Since the cascade does not have any own inputs and outputs, specialized blocks are used as connectors to receive input data from an HTTP request and to pass response data into a template engine (view). The processing of any HTTP request is split into two stages. During the first stage, the cascade is evaluated, and response data are prepared as a result. In the second stage, the template engine generates a final web page using the prepared data.

When a development mode is enabled, each web page contains automatically generated visualization of a cascade used to create that page (see Section 10.3.2), so the debugging tool is always at hand.

Also, a simple profiler is embedded in the framework. An average time needed to evaluate a cascade in real applications (ca. 20–40 blocks per page) is approximately 30 ms, and overhead of the cascade is less than 3 ms of that time, which is as good as widely used web frameworks.

10.6.2 Modular Applications

Extensible applications typically declare places, where they can be extended, which is usually done using hooks or virtual methods. The cascade uses a different approach. It offers the user a set of blocks, and it is up to him what he will build. Then, when there is some special need, which cannot be satisfied using available blocks, an additional set of blocks shall be created.

The cascade introduces a unified mechanism allowing various blocks from foreign sets to be connected into one structure, but it does not specify how this structure should look. It is a job for the framework built above the cascade to specify common fragments of these structures and define configuration style, so the solid base for application is created.

10.6.3 Rebuild rather than Modify

The cascade tries to make maximal use of block reusability. When a new part of an application is to be created, many of the required blocks are already available for use, so the new part of the application can be built in a little time. Thanks to this, a need for adjusting existing structures to suit current needs is significantly reduced, because they can be thrown away and easily replaced.

10.6.4 Generating the Application

It is possible to generate these structures algorithmically so that we can avoid repetition while composing the blocks. A block that inserts additional blocks into the cascade is not limited to how it should get a list of these blocks. It may be from a static configuration file or a database, but the block can interpret these data and use them as templates or as parameters of a predefined template.

The cascade is designed to support this behavior. The dynamic insertion of blocks into the cascade is its key feature. The way how the cascade is built (see Section 10.3) makes it very easy to attach new blocks to existing sources of data. Because the order of blocks is not significant while inserting them into the cascade, the generating blocks can be much simpler than traditional code generators. Also, it is easy to use multiple blocks to generate a single structure, since connections can be made across namespaces (see Section 10.4.1).

When this approach is combined with sufficient metadata about entities in the application, the results may be very interesting. At this point, we may notice similarities with generative programming (see Sec. 3.19) and how Smalldb state machines provide metadata (see Sec. 6.7).

10.7 Conclusion

In practice, the cascade made development more effective, because it supports better code reusability while creating a new application, and it helps the developer to analyze an old code when modifying or extending an existing application. The graphical representation of the code is beneficial when tracking from where broken data arrived and what happened to them on their way. It significantly reduces the time required to locate a source of problems.

The dynamic nature of the cascade allows the programmer to cover a large number of options while using a fairly simple structure to describe them. Moreover, because the cascade drives the execution order, the programmer does not have to care about it. That means less code and less space for errors.

A main contribution of the cascade is extending the time-proven function blocks approach with new features making it suitable for many new use cases and preparing the base ground for further research.

Chapter 11

Web Page Composition

When a web application produces results requested by the user, it typically presents the results as a web page. To do so, the application needs a mechanism to convert raw data structures, for example, produced by the cascade from the previous chapter, into a human-friendly document.

In the MVC architecture, a controller generates output data, and a view renders them for the user (as explained in Chapter 5). This arrangement introduces an implicit, somewhat hidden dependency between the view and the controller, as the view must know what kind of data the controller provides to render them correctly. A traditional approach to overcome this issue is to assign each controller its template and let programmers maintain the consistency.

The compositional approach to software construction, either manual or automated, means that individual controllers are composed of multiple components, and each component may require a specific template fragment to render its output. Therefore, the compositional approach in the controller requires a compositional approach in the view as well.

One of the widely-used template engines designed for HTML 5 [122] is Twig [123]. In this chapter, we will use Twig syntax for template examples. Twig uses two language constructs. First, `{{expression}}` outputs the value of the expression, which can be a variable name. The second is a tag enclosed in `{% ... %}`. They can be unpaired, for example, `{% include filename %}`, or they can come in pairs, where the closing tag is prefixed with "end", for example, `{% block foo %}...{% endblock %}`.

11.1 Basic Templates

A template is a mapping from a data object to its visual representation. Input is a data structure or an object; output is typically an HTML code, but templates may produce PDF documents or images too.

When composing templates, we usually distinguish two kinds of templates. The first kind is dedicated to views of individual entities. The other kind represents the surrounding HTML page with placeholders for actual content — these are typically referred to as "layouts".

Probably the oldest approach to template implementation is using include statements to separate repeating fragments of templates to shared files. Such a simple template to display a piece of text with a title would look like this (using Twig syntax):

```
{% include "header.html.twig" %}
<article>
  <h1>{{title}}</h1>
  <div>{{textBody}}</div>
```

```
</article>
{% include "footer.html.twig" %}
```

The files `header.html.twig` and `footer.html.twig` contain opening and closing tags of the HTML document as well as some shared navigation elements.

Unfortunately, this approach does not scale well, and maintenance of such templates quickly becomes too difficult. Moreover, passing parameters to the included files usually lacks elegance.

11.2 Inheritance-based Templates

Inspired by object-oriented programming, template engines adopted inheritance mechanisms to provide extensible templates. Instead of including the shared parts of HTML pages, individual templates use a "layout template", which defines "blocks", and then populates these blocks with data. The inheritance is currently the mainstream approach to template engines for web applications.

The example from the previous section would use the following layout template, stored in `layout.html.twig` file:

```
<html>
  <head>
    <title>{% block title %}Untitled{% endblock %}</title>
  </head>
  <body>
    {% block content %}{% endblock %}
  </body>
</html>
```

Then we modify the template to render the piece of text with a title. Instead of the "includes", the template uses the following "extends" statement and two block overrides:

```
{% extends "layout.html.twig" %}
{% block title %}{{title}}{% endblock %}
{% block content %}
  <h1>{{title}}</h1>
  <div>{{textBody}}</div>
{% endblock %}
```

Note that in this case, the template overrides two blocks, `"title"` configures the title of the HTML document and has a default value; the `"content"` is a placeholder for the, well, content. Both blocks can be placed at arbitrary locations in the layout without changing anything in the templates that use the layout — this was not always possible with the includes. The blocks provide a mechanism that inverts the includes so that the templates are compact and properly structured. It also allows us to build a hierarchy of progressively specialized layouts.

This inheritance-based approach is quite popular in modern frameworks, for example, Twig (used by Symfony), Smarty. It provides sufficient flexibility, and the maintenance is relatively easy. However, when we begin to compose web pages, we find out an intriguing property — it is the template that defines the content of the web page, not the controller, which only selects the template. Could we return the control over the web page content to the controller?

11.3 Slots and Fragments

For the needs of software synthesis and compositional approach, we introduce an approach inspired by the Drupal content management system. It utilizes slots and fragments to compose a web page in an as flexible way as possible.

The idea of slots and fragments is very simple: A layout defines slots, and the controller populates them with fragments. The fragments are triplets of a template name, a set of template parameters (data to render), and a weight to determine the order of fragments within a slot.

The core concept of this approach is that the controller defines what the view should render. The earlier described approaches left this knowledge to the view, and the controller only provided the required data. It is important to preserve the separation of concerns between the controller and the view. The controller should specify what to render, but it must avoid specifying how to render — that is for the view to handle.

If we adjust the example from the previous sections, the layout with slots will be as follows:

```
<html>
  ...
  <body>
    {% slot content %}
  </body>
</html>
```

The fragment template for the text with a title is mostly the same:

```
<h1>{{title}}</h1>
<div>{{textBody}}</div>
```

The difference is that neither the fragment or the layout do not refer to each other in any way. Therefore, the controller must compose the page accordingly (using PHP):

```
$template->setLayout("layout.twig.html");
$template->slot("content")->add(50, "text.twig.html", $textData);
```

In this example, the controller configures the template engine to use the given layout template, then it gets the "content" slot and adds the text fragment of weight 50, the given template, and data. As we can see, the controller does not bother with rendering details; it only tells the template engine to put text in that location.

There are a few details we did not solve, however. The knowledge of the slots defined in the layout leaks into the controller and the weights of the fragments are arbitrary numbers instead of properly defined relations between the fragments. Also, the use of file names to refer to the templates may be suboptimal.

The issue of template file names and slot names could be resolved by using native classes to represent the templates, similarly to React components, or we could generate such classes from the template files. Then the template classes could be injected using dependency injection or service locators to decouple them from the controller. Once we would have the template classes in place, they could declare slots, and the compiler would check the consistency for us. Moreover, the controller could query the template for available slots. Indeed, there is some space for improvements; however, all of these unsolved issues are a matter of engineering touch and do not present any fundamental problems.

11.4 Fragment Placement and Ordering

We can view each layout template as a two-dimensional space. The slots form the first dimension, and weights of the fragments the second dimension.

The slots are named regions in the web page defined by the layout. To place a fragment correctly, the controller needs to identify the appropriate slot. The usual approach is that a website designer, who creates the layout templates, defines a naming convention shared across the application. The slot names are usually passed to individual components as parameters, preferably with a sensible default value.

Weights of the fragments drive the ordering of the fragments within a slot, where the weight is a dimensionless number with defined upper and lower bounds to denote the begin and end of a slot. For convenience, we define a trivial rule saying to put the light fragments on top.

The sorting algorithm used for arranging the fragments must be stable, meaning that fragments added with the same weight stay in the order they have been added to the slot. This property is essential when a controller adds several fragments with the same weight into a single slot. Changing the order of the fragments in such cases might be very confusing to the users.

In terms of semantic information, the use of the weights may seem somewhat inappropriate; however, practical implementations showed us that defining an approximate position within the page is sufficient to satisfy users' expectations. A more semantic way of defining the order of the fragments, for example, using relations between the fragments, might seem to be useful, but such a mechanism would introduce significant complexity that does not seem to be necessary.

11.5 Conclusion

Templates are an important part of every web application, and current state-of-the-art approaches do not expect use cases with generated content. When we synthesize our applications, we need to deal with this issue, and the concept of slots and fragments provides us with a simple and practical mechanism. The real-world implementation of this mechanism fits within a few hundreds of lines and can be implemented as an extension of an existing template engine, e.g., Twig. Therefore, we gain capabilities required for software synthesis without losing the power of well-established tools.

Chapter 12

Software Synthesis as a Planning Problem

The previous chapters provided us with various tools and concepts useful for software synthesis, and thus we should discuss potential approaches to their use and what possibilities they open. This chapter is meant as a collection of ideas suitable for further research. Some of the ideas were present in the very beginning and influenced the design of the Smalldb framework or the Cascade; others emerged while searching for the STS algorithm and experimenting with the Cascade.

As we are leaving the models and specifications towards the software synthesis, we are leaving the scope of this thesis, and thus we will only scrape the surface of the possibilities. The in-depth analysis and unraveling of these ideas would easily take up a few more dissertations.

12.1 Planning vs. Software Synthesis

When we look at software composition in a broader context, we can identify three related tasks of increasing difficulty:

- Scheduling: Tasks are given. The goal is to find an optimal order of the tasks.
- Planning: A palette of operators is given. The goal is to find an optimal order of the operator instances; some operators may be used multiple times, some not at all.
- Software synthesis: A palette of operators is given but potentially incomplete. The goal is to find an optimal order of the operator instances and specify missing operators needed for a complete solution.

Both planning and software synthesis require operators. The operators are atomic parametrized actions that modify the state of the application. In some planning techniques, e.g., HTN [119], tasks may be hierarchically composed of subtasks and operators. By combining the operators, we build a path from the initial state to the desired state.

Planning applications are relatively straightforward tasks where we know what we require to achieve each goal, but the goals are variable, or the environment is changing, and thus we need to adjust the plan for each particular goal. The typical feature of the planning is a certain repetitiveness that provides us with a stable class of problems. Therefore, if the available set of operators was sufficient to achieve the previous goals, it is likely it will be sufficient for future goals as well.

Software synthesis is, in many ways, similar to the planning, but it is not repetitive. The goals of software synthesis are typically unique and often require a few new operators. For example, if we have a page in a web application to edit properties of an entity and we want to implement a similar page for another entity, we need a new HTML form to handle the different entities. Human programmers usually generalize the previous implementation; they identify the overall structure of such page and then concretize it for the new scenario, in this case, for the other entity.

Unfortunately, computers are not very good at generalizations, and since the planning problems alone are almost always NP-complete, the searching of the missing operators adds even additional complexity.

12.2 Operator Factory

Both the planning and software synthesis require operators as their building blocks. The question is, where to get these operators. In terms of software synthesis, the operators are functions or methods. They take parameters to produce a return value and potentially some side effects. If we use the cascade presented in Chapter 10, the cascade blocks are our operators. The visual nature of the cascade is quite helpful in grasping the idea.

Aside from some data manipulation primitives, we need operators that can access and modify data, i.e., to interact with the model layer of the application. Since we use Smalldb state machines, we can easily generate such operators from the state machine definitions — one operator to load the state machine state, one operator to search the state machine space (see Section 6.5), and one operator for each transition of the state machine. For the cascade, this means to generate a set of blocks for each defined state machine.

Creating such operators (and blocks) is quite a simple task; it only requires us to set up a proper infrastructure. The generative approach presented in Section 3.19 or even a simple factory [18] will likely do well.

12.3 Sunshines

When looking at a state diagram of an entity from the perspective of the current state and immediate action, only current state and available transitions are visible, leaving the rest of the diagram too far to be seen. On paper, the cutout of the diagram looks like little sunshine (with some imagination), see Figure 12.3.1.

When designing a planning algorithm, we can utilize another feature provided by the state machines. It is likely we will know in which state a currently considered state machine is at the given time. Therefore, we can reduce the number of considered operators to the rays of a sunshine.

The sunshine can tell us which actions are relevant while planning the next steps. This reduces the number of possibilities to consider, sometimes to a single possible action, and thanks to the number of inappropriate actions eliminated, the synthesis should produce meaningful results faster.

In terms of planning algorithms, we can consider the sunshine as a precondition of the operators derived from the state machine definition.

In terms of software synthesis, the sunshine tells us what actions the user can perform at the moment. This corresponds with the buttons and links the user should see. Therefore, we can use the sunshines to generate, for example, navigation widgets and menus, and also validate that no available action is not inaccessible.

The sunshines are a simple concept, but they represent the hypertext nature of web applications and provide us with basics to turn a group of components and HTML forms into a coherent application.

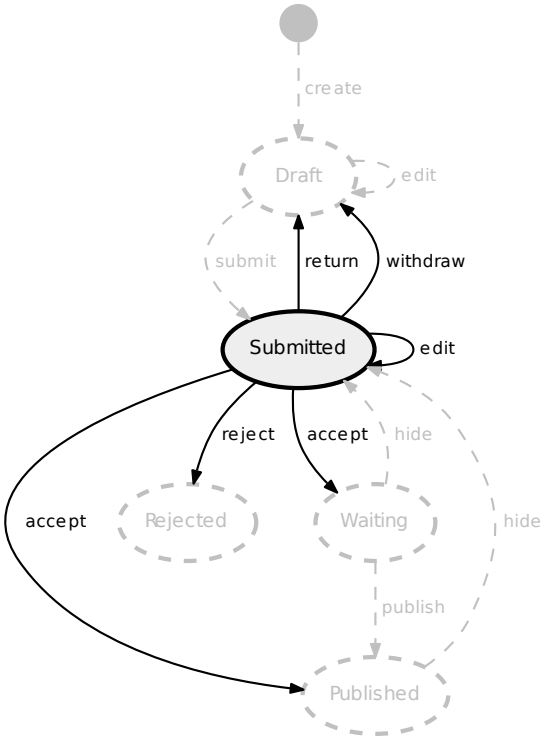


Figure 12.3.1: Sunshine example — an article in a content management system

12.4 Storylines

An attempt to generalize programs implemented in the cascade provided us with an idea of storylines. The core concept is that users intuitively understand each operation in an information system as a sequence of elementary steps, which, in our case, are represented by blocks in the cascade. For example, an edit operation is implemented as a block sequence “load” — “form” — “update”. The storylines are not strictly linear; instead, they form directed acyclic graphs. For example, the edit operation requires an ID of the edited entity to be passed directly from the “load” block to the “update” block, bypassing the “form” block so that the user cannot edit a different entity.

The storylines do not include unnecessary details, like how individual blocks are connected unless required by the meaning of the storyline. They are meant as a guideline for a planning algorithm to understand what a user wants and expects. Therefore, if the user says: “I want to edit this entity.”, then the planning algorithm knows it needs the three blocks specified by the storyline and connect them somehow into a working application.

The simplicity and lack of detail allow us to compose the storylines together. For example, the “form” block contains the logic of the form, how to interpret an HTTP request, and how to validate the data, but it does not render the form. Therefore, we need another block called “form_view”. The separation of the logic and rendering of the form allows us to control the visibility and location of the form on the web page and, for example, display errors when the update block fails. To utilize the “form_view” block automatically, we may define a storyline

"form" — "form_view". Then, we may combine these two storylines to let the user edit the entity and see the form.

The mechanism of storyline composition may also provide us with a tool to extend an existing application. We may add new functionality by defining a new storyline that will attach to existing blocks and provide additional features.

As a side effect, this approach enables the planning algorithm to detect a missing block relatively easily. When a storyline defines the use of a block, but there is no such block available, it can ask programmers to implement the missing block.

From the other end, if we have multiple implementations of the same operation but for different entities, we may be able to generalize the program by shared storylines, i.e., by detecting recurring sequences of similar blocks. Then we may use the extracted storyline to implement the same operation for a new entity automatically and maybe even refactor the original implementations by reconstructing them from the new storyline.

The storylines are a concept that influenced the design of the cascade, and they could inspire further research in this area. They are expected to introduce a relation between user's words and particular plans provided by planning algorithms. An intriguing question is whether the storylines could represent a programmer's experience in a way that planning algorithms could use.

Part IV

Endgame

Chapter 13

Experiments, Results, Applications

The previous chapters presented a number of novel concepts, and before we conclude this thesis, it is time to discuss and summarize their practical applications and contributions. Some of them were already presented in the form of examples and use cases spread across this thesis, so let us put them into a broader context.

13.1 Smalldb Framework

Smalldb framework, built around Smalldb state machines, see Chapter 6, was successfully used and deployed in real-world applications, including few commercially successful projects (that we can disclose in limited detail only due to business reasons). The Smalldb framework itself is an open-source project published under the Apache 2 license and available at https://smalldb.org.

13.1.1 Application of the Formal Model

The first notable project was a real-time auction application where users were bidding both in the presence and remotely over the web application while watching a live video stream. Smalldb state machines were used to control the auction and life cycles of individual auctioned items.

While the application was not particularly extensive, it required absolute reliability and correctness. The formal framework provided by Smalldb proved itself invaluable. The application was designed around the state machines. The notifications about state changes were broadcasted to all users, and the users were sending requests for transition invocations to bid on the items. As a consequence, the implementation of the user interface was quite simple — the user interface only had to collect the updates of the global state and render them to the user and let the user send some requests back. The server side was not much more complicated. The received requests were passed to the Smalldb state machine, and each transition also emitted an event to notify the clients. The rest was encapsulated in the state machine definitions and transitions implementation.

Thanks to the elegant use of the formal model, the application logic was implemented without a single error and worked perfectly from the very beginning, including the testing. The only changes were some minor user interface adjustments and one bug in the network communication, unrelated to the state machines.

13.1.2 Designing the Business Processes

The second notable project was an education support system with a rather complicated way to order courses, lectures, and exams, as described in the case study in Section 7.9. This project

showed that even though we knew how to model processes with interacting people, we struggled to capture nontrivial interactions with the information system. At the moment, we gave up on the business processes and used Smalldb state machines instead.

The discussions over the state machines and user interface prototypes converged to an acceptable solution and understanding on both sides in about the first four short iterations. Then the state chart sheet (Fig. 7.9.1) guided the implementation progress and served as a product specification and also its documentation. The fact that the customer was able to locate flaws in the state diagrams confirmed the qualities of this approach.

13.2 STS Algorithm: From BPMN to Smalldb

The struggle with capturing the interactions between people and information system was the motivation for a research in this area (see Sec. 13.1.2) and it concluded with introduction of the STS Algorithm.

A working implementation of the STS algorithm is included in the Smalldb framework. To use the STS algorithm, a programmer just needs to reference a BPMN file from the state machine definition and specify which participant represents the state machine.

The functionality of the STS algorithm was demonstrated on examples — see sections 8.10 to 8.15. The examples from Chapter 8 and a few others are included in the tests suite of the STS algorithm. The Pizza Delivery example (Section 8.15) is designed to reflect the features of the practical application that motivated the creation of the algorithm.

13.3 Business Process Simulation

The understandability and accessibility of formal modeling and verification of business processes were tested on our students in the form of semestral assignments, as presented in Chapter 9. The experiment concluded that the concept of using formal tools is comprehensible even without a broad background in formal methods, but the methods stand and fall on the quality and availability of the required tools.

13.4 Cascade

Initially, the cascade (see Chapter 10) was designed as the heart of a standalone application framework. As such, it was used in a few experimental applications. The framework included a visual editor to interactively compose the cascade, run-time visualization of the executed cascade on each page, and Smalldb integration.

The Smalldb integration with the cascade utilized metaprogramming techniques to generate blocks from Smalldb metadata. Since all the blocks were created using a single (extensible) block factory, it was quite straightforward to inspect Smalldb definitions and provide a generic set of blocks for each defined Smalldb state machine. The cascade visual editor then offered the generated blocks to programmers and let them place the new functionality anywhere on the website. The programmers then extracted repeating patterns of block connections into cascade templates (using simple placeholders in the configuration files) and let them generate an administration interface automatically. From this perspective, the cascade seems to be useful, and the template experiment showed that the cascade is indeed machine-friendly.

The overall performance of the cascade is comparable to other frameworks; the cascade evaluation takes about 3 % of the application CPU time. A web application of 35 SQL tables had 119 blocks implemented in PHP and 80 composite blocks created using the visual editor (these reflect the number of pages/screens in the application). Another application of 16 SQL tables had 32 blocks implemented in PHP, and 49 composite blocks — in this case, the number of composite blocks was higher due to various reports and views on relatively a few entities. The application that utilized integration with Smalldb was of 31 SQL tables, and thanks to the Smalldb integration, it had only 13 blocks implemented in PHP, but it consisted of 128 composite blocks, and another 137 blocks were generated automatically from the state machine definitions as a model API. These counts do not include the framework library of 61 blocks implemented in PHP.

In the matters of the developer experience, there were certainly some rough edges due to the experimental nature of the whole framework. For example, when adding a simple functionality, the cascade requires the programmer to create a block, implement the functionality there, and then use the block somewhere in the application — this introduces some overhead that does not always pay off, but it should be possible to mitigate it with some framework improvements.

The practical usability of the cascade stands and falls with the following three things:

- Block factories need to automatically provide blocks to interface with other components like model layer, forms, templates (view), or to access input data. Creating manually such a boilerplate is not acceptable in terms of productivity.

- The visual block editor is crucial for the use of cascade. Configuring the connections in any text format is way too tedious.

- Development overhead to implement individual blocks must be as low as possible, comparable to the creation of a regular class.

However, probably the most important "detail" is in the integration with the existing frameworks and tools. The initial approach to build an application framework around the cascade proved rather suboptimal because such a framework needs to address many features not relevant to the cascade. The problem is not technical — the cascade is versatile enough to incorporate these features and cover entire application architecture. In fact, it is quite refreshing to see the framework internals next to the application logic with all the connections in a single generated diagram. The issue is a matter of resources and time to develop such a framework from scratch because the cascade is too different to benefit from existing frameworks.

Therefore, it was decided to scale down the scope of the cascade and use it to implement only the controller component. Then, the cascade becomes a component that can coexist with other frameworks and legacy implementations. This experience significantly influenced the design of the Smalldb framework. The focus at the model layer had proved itself and allowed straightforward integration with 3rd-party frameworks.

13.5 Web Page Composition — Slots & Fragments

The concept of slots and fragments for web page composition (see Chapter 11) is quite simple and already proven and widely used by Drupal CMS. Our contribution is in its identification as a generic technique useful with software synthesis.

The mechanism of slots & fragments was included in the framework built around the cascade, and later, it was separated into a standalone library and a Twig extension so that we can use other features of this well-established template engine in combination with the slots.

The slots & fragments, both as part of the cascade framework and a standalone library, was successfully deployed in a few practical applications (some of them were mentioned in Sec. 13.4). According to our experience and reactions of colleague programmers, this approach seems to be more elegant than traditional approaches even if the structure of templates is static and known in advance. However, measuring the impact on programmer productivity in this aspect is somewhat tricky.

Chapter 14

Conclusions

To answer the question of how to tell what we want, we started by identifying what inputs programmers have available (Chapter 2). We found that a specification that is human-friendly can also be machine-friendly and that there are specifications useful for automated processing, for example, user interface designs, data structures, and business processes. However, none of these specifications provide a complete specification of the software product (which is a good thing), and thus, we change our perspective so that we see the software synthesis as an assistive tool that spares programmers of tedious and repetitive tasks (Sec. 1.3).

The assistive approach and the use of partial specifications imply yet another concept we need to grasp — the specifications and synthesis need to become part of the development process (Sec. 7.9), it is no longer just a tool that programmers use. We use the specifications to discuss requirements and expectations with the customer, and then the programmers use the very same specification as the input for the software synthesis.

To synthesize, we need to reason. To reason, we need a reasonable model. Turing-complete languages are expressive, but it is too difficult to reason about them. Instead, we propose to use simple formal models, like Smalldb state machines (Chapter 6). Because such a simple model cannot describe all the details, we leave well-defined gaps in the model, to keep the model simple, and then we fill the gaps using traditional programming languages. As long as we preserve the defined properties of the gaps, we do not break our simple model, and we can reason about the application.

In order that we can effectively build an application around a formal model, Chapter 7 presented an architectural pattern that integrates and utilizes such a model. The architecture combines the formal model with well-established approaches — the layered architecture and command–query separation.

Once we got the formal models integrated into the implementation, we can look for a relationship between the input specifications and the model in the implementation. Chapter 8 presented such a relationship by introducing the STS algorithm that analyzes a BPMN diagram and implements a given participant using a Smalldb state machine. The STS algorithm can also detect some semantic errors in the BPMN diagrams, which was a somewhat unexpected benefit. Most importantly, the development of the STS algorithm showed us how to draw BPMN diagrams where users interact with a web application, not only with other users.

As an alternative approach, Chapter 9 briefly presented the use of Uppaal to model and verify a business process as a network of interacting automata, where some automata represent an application and other its users. The understandability of the approach was successfully tested on students, but we realized the importance of the proper tools for practical applications.

Once we have models and specifications in place, we tackled some of the possibilities of

software synthesis in Part III. The cascade presented in Chapter 10 provides a machine-friendly way of composing the application logic of web applications. The concept of slots & fragments complemented the cascade with a compositional way of generating web pages, the views of the application, in Chapter 11. The cascade is an example of a tool that can utilize metadata provided by the formal models like Smalldb state machine. It shows one of the possibilities where software synthesis can take place.

In Chapter 12, we discussed the relation between planning and software synthesis and identified the fundamental difference — the problem of potentially missing operators. This finding confirms the viability of the assistive approach to software synthesis.

Most of the concepts presented in this thesis are supported by practical applications, or inspired by the real-world problems we encountered. Smalldb framework and its architecture taught us how to integrate formal models with the software development process, not only with the software itself, and the STS algorithm presented a formal tool to utilize and benefit from such formal models. Now, we have a practical framework, where formal models have their place, and we can continue with more advanced techniques and methods.

Appendix A

Author's Publications

A.1 Publications Related to the Thesis

A.1.1 Impacted Journal Articles

- J. Kufner and R. Mařík. *Restful State Machines and SQL Database.*
  In: IEEE Access, vol. 7, pp. 144603-144617, 2019.
  doi:10.1109/ACCESS.2019.2944807
  Journal impact factor: 4.098 (2018).

- J. Kufner and R. Mařík. *From a BPMN Black Box to a Smalldb State Machine.*
  In: IEEE Access, vol. 7, pp. 56276-56296, 2019.
  doi:10.1109/ACCESS.2019.2912567
  Journal impact factor: 4.098 (2018).

A.1.2 Other Excerpted Publications

- J. Kufner, R. Mařík. *Self-generating Programs – Cascade of the Blocks.*
  In: Information and Communication Technology. ICT-EurAsia 2014.
  Lecture Notes in Computer Science, vol 8407. Springer, Berlin, Heidelberg.
  doi:10.1007/978-3-642-55032-4_20

- J. Kufner, R. Mařík. *State Machine Abstraction Layer.*
  In: Information and Communication Technology. ICT-EurAsia 2014.
  Lecture Notes in Computer Science, vol 8407. Springer, Berlin, Heidelberg.
  doi:10.1007/978-3-642-55032-4_21

Bibliography

[1] Josef Kufner and Radek Mařík. State Machine Abstraction Layer. In *Information and Communication Technology*, volume 8407 of *Lecture Notes in Computer Science*, pages 213–227. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-55032-4_21. 1.5, 7.1, 8.1, 8.2.3, 8.3, 6, 9, 8.10

[2] Josef Kufner and Radek Mařík. Restful State Machines and SQL Database. *IEEE Access*, 7:144603–144617, 2019. doi:10.1109/ACCESS.2019.2944807. 1.5

[3] Josef Kufner and Radek Mařík. From a BPMN Black Box to a Smalldb State Machine. *IEEE Access*, 7:56276–56296, 2019. doi:10.1109/ACCESS.2019.2912567. 1.5, 2.4, 7.6.2, 7.10.3

[4] Josef Kufner and Radek Mařík. Self-generating Programs – Cascade of the Blocks. In *Information and Communication Technology*, volume 8407 of *Lecture Notes in Computer Science*, pages 199–212. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-55032-4_20. 1.5

[5] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887. 1.6, 6.2, 7.1, 7.2, 7.3

[6] CMMI Product Team. Capability maturity model® integration (CMMI SM), version 1.1. *CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1. 1)*, 2002. 2.1

[7] UML, OMG. Superstructure specification. Technical report, 2011. URL: http://www.omg.org/spec/UML/. 2.1, 3.2, 3.11, 8.16.1

[8] OMG. BPMN 2.0. *BPMN/2.0*. URL: http://www.omg.org/spec. 2.1, 3.2, 6.9.2, 8.1, 8.2.1, 8.16.2

[9] K.H. John and M. Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making AIDS*. Springer-Verlag, 2001. URL: http://books.google.cz/books?id=XzlYGLulBdIC. 2.1, 3.2, 3.9, 10.1, 10.5

[10] yEd Graph Editor. URL: https://www.yworks.com/products/yed. 2.1

[11] Camunda Modeller. URL: https://camunda.com/products/modeler/. 2.1, 5.3, 8.16.2

[12] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works*, 122:14, 2006. URL: http://www.thoughtworks.com/ContinuousIntegration.pdf. 2.1

[13] E Michael Maximilien and Laurie Williams. Assessing test-driven development at IBM. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 564–569. IEEE, 2003. 2.1, 3.14

[14] Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2017. 2.1, 3.14

[15] Oscar Sánchez Ramón, Jesús Garcıa Molina, Jesús Sánchez Cuadrado, Jean Vanderdonckt, et al. GUI Generation from Wireframes. In *14th Int. Conference on Human-Computer Interaction Interaccion*, 2013. 2.1

[16] EasyAdminBundle (a Symfony component). URL: `https://symfony.com/doc/master/bundles/EasyAdminBundle/`. 2.2

[17] Javier Eguiluz. Introducing the Symfony Maker Bundle, 2017. URL: `https://symfony.com/blog/introducing-the-symfony-maker-bundle`. 2.2

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 2.6, 3.3, 12.2

[19] Krzysztof Czarnecki. Overview of Generative Software Development. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 326–341. Springer Berlin Heidelberg, 2005. URL: `http://dx.doi.org/10.1007/11527800_25`, `doi:10.1007/11527800_25`. 3.2

[20] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002. URL: `http://doi.acm.org/10.1145/505145.505149`, `doi:10.1145/505145.505149`. 3.2, 3.11, 4.5

[21] George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. 3.2, 3.11, 6.4.3, 7.8

[22] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2019. URL: `https://doc.rust-lang.org/book/`. 3.3

[23] Andrei Chiş. Towards object-aware development tools. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 65–66. ACM, 2016. 3.4

[24] Krzysztof Czarnecki and UlrichW. Eisenecker. Components and Generative Programming. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 2–19. Springer Berlin Heidelberg, 1999. URL: `http://dx.doi.org/10.1007/3-540-48166-4_2`, `doi:10.1007/3-540-48166-4_2`. 3.5

[25] Onur Aydın, Nihan Kesim Cicekli, and Ilyas Cicekli. Automated web services composition with the event calculus. In *International Workshop on Engineering Societies in the Agents World*, pages 142–157. Springer, 2007. 3.5

[26] Quan Z. Sheng, Xiaoqiang Qiao, Athanasios V. Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decade's overview. *Information Sciences*, 280:218–238, 2014. URL: `http://www.sciencedirect.com/science/article/pii/S0020025514005428`, `doi:10.1016/j.ins.2014.04.054`. 3.5

[27] Biplav Srivastava and Jana Koehler. Web service composition-current solutions and open problems. In *ICAPS 2003 workshop on Planning for Web Services*, volume 35, pages 28–35, 2003. 3.5

[28] Aurora Ramírez, José Antonio Parejo, José Raúl Romero, Sergio Segura, and Antonio Ruiz-Cortés. Evolutionary composition of QoS-aware web services: A many-objective perspective. *Expert Systems with Applications*, 72:357–370, 2017. URL: `http://www.sciencedirect.com/science/article/pii/S0957417416305887`, `doi:10.1016/j.eswa.2016.10.047`. 3.5

[29] Guodong Fan, Ming Zhu, and Xiaoliu Cui. Optimizing Web Service Composition with Graphplan and Fuzzy Control. *Journal of Ubiquitous Systems & Pervasive Networks*, 11(2):15–21, 2019. 3.5, 3.18

[30] Hong Yul Yang, E. Tempero, and H. Melton. An Empirical Study into Use of Dependency Injection in Java. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pages 239–247, March 2008. `doi:10.1109/ASWEC.2008.4483212`. 3.6

[31] K. Jezek, L. Holy, and P. Brada. Dependency injection refined by extra-functional properties. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pages 255–256, Sept 2012. `doi:10.1109/VLHCC.2012.6344541`. 3.6

[32] Douglas R Smith. Constructing specification morphisms. *Journal of Symbolic Computation*, 15(5):571–606, 1993. 3.7

[33] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Issues in the practical use of graph rewriting. In *Graph Grammars and Their Application to Computer Science*, pages 38–55. Springer, 1996. 3.8

[34] Detlef Plump. *Term graph rewriting*. Computing Science Institute Nijmegen, Faculty of Mathematics and Informatics, Catholic University of Nijmegen, 1998. 3.8

[35] Barbara König. Analysis and Verification of Systems with Dynamically Evolving Structure. *ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/HABIL-2004-01/HABIL-2004-01.pdf*, 01 2005. 3.8

[36] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016. 3.10

[37] Edward F. Moore. Gedanken Experiments on Sequential Machines. In *Automata Studies*, pages 129–153. Princeton U., 1956. 3.11, 6.4.3

[38] *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. W3C, 2015. URL: `https://www.w3.org/TR/scxml/`. 3.11

[39] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. URL: `http://www.sciencedirect.com/science/article/pii/0167642387900359`, `doi:10.1016/0167-6423(87)90035-9`. 3.11, 3.13

[40] Gerd Behrmann and Alexandre David and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editor, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer,*

Communication, and Software Systems, SFM-RT 2004, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004. 3.11, 6.8, 7.8, 7.10.3, 9.1

[41] Hana Kubátová, Karel Richta, and Tomáš Richta. Petri Nets versus UML State Machines. *SDOT 2013*. 3.12, 8.2.4

[42] Éric Badouel, Benoît Caillaud, and P. Darondeau. Distributing Finite Automata Through Petri Net Synthesis. *Formal Aspects of Computing*, 13(6):447–470, 2002. URL: `http://dx.doi.org/10.1007/s001650200022`, `doi:10.1007/s001650200022`. 3.12

[43] Simona Bernardi, Susanna Donatelli, and José Merseguer. From UML sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the 3rd international workshop on Software and performance*, pages 35–45. ACM, 2002. 3.12, 8.2.4

[44] Michele Colledanchise and Petter Ögren. Behavior Trees in Robotics and AI: An Introduction, 2017. `arXiv:1709.00084`. 3.13

[45] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 01 1984. `arXiv:http://oup.prod.sis.lan/comjnl/article-pdf/27/2/97/981657/270097.pdf`, `doi:10.1093/comjnl/27.2.97`. 3.15

[46] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. 3.15

[47] Brian Hayes. Thoughts on Mathematica, 1990. 3.15

[48] Charles Simonyi. Intentional programming: Innovation in the legacy age. In *IFIP Working group*, volume 2, pages 1024–1043, 1996. 3.16, A.1.2

[49] Charles Simonyi. The death of computer languages, the birth of intentional programming. In *NATO Science Committee Conference*, pages 17–18. Citeseer, 1995. 3.16

[50] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997. 3.17

[51] Xianjun Chen, Zhoupeng Ji, Yu Fan, and Yongsong Zhan. Restful API architecture based on laravel framework. In *Journal of Physics: Conference Series*, volume 910, page 012016. IOP Publishing, 2017. 3.17

[52] Douglas R Smith. A generative approach to aspect-oriented programming. In *International Conference on Generative Programming and Component Engineering*, pages 39–54. Springer, 2004. 3.17

[53] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016. 3.18, 7.10.2

[54] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016. 3.18

[55] Ilche Georgievski and Marco Aiello. An overview of hierarchical task network planning. 2014. 3.18

[56] Evren Sirin, Bijan Parsia, Dan Wu, James Hendler, and Dana Nau. HTN planning for Web Service composition using SHOP2. *Journal of Web Semantics*, 1(4):377–396, 2004. International Semantic Web Conference 2003. URL: `http://www.sciencedirect.com/science/article/pii/S1570826804000113`, `doi:10.1016/j.websem.2004.06.005`. 3.18

[57] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):281–300, 1997. URL: `http://www.sciencedirect.com/science/article/pii/S0004370296000471`, `doi:10.1016/S0004-3702(96)00047-1`. 3.18

[58] D. Arellanes and K. Lau. Exogenous Connectors for Hierarchical Service Composition. In *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*, pages 125–132, Nov 2017. `doi:10.1109/SOCA.2017.25`. 3.18

[59] Krzysztof Czarnecki. *Generative Programming*. PhD thesis, Technical University of Ilmenau, 1998. 3.19

[60] Jiri Kubalik, L Rothkrantz, and J Lažanskỳ. Genetic Algorithms with Limited Convergence. In *Information Processing with Evolutionary Algorithms*, pages 233–253. Springer, 2005. 3.20, A.1.2

[61] John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, 1992. 3.20

[62] John R. Koza. *Genetic programming II: automatic discovery of reusable programs*. MIT Press, Cambridge, Mass, 1994. 3.20

[63] John R. Koza. *Genetic programming III: darwinian invention and problem solving*. Morgan Kaufmann, San Francisco, 1999. 3.20

[64] John R. Koza. *Genetic programming IV: Routine human-competitive machine intelligence*, volume 5. Springer, New York, 2003. 3.20

[65] Jiří Kubalík, Eduard Alibekov, Jan Žegklitz, and Robert Babuška. Hybrid single node genetic programming for symbolic regression. In *Transactions on Computational Collective Intelligence XXIV*, pages 61–82. Springer, 2016. 3.20

[66] Kubalík Jiří. Using genetic algorithms with real-coded binary representation for solving non-stationary problems. In *Adaptive and Natural Computing Algorithms*, pages 222–225. Springer, 2005. 3.20

[67] Christopher M Bishop. *Pattern recognition and machine learning*. Information science and statistics. Springer, New York, NY, 2006. Softcover published in 2016. URL: `http://cds.cern.ch/record/998831`. 3.21

[68] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986. 3.21

[69] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. 3.21

[70] Jaegul Choo and Shixia Liu. Visual analytics for explainable deep learning. *IEEE computer graphics and applications*, 38(4):84–92, 2018. 3.21

[71] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *ACM SIGPLAN Notices*, volume 51, pages 522–538. ACM, 2016. 3.22

[72] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–268. ACM, 2019. 3.22

[73] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program Synthesis by Type-Guided Abstraction Refinement. *Proceedings of the ACM on Programming Languages*, (12), 2019. URL: `https://doi.org/10.1145/3371080`. 3.22

[74] Roberto Cignoli. Injective de Morgan and Kleene algebras. *Proceedings of the American Mathematical Society*, pages 269–278, 1975. 4.1

[75] Philip Hugly and Charles Sayward. Are all tautologies true? *Logique & Analyse*, 125-126:3–14, 1989. 4.1

[76] Thomas Lukasiewicz. Probabilistic Logic Programming. In *ECAI 98*, 13th European Conference on Artificial Intelligence, pages 388–392. John Wiley & Sons, Ltd., 1998. 4.1

[77] Jaroslav Peregrin. Logika a logiky. *Academia, Praha*, 2004. 4.2

[78] Ronald Fagin. *Reasoning about knowledge*. MIT Press, Cambridge, Mass, 2003. 4.2

[79] O. Gasquet, A. Herzig, B. Said, and F. Schwarzentruber. *Kripke's Worlds: An Introduction to Modal Logics via Tableaux*. Studies in Universal Logic. Springer Basel, 2013. URL: `https://books.google.cz/books?id=QPrTbwAACAAJ`. 4.3

[80] Giovanna D'Agostino, Angelo Montanari, and Alberto Policriti. Modal Logic and Set Theory: a Set-Theoretic Interpretation of Modal Logic. 4.3

[81] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988. 5, 7.1

[82] OASIS Standard. Web Services Business Process Execution Language Version 2.0. 2007. URL: `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`. 5.3, 8.16.3

[83] A. Gill. *Introduction to the theory of finite-state machines*. McGraw-Hill electronic sciences series. McGraw-Hill, 1962. 6.4

[84] K. Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. 6.4.4

[85] John Ellson and Emden R. Gansner and Eleftherios Koutsofios and Stephen C. North and Gordon Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*, pages 127–148. Springer-Verlag, 2003. 6.7, 10.3.2

[86] Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall, Upper Saddle River, 2nd edition, 1997. 7.1, 7.6.6

[87] Robert C Martin. Principles of OOD. *línea]. Available: http://butunclebob. com/ArticleS. UncleBob. PrinciplesOfOod.[Último acceso: 29 Agosto 2016]*, 1995. 7.1, 7.6.6

[88] M. Fowler. Yagni. 2015. URL: `https://martinfowler.com/bliki/Yagni.html`. 7.1

[89] G. J. Holzmann. Points of Truth. *IEEE Software*, 32(4):18–21, July 2015. doi:10.1109/MS.2015.103. 7.4.1

[90] Fowler, M. *Patterns of Enterprise Application Architecture*. A Martin Fowler signature book. Addison-Wesley, 2003. URL: https://books.google.cz/books?id=FyWZt5DdvFkC. 7.4.3, 7.6.6, 7.7

[91] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995. 7.4.3

[92] Marti Hearst. Design recommendations for hierarchical faceted search interfaces. In *ACM SIGIR workshop on faceted search*, pages 1–5. Seattle, WA, 2006. 7.6.1

[93] Doctrine ORM, 2019. URL: https://www.doctrine-project.org/projects/orm.html. 7.6.3

[94] Dhanji R Prasanna. Dependency injection. 2009. 7.6.3

[95] Martin Fowler. Command Query Separation, 2005. URL: https://martinfowler.com/bliki/CommandQuerySeparation.html. 7.6.6, 7.10.1

[96] Greg Young. CQRS documents by Greg Young. 2010. URL: http://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf. 7.6.6, 7.10.1

[97] Martin Fowler. Event Sourcing, 2005. URL: https://martinfowler.com/eaaDev/EventSourcing.html. 7.6.6, 7.10.1

[98] Scott W Ambler. The design of a robust persistence layer for relational databases, 2005. URL: http://www.ambysoft.com/downloads/persistenceLayer.pdf. 7.7

[99] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the Situation Calculus. *Computer and Information Science*, 3(18), 1998. 7.10.2

[100] Michael Thielscher. Introduction to the Fluent Calculus. *Electronic News Journal on Reasoning about Actions and Change*, 3:12–20, 1999. 7.10.2

[101] Murray Shanahan. The event calculus explained. In *Artificial intelligence today*, pages 409–430. Springer, 1999. 7.10.2

[102] Arthur Bit-Monnot. *Temporal and hierarchical models for planning and acting in robotics*. PhD thesis, 2016. 7.10.2

[103] Pathathai Na Lumpoon. *Toward a framework for automated service composition and execution: E-tourism Applications*. PhD thesis, 2015. 7.10.2

[104] Zinovy Diskin and Tom Maibaum. Category theory and model-driven engineering: From formal semantics to design patterns and beyond. *Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice*, page 173, 2014. 7.10.3

[105] Remco M. Dijkman and Marlon Dumas and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294, 2008. URL: http://www.sciencedirect.com/science/article/pii/S0950584908000323, doi:10.1016/j.infsof.2008.02.006. 8.2.4

[106] Michael Barr and Charles Wells. *Category theory for computing science*, volume 1. Prentice Hall New York, 1990. 8.5

[107] Harold Simmons and Andrea Schalk. Category Theory in Four Easy Movements. *Online Book, University of Manchester*, 2003. 8.5

[108] Reinhard Diestel. Graph Theory. *Graduate Texts in Mathematics*, 173, 2005. 8.8

[109] Camunda BPMN Engine. URL: `https://camunda.com/products/bpmn-engine/`. 8.16.2

[110] Lianping Chen. Microservices: Architecting for Continuous Delivery and DevOps. In *IEEE International Conference on Software Architecture (ICSA)*, 2018. 8.16.2

[111] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. URL: `http://www.sciencedirect.com/science/article/pii/0890540187900526`, `doi:10.1016/0890-5401(87)90052-6`. 8.16.4

[112] Ali Khalili and Armando Tacchella. Learning nondeterministic mealy machines. In *International Conference on Grammatical Inference*, pages 109–123, 2014. 8.16.4

[113] Arlindo L Oliveira and João P Marques Silva. Efficient search techniques for the inference of minimum size finite automata. In *Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No. 98EX207)*, pages 81–89. IEEE, 1998. 8.16.4

[114] Dennis M. Ritchie. The Evolution of the Unix Time-sharing System. *Communications of the ACM*, 17:365–375, 1984. 10.1

[115] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. 10.3.1

[116] Tomaz Curk, Janez Demsar, Qikai Xu, Gregor Leban, Uros Petrovic, Ivan Bratko, Gad Shaulsky, and Blaz Zupan. Microarray data mining with visual programming. *Bioinformatics*, 21:396–398, February 2005. URL: `http://bioinformatics.oxfordjournals.org/content/21/3/396.full.pdf`. 10.5

[117] K. Konstantinides and J.R. Rasure. The Khoros software development environment for image and signal processing. *Image Processing, IEEE Transactions on*, 3(3):243–252, 1994. `doi:10.1109/83.287018`. 10.5

[118] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. YALE: Rapid Prototyping for Complex Data Mining Tasks. In Lyle Ungar, Mark Craven, Dimitrios Gunopulos, and Tina Eliassi-Rad, editors, *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 935–940, New York, NY, USA, August 2006. ACM. URL: `http://rapid-i.com/component/option,com_docman/task,doc_download/gid,25/Itemid,62/`, `doi:10.1145/1150402.1150531`. 10.5

[119] Shirin Sohrabi, Jorge A. Baier, and Sheila A. McIlraith. HTN planning with preferences. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 1790–1797, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc. URL: `http://dl.acm.org/citation.cfm?id=1661445.1661733`. 10.5, 12.1

[120] Riccardo Rasconi, Nicola Policella, and Amedeo Cesta. SEaM: Analyzing Schedule Executability Through Simulation. In Moonis Ali and Richard Dapoigny, editors, *IEA/AIE*, volume 4031 of *Lecture Notes in Computer Science*, pages 410–420. Springer, 2006. URL: `http://dx.doi.org/10.1007/11779568_45`. 10.5

[121] Niko Schwarz, Mircea Lungu, and Oscar Nierstrasz. Seuss: Decoupling responsibilities from static methods for fine-grained configurability. *Journal of Object Technology*, 11(1):3:1–23, April 2012. URL: http://www.jot.fm/contents/issue_2012_04/article3.html, doi:10.5381/jot.2012.11.1.a3. 10.5

[122] *HTML Living Standard*. WHATWG, 2019. URL: https://html.spec.whatwg.org/multipage/. 11

[123] *Twig Documentation*, 2019. URL: https://twig.symfony.com/doc/3.x/. 11

Index