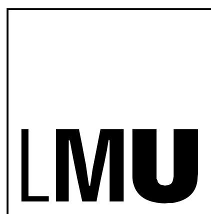
SOFTWARE ENGINEERING

Elite Graduate Program

Master's Thesis

Uselets: UIs using Actors as an Abstraction for Composable Communicating Components

Thomas Weber



Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

SOFTWARE ENGINEERING

Elite Graduate Program

Master's Thesis

Uselets: UIs using Actors as an Abstraction for Composable Communicating Components

Autor:	Thomas Weber
Matrikelnummer:	1450761
Beginn der Arbeit:	1. August 2017
Ende der Arbeit:	1. Februar 2018
Erstgutachter:	Prof. Dr. Alexander Knapp
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Betreuer:	Prof. Dr. Alexander Knapp

Universität
Augsburg
University

TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Institut für Software & Systems Engineering

Universitätsstraße 6a D-86135 Augsburg

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Teile dieser Arbeit sind im Zuge der ICFP 2017 Student Research Competition veröffentlich worden.

Augsburg, den 31. Januar 2018 Thomas Weber

Acknowledgments

Let me express my gratitude for the people at LFCS, University of Edinburgh, especially James Cheney, Simon Fowler, and Sam Lindley, for hosting me there while working on this thesis. Not only is some of their work the basis and inspiration for this thesis, their feedback and encouragement has been invaluable, too.

Also, my thanks to Prof. Alexander Knapp for his guidance and feedback throughout the writing of this thesis and the Masters program in general.

Lastly, to all those who read this thesis and gave feedback on it: Thank you for finding the time and diligence.

Abstract

Applications with rich user-interaction, especially web-apps, have become part of the day-to-day lives of most users, both for work and leisure. More complex applications always mean more complex development though, including for the user interface. Using reusable off-the shelf components can help here: By encapsulating functionality, parts of the application can be developed independently, which makes them easier to understand, write, and maintain. But slicing the application into small parts creates the need for composition, coordination, and communication, which brings its own challenges.

This thesis proposes a concept for composable, communicating UI components called Uselets, that use Actor-style message passing.

Uselets can be used both to abstractly model the UI and to implement it, both without what is usually part of the expertise of software developers: presentation and design aspects. Visualization is delegated, creating a clear separation of development domains and concerns. Instead the focus is on the structure and interaction logic, from which the resulting user interface is inferred automatically.

By building the Uselet abstraction on top of the Actor model, they provide simple composition and a unified interface for interaction. This allows developers to focus on the core aspects of any user interface: Structure and interaction.

The examples and implementations provided with this thesis focus on the creation of web-based UIs, but the concept of Uselets could find much further use for all kinds of applications.

Contents

1 Introduction **1**

2 Background Information **5**
- 2.1 Web-Development . . . 5
  - 2.1.1 A Brief History . . . 5
  - 2.1.2 Main Challenges . . . 6
  - 2.1.3 Main Advantages . . . 7
- 2.2 Concurrency and Actors . . . 8
  - 2.2.1 The Actor Model . . . 8
- 2.3 Links . . . 9
  - 2.3.1 Core Syntax . . . 9
  - 2.3.2 Type System . . . 9
  - 2.3.3 Formlets . . . 10
  - 2.3.4 Concurrency . . . 11
  - 2.3.5 Client-Server-Interaction . . . 12

3 Uselets Example **15**

4 Uselets Concepts **21**
- 4.1 Abstractions . . . 22
- 4.2 Usage Methodology . . . 28
- 4.3 Key Benefits . . . 32
- 4.4 Visual Notation . . . 33

5 Uselets Implementation **35**
- 5.1 Uselets in Links . . . 35
- 5.2 Uselets in JavaScript . . . 37
- 5.3 Implementation Differences and Challenges . . . 40
- 5.4 Testing . . . 42
- 5.5 Debugging . . . 44

6 Case Studies **47**
- 6.1 Kanban Board . . . 47
- 6.2 TodoMVC . . . 51

7 Evaluation **55**

8 Related Work **59**

9 Conclusion and Future Work **63**
- 9.1 Future Work . . . 63
- 9.2 Conclusion . . . 66

1 Introduction

These days software applications, especially web-based ones, have become ubiquitous. But to make them *accessible* to a large number of people, there is a definitive need for quality user interfaces. Developing larger UIs is a cumbersome task though, prone to errors, inconsistencies etc. A common solution is breaking the UI into smaller components that can be developed individually (e.g. [34, 44, 36]). But wiring these together is frequently a tedious process, full of repetition and boilerplate.

The act of breaking a large monolithic system into smaller parts is very similar to what you do, when you create a distributed system. To tackle the challenges of distributed systems, people have come up with various abstractions and models to hide away the low-level details, for example the Actor model [2, 51]. Leveraging the Actor model, this thesis introduces Uselets, a high-level, component-based abstraction for web-based UIs, inspired by Formlets [24], composable web-forms.

Other than the name suggests, there is only limited overlap, in implementation or usage, to the original Formlets though. In fact, the application domain for those two is already quite different: Formlets focus on forms that are submitted to the server where they are evaluated, while Uselets are meant for primarily client-side interaction with only the addition of client-server-communication. Yet they do share the major design goals: Good composability while hiding low-level details.

Uselets attempt to transfer the benefits of Formlets to more general UI elements. Besides that, they provide additional improvements, such as a more uniform way of defining interaction and a stricter separation of presentation and logic.

- By modelling the UI like one would model an Actor system, the focus is on the internal logic and possible interactions. The presentation is delegated to clearly defined holes for people with design expertise to fill.
- Each Uselet is fully self-contained, making it easy to compose with others, simplifying the development of larger applications. With that, one can imagine repositories of pre-made Uselets that can be re-used in a plug-and-play fashion.
- The development of custom Uselets focuses exactly on the two important (non-presentation) aspects of a UI: In what structure of data is to be presented to the user and how the user can interact with it. Boilerplate and configuration overhead is kept to a minimum.
- All interaction is unified as message-passing.
- Interaction and the resulting change in the system is defined in a declarative way.
- With all change in the system being the result of an interaction, i.e. a message, Uselets are simple to test and to debug (see Sect. 5.4 and 5.5).
- To implement Uselets, only very few features are necessary, making the implementation in different languages simple (see Chapter 5).

To bridge the gap from an abstract model to a concrete implementation, Uselets serve both as the conceptual model for the problem space and concrete framework for implementation

in the solution space. To showcase the viability of the abstract for building concrete HTML-based applications, two implementations, one in the functional programming language Links [23] and the other in JavaScript, are provided.

While the languages of those two implementations differ greatly, the underlying model is the same, and the user code is very similar (see Sect. 5.3 for a comparison).

To demonstrate the viability of Uselets for developing applications, two case studies (Chapter 6) are provided. Since Uselets allow specifying an application in an abstract way, i.e. without a concrete implementation, Chapter 7 evaluates Uselets in the context of criteria proposed in the literature for such abstract specifications.

With Uselets we get a new framework for creating web UIs and potentially UIs in general. Given the large number of existing frameworks to improve UI and web development, the desirability of yet another one needs some justification. The comparison with some popular frameworks, as well as plain JavaScript as the base-language (Chapter 8), demonstrates the different approaches and their different focus, advantages and disadvantages.

Finally, Sect. 9.1 proposes a number of improvements and potential other applications for Uselets.

Motivation

While there are many ways for creating user interfaces, web-based applications and their UIs are, together with mobile apps, some of the most important and popular ways for creating software for end-users [62]. From their humble origins as fully static content, web-based user interfaces have evolved into full, rich, and dynamic applications. The many tools and APIs available nowadays certainly offer great potential but also immediately add complexity.

The level complexity of those applications unfortunately makes them prone to errors. Understanding them to their full extend becomes pretty much impossible for one individual. Extensive testing is necessary but even that rarely is sufficient. Approaches to break down complex systems into smaller, more comprehensible parts usually only push the complexity to a different level, e.g. the added overhead of composing the system.

Obviously, having errors in the system is undesirable, regardless of at what level they occur. Not only do they impair system reliability and correctness, they also impart additional costs for fixing, especially if they are found only late during development or in productive use.

A common solution, when the complexity of something exceeds one's capabilities of understanding, is to "use something that is simpler, safer or cheaper than" [80] the original thing, i.e. a model. Developing systems on the foundation of a model is a well-known practice in software engineering (and engineering in general) with different abstractions and models available. Such models offer a higher-level view of the system while hiding away low-level and implementation details. This allows software engineers to focus on the important aspects of their applications, i.e. focus on the problem space, not on the solution space [81], and potentially even enables domain experts to participate in the design process.

Not only are systems often easier to describe like this, they are frequently also simpler

to understand since the description is not polluted by unnecessary implementation details. This again helps developers and non-developers understand what is going on.

Some abstractions and models even allow the use of verification methods like model checking [13]. By formally verifying properties of the model, developers can give certain guarantees for the behaviour of the system based on that model. Such guarantees are desirable for developers, especially if the system is embedded into other systems, because those surrounding systems can be developed assuming those guaranteed properties. It also is beneficial for users of the system, since developers can ensure certain requirements will hold, such as system reliability or correctness.

Guarantees like that of course only hold if the system actually implements what the model describes. Manual implementation of the model still remains error-prone, but with a sufficiently detailed model it can be possible to generate either the whole code or parts of it, ideally yielding an executable model. This reduces the amount of code developers have to write, and brings a number of additional benefits: First and foremost, generated code should have less errors than manually written code. This of course depends on the generator to be bug free and the translation from model to code to be correct. But a single generator written and maintained by experts should be more reliable than having many applications manually written by developers of different levels of expertise. This also lowers the entry barrier, since developers are not required to have intricate knowledge of their platform and language but can rely on the code generator to produce code of adequate quality. Assuming they are possible, a code generator can also perform a number of optimizations based on the target platform.

Lastly, having code generated from an abstract model gives greater platform independence. While the web is a dominant platform for developing and deploying applications, it certainly is not the only one. Native mobile applications, desktop applications etc. are an important part of peoples' daily lives as well. Since not every platform is available to every user, it is often necessary to build an application multiple times, for each relevant platform. Developing applications by first creating an abstract model, and basing the implementations on that model, could potentially reduce the work of developing for multiple platforms significantly: With an adequate model, it could be possible to generate the majority of code from that model, so that developers only need to fill in the remaining platform-specific parts that could not be generated.

Based on this information, there are a number of criteria that an abstract model should have (e.g. [82, 87]). We can use these criteria to assess the quality of our abstraction, whether it satisfies these and other criteria (Sect. 7), e.g.:

- The abstraction should be conceptually simple to understand for both programmers and non-programmers.
- In particular no extensive knowledge of the underlying platform and its intricacies should be necessary.
- The implementation should either be generatable from the specification or the specification acts as an executable model, i.e. it can be used as runnable code.
- System properties should be verifiable either fully automated or semi-automated.
- The system created via the model should be less error-prone than manually written

code, i.e. there should be fewer potential sources for errors and errors should be easier to identify and diagnose.

2 Background Information

To gain a full understanding of the Uselets abstraction, some background information is necessary. For this purpose, the following chapter serves as an introduction to some general web-development concepts and the Actor concurrency model. It also gives a brief overview of the Links programming language, as the majority of examples are written in Links.

For readers with some general knowledge of these topics, or those interested in but a high-level overview of Uselets, it is perfectly possible to skip this section for now, jump directly to the core description of Uselets (Chapter 4), and to refer back to this chapter merely as reference for more background information.

2.1 Web-Development

Since Uselets are primarily a programming abstraction for web-based UIs, the following section will give a brief overview on web-technologies, their historical development, and the advantages and challenges of developing for the web.

2.1.1 A Brief History

The internet and its attached technologies have become one of the most important software platforms. At its origin though it was just meant to exchange simple, annotated text documents. Beginning of the 1990s the HyperText Transfer Protocol (HTTP) and the HyperText Markup Language (HTML) were introduced by Tim Berners-Lee and others [7] to exchange documents at CERN in Switzerland.

Both these technologies are still under active development even today, resulting in HTTP/1.1 [37], which was state of the art until 2015 when HTTP/2 was introduced [6]. It was further extended with additional specifications, e.g. for authentication, caching etc. [38, 39].

HTML too reached a long lived standard end of the 1990 with HTML 4.01 [78]. XHTML added to that soon after with a stricter, XML-based syntax [47]. Both were de facto replaced with the advent of HTML5 in 2014 [52]. HTML5 greatly revamped parts of HTML and added a large number of new elements for new types of content. Next to the current HTML specification, additional technologies, e.g. WebSockets, WebGL, the current iterations of CSS, the styling language of the web, etc., which are often times considered part of HTML5, have their own specifications.

Since the adoption of HTML features heavily relies on when browser-vendors implement them, many features have been introduced gradually. This continues to be the case; many of the newer additions to HTML5 in the succeeding versions 5.1 and 5.2 and orthogonal specifications have different levels of browser support.

These technologies, while the foundation of web-applications, merely provide static content though. To enable full dynamic applications, like desktop or mobile applications, on the web, JavaScript, or ECMAScript, the scripting language of the web, was necessary.

Introduced shortly after HTML, JavaScript, in its beginnings LiveScript for Netscape Navigator and JScript for Internet Explorer, was originally not a single language, due to the then ongoing "browser-war", i.e. the rivalry between Netscape and Microsoft on browser-market dominance. Soon after, in 1996, it was standardized as ECMAScript, which was subsequently updated and improved, leading to the current state of the ECMAScript standard and the JavaScript language.

At the beginning JavaScript was used primarily to add simple interaction to static content. This was the case roughly until AJAX, i.e. asynchronous data transfer, and related technologies were proposed in 2005 [43]. This led to the development of many now popular libraries like jQuery [85] and Dojo Toolkit [56].

This paved the way for many rich applications that made up what is commonly referred to as Web 2.0 [74, 83], defined by characteristics like rich user interaction, user participation, and software-as-a-service, via rich APIs, commonly REST APIs [40]. Those principles and technologies are still the foundation for applications today, even though it is not certain whether Web 2.0 is still current or a new version would be adequate.

More recent additions to the family of HTML and JavaScript specifications are for example WebWorkers [89], which introduce concurrency to JavaScript. Four further specifications, HTML Imports, Shadow DOM, Templates, and Custom Elements ([18, 20, 19, 16] respectively) intend to improve modularity, reusability, and separation of concerns for web development by paving the way for WebComponents [21].

Modern JavaScript provides a high-level language with a great number of APIs to access the many features and extensions available in modern browsers. While there were competing concepts to add interactivity to web applications, JavaScript is now the dominant tool, used by the overwhelming majority of websites [77].

Support for HTML based applications and progressive web apps on mobile platforms adds to the spread of HTML and JavaScript based UIs beyond the web-browser. Besides its use for client-side programming JavaScript has also been widely adopted for other uses, such as server-side programming (e.g. [71, 91]). Shedding its original bad reputation [25], it is now used in many projects beyond just extending static content on web pages.

Again, as with HTML features, JavaScript's newer language features are subject to varying browser and runtime environment support, leading to a number of fixes, polyfills, etc. Another common solution in recent times is source-to-source compilation or "transpilation". Developers do not write JavaScript but a language that is a superset of JavaScript [66], a different previously existing language [31] or a completely new language. This, amongst other benefits, allows developers to write their code without worrying about compatibility, because that is ensured by the transpiler.

2.1.2 Main Challenges

As with any programming environment, especially one that has grown as dynamically as the web has, it is no surprise that certain challenges exist for developers.

As mentioned above, many features are not immediately adopted by every browser. Since the web-browser is the primary environment for web-based UIs this requires developers to

build their applications with the different browsers in mind, always double checking and providing fallbacks should a feature not be available.

JavaScript also has a number of quirks and idiosyncrasies in its language design that are often not immediately clear to beginners. These can result in everything from poor code legibility to highly unexpected behaviour [69, 25]. Consequently, good tool support with e.g. a linter is necessary to ensure good practices and prevent common pitfalls and the use of poor code constructs.

This is especially true for amateur programmers or beginners. Since JavaScript has a fairly low entry barrier, it is a common language for people with little programming experience. This, and the fact that on the web it is simple to look up other people's code, leads to a lot of bad practices being propagated.

Some might argue [25] that the quality of the ECMA standard and other literature on JavaScript have played their part in the prevalence of sub-par JavaScript and the resulting poor reputation of the language.

2.1.3 Main Advantages

But while those are undeniably challenges, the web as platform has quite a few advantages as well. They are responsible for the popularity and importance of the web.

Its enormous reach is certainly one of the biggest advantages. Nearly every personal computer and mobile devices has a web-browser. Since this is about the only requirement on the client side, a web-based application is immediately available to almost everyone. This also cuts out software distribution, as hard-copy or download, entirely. But many web-applications can also be run offline if needed, e.g. as progressive web-apps [8], where only missing data is fetched from some server.

The simplicity of deployment and execution, especially locally for inspection during development, makes JavaScript especially attractive for beginners, since results are immediately visible and it is easy to try out different solutions and see their effect. The, by now, extensive tool support in browsers for experimentation, inspection, debugging etc. simplifies development too, while the enormous eco-system of libraries, frameworks, examples, etc. gives beginners easy starting points to learn web-development and, at the same time, allows professionals to prune repetitive work by extensive code-re-use.

Thanks to source-to-source compilation, which is widely available for JavaScript (e.g. [31, 66]), it is sometimes not even necessary to learn new languages, when a developer already knows one and is comfortable with it. And even then, the range of languages that can be translated to JavaScript is wide enough, such that everyone can find one that suits them best. This effectively negates the issue of browser compatibility for many scenarios, since the transpilers can take care of that. They can also improve code further during compilation, e.g. optimize for performance.

And while browser adoption is an issue, ultimately web-technologies are standardized, which adds to platform independence and does not force users and developers to lock into some proprietary eco-system.

The web is a ubiquitous platform. Developing for it is simple and accessible. When developing new tools or abstractions for web-development, be it for user-interfaces or any

other part, it clearly is important to maintain these advantages.

2.2 Concurrency and Actors

High-level abstractions and models are very common in areas like distributed systems, where they are used heavily to hide low-level implementation details. Due to the inert complexity of having a distributed system, it is very challenging to ensure all the potential pitfalls, like race-conditions and deadlocks, are dealt with. Abstractions for distributed system hide away all these challenges and low-level details, allowing developers to focus on their applications logic.

Over the years there have been a number of systems, some more theoretical, e.g. process-algebras like Milner's π-calculus [68] and CCS [67], and some that have led to very practical implementations, such as CSP [53] or the Actor model [51].

Many of them are based on some form of message-based communication. The system consists of sources of computation, the distributed parts of the system, that communicate by sending each other messages. The exact scheme of communications depends on the model, e.g. the channel-based communication of CSP, where there are dedicated FIFO channels for messages, or more direct message sending, where the endpoints of the communications are the computing elements themselves, as is the case in the Actor model.

2.2.1 The Actor Model

As mentioned, one of those models for describing distributed systems is Hewitt's Actor Model [51]. This model has found a number of practical implementations in many programming languages, such as the built-in concurrency in Erlang [4, 88] or Links [23], or frameworks such as Akka [63].

The Actor system uses the name-giving *Actors* as its core primitive. An Actor is a single source of computation that, aside from its computation, can perform three actions:

- **Send** messages to other Actors
- **Receive** messages and execute a behaviour depending on the incoming message
- **Spawn** new Actors

With these three operations, "Actors are intended to (be able to) model **all** digital computation" [49]. The relative simplicity yet expressiveness of those three rules makes this model a popular one, evident in the many implementations of the Actor model. Besides the practical application, various people have also come up with formal semantics to allow reasoning about Actor systems [46, 3, 14].

While the third rule suggests a parent-child relation between Actors and thus a tree topology, Actors in an Actor system do not necessarily follow a strict, pre-defined topology. To send a message to an Actor a sender must only know the recipients address. Since messages itself can contain addresses, the relationship which Actor knows which and thus the topology is flexible.

Yet a topology can be defined by e.g. by hardwiring a rigid structure or implicitly by addressing Actors with a scheme that yields a certain topology. In the second version the

topology is not enforced though, since a topological addressing scheme does not prevent Actors to look up other addresses and send messages outside the topological bounds.

The topological flexibility means that composing of Actors is simple: If the address of an Actor is known to other Actors in a system, e.g. by receiving it in a message, the Actor is part of the system. There are additional, slightly more elaborate schemes to compose larger systems of Actors (cf. [2]).

With these properties the Actor model alleviates a number of issues of concurrency, most notably the complexity and composability. The proximity of conceptual model and practical implementations minimizes the overhead of transitioning between those two. Extensive research into the properties of Actors [3, 57], modelling, and verification [32, 33, 55, 61], plus a myriad of implementations for various languages [4, 57, 63] make the Actor model a good foundation for a distributed abstraction.

2.3 Links

As one of the Uselet implementations, and the majority of the examples in this thesis utilize Links [23], this section will provide a brief introduction to this language. [1]

Links is a strongly typed functional programming language with primary focus on web development. It aims to ease development by providing a single language for all three common “tiers” of web-programming: Database, backend and frontend.

It also serves as the research vehicle for various language features at the LFCS, University of Edinburgh, where it is being developed.

Some of those features will be described in the following sections, with a focus on those that are of relevance for the Uselets implementation. For more in-depth information on Links, see the language documentation.

2.3.1 Core Syntax

Links’ core syntax provides all the features one would expect from a functional programming language: Primitive types, functions, basic arithmetic and boolean operations, and some basic data-structures like lists, tuples, and records.

Value declarations with the “var” keyword are, other than the keyword suggests, constant. Functions are declared with the keyword “fun” and can be preceded by a type signature “sig”

2.3.2 Type System

Links is a strongly statically typed language. It has HM type inference, so type annotations, while possible, can be omitted in many cases. Besides the primitive types for values and the built-in data-structures, variant types can be used as custom type constructors.

[1] Since Links is still under active development, the following section describes the Links languages at its Dalry release. Some sections may not be accurate for other releases.

Effects

Along with types for values and variant types, Links also supports effect types. Functions are annotated, next to their input and output, with effects that can happen during run-time. Absence of an effect guarantees it will not happen, presence means it *might* happen. There are numerous variations of syntactic sugar for annotating effects.

Effects are for example the `wild` effect which represents that the function does not execute code on the database, or the `hear` effect which describes what message are accepted and sent in a message passing environment.

Variant types and effect types are represented as polymorphic row types: Unless explicitly constraint, a term that is typed with a certain row type can be used where a row type is expected that subsumes that of the term. For example, a term of variant type `A` can be used in an instance where a type `[| A | B |]` is expected. That is because the term `A` will be typed, by the inference, as an open row type `[| A | _ :: Any ]`, i.e. a polymorphic type that includes the type-constructor `A` and potentially others.

2.3.3 Formlets

Part of the original work on Links were Formlets [24]. Formlets are a high-level abstraction for composable, type-safe web-forms, and act as inspiration for Uselets.

Their goal is to solve the issue that arises when data is entered in a form on a web-client and submitted to the server. For trivial examples this is straight forward enough, but as soon as the entered data has a more complex structure or the code has to be updated frequently, due to e.g. changes in the requirements, this can lead to inconsistencies and other problems.

A single Formlet encapsulates both sides of this scenario, the client side with some widget, for entering the data, and the server side that parses the entered data into a desired data-structure. Formlets can then be composed to construct more complex data-structures from primitive ones, while on the one side the widgets are put together and on the other the parsing composes too, yielding the desired data at the end. See Code 2.1 for an example Formlet for a Date, composed from three Formlets for numbers.

```
1  fun dateFormlet () {
2    formlet <#>
3      Day:   {inputInt -> day}
4      Month: {inputInt -> month}
5      Year:  {inputInt -> year}
6    </#>
7    yields {
8      Date(day, month, year)
9    }
10 }
```

Listing 2.1: A Formlet for entering a date. It is composed from three inputs for integers that are bound to the names *day*, *month*, and *year*, respectively. The `yields` clause describes how the submitted data should be processed, in this case passed to the `Date` type-constructor.

To embed a Formlet in a page it has to be rendered, which requires a *handler*. A handler is a function that performs some action with the parsed user input, returning

some XML/HTML as result. The rendering yields the initial XML/HTML, which can be embedded into a page (see Code 2.2).

```
fun dateHandler (date) {
  <h1>Date submitted</h1>
}

var initialXml = render(dateFormlet(), dateHandler);

page <div>
  { initalXml }
</div>
```

Listing 2.2: Rendering of a Formlet, using a simple handler function. Once rendered the returned initial XML can be embedded into a page.

The whole of Formlets is based on the theoretical framework of applicative [24], and later monoidal [79], functors. Additional work on Formlets extended it with initial or default values [79] and input validation [22].

2.3.4 Concurrency

The built-in concurrency model for Links is Actor and channel based. Using `spawnAt` or its specializations, `spawn` and `spawnClient`, the programmer can create an Actor-process on client or server. An Actors behaviour is defined by a `receive` block. A `receive` suspends function execution until a message is received. Pattern matching allows switching behaviour depending on the incoming message (see Code 2.3). The handling `receive` must again return a `receive` block if the Actor should continue to run. This can either be done recursively to maintain the same Actor behaviour or it can be switched for a different one, hot-swapping Actor behaviour.

```
fun actorHandler () {
  receive {
    case MyMsg ->
      print("MyMsg");
      actorHandler()
    case _ ->
      print("Unknown message");
      actorHandler()
  }
}

var serverProcess = spawn { actorHandler() };
var clientProcess = spawnClient { actorHandler() };
```

Listing 2.3: Actor instantiation. `actorHandler` recursively defines the Actor behaviour for each message. `spawn` spawns a process on the server, `spawnClient` on the client.

Channel-based Communication with Session Types

A more recent extension of Links are session types. They allow encoding a sequence of message-passing-interaction in a π-calculus-like style. Distributed entities can then

communicate via channels for which the communication protocol is specified and statically checked. To ensure the session type statically, the cannel is not modified when used but instead the read and write operation always return an updated channel for the next step in the session type.

```
typename Add = [&| ?Int.?Int.!Int.End |&];

sig add : (Add) ~> ()
fun add (s) {
  var (x,s) = receive(s);
  var (y,s) = receive(s);
  var _ = send(x+y,s);
  ()
}

sig user : (~Add) ~> Int
fun user (s) {
  var (x, s) = receive(send(6,send(7,s)));
  x
}

user(fork (add)) # 13
```

Listing 2.4: A simple distributed calculator example.

In the above example[2], fork creates a new process executing add, which communicates with user via a channel that first expects user to send two numbers, to which add then responds with a number of its own (as specified by the session type ?Int.?Int.!Int.End). The same channel cannot be used for all three operations, because the types would not match. send therefore returns the channel one step further in the communication protocol, and receive returns the next channel plus the received value. This way Links has both Actor-style and channel-based distribution (although both could be emulated by the other [41]).

2.3.5 Client-Server-Interaction

The primary goal of the Links language was to reduce the harsh borders between the “tiers” of developing a typical web application, i.e. between database and backend and backend and frontend.

For database access it provides a syntax to define and query tables in a relational database system. Since this thesis focuses on user interface and frontend aspects, this is not discussed here. But the communication between client and server, i.e. front- and backend is the primary source for data in the client-side application, so of interest for Uselets.

To break the bounds between the tiers, one writes Links code as if everything happens in the same tier. The language then handles communication. In most cases this happens by providing the function twice, once as the Links version in the backend and once as a client version as transpiled JavaScript. So, in most cases, no network communication will occur.

[2] For this and more examples, also see the Links repository

Some function cannot be executed on the client though, e.g. because they access the database or other data that is private to the server. Additionally, functions can be explicitly annotated to run only on the server or only on the client (see Code 2.5). This creates a need for network communication.

```
1 fun serverOnly () server {
2   # This code is always executed on the server
3 }
4
5 fun clientOnly () client {
6   # This code is always executed on the server
7 }
```

Listing 2.5: Links functions annotated as client- and server-only.

Links creates a function stub for these cross-tier functions, which makes a remote-procedure-call in continuation-passing-style, i.e. remainder of code that needs to be executed after the RPC is sent with the request as encoded continuation. For example, when a client calls a server-only function it encodes the handling of the return value as continuation and sends a request with that continuation to the server. The server performs the server-only actions and executes the continuation. Since the handling of the response is client code again, a response is sent with the return value encoded into the continuation. The client then carries on with the suspended execution in the continuation, now with the return value from the server-function. All this is wrapped under the hood of the language and opaque to the developers.

3 Uselets Example

Consider writing a simple todo-list application. For the purpose of this example it only needs to offer the very minimum of functionality:

- It should have a header, delivered by an external designer.
- There should be a todo-list.
- The todo-list shows each todo-item with its completion status.
- Each item can be marked completed.
- There should also be a form to add new items.

The core idea of Uselets is to have easily re-usable components that allow us to compose our application from off-the-shelf parts. We will begin by writing in a top-down approach, assuming we have all the necessary parts available in some repository and then, realizing that we do not, write those sub-components.

An user interface based on Uselets has two building-blocks: The components that create the structure and messages that facilitate interaction. Clearly written requirements allow us to deduce both: The structure is what is present, so e.g. the *header* and the *todo-list* which consists of the *items* and the *form*. And for everything that can happen, i.e. *adding* and *removing* items, there is a message.

The example discussed here will focus on the Uselet-part, using the standard tools provided by the Links language to embed our todo-list into a webpage and handle client-server communication.

With the knowledge that our app consists of a header and the actual todo-list, we can begin:

```
1 var myApp = header <+> todoList;
```

Listing 3.1: The todo-app, composed of a header and the actual list.

We here assume a `header` and a `todoList` component that we compose with the `<+>`-combinator. Each Uselet application has a tree-structure with the Uselets as subtrees. The `<+>`-combinator combines two Uselet-subtrees on the same level, i.e. as siblings.

```
1 (<+>): (Uselet,  Uselet) ~> Uselet
2  (+>): (XmlItem, Uselet) ~> Uselet
```

Listing 3.2: Type signatures of the Uselet combinators: `<+>` for composing Uselets as siblings, +> to embed a Uselet in a piece of XML. Effect types parameterizing the Uselet type and the function-arrow are omitted for clarity from here on.

Since we have neither the `header` nor the `todoList` component yet, we now have to write these. Those components are representative of the two types of components a Uselet system, or any UI for that matter, consists of, static and dynamic content respectively. The `todoList` has interactive elements, so it is a dynamic component. The `header` on the other hand never changes, so it is fully static.

There are two primitives to create static content, `textUselet` for static text and `xmlUselet` for arbitrary static XML/HTML. These static components make up the leaves in the Uselet tree-structure.

```
textUselet: (String) ~> Uselet
xmlUselet: (Xml) ~> Uselet

uselet: (
  String,
  model,
  (Ctx) ~> (model, message) ~> model,
  (Ctx) ~> (model) ~> Uselet
) ~> Uselet
```

Listing 3.3: The signatures of the Uselet constructor-functions.
`textUselet` to construct static text
`xmlUselet` to construct static XML
`uselet` to construct dynamic Uselets
Effect types are omitted for clarity.

The idea is that the developer working on the interactive elements should not have to care about static content. This is up to dedicated designers, who provide that kind of content to embed, giving a clear boundary between application logic and interaction aspects and presentation.

Assuming our designer has delivered a very simple header to fill our `xmlUselet`, which in Links can be done with literal XML syntax, the header-component is complete:

```
var header = xmlUselet(
  <header><h1>Todo-List</h1></header>
);
```

Listing 3.4: A static Uselet for the header.

We now move on from static content to the first dynamic part of the application: the todo-list-component. For this we have to figure out the core aspects of any UI:

- What is the underlying data?
- How can a user interact with the data?
- And how is it structured internally?

For the list-component the underlying data-model is obviously a list of items. To distinguish between checked and unchecked items we will use a pair of string, the content, and boolean, the checked-status:

```
typename TodoItem = (String, Bool);
```

Listing 3.5: The type of a single todo-item: Its textual content and the checked-status.

Now to create the Uselet we have to provide the three components mentioned above plus an additional name.

```
1 var todoList = uselet(
2   "todo-list", # name
3   [ ], # data
4   updateFn, # interaction
5   viewFn # structure
6 );
```

Listing 3.6: The creation of the `todoList`.

We choose “todo-list” as name. The internal data-model is initialised with the empty list, but defining `todoList` as a function that is passed some initial data is equally feasible.

Now it remains to provide the `updateFn` function for the interaction and `viewFn` to produce the internal structure.

`updateFn` takes three parameters:

- The Uselets context (described in more detail in Sect. 4.1),
- the current model,
- and the incoming message that triggered this update.

The return value is the (possibly) updated model. For this reason, we refer to this function as the “update-function”.

As described in the requirements, we have two possible interactions: Adding and removing items. Using e.g. pattern matching we select the behaviour based on the incoming message-type. When we receive our `AddItem` message, we prepend the new item, when we see `RemoveItem`, we filter it out of our list. Any other message will be discarded and the data remains unchanged:

```
1 fun updateFn (ctx) (model, msg) {
2   switch (msg) {
3     case AddItem(item) ->
4       (item, false) :: model
5     case RemoveItem(item) ->
6       filter(fun (i) { i <> item }, model)
7     case _ -> model
8   }
9 }
```

Listing 3.7: The update-function for the todo-list.

This is everything required in terms of interaction and data manipulation, allowing us to move on to the rendering of the internal component structure.

Similar to the update-function, the view-function, `viewFn`, takes the Uselets context as well as the current data-model. From that it produces a Uselet substructure. We refer to this function as the “view-function”. The elements of this substructure are the descendants of our `todo-list` in the overall Uselet tree.

As described in the requirements, we want to show all list items as well as the form for creating new items.

```
fun viewFn (ctx) (model) {
  fold_left(fun (a, b) { a <+> b },
    map(fun (i) {
      # individual items
      todoItem(i, fun () { ctx.toSelf(RemoveItem(i)) })
    }, model)
  )

  <+>

  # form to add items
  createItemForm(
    # add-item-callback
    fun (str) { ctx.toSelf(AddItem(str)) }
  )
}
```

Listing 3.8: The view function of the todo-list.

This we do using the standard tools of `map` and `fold` to produce a `todoItem` Uselet per item and compose those together.

The components used here, `todoItem` and `createItemForm` both interact with the todo-list component. Generally, interaction is facilitated through a Uselets context, which provides the means to pass messages to e.g. other Uselets. If “todoItem” and “createItem-Form” were dynamic Uselets created with the “uselet” function they will have their own context for communicating with “todoList”. They are static ones though, so they do not have their own context and rely on their parent Uselet to pass them a callback function (the anonymous functions that call `ctx.toSelf` in the above example).

With that we have written all parts of our todo-list component and it remains to provide the `todoItem` and `createItemForm` component.

As mentioned above they are both static Uselets, like the `header`, so we provide a `xmlUselet` to fill with HTML provided by the assumed design team. Because these components provide interaction, the designer must now attach the passed callback to the interaction-endpoints in the HTML.

In Links this is done via attributes prefixed by ‘`l:`’.

```
fun todoItem ((str, checked), rmCallback) {
  xmlUselet(
  <li class="{ if (checked) "checked" else "" }">
    <input
      checked="{ if (checked) "true" else "false" }"
      type="checkbox"
      l:onclick="{ rmCallback() }"
    ></input>
    stringToXml(str)
  </li>
  )
}
```

Listing 3.9: The markup for a single list-item.

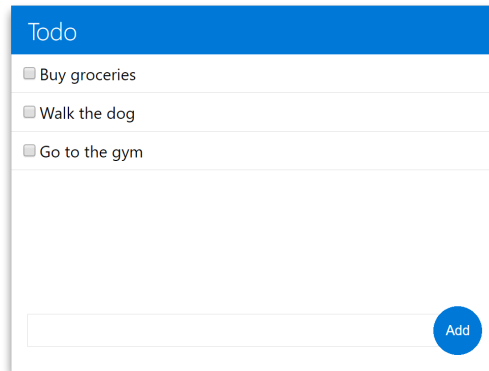


Figure 3.1: The todo-application built with Uselets

The `createItemForm` is constructed analogously.

With that we have constructed our Uselet structure and can hook it into a page using `useletApp` which spawns the Uselet-Actors and returns the initial XML/HTML. This can be embedded into a page using the standard Links `page` keyword.

```
1 page <body>
2   { useletApp("app", myApp, fun (_) { () }) }
3 </body>
```

Listing 3.10: Embedding of Uselets into a page in Links.

The `useletApp` again expects a name as the first argument followed by our Uselet structure. The last parameter is for injecting additional values into the context, which we did not use here.

This finalises the application logic, producing, after some styling from our favourite designers, the application as shown in Fig. 3.1.

The application is extended further for comparison with the TodoMVC application [75], an example application to showcase differences between MV* toolkits, in Sect. 6.2.

4 Uselets Concepts

With that we have seen how to build an application from Uselets, but why are things the way they are, what are the underlying concepts and the high-level model, the previous example is based on? This section will explore Uselets conceptually, its abstractions and how they can be achieved in concrete implementations, the internal workings, and the design rationale behind Uselets.

To understand the motivation behind some design decisions we first have to understand what the important aspects of a UI are though, i.e. what development of UIs should focus on. At its core a user interface serves to facilitate two things: It must provide some access to its underlying data to a user. For a developer this means the data has to be structured and then visualized. To ensure a clear separation of application logic, which includes the structuring, and the presentation, i.e. the visualisation, Uselets focus on the structuring, leaving the visualisation aspects to others by providing holes for them to fill as seen above. Now that the user has access to the data, she will want to interact with the data. This is the second thing a UI has to enable.

When developing UIs, this is exactly what a developer should focus on without having to worry about too many internal details. So, tools, such as Uselets, should abstract away low-level details and provide straight forward, boilerplate-free ways to define the structuring and interaction.

Especially decomposing a system in smaller parts easily introduces additional overhead and boilerplate, with a major aspect being the configuration and wiring of components. With Uselets being component-based, and with the intent to support simple off-the-shelf component re-use, it is important to minimize that overhead.

Taking a page from the playbook for distributed systems, this is achieved by basing the Uselet abstraction on the Actor model for distributed systems [51]. In the Actor model, a system is composed of Actors as the core primitive. An Actor is a self-contained unit and can perform three actions:

- Send messages to other actors.
- Receive such messages and perform a behaviour according to it.
- Spawn new child Actors.

Each Uselets builds on top of an Actor that:

- Interacts with other Uselets via messages.
- When receiving a message updates its data.
- After updating the data creates its internal Uselet structure, "spawning" the Uselets necessary.

So, there is a clear mapping from the actions an Actor can perform to all the actions a Uselet performs.

Uselets build on top of Actors, so it is possible to develop the UI by essentially modelling an Actor system. To remove the visualization aspect, the UIs structure is then inferred from the structure of the Actor system.

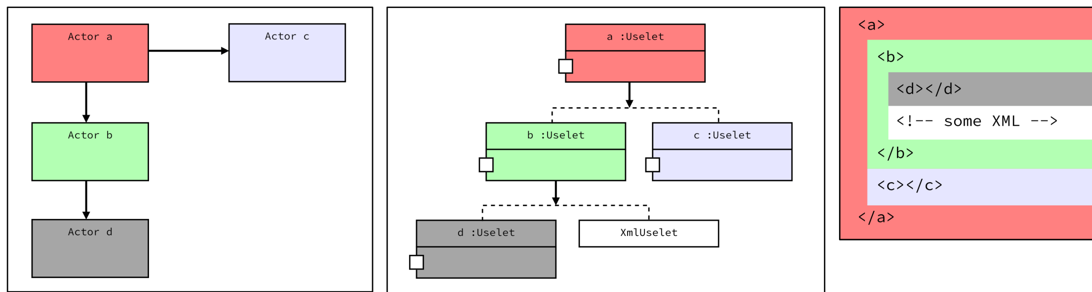


Figure 4.1: A Uselet hierarchy is built on top of an equivalent Actor hierarchy. In structure, the resulting UI is equivalent to both.

It may need mention that the visual structure should be independent from the data-structure of a UI, i.e. for web-applications, the HTML should not dictate how the application looks or vice versa. Since the inferred structure in a Uselet system is the data-structure, not the visual, this is enforced very strictly.

Arranging the Uselets in a hierarchy, a tree structure to be specific, therefore corresponds to the first part of an user interface: The structuring of data. The other aspect, the interaction, happens via messages passing. Uselets, or their underlying Actors, send each other messages, treating each other as black boxes. The fact that Uselets don't require information or make assumptions about other Uselets gives us the simple composability we desire.

This way we focus on the things that matter most: The structure of our application, via composing Uselets in a tree structure, and the interaction, represented by the messages accepted and sent.

4.1 Abstractions

The whole point of an abstraction is to hide away details that are not relevant for conceptual creation of the application. This section outlines some of the aspects of UI development that are abstracted away with Uselets. To minimize the gap between conceptual model and concrete implementation, it also gives some general guidance on the implementation of those aspects.

One of those details removed via Uselets is the actual UI. Programmers no longer manually create e.g. HTML but instead compose components with the functionality they need. This happens at every level of abstraction: At high-level components for user tasks are combined. Consider for example the task of entering the data to sell a car. This involves entering various information about the car, e.g. the VIN (Vehicle Identification Number), colour, mileage, upload images etc. These sub-tasks can be separate components which are then composed together, giving a `EnterCarInfo`-Uselet with components for each of the sub-tasks as children e.g. a `EnterVIN`-Uselet, `EnterColour`-Uselet, etc. Each of those components is built in the same way again: Composed of sub-components. Only at the lowest level of this hierarchy, holes are left for the necessary HTML provided by UI design experts.

To make this aspect of the Uselet model executable, i.e. provide a mechanism for implementation, we utilize the hierarchy inherent in XML based interfaces such as web pages: The structure of the Uselets is mirrored in the structure of the UI representation, in HTML for example via custom elements for each Uselet with the sub-components as its children (see Fig. 4.1).

With this the UI representation, i.e. the HTML, can be inferred from the modelled Uselet hierarchy, down to the holes left to fill with concrete HTML. This removes the need to manually construct the UI and then either programmatically fill it with the data or use some sort of data-binding to make the underlying data visible.

In a concrete implementation of Uselets, it becomes necessary to label the Uselets to identify them and their corresponding elements in the actual UI. The labelling must also handle the re-labelling whenever the data, and thus the structure changes.

For this reason, the initial composition of Uselets can merely be an empty husk, that has to be "filled" i.e. labelled upon system initialization. So when initializing the system, i.e. just before hooking it into the actual page, the labels are assigned by passing a namer through the composition of Uselets.

Therefore an uninitialized Uselet can be viewed as an unevaluated function that takes the namer as an argument and, when called, returns the initial UI representation, the HTML or XML, and spawns the underlying Actor that is responsible for updating it as a side-effect:

```
1  typename Uselet = (Namer) ~> (Xml, Namer)
```

Listing 4.1: A Uselet is a function that takes a *Namer* and produces the initial *Xml*, the updated namer *Namer* and *spawns* the underlying Actor as a side-effect. Effect types are omitted for clarity.

Since a Uselet system is usually not a single Uselet but the composition of multiple ones, it is necessary to return the updated namer as well, to pass it through to the next Uselet in the composed system:

```
1  (<+>) = λ u1, u2 → λ namer --spawn', spawn''-->
2        let (html', namer') = u1(namer)
3     in let (html'', namer'') = u2(namer')
4     in (html' ++ html'', namer'')
```

Listing 4.2: Combining two Uselets yields a new Uselet, i.e. a function that passes the namer through the two originals and yields their combined HTML representation.

This whole scheme is very similar to the way Formlets [24] handle naming. But while the forms used in Formlets were flat in structure, Uselets are composed in a tree-hierarchy. Consequently, the naming scheme must accommodate that.

Labelling a tree is simple via either passing the namer breadth- or depth-first through the tree. Since a Uselet-tree can change its structure due to some interaction and a resulting update of the Uselet system, this is not possible here. Thus, we use a different, hierarchical approach for Uselets.

The scheme described above, passing the namer from one Uselet to the next, is used for siblings, i.e. elements in the tree structure with the same parent. When a Uselet is named

though it maintains a “child-namer” in local scope. Such a child-namer can generate names like the one used for the parent, but all generated names are now dependent on the parents name.

This way all names are dependent on the position in the hierarchy, which means they reflect the hierarchical nature of Uselets, and are unique. Since the hierarchy is used to infer and create, and thus is equivalent to, both the UI representation, i.e. the HTML/XML, and the underlying Actor hierarchy for message passing communication, these generated names can be used as both identifier for the HTML/XML nodes and as addresses for the Actors, giving the Actors their tree structure as well.

Writing the structure once in an abstract way and creating the equivalent structure in the UI representation and Actor domain has the advantage that the system can have different views: For those creating the model, none of those implementation details matter. From the perspective of those concerned with the design of the application, one might argue that the Uselet structure *is* the HTML structure while for those that define interaction, and data manipulation, there is very little HTML or presentation involved; they see the UI primarily as an Actor system. So, it becomes possible to model the UI entirely using an existing and well established model, the Actor model, and then hand it off to be styled, yet the designers don’t have to concern themselves at all with the underlying Actors system.

But the so inferred and named HTML/XML structure is but the initial representation. It has to be updated whenever the underlying data changes, i.e. whenever a message is received. To provide a high-level updating mechanism, the scheme used in Elm [29] is adopted:

Messages are the only source of change in the system. Whenever a message is received it potentially triggers a change of the underlying data. A change of the data results in a change of the application structure and thus the UI representation. (See Fig. 4.2.)

Elm uses a virtual DOM update to transfer the changes made to the UI representation to the actual UI, an approach adopted by the current Uselets implementations as well. There may be some applications where either a virtual DOM update is not possible or a full update is preferable or sufficient. In those cases, replacing the full UI with the updated version is also quite feasible.

Uselets have the advantage, that, when a component updates, the changes to the UI are restricted to only a specific part of the DOM. Since exactly and only one Uselet is responsible for a part of the DOM with its children responsible for the sub-structure, no overlap in responsibility exists between the components. This way, the virtual DOM update, could be performed in parallel without the danger of interference. Furthermore, the part of the DOM that needs to be inspected, is comparatively small, since the virtual DOM update has to only look at the corresponding DOM elements for each Uselet, not the whole page.

This overall scheme has the advantage that there a exactly two aspects that need to be described: How the data changes and what structure its representation has. Everything else is low level detail that is hidden away, which is exactly what the abstraction should do.

Additionally, in concrete implementations those two aspects can be described in a straight forward, declarative fashion as mappings. The change of the data is a mapping from the old to the new data, based on the message. Such a mapping can be represented as a simple

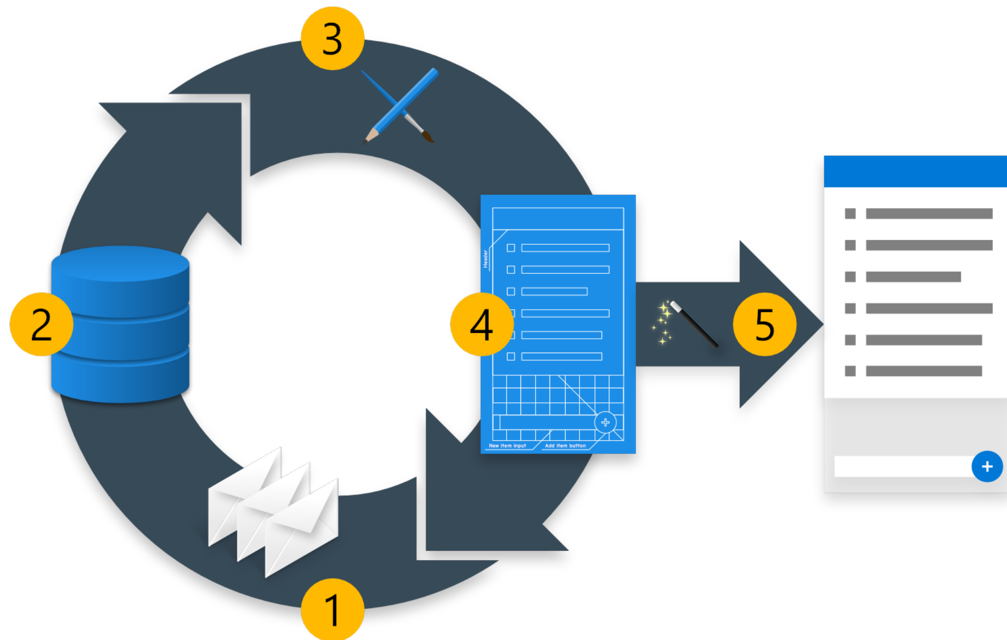


Figure 4.2: The update cycle of Uselets: When a message is received (1), the internal data (2) is updated (3) and rendered as the Uselet substructure (4) a kind of blueprint for the UI. Using virtual DOM update (5), the real UI is modified to reflect the changes.

function and is referred to as the "update-function".

The "view-function" is the mapping from a snapshot of the underlying data to a Uselet subtree. The subtree defines how the component is internally structured, i.e. how sub-components are composed. In the hierarchical modelling, the view-function produces exactly the children, i.e. the Uselets one layer lower than the current component.

Since these two mapping are sufficient to complete the update-cycle, writing a Uselet in a concrete implementation is as simple as providing those two functions. Thus, modelling the UI is just defining messages as triggers of change, outlining how the data is affected and how the changes in the data affect the sub-structures. Since these steps translate directly into a Uselet implementation, Uselets prune most of the low-level development details.

Additionally, the abstract specification of the UI can serve as documentation: With message passing as the single way for communication and change, the messages fully describe the interface of each Uselet.

This not only applies to inter-Uselet communication but to external communication as well. For example, communication with the server, in the web-context via an AJAX call, is hidden under a layer of message passing, giving a single unified interface for all communication.

If the server side uses Actors as well, this means the whole system, client, server, and potentially other clients, form a single distributed Actor system, which can be modelled in a single cohesive model. Even client to client communication via the server is then possible within this single conceptual model, it is just sending messages to yet another Actor.

External communication partners have to be made available to a Uselet though. For this a Uselet has a context, which can be extended. To extend this context implementations can use the same mechanism used for naming proposed above, i.e. when initializing the Uselets with the namer they also extend the context.

```
1  typename Uselet = (Namer, Ctx) ~> (Xml, Namer)
```

$$
\begin{array}{ll}
1 & \lambda\ u_1, u_2.\rightarrow \lambda namer, ctx^{+} \xrightarrow{\text{spawn}_1\ \text{spawn}_2} \\
2 & \quad\quad \textbf{let}\ (html',\ namer') = u_1\ namer\ ctx^{+} \\
3 & \quad \textbf{in let}\ (html'',\ namer'') = u_2\ namer'\ ctx^{+} \\
4 & \quad \textbf{in}\ (html' +\!\!+\ html'',\ namer'')
\end{array}
$$

Listing 4.3: Uselets with the option to add context are functions that takes a *Namer* and the additional context information *Ctx* and produces the initial Xml and the updated Namer, while spawning the Actor as a side-effect. The additional context has to be passed to all Uselets when combining them with the combinator, just as the namer has to be passed along.

The approach of viewing every part of the system as an Actor extends to user interaction as well: All User interaction results in a message being sent to the system, which in turn triggers the update-cycle described above.

Conceptually one might argue that the user herself is then no more than an Actor in the Actor model sense, passing messages to the system to facilitate interaction. This does however raise a question whether *conceptually* the message is then sent from the user or from the Uselet that binds the message passing. The answer has some implications for implementations: If trying to type the messages sent between user and Uselets, e.g. using session types, one would have to give a type to the user. The user is not part of the system though, so there is no point in the code, where one could annotate the users message-type. Additionally, the translation from DOM event to message has to happen somewhere within the system. This is the point where the type-annotation will have to be, but that leads to a mismatch with the conceptual view, that the user is an external Actor.

Regardless, from the perspective of a Uselet there are the following potential recipients for messages, or *acquaintances* [50], available from its context (similar to e.g. [63]):

- external Actors, e.g. the proxy-actors mentioned above
- the Uselet itself
- the Uselet's parent
- the Uselet's children
- any specific Uselet for which the address is known

In a concrete implementations Uselets have the option of directly sending messages to one of these recipients or broadcasting to all Uselets or Actors in the system. Broadcasting obviously imparts a certain overhead, since not all Uselets have behaviour for all messages but would receive them anyway. But this is necessary so that Uselets do not require manual wiring for composition or any information about their potential communication partners and environment. (There may be some mechanisms to reduce the number of messages sent, see Sect. 9.1)

The fact that a message can trigger a change in the Uselet tree is also a challenge for implementations. Sending a message to a component that is removed immediately after is undesirable. But missing sending to a recently spawned component is equally bad.

Since Uselets are hierarchical and a message can always only trigger a change in a subtree, it is sufficient though to send broadcasted messages top-down through the layers of the tree, waiting for a layer to complete its update-view-cycle. Since that means that a parent always receives, and thus updates, before its children, the subtree is already updated when the components in it receive the message.

Directed sending does not have these issues, but is mostly only relevant when a Uselet communicates within a well-known substructure, usually only sending to itself and its children.

One can argue that broadcasting is akin to *public* in an object-oriented setting, as this is what the environment sees and has access to, while direct messaging is a fairly liberal *protected*.

Regardless of whether they are broadcast or directed, in the current implementations, messages are always sent through a dispatcher (cf. [35]). This is an approach advisable for a number of reasons:

- The sending order mentioned above, parent before child, can be enforced since the dispatcher knows the structure of the system.
- It enforces unidirectional dataflow: Every message flows exactly only from source, through the dispatcher, to the target.
- This makes the dispatcher *the* starting point for debugging, since, as mentioned above, all change in the system is the result of a message and each message passes through the dispatcher.
- Implementing a dispatcher is straightforward in languages with Actor-style concurrency; it is just another Actor that forwards messages. But it is also possible in languages without Actors, e.g. JavaScript, allowing for more similar implementations and fewer differences in usage. (See Sect. 5.3).

To be reachable via the dispatcher, Uselets register with it upon instantiation. This could be used for life-cycle hooks in future iterations (see Sect. 9.1).

This also works as an alternative way of adding external communication partners. Instead of being available in the context, they register with the dispatcher with a certain address, to which messages can be sent. This enables things like e.g. service-discovery or hot-swapping at runtime or replacing them with mock-actors for testing purposes.

This is however only one messaging scheme for concrete implementations. It is equally possible to use the references to parent and children from the Uselets context to pass messages along the tree structure. This does make broadcasting more challenging though, since broadcast-messages will have to propagated through the whole tree along its structure. In general though, Uselets as a model do not make assumptions on how messages are sent.

4.2 Usage Methodology

With the abstractions provided by Uselets, building an application from them can be done by modelling the system similar to how one would model an Actor system. While, in principle, developers have all the freedom the model leaves to experiment with, certain aspects of a Uselet system can and should follow a certain methodology. This section will recommend a number of systematic approaches for common modelling and design scenarios.

From textual requirements to Uselets

The first step when modelling a Uselet-based application is to identify what components are required and how they communicate, i.e. create the parts that make up the model

A good first approach here is to identify everything present in from the requirements documentation. This should happen in a top down approach: Beginning with the individual tasks and operations that need to be performed, followed by their arrangement in sequence or parallel. Moving one layer of abstraction down, the sub-task of this first batch need to be identified following the same procedure. This process is repeated until either the model is fully complete or has reached a satisfactory level of with the remaining sub-structure are up to the developers of the concrete implementations.

In a second step the modeller can then identify which of the components have to be newly created and which can be reused, either within the project or from an external repository of components.

This approach is not specific to Uselets but can be applied to most component-based frameworks and libraries.

A bit more specific to Uselets is the modelling of the interaction. Since every interaction happens via a message, everything the requirements describe to *happen* should have a message. Again, developers may try to consolidate redundancy and overlap within the messages.

With the messages and components identified it remains to figure out which component sends which message. This should be clear from what UI element triggers the interaction. The Uselet that holds that UI element sends the message. Message may also not originate within the system but from an external source. Likewise, some messages will be sent outside the system, so their recipients need to be noted in this step as well.

Lastly it remains to define the effect of the messages in the message recipients. With the structure, the messages, the behaviour per message, and external communication partners defined, all parts of the Uselet system are then assembled.

Communication Patterns

Broadcasting the most common way for Uselets to communicate. This is because Uselets should not make assumptions about their environment so they are easily composable in any configuration.

Just like in a normal distributed system, broadcasting does incur a certain overhead though, so directly sending is also possible.

Sending messages to a Uselet directly should only be done when the type and structure of the recipient is well known. This is the case for example when a Uselet is sending messages to its children. In this scenario a directed message is a kind of “private” message, since it is only sent within the Uselet.

An alternative scenario for directed message sending is when the Uselet has previously figured out all the relevant recipients e.g. via some form of WHOIS broadcast [30].

Such a WHOIS protocol would need to be implemented either in every Uselet or in a single entity every message has to pass through, like the dispatcher above. So, this has to be built in at the framework level and is not up to individual developers.

Communication from external sources to the Uselet system is always broadcasted to all components, while messages sent from Uselets to those external communication partners is always directed.

Since Uselets can simply discard messages, it is not harmful to send messages to Uselets that do not know how to handle them. But proper recipients for the message are all those Uselets whose underlying data is affected by the message. This of course depends on which component holds the data, i.e. the data locality.

Data locality

Data locality, i.e. which data is held by which Uselet, should, in general, follow the principle of the Information Expert, i.e. information should lie exactly in that component that uses it [60].

Often times data is shared between components though, so identifying that single component that holds the data is non-trivial. Duplicating the data is one possibility. Since most messages, i.e. reasons for altering data, are broadcast, keeping the data in sync is quite feasible, though not without challenges. When the data is retrieved from a server, this is mitigated, since messages from the server, as the authoritative information source, are always broadcast. But since the data-updates have to be part of every component that holds that data it can lead to code duplication. So, having the data redundantly in the application, lying in Uselets of the same type, is no problem, but sharing across different kinds of Uselets is undesirable.

If parts of the data must be used by Uselets of different types, the data should be held in an ancestor component and passed to the children. Within the hierarchy of Uselets it should be placed at the lowest possible but highest necessary point, i.e. when multiple Uselets require access to the data, their least common ancestor should hold the data and give them access.

Access to that data can be for example by passing references, if the underlying language supports that. An alternative is the passing of closures that return the data required.

Finite State Machines

One benefit of using Actors as the underlying abstraction for Uselets is that Actor behaviour can be described as finite state machines. This does translate to Uselets as well: The

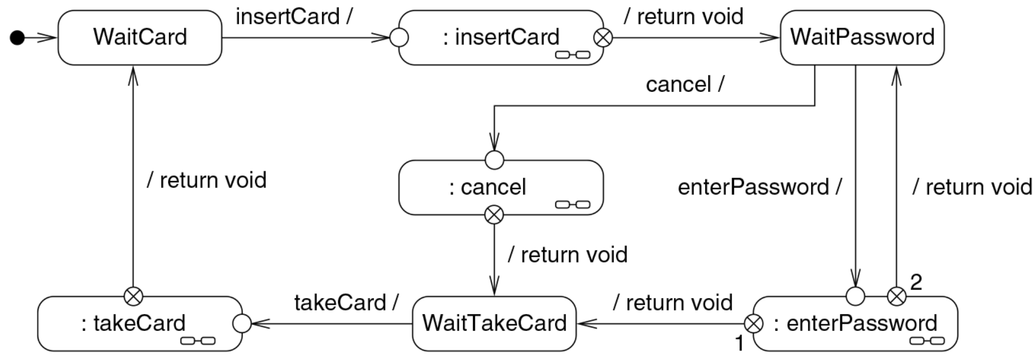


Figure 4.3: The UML state machine for the ATM example, as created in [48].

current state of the internal data can be seen as the state of the FSM and the incoming messages trigger transitions.

This means that the behaviour of an individual Uselet can be described as a single FSM, and that of multiple Uselets by having parallel FSMs. This provides an alternative model as starting point which can then be translated to the Uselet model. State machines are reasonably easy to understand and can help non-programmers grasp the working logic, which in turn can furthers customer-engagement and thus the quality and fit of the resulting software.

Consider the example of an ATM (a common example case, cf. [9, 48, 86, 90]. Its behaviour can be modelled as a Finite State Machine, as seen in Fig. 4.3. Assuming we are trying to build a single ATM Uselet, the translation from UML state machine to Uselet is straight-forward: The stable states of the state machine are introduced as a variant type or an enumeration. They are the data-model of the Uselet. Activity states are functions that, if branching, return an identifier for the next state. An alternative approach for activity states, especially if they contain external communication, is to build them as service Actor that executes the states internal behaviour when triggered via a message, i.e. the inbound transition, and responds with a message as result, i.e. the outgoing transitions of the activity state. The update function is equivalent to the transition relation in the FSM, see Code 4.4.

```
typename ATMState = [| WaitCard
                     | WaitPassword
                     | WaitTakeCard
                     | Error |];

typename ATMMessage = [| InsertCard
                       | EnterPassword
                       | Cancel
                       | TakeCard |]

var atmModel = WaitCard;

fun atmUpdateFn (ctx) (mod, msg) {
  switch ((mod, msg)) {
    case (WaitCard, InsertCard) ->
      insertCard();
      WaitPassword
    case (WaitCard, _) -> Error

    case (WaitPassword, EnterPassword) ->
      ctx.consortium ! VerifyAccount;
      AwaitVerificationResult

    case AwaitVerificationResult(nextState) ->
      if (nextState == 1)
        WaitTakeCard
      else
        WaitPassword

    case (WaitPassword, Cancel) ->
      cancel();
      WaitTakeCard
    case (WaitPassword, _) -> Error

    case (WaitTakeCard, TakeCard) ->
      takeCard();
      WaitCard
    case (WaitTakeCard, _) -> Error
  }
}
```

Listing 4.4: The ATM types, model, and update function.

Some modifications of the resulting code are sensible. The state of the Uselet can be extended to hold necessary data, e.g. the card in the ATM example. Messages can be parameterized to pass information around, the `EnterPassword` with the password for example or the `InsertCard` message with the card.

External communication should be modelled as messages passed to the external communication partner. In the ATM example an external entity “consortium” takes care of password and account verification. It can be introduced as actor available in the context. Instead of transitioning through the `enterPassword` activity state to either `WaitPassword` or `WaitTakeCard`, the FSM would send a `VerifyAccount` message and

enter the `AwaitVerificationResult` state. In this state the ATM accepts only messages with the verification result, `BadAccount` and `BadPassword`, transitioning to the next state accordingly.

The resulting Uselet models the behaviour of the ATM. It only remains to also give some method of rendering for the missing view function.

One way is to create separate Uselets for each of the UIs that should be displayed in each state, e.g. a component that requests the card, one for entering the PIN etc. and switching between those in the parent ATM component (see Code 4.5).

```
1 fun atmViewFn (ctx) (mod) {
2   switch (mod) {
3     case WaitCard -> waitCardUselet
4     case WaitPassword -> waitPasswordUselet
5     case AwaitVerificationResult -> awaitUselet
6     case WaitTakeCard -> waitTakeCardUselet
7     case Error -> errorUselet
8   }
9 }
```

Listing 4.5: The ATM view function.

Because there are translations from other specification models, e.g. interaction diagrams, to FSMs [48], this allows developers and non-developers to model their requirements in a way most suitable for them and then transform them, in one or more steps, into ready-to-use Uselets.

4.3 Key Benefits

With the concepts of Uselets outlined above, this section summarizes the key benefits arising from Uselets for developing user interfaces.

Uselets allow developing UIs in an abstract way, based on the Actor model for distributed systems. Basing the Uselet abstraction on Actors as well established foundation makes understanding them easier, provides at least some basic tools support and simplifies porting them to different languages due to the existence of Actor libraries or native language support.

Uselets serve both as a conceptual model for abstract, implementation-independent modelling, and at the same time also as implementation framework. The proximity of those two helps creating implementations that actually reflect what was modelled.

Actors in the original model were the composable building blocks of the system. Uselets too are easily composable, allowing developers to construct large applications from small, understandable parts. Not only does this break up development of such large applications into smaller work packages, which makes incremental development, and work in teams simpler. It also sets the foundation for repositories of ready-to-use components that can simple be downloaded and used.

Conceptually every part of the system, Uselets, user, and external communication partners are all viewed as Actors, giving a single unified interface for all change in the

system: Messages. Communication can then, with appropriate tooling or type systems, be encoded and statically checked.

Change in the system can be described declaratively as a mapping from old data and triggering message to new data. Again, with an appropriate tooling or types it may be possible to statically check whether this mapping is comprehensive, i.e. whether each interaction is covered.

Analogously, describing the mapping from data to the resulting substructure of the Uselet can be done declaratively, yielding a tree structure for the application. Inferring the UI representation, HTML for web-applications, eliminates the need to write UI code manually. Ultimately though, designers will have to have control over the elements on the page. This is done via explicitly placing "holes" for them to fill in the leaves of the Uselet tree, creating a strict separation of application logic and presentation. Changes to the tree are automatically propagated to the actual UI, removing a usually tedious part of development.

4.4 Visual Notation

When implementing Uselets, writing them down is trivial, in that the code of the implementation language can serve as the specification of the Uselets. Usually before implementation starts, there is a design phase though. In this phase the Uselet hierarchy should be outlined, but the language that is ultimately chosen is not set yet. This section outlines a visual notation that allows describing Uselets and their hierarchy without a specific implementation and with the option to defer certain implementation details.

Fig. 4.4 shows an example of the visual notation with the key components:

1. Uselets are denoted as boxes with two sections. The upper section contains the Uselets name. The lower section the data-type of the data-model. If a Uselet has no data, e.g. because it is static, the lower section can be omitted.
2. A Uselet has ports for incoming and sent messages. Expected messages are noted on ingoing arrows, sent messages on outgoing arrows. These arrows can be connected to the message source or target, if it is present. Notes on the message describe the behaviour of the update function for a message.
   Private messages, i.e. those sent to the component itself, are usually implementation specific and not represented. Since most public messages are broadcast, they commonly have no specified target.
3. To describe the hierarchy of Uselets, an outgoing, filled arrow points to the children of the Uselet, i.e. the result of the view function. Since those are dependent on the state of the Uselet, the outgoing arrow can branch with conditions attached to the branches.
4. Since it is not just a single child but can be multiple, composed children, multiple Uselets that are composed together to for a single descendant are connected by a dashed line.
5. (not shown) To enable conditional composition and repetition, components that are composed can be annotated with a circle and `foreach` or a square and an `if`

;

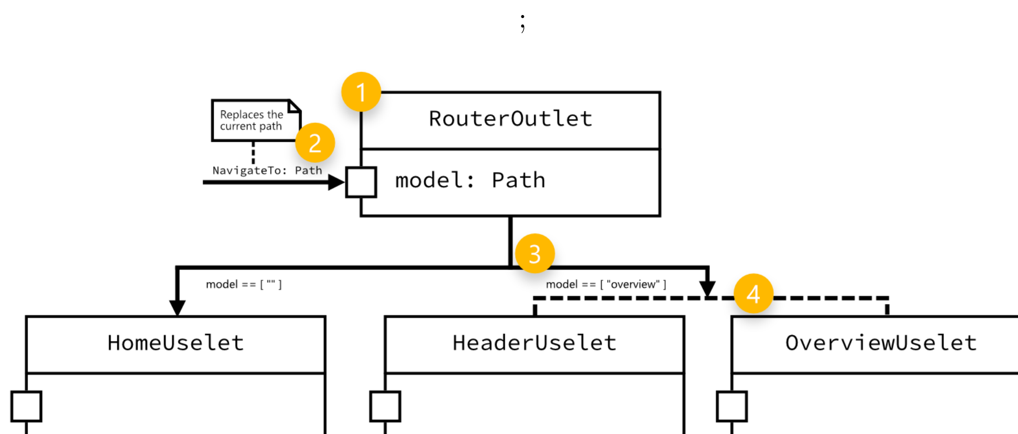


Figure 4.4: Visual notation for Uselets

statement to signify that the component should be repeated respectively added only when a condition holds.

5 Uselets Implementation

A number of details of Uselets, while outlined conceptually, remain implementation specific. The following section will describe the two provided implementations, in Links and JavaScript, showcase some of those details, and discuss the differences. It will also introduce testing and debugging of Uselets.

5.1 Uselets in Links

The original idea for Uselets came based on the challenges when developing larger user interfaces with Links. So, the first implementation also was done in Links. Consequently, many of the design decisions for Uselets were made with features of the Links language in mind. Most notably, Links has built-in Actor-style concurrency, resulting in the Actor model as the underlying model of the abstraction.

Uselets in Links are, exactly as described above, functions that take the namer and contextual information and return the initial representation and the updated namer while spawning the underlying Actor.

A namer in Links is a function that returns a name or “address”, which, in the current implementation, is but a list of integers. The hierarchical nature of the addresses is encoded in the structure of the list: The head of the list is the part that distinguishes siblings, while the tail is the address of the parent. This is fully sufficient to label the Uselet tree and makes figuring out the address of a Uselets parent trivial.

Access to the parent, children, and other associates is provided via each Uselets context. Message are always sent via a dispatcher though, as described above. It can also include additional Actors for external communication. These so called “service-Actors” act as proxies and are passed in a record, a dictionary mapping from service-name to service-Actor.

It remains the responsibility of the developer to spawn those service-Actors during system start-up and pass them to the Uselets. When this information and the namer are passed to an individual, uninitialized Uselet it:

1. Creates a new address with the namer
2. From that address creates a child-namer
3. Constructs the full context from the passed and constructed information
4. Spawns the underlying Actor with the update behaviour
5. Creates the initial UI representation
6. Returns the initial representation and the updated namer

Creating the initial representation is done using the view-function, described above, with the initial data-model. These two plus the update-function and a name are provided when creating the Uselet, i.e. when creating the function. Those parameter are passed to a factory function `uselet` that creates uninitialized Uselet instances.

```
1 uselet: (
2   String,
3   model,
4   UpdateFunction(model),
5   ViewFunction(model)
6 ) ~> Uselet
```

Listing 5.1: The `uselet` factory function creates dynamic Uselets from their name, their underlying data, an update-function and a view-function. Types are simplified, disregarding effect-types. label

Uselets created with this factory-function are the dynamic ones. Static Uselets are created with their own factory-functions, one for text and one for XML. These create empty Uselet husks, i.e. functions that too take a namer and contextual information but mostly ignore it and only pass it along when composed with other Uselets.

```
1 textUselet: (String) ~> Uselet
2 xmlUselet: (Xml) ~> Uselet
```

```
1 textUselet = λ text → λ namer, ctx —spawn→
2   (stringToXml(text), namer)
3 xmlUselet = λ xml → λ namer, ctx —spawn→
4   (xml, namer)
```

Listing 5.2: `textUselet` and `xmlUselet` both produce Uselets that ignore additional context and pass the namer through.

Composition of Uselets uses binary infix operators that take two Uselets, two functions, and creates a new one that passes along the parameters during initialization.

Along this binary composition operator there is a second infix operator to embed a Uselet in a piece of XML. This is one of the few instances where the Uselet developer can explicitly manipulate the UI representation.

This composition produces, as described above, an empty husk of uninitialized Uselets, i.e. one large function that passes parameters through the underlying tree structure. This function is called when the Uselets are embedded into a `Page`, Links type for web-pages as they are sent to the browser. This is done with the `useletPage` function, which takes a Uselet structure and additional contextual information and creates a `Page` containing the Uselet-defined application. This page can then be used in conjunction with the standard tools of the language to serve it to users. A similar function that merely produces the initial XML without embedding into a page exists also.

When the page is passed to the client, Links spawns the Actors for updating the Uselets based on the incoming messages. These Actors use the update mechanisms outline above to keep the data model and its internal structure up-to-date. The actual DOM, the HTML used by the browser, is updated using a virtual DOM library via the JavaScript FFI. The XML representation of the Uselet structure is passed to that library, which determines the necessary changes to keep the DOM in sync, and applies those.

Communication by message-passing is quite straight forward, as Uselets merely build on top of the language message-passing. Additionally, the fact that there are no bounds

between the "tiers" of the application, i.e. client-server-communication works the same as intra-client or intra-server, allows for simple external communication, by simply calling location-annotated functions.

Due to Links strong static type system with effect types, the type of the overall application is very prone to become excessively long. Individual Uselets can be described with reasonable types, by using open row types to leave some of the messages accepted unspecified: Only a number of the messages accepted by that Uselet are specified as variant type, and a wildcard signifies that other messages are also possible. But by composition these types are combined into a variant type that combines the variants from all the Uselets in the composed system. This quickly leads to large polluted types, that, while statically checked, offer only limited value to developers.

5.2 Uselets in JavaScript

Using JavaScript to build Uselets, being a very different language to Links, results in a number of differences and challenges. This section describes the provided JavaScript implementation and its idiosyncracies.

While in Links Uselets built on top of the native Actor style concurrency, JavaScript is inherently single-threaded. WebWorkers, JavaScript's concurrency-mechanism, do provide a form of concurrency with message-passing but the constraint that WebWorker threads cannot manipulate the DOM proves a serious obstacle, since each Uselet should be able to update its corresponding DOM subtree.

It thus becomes necessary to emulate the message passing. When using a dispatcher, this is straight forward enough: Upon instantiation of a Uselet it registers with the dispatcher with an address or name. When another Uselet wants to pass a message, it has to just pass it to the dispatcher. The `send` call on the dispatcher is wrapped as closure, available from the `this` reference. The dispatcher then looks up the correct recipient from the registered Uselets and passes the message to its `receive`. By wrapping all this in closures local to each Uselet, the whole mechanism of the dispatcher can be hidden away. With this the messages can be enriched further in those closures with e.g. the address of the sending Uselet for responses, without exposing these details to developers using Uselets. This is the same mechanism used in the send-functions available in the Uselets context in Links, leaving developers with an interface that looks very much the same as it does in Links (see Code 5.3).

```
1 this.send(message, recipient);
2 this.broadcast(message);
```

```
1 ctx.send(message, recipient);
2 ctx.broadcast(message);
```

Listing 5.3: Message passing in JavaScript via local closures bound to the `this`-reference (upper) and Links via the context (lower).

The definition of individual Uselets leverages JavaScript's object-orientation. Uselets written by developers inherit from a Uselet class and e.g. the contextual information of an

Uselet instance, are members of the objects, available through the `this` reference.

This gives two ways of creating new Uselets: Either extend the Uselet class in classic object-oriented fashion, with a new name, constructor etc., or call the Uselet constructor-function and create an anonymous Uselet without its own class, just as instance of the Uselet class (see Code 5.4).

Those two versions are directly analogous to the two variants in Links, creating a Uselet by calling the `uselet`-function and assigning it to a variable versus writing a function, a constructor, that internally calls the `uselet`-function, as if it were a `super`-call.

```
1  class MyUselet extends Uselet {
2    constructor () {
3      super(
4        'my-uselet',
5        /* model */,
6        /* update function */,
7        /* view function */,
8      )
9    }
10 }
```

```
1  const anonUselet = new Uselet(
2    'anon-uselet',
3    /* model */,
4    /* update function */,
5    /* view function */,
6  );
```

Listing 5.4: Named (upper) and anonymous (lower) Uselets.

The parameters for the Uselet constructor are, as in Links, described above: A string to assign a name to the Uselet, an arbitrary internal data model, an update-function that maps from the current data-model and an incoming message to the updated data-model, and lastly the view-function producing the Uselet substructure from the data-model.

Static Uselets are created with constructor functions `XmlUselet` and `TextUselet` like in Links. Since those have no dynamic elements to them, and thus the `this`-reference is of no importance, their instantiation does not require the `new`-keyword, while the dynamic ones do.

The lack of static type-checks in JavaScript allows overloading those constructors. The constructor for XML therefore accepts both a string representation that is parsed into desired XML and instances of the native JavaScript HTML-nodes. An optional parameter allows passing additional attributes, most notably event-handlers for the HTML. This is particularly relevant when the first argument is a string representation of the HTML, where it is not possible to directly write the event-handlers.

Native JavaScript nodes are used to interface with the DOM for updates. They are passed to a virtual DOM library, which, like in Links, takes care of diffing and updating of the actual DOM. Since the real DOM is available in JavaScript, there is no need to use an ID to identify the correct DOM nodes per Uselet, but instead they can be remembered by reference in the scope of a Uselet. Vice versa the Uselet object can also be made available

from the DOM-node by adding it as additional property to the node object. This eases debugging; the developer can select a DOM node using the standard tools available in JavaScript or browser tools and get access to the Uselet and its values from there.

Since JavaScript does not support custom infix operators, composition is performed by functions. The `asSiblings` function is roughly equivalent to the `<+>` operator. It composes Uselets on the same layer in the hierarchy. The main difference to the Links version again results from the dynamically typed nature of JavaScript. While in Links the composition is always binary, in JavaScript function can take an arbitrary number of arguments and composes *all* of those. So, the function called with a single Uselet is just that Uselet. When called with two Uselets it is the same as `<+>`, and when called with more arguments it is like multiple applications of `<+>`.

```
1 asSiblings(uOne) //= uOne
2 asSiblings(uOne, uTwo) //= uOne <+> uTwo
3 asSiblings(uOne, uTwo, uThree) //= uOne <+> uTwo <+> uThree
4
5 childOf(elem, uselet) //= elem +> uselet
```

Listing 5.5: Uselet composition using `asSibling` for same-level composition and `childOf` for embedding a Uselet in a HTML node.

As described in Sect. 4 and in the Links implementation, the result of Uselet composition is the uninitialized Uselet skeleton, by the same mechanism, i.e. by Uselets being functions that are not yet called. The initialization executes the naming scheme and fills in all the contextual information. Unlike the Links implementation the return value of this functions is just the generated initial HTML, but not the namer. The namer is a stateful, so since it updates its state and maintains it internally, it is not necessary to return an updated namer.

Since JavaScript does not provide Actor style concurrency, these functions of course do not spawn processes to update the DOM. This all happens, as is JavaScript's nature, asynchronously by callback, but using the very same mechanism described above: When the dispatcher signals that a message has been received the update function calculates the updated data-model, which is then passed to the view function, which produces the desired structure. This Uselet structure is passed to the external virtual DOM library, which patches the actual DOM.

Triggers for this cycle are internal or external communication. External can be a DOM Event, which usually is the result of a user interaction. Binding user interaction to message sending can be done either manually, by attaching a function that calls the `send` or `broadcast` functions or more declaratively e.g. by passing a mapping from event to message as the second parameter of the `XmlUselet` constructor.

External communication depends on the exact communication type, where again "proxy-Actors" serve to hide away the low-level details under a layer of simple message-passing. These "Actors" in JavaScript of course are no real Actors but objects with a receive-function that asynchronously perform a task, responding with a message on success.

Since the dispatcher, with the lack of a type-system, is agnostic as to what registers, these communication-proxies can either be registered with the dispatcher under some name,

or can, like in Links, be passed to the constructor as part of the additional contextual information.

5.3 Implementation Differences and Challenges

With JavaScript and Links being fundamentally different languages, there are of course a significant number of differences between the implementations. Additionally, since Uselets are primarily a concept, originally based on Links, there are a number of challenges actually implementing it in Links, but especially when porting it to a more imperative language with object-oriented elements like JavaScript. The following section explicitly lists the most important differences between the implementations, the challenges of the implementations, and some additional notes for potential further implementations in other languages.

On the surface level there are of course a number of syntactical differences. These have different origins, but usually either the lack of a language feature, like the missing infix operators in JavaScript, or the addition of a language feature that allows some things to be solved in a different, more idiomatic way, e.g. the use of object-orientation and the resulting use of `this` instead of a dedicated `ctx`. Other differences include:

- Links uses the native literal XML syntax and its `l`-attributes for interaction and messages, whereas JavaScript relies on the built-in node types and event listeners with some additional syntactic sugar.
- Messages are dedicated variant types in Links, which are not available in the dynamically typed JavaScript. Instead messages are simple objects that are dynamically checked to have the appropriate fields.

While these of course decrease the cohesion of the Uselet programming model across languages, they are rather minor. The fact that they offer familiar way of doing things even helps with the cohesion within one language.

Ultimately these syntactical differences are minor though, resulting in very similar user-code, as shown exemplary in Fig. 5.1.

Of much greater interest are the differences in the implementation of the Uselet framework, since they showcase the challenges and offer examples on how to port Uselets to even more languages.

As mentioned above the underlying concurrency model and options differ greatly between the two languages used for this thesis. Since the Uselet abstraction is based on a concurrency model, this is especially relevant. Links does offer easy to use Actor-style concurrency, while JavaScript offers almost no concurrency at all. Since those are very much the ends of the spectrum, it is fairly safe to assume that every other language will fall somewhere between those two extremes.

For most languages that support concurrency, and even some that do not or only very limited like JavaScript, Actor-libraries do exist. Building Uselets on top of those should prove to be not much different than building Uselets on top of Links.

Since the purpose of Actors is primarily to detach application logic from presentation and provide a single unified interface for communication, it is not necessary to bring the full power of Actors when they are not natively available to begin with.

```
1  uselet(
2    "todo-list",
3    [ ],
4    fun (ctx) (mod, msg) {
5      switch (msg) {
6        case AddItem(i) ->
7          i :: mod
8
9
10       case _ ->
11         mod
12     }
13   },
14   fun (ctx) (mod) {
15     fold_left(fun (a, b) { a <+> b },
16       map(fun (i) { itemUselet(i) }, mod)
17     )
18   }
19 )
```

(a) Links

```
1  new Uselet(
2    "todo-list",
3    [ ],
4    function (mod, msg) {
5      switch (msg.type) {
6        case 'ADD_ITEM':
7          const newList = mod.slice();
8          newList.push(msg.data);
9          return newList;
10       default:
11         return mod;
12     }
13   },
14   function (mod) {
15     asSiblings(
16       ...(mod.map(i => new ItemUselet(i)))
17     )
18   }
19 )
```

(b) JavaScript

Figure 5.1: Exemplary comparison of user code for the Links and JavaScript implementation.

Using the approach of a dispatcher and then even wrapping that in local functions (as seen in Code 5.3) allows having a nigh identical interface even though no real message passing happens. So, when Actors are unavailable altogether the approach used for this thesis' JavaScript implementation is a good starting point to emulate message passing. Other techniques, e.g. for JavaScript using the built-in asynchronous events, may also prove to be feasible.

This covers internal communication but message passing should act as unifying interface for external communication and user interaction as well. For external communication proxy-Actors, as described above, work for both native Actors and when Actor-like behaviour is emulated.

In both implementations, Links and JavaScript, these proxy Actors work in the same way: They accept messages, depending on the message perform an action, in this case the external communication, and potentially respond with the response from the external communication partner. Native Actors of course make the implementation easier, especially when no additional overhead is necessary to communicate across domain-bounds, e.g. client-server in Links. In this scenario, messages can be sent to those proxy-Actors, directly or via the Dispatcher, and they respond with messages in turn.

When no Actors are available, those proxy-Actors are, as described above, mere objects that provide a `receive`. This receive can be called directly with a message, e.g. when these proxies are available to the Uselets via their context, or via the dispatcher, where they register with a name. Internally they have to perform everything usually necessary to facility external communication. Especially in object-oriented languages inheritance can help hiding away these kind of details even further with a `ProxyActor` class that provides all the low-level communication details and only has to be extended with the domain-specific parts of the specialized proxy-Actor.

While the user in a Uselet application is conceptually an Actor too, this is not affected by the presence or absence of Actors in the language. What is language-specific though is the way user interaction is translated into messages. While Links uses its `l`-attributes, which is very similar to many templating languages and quite declarative, the JavaScript implementation relies on the standard way of attaching event listeners to DOM events. Since event-listeners require a bit more understanding of JavaScript, the constructor allows for an alternative way, using the attributes object to directly map from event to message sent, as described above.

There are numerous templating languages with even binding (e.g. for [76, 44]); this should be the primary mechanism to maintain declarativeness. If that is not possible, there is no harm in using the tools the language provides, like the standard JavaScript event binding. Added syntactic sugar, like the attributes object in the `XmlUselet` constructor or Links' `l:`-attributes can provide a reasonable compromise.

5.4 Testing

An important part for any software is testing it, which makes testability an important part for software models and frameworks. For Uselets there are two important testing scenarios:

Uselets in isolation and in composition.

Both are greatly simplified by having messages as the single source of change in the system: For unit test, i.e. testing a single component's behaviour in isolation, the Uselet in question can be initialised with a certain data-model, giving the pre-state. Then a message or a series of messages is sent to the Uselet after which the data-model, i.e. the post-state, can be inspected. Additionally, not only the data-model but also the resulting UI representation can be assessed whether the result is the intended one.

The second testing scenario is a form of integration-test with other Uselets and other external communication partners, like the server or other clients. Part of this testing can be done exactly like unit testing: Bring the system into a pre-state, send messages, and observe the state of the Uselets as post-state. Besides the post-state the interaction between the composed components are also of interest. Since, with a dispatcher, the dataflow is fully unidirectional through the dispatcher, observing the messages passing through there allows checking the interaction.

All these tests can happen in a pure, headless test environment: No browser is required, since the interfacing with the DOM happens by virtual DOM update, which can be omitted, while proxy-Actors for communication with external data sources can be replaced with mock-Actors.

Since messages are the single source of change in a Uselet application, there is no other way the component can be updated than as the result of a received message, making it sufficient to test the behaviour for each message or sequence of messages.

Code 5.6 shows an example of a test case using the rudimentary testing framework for the JavaScript Uselets implementation.

```
1  describe('InPlaceEdit', () => {
2
3    describe('Initialization', () => {
4      const ipe = initUselet(new InPlaceEdit('foo'));
5
6      it('should start in display mode', () => {
7        assert(ipe.model.editable).toBe(false);
8      });
9    });
10
11   describe('Interaction', () => {
12     const ipe = initUselet(new InPlaceEdit('foo'));
13
14     it('should toggle editable and non-editable', () => {
15       const msg = new ToggleEditMessage()
16
17       ipe.receive(msg);
18       assert(ipe.model.editable).toBe(true);
19
20       ipe.receive(msg);
21       assert(ipe.model.editable).toBe(false);
22     });
23   });
24 });
```

Listing 5.6: A test case for the `InPlaceEdit`-Uselet using the rudimentary JavaScript testing framework. `describe` and `it` describe the test cases, as done by some popular testing frameworks. Access to a Uselets model and receive-function is restricted to testing-mode, as is the stand-alone initialization with `initUselet`.

5.5 Debugging

If runtime errors occur regardless of testing, the messages as single source of change can improve debugging as well: Reproducing an error is as simple as replaying the sequence of messages up to the error. Storing and then replaying a sequence of messages not only makes it easy to reproduce an error, it also allows jumping to an arbitrary point in the history of the interaction that resulted in the error, allowing developers to inspect the path that led to the problem. From that it is possible to perform different interactions to determine, whether e.g. an earlier fault has repercussions for other interaction scenarios as well. This yields a time-travel-debugger (TTD) for Uselets.

Due to the distributed nature of Uselets, and the fact that a message received possibly rearranges parts of the Uselet-tree, it is unfortunately not as trivial to replay a sequence of messages, as in a system with a single message-receiving entity, like Elm [26, 28]: Unless done in the proper order, it becomes possible to send messages to entities that should not receive them, e.g. when a Uselet is created before the message propagated through the system, but it should have been after. Vice versa it is also possible to miss Uselets, e.g. when they are only created due to message-induced changes in their parent.

There are multiple approaches: Extracting the full state of the system per message and applying it again, when replaying. Or just logging the messages sent including their recipient, and, for replaying, resetting the system to its initial state and sending each message up to the desired point in the history. And of course, there is the possibility of mixing those approaches, by storing the messages plus snapshots of the system in certain intervals (say every 100 messages) to minimize the updates when replaying long message-sequences (cf. [26]).

Extracting the state of a distributed system is a well-researched topic [12, 59] and, while often still challenging, is not too difficult for Uselets, especially since the topology and every member of the system is known to the dispatcher. So, the full snapshot of the system can be constructed by the dispatcher by requesting the data of every registered Uselet before resuming message-passing. The resulting snapshot is a tree of data-models.

Re-applying those models requires some care though, since a change in a component's data leads to a change of the component-subtree. Thus, the data has to be applied top-down, from root to leaves, ensuring that each layer in the tree has completed its `update-view`-cycle before moving on to the next layer. Additionally, message passing must be put on hold for the duration of the model-application, so that no message can alter the Uselet structure while it is fed the models.

The overhead of storing the data of the full system is mitigated in part by the distinction between broadcast and single-cast messages. While for broadcast messages is necessary to

take a snapshot of the full system, every component is potentially affected by the message, for single-cast messages only the data of the exact recipient should be stored and re-applied.

Since the majority of messages is broadcast though, and each broadcast results in a full snapshot of the whole application, this approach can quickly lead to large amounts of data. The alternative approach of only storing the messages with the recipient lowers the amount of data stored, but increases the workload for replaying: The state cannot simply be applied but has to be re-constructed stepwise by applying each message up to this point, one after the other, triggering the whole update cycle for each.

To ensure the Uselets are updated correctly the messages have to be sent in a top-down approach, like the data is applied for the snapshot-method, ensuring the update cycle to be complete and child components to be spawned before messages can be sent to them. Again, since the dispatcher is aware of the application structure, this is not an issue.

The exact method for identifying the problems, when in a replayed state, then depends on the underlying language and platform. For HTML, or web-based Uselets, the developer-tools provided by modern browsers offer a good number of tools to inspect variables, the call stack etc.

A proof-of-concept debugger using the first approach exists for the JavaScript implementation: When debugging every message sent between Uselets triggers the debugger to take a snapshot of the system by extracting it from the Uselet registered with the dispatcher. Selecting one specific state to apply, forces the data-model of each registered Uselet to conform to the snapshot, triggering the update cycle for each, resulting in the whole system being in the state it was when the snapshot was taken. Due to JavaScript's single-threaded nature, this can safely happen without any message, sent in the mean-time, interfering.

An overlay interface as shown in Fig. 5.2 in the browser allows interactively stepping to specific states in the history.

The Links implementation suffers a little from its functional nature here, and the fact that the Links language is still immature. Since the behaviour of an Actor is effectively a suspended function execution, awaiting a message, and no way to extract the functions parameters during that execution exists, a debugger has no way to extract a Uselets data model. Conversely there is also no way to alter the parameters during execution, so applying a state at an arbitrary point is not possible either. A mechanism where there is a special message to force a Uselet into a specific state could be conceived, but the static check of the message's type would prevent this as well: Since the message would need to hold the desired target state, to pass it to the Uselet, its type would have to allow every possible model, i.e. unify those, which is usually not possible.

The second approach, storing only the messages, in order, with their recipients sounds more promising. Since it requires intercepting every message in the dispatcher and resetting the system to re-send a subset of messages, it would significantly increase the complexity of the dispatcher, leading to its omission from the scope of this thesis.

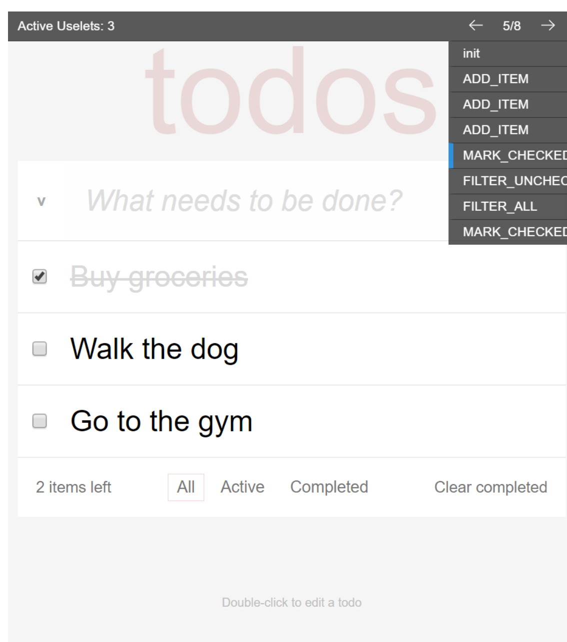


Figure 5.2: The debugger overlay with the message history on top of the TodoMVC application.

6 Case Studies

6.1 Kanban Board

To showcase a proper application built with Uselets, the following section will describe a kanban board, created with Uselets.

A kanban board is a tool frequently used for e.g. agile software development. A physical board usually consists of multiple columns marked on a surface or wall, where small cards or post-its are hung and moved around. The digital equivalent too consists of columns with text-blocks, and emulates physically moving them via drag and drop plus forms to create new cards and edit existing ones.

The combination of keyboard and mouse-based interactions in such an application covers the majority of possible user-actions for a web-application, making it quite suitable as a case-study. This provides some non-trivial interaction scenarios for, what is typically, a single-page-application.

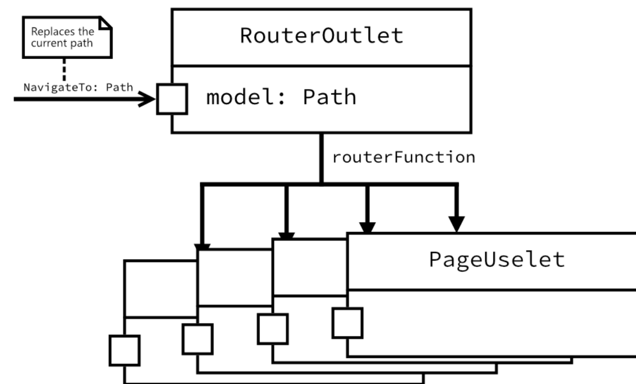
Based on the requirements for a kanban-board, the list of Uselets in this application consists of Uselets for the cards, the columns, and an overview of the available boards. To allow the adding, and editing of cards, it is extended by a component for editing text in place and a modal, i.e. a dialog box which can then hold the form for creating a new card.

Since, in a traditional webpage, the overview and each individual board are separate pages, Uselets will have to support multi-page applications. This can be achieved with the history API [17]. In the abstract hierarchy (Fig. 6.1), those individual pages are all on the same layer, children of a Uselet that switches between the "pages". The `router-outlet` Uselet maintains the current path as model, rendering the view according to a routing-function, i.e. a mapping from path to view. Instead of navigating via Links, navigation is a message to that component, resulting in a change of the path. Via the history API this path-change can be pushed to the browser history and vice versa, changes to the path can trigger messages. For the end-user it looks like she is transitioning between pages, although it is just messages sent to the `router-outlet` component and an update of the UI representation based on the routing-function.

Of those pages, the overview page is the simplest one. It is just a list of available boards with the option to add a new one. All this can be achieved using standard Links and loops in the XML literal syntax. With Uselets it is a list-component, rendering a repetition of content, in this case a links to the individual boards. Such a list component is a common UI element and it is not unlikely that different flavours of lists would be available pre-made.

The pages for the individual boards are more complex. They consist of column-Uselets which in turn are composed of Uselets for each card, which contain in-place-edit Uselets to enable editing of the card text. The static composition is still fairly trivial; at the highest level is a list of columns, i.e. re-use of the list-component. Within the columns are lists of cards. Getting that to look like columns and cards is not part of the Uselet-based development but is delegated to the designers.

With the general structure layed out, the Uselets now have to send messages to facilitate



(a) Abstract hierarchy of the router. The current path is the model, which is altered by incoming `NavigateTo` messages. The children of the router are the individual pages, selected by the `routerFunction`.

```
1  fun routerOutlet (routingFunction, initialPath) {
2    uselet(
3      "router-outlet",
4      splitAt('/', initialPath),
5      fun (ctx) (mod, msg) {
6        switch (msg) {
7          case NavigateTo(path) ->
8            historyPushState((), "", path);
9            splitAt('/', path)
10         case _ -> mod
11       }
12     },
13     routingFunction
14   )
15 }
```

(b) A concrete Links implementation. The model is a string array of URL sections, the update-function consumes a `NavigateTo` message for navigation, the view-function uses the provided mapping from path to Uselet. `historyPushState` uses the JavaScript FFI to modify the browser history.

Figure 6.1: The `router-outlet` component for single-page routing

interaction. As common for single-page applications, the form for creating new cards is inside a modal, a pop-up dialog box. This is a good example to show the difference between broadcasting and direct message sending: Opening and closing of the modal happens, of course, via a message. This message cannot be broadcast, since there may be multiple modals on the page, which would all react to the message. Since the message only affects the modal itself, it can be sent via direct-sending back to the Uselet itself, triggering the desired behaviour without affecting other modals.

When submitting data with the form in the modal, the data is sent to some external API service. The response from that API service is broadcast, so that every component that relies on data from that API can update accordingly without the need for some subscription mechanism. Removing happens analogously.

Editing in place is very similar to use of a modal: A private message has to be sent to the component for in-place editing to toggle between editing and display mode. When committing an edit, the change to the data has to be sent to an API again, with the response broadcast.

Drag and drop functionality, while widely supported in the JavaScript world with a dedicated API, is still somewhat challenging in Links. The common way is to store the element that is being dragged when the mouse is pressed, and switch it with or insert it before the element under the cursor when the mouse is released. This can be used for Uselets as well, but it is not necessary to rely on the DOM elements. Instead we can use e.g. the index or an ID of the elements dragged and dropped on. Since not at all times an element is dragged, the model has an option datatype for storing the dragged element, containing `Nothing` when no interaction takes place and `Just` the id or index when dragging. With the use of the history API the dragging and dropped element, index or ID could be made available from the triggering DOM event. Then there is no need to pollute the data-model with interaction specific data.

This, and some static content, yields the full hierarchy as seen in Fig. 6.2, and encoded in Links the kanban application. With some styling, this gives us a proper web app application as seen in Fig. 6.3.

Generally, the resulting application performs as specified. There are some quirks in its behaviour though. For example, when entering text into an input and then triggering a message elsewhere, it is possible for the input to lose focus when it it part of the Uselet sub-tree that is restructured as part of the update. The virtual DOM update cannot always just adjust the attributes and properties of the input, resulting in a re-creating. The newly created input then does not have the focus.

When extending drag-and-drop to update a cards position while it is being dragged, it may end up somewhere, where it should not be. This happens when dragging and dropping cards very quickly: JavaScript's asynchronous nature can have the events and thus the messages not necessarily in the order one would expect, resulting in the odd positioning of cards during dragging.

But with use of the proper APIs for drag-and-drop and future improvements to the DOM-update-algorithm, those are but temporary challenges.

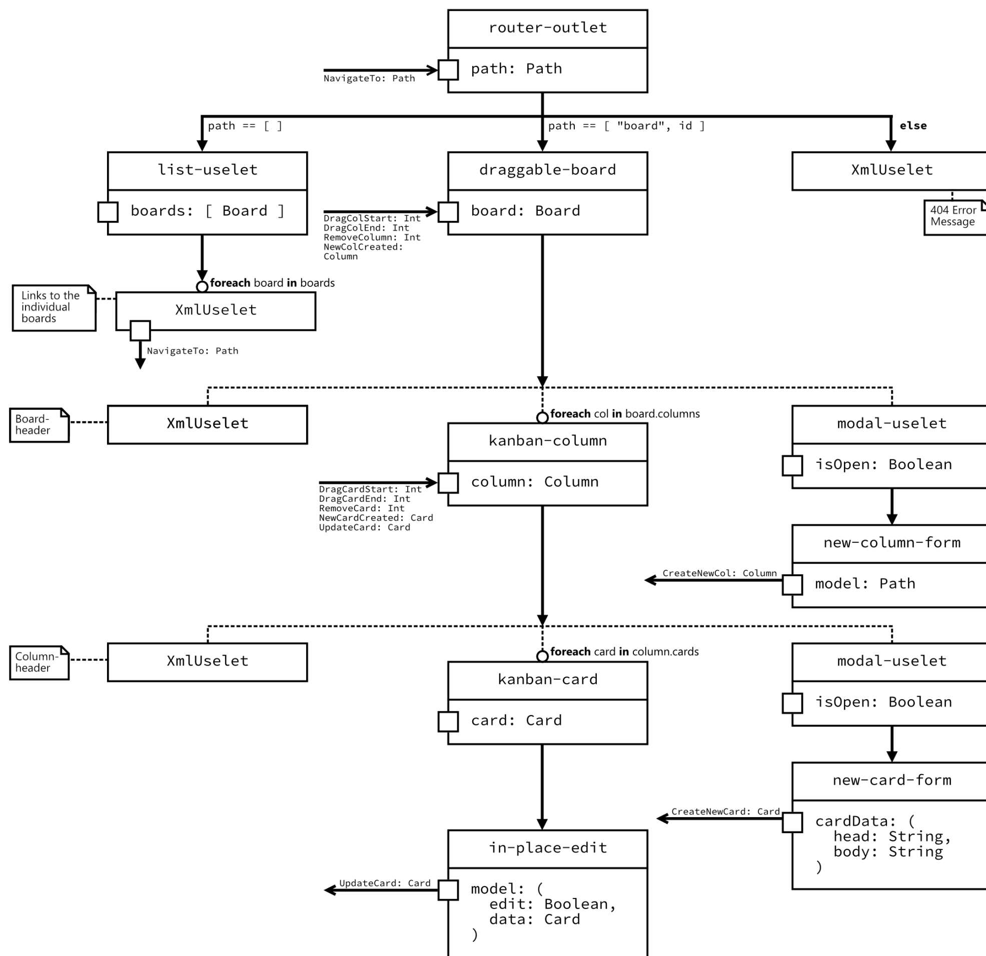


Figure 6.2: The overall hierarchy of the Uselets in the kanban application.

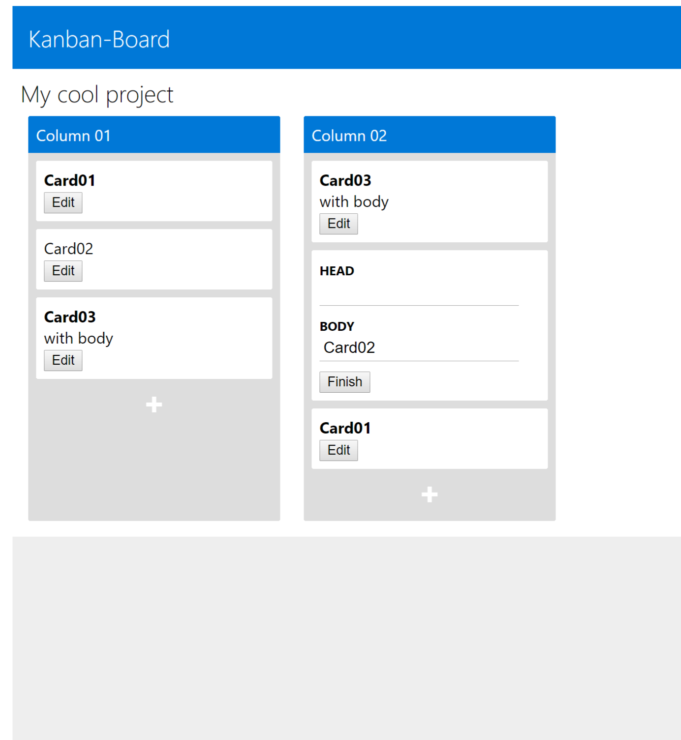


Figure 6.3: The kanban application after styling.

6.2 TodoMVC

This second case study focuses on comparing development with Uselets to the use of other frameworks and libraries. It does that by implementing TodoMVC [75], a todo-application with exactly this goal: Comparing different implementations using the same application. This application uses the JavaScript Uselets-implementation for better comparability with the frameworks and libraries TodoMVC has been implemented in before.

The application itself is a basic todo-app, not unlike the one built as the introducing example 3. Next to the functionality built, i.e. adding and removing items, we have to add some more features to create a proper copy of TodoMVC:

- Todo-items exist in both checked and unchecked state and can be toggled between those.
- Additionally, all items can be toggled at the same time. If one or more items are not checked, all items are marked as checked. If all items are currently checked, they are all unchecked.
- The content of an item can be edited in-place: Double clicking it replaces the displayed text with a text field that contains the list item. The text can then be edited and the changes are committed with the return key.
- There is an indicator for how many of the items are not yet marked completed. The indicator distinguishes between one “item” and multiple “items”.
- List items can be filtered by their status: Either all items are displayed, just those

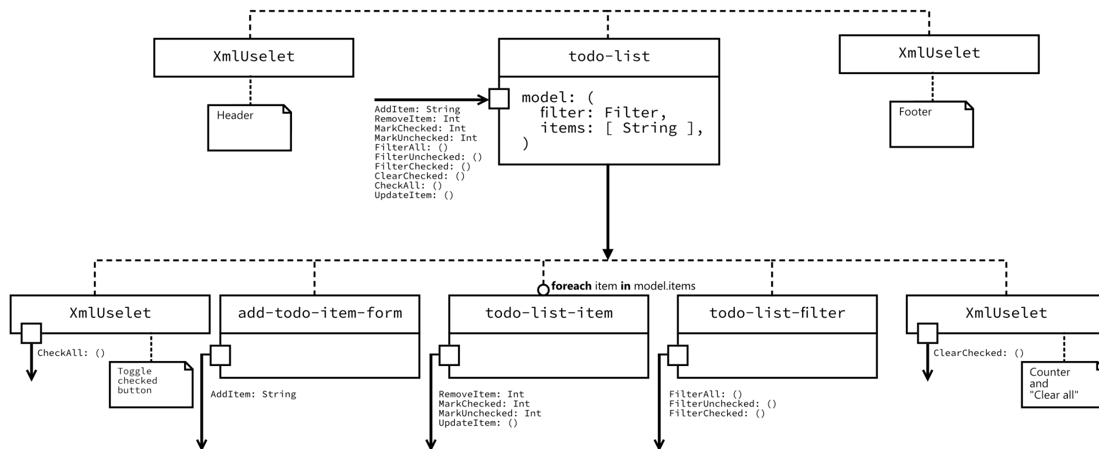


Figure 6.4: The Uselet hierarchy for the TodoMVC application.

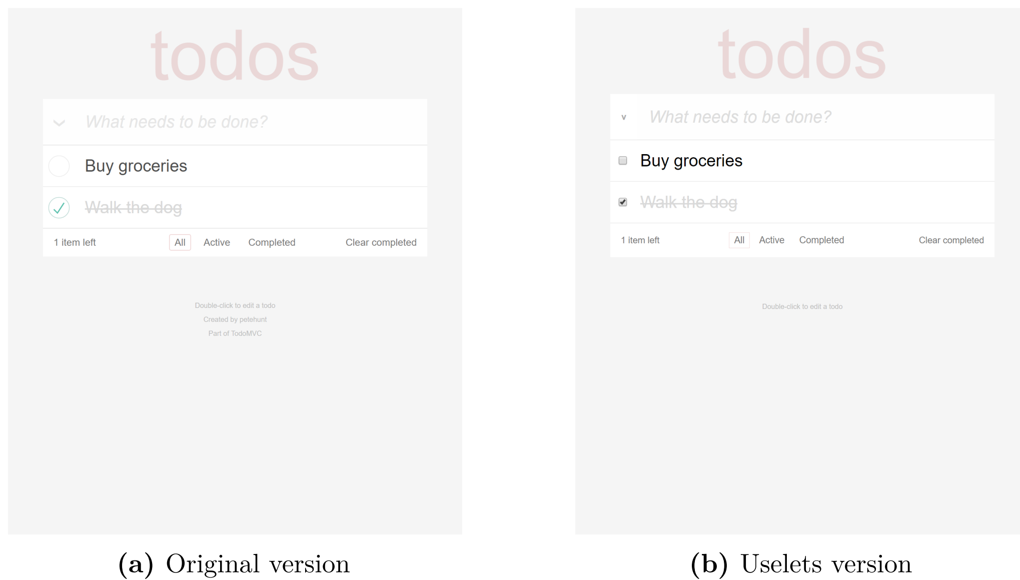
checked or just those unchecked.
- All checked items can be removed at once.

Based on this description we can now extend our previous list of necessary components. For the full TodoMVC application we then need the following components, arranged as shown in Fig. 6.4:

- The todo-list component as collective parent
- A form for adding items
- The component for the individual list items
- A component for editing text in-place
- The filter selector
- An indicator for the number of unchecked items

The added interaction options of course also mean additional messages, yielding the following list of messages and interactions for the full TodoMVC. (Some of the messages could be parameterized to consolidate multiple messages into one. To maintain a direct mapping from requirements and interactions to messages, this is not done here.)

- `AddItem` for adding new items to the list
- `RemoveItem` for removing items
- `MarkChecked` to mark an item as completed
- `MarkUnchecked` to unmark an item
- `FilterAll` To set the filter to show all items
- `FilterChecked` To set the filter to show only checked items
- `FilterUnchecked` To set the filter to show only unchecked items
- `RemoveAllChecked` clears all items that are currently marked completed
- `ToggleAll` toggles the completion status of all items



(a) Original version **(b)** Uselets version

Figure 6.5: Comparison of the TodoMVC implementation using Uselets with the original TodoMVC version.

- `ToggleEdit` switches between editing a text in-place and just displaying it
- `UpdateItem` commits the changes to an item

In the lists of components and interactions we can already find a component that we have previously written, namely the one for editing a text in place. In larger project reuse of components can be even broader, with common interaction and UI patterns, e.g. draggable lists, navigation-drawers, menu-bars, etc.

Implementing all the components that cannot be found in previous projects or repositories, we can compose the TodoMVC application, wiring together all communication.

Ultimately, this yields an application that provides the same functionality and, some minor graphic differences aside, is visually nigh identical to the original TodoMVC application (see Fig. 6.5).

But while the appearance and functionality is the same as the many existing versions, some fundamental differences can be found. This starts with the amount of source code necessary for writing the application, as seen in Tab. 6.1 [1]. The majority of inspected implementation requires developers to write more code to achieve the same functionality.

Most notable exceptions are the version using jQuery, which has the disadvantage though that it does not abstract and developers must write e.g. UI updates by hand. The second exception is Angular2, which is written in TypeScript. Both, TypeScript code and generated JavaScript, are shorter than the version using Uselets. For the JavaScript code this is in part because, as generated code, it is written very compact but unfortunately not very legible. Angular2, and some other frameworks, require a build system though, which developers not only have to install but also to configure, adding a considerable overhead.

[1] As measured by CLOC `http://cloc.sourceforge.net`

	LOC
JavaScript	241
JavaScript ES6	347
jQuery	171
Angular	256
Polymer	383
React	379
Angular2	112
Closure	631
Elm	434
Uselets	247

Table 6.1: Number of source lines of code to build the TodoMVC application for plain JavaScript, some popular frameworks, and some transpiled languages.

The Uselet implementation on the other hand is just the Uselet library code, the components, and a few lines to set up the application. All this can be imported either via the traditional script-tag or the recent addition of HTML imports [18]. This means Uselets have less setup overhead than transpiled languages, provide higher abstraction than the vanilla implementations, and needs less code than other comparable frameworks.

The Uselet code is exactly the components and messages we deduced from the requirements. The messages do take up more space than absolutely necessary, since there are dedicated constructors for them, although they could be created and used in literal object notation. Some of the transpiled languages have the advantages of more concise syntax, e.g. infix operators or a short notation for UI elements, which, when applied to Uselets, could reduce the amount of code even further. The only line that is not an Uselet or a message is the one hooking the Uselets into the page.

In terms of generated UI code, Uselets have the advantage over the competitors that it relies on custom elements, which gives the HTML elements semantic information without polluting the markup with classes etc. This also affects the stylesheet, although not significantly: Usually the HTML has to contain a number of classes to serve as selectors for the CSS, with some shallow nesting. Tag names could be used too, but they are often ambiguous, hard to understand, and complex nesting is necessary to select the correct elements. Since Uselets use custom elements, the tag names alone can serve as selectors, e.g. the selector `todo-list-item` vs. `.todo-list li` used in other implementations. Length-wise this changes little — at least for this example. When this replaces deeper nestings, it should positively affect legibility and understandability though.

Those two case studies show that implementing web-applications using Uselets is feasible. Not every form of interaction may yet be fully identical to the native APIs, but they are possible. Rich interactive applications can be written just as well as more traditional web-pages by emulating routing. While other frameworks may do certain things better, the low-overhead, highly abstracted nature of Uselets makes them a viable alternative to some existing tools.

7 Evaluation

Multiple criteria have been described in the literature to compare and assess the quality of models and abstractions.

Uselets specify the behaviour of the application, so Meyers “seven sins of the specifier” [65] or a version adapted to modelling tools should therefore be avoided. Ideally a model is built in a way that sins like those are inherently prevented. Tab. 7.1 lists the sins and whether they apply to Uselets.

The process of modelling a system first in an abstract fashion before a concrete implementation is, while widely accepted as good practice. It has spawned a whole branch of model-driven development processes. Selic lists a number of criteria for models and abstractions used for such processes, that are necessary for their success. (See Tab. 7.2.)

Yet, model-driven development and other processes using high-level abstractions area not very widely adopted. Some [84, 70, 87] have voiced criticism of some properties of modelling tools that may limited their success. Since Uselets as an abstract view of an interface serves that purpose, modelling the UI abstractly before implementing it, Tab. 7.3 lists some of the criticism voiced and how Uselets attempt to improve that.

Original formulation	Do Uselets suffer from this sin and if not, how do they prevent it?
Noise: The presence in the text of an element that does not carry information relevant to any feature of the problem.	Uselets model the interface and application logic with exactly the important features: The interaction via message passing and the structure of the application based on the underlying data. Thus, noise is kept to a minimum.
Silence: The existence of a feature of the problem that is not covered by any element of the text.	Based on the expressiveness of the Actor model, interaction should be fully covered. Since the UI representation is implicit in the Uselet structure, some visual aspects may be challenging to implement.
Overspecification: The presence in the text of an element that corresponds not to a feature of the problem but to features of a possible solution.	Uselets are meant as tool for creating solutions, so this is deliberately the case.
Contradiction: The presence in the text of two or more elements that define a feature of the system in an incompatible way.	This can occur due to name clashes, i.e. when two Uselets are specified using the same name but with different behaviour or two messages carrying different data.
Ambiguity: The presence in the text of an element that makes it possible to interpret a feature of the problem in at least two different ways.	Since messages are defined by every developer for their own Uselet and no name-spacing or similar mechanism exists yet, name-clashes can occur when two messages have the same name, leading to ambiguity and unexpected behaviour. Static checking of message types in e.g. Links prevents them from carrying different data, but this can still lead to unexpected behaviour, when one component binds a message to some user interaction that triggers multiple effects across the application. The same thing can happen with messages from external communication partners. As mentioned, a mechanism like name-spacing messages, encoding the senders address and checking it, etc. could prevent this in future iterations.
Forward reference: The presence in the text of an element that uses features of the problem not defined until later in the text.	Uselets in no way enforce the order in which components are defined, so forward referencing is possible. Since Uselets are meant to be self-contained, their description too should be self-contained. So, the description of a Uselet may forward reference other components for outside communication but does not functionally rely on their presence. Since communication partners are treated as black-boxes, for the reader a forward reference makes no significant difference, since the referenced Uselet can be viewed as a black box, too.
Wishful thinking: The presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature.	Uselets are a model that can immediately be translated into executable code. Unless the model defines impossible side-effects, everything described by Uselets is implementable.

Table 7.1: Meyers “seven sins of the specifier” for formal specification [65] and how they apply to Uselets or how the model inherently prevents them.

Abstraction	As described in Sect 4.1, Uselets abstract away a considerable amount of implementation details, allowing developers to focus on the two main aspects of the user interface, interaction and structure.
Understandability	Describing interaction between elements in the software system as message passing should appeal to the intuition of many, since it is a concept with real-world equivalents, e.g. sending mail. This should improve understandability over some more complex communication concepts like asynchronous HTTP requests, even-handling etc. In terms of the UI representation, Uselets have developer-defined names which should reflect the purpose of a component. This makes it easier to understand the purpose of a component compared to some of the default HTML elements that are generated from that.
Accuracy	Since Uselets are directly translated into code the system performs exactly as modelled. The structure of the model is reflected in the structure of the UI representation and message passing for interaction is used in the implementation as well.
Predictiveness	Messages are the only change in the system, both in the model and the resulting implementation, and the mapping of message to behaviour is an integral part of the model, so a Uselet is quite predictable. Model checking or similar verification methods (see Sect. 9.1) can even automate this.
Inexpensive	Because Uselets can immediately be translated into a working implementation - Uselets really are an nigh executable model - and the model strips away many implementation details, the workload should be significantly lower when using Uselets, although of course the time savings vary from project to project. Especially when developing cross platform, and each platform can deal with Uselets, each platform after the first only provides only little work, since porting Uselets is easy.
Observability	Creating an implementation that behaves exactly like the model makes it straight forward to observe modelled behaviour in the implementation. For Uselets messages as the single source of change are the items of interest, and a debugger that traces the messages, like the TTD provided (see Sec. 5.5) help with that.

Table 7.2: Some Quality criteria for software models [82] and how Uselets comply with them.

High Threshold: The modeller has to learn a new language of method of specification.	While Uselets can be used for fully abstract specification, they can also be used for concrete implementation, evident in the provided implementations and examples. Assuming a Uselet implementation is available in a language known to the developer, there is only a very little learning-overhead.
Low Ceiling: The modelling system is limited or conventional methods produce better results	With the strict enforcement of separation of presentation and logic some of the generated UI code may prove more challenging to style than code written specifically for a design. So, with respect to rendering, Uselets are constraining. By basing communication on the Actor model with its expressiveness, in terms of interaction Uselets should not prove constraining though.
Unpredictability: The connection between model and actual result is difficult to understand and thus unpredictable.	Since Uselets are designed as a highly executable model, the resulting implementation behaves exactly as modelled.
Lack of propagation of modification: Changes of the model are not propagated to the system or vice versa.	This aspect is dependent on the method used to create the Uselets and the tool-support. Assuming a dedicated, e.g. graphic, modelling tool with code generation for Uselets, abstract modelling is made simple but the synchronization is up to that tool. On the other hand, if Uselets are specified directly in the target language, the result is more platform specific and less easy to migrate but every change is immediately synched, since the written specification *is* the application as executable model.
System dependent and private models: The models and their tools are proprietary or in other ways restricting.	Uselets are based on the Actor model as a well-established model. The provided implementations and the languages they are built in are publicly available. Since currently no dedicated tools exists for Uselets, no issues with closed or proprietary systems exist.

Table 7.3: Common criticism of models and their tools, as summarized in [87]

8 Related Work

Uselets are far from the first approach to tackle all the challenges of UI development in general and the web- or JavaScript based development in particular. In fact, *especially* for web-based UIs, the number of frameworks, libraries, transpiled languages etc. is tremendous.

They range from simple libraries, full frameworks (e.g. [76, 44, 36]) to transpilers for either existing or completely new languages (e.g. [29, 31, 66]).

There may be all kinds of reasons for this, but one certainly is the complexity of modern web applications; JavaScript as the web's scripting language certainly was not intended for many of the things it is used for today. Especially for DOM manipulation a myriad of tools exists to hide away the imperative, low-level tools available in the language.

Another aspect of vanilla JavaScript frequently addressed by libraries and frameworks is external communication, where the built-in tools require a good deal of boilerplate to set up and use.

Both aspects have found their way into the Uselet-abstraction, with communication being unified as message passing and the DOM manipulation being hidden away under a virtual DOM.

Comparing Uselets with some of those existing tools shows a number of differences, e.g. the size and overhead of their setup, ranging from simply linking some script to the page (e.g. [76]) using the existing tools, to complete build setups with compilers or transpilers, that allow more domain-specific tools and languages (e.g. [44, 36]). The former being simple, while the latter usually requires a larger amount of configuration. Uselets fall on the lower end of the spectrum requiring only a few imports [18] in both implementations, keeping the setup overhead low.

With respect to their level of abstraction, there is also the full spectrum, some maintaining low-level details (e.g. [44]) while others are considerably more abstract (e.g. [29]), with Uselets of course being highly abstracted.

Especially the younger tools adopt a component-based approach ([44, 36, 34]), more or less following the W3Cs custom-element proposal [16].

Some of these frameworks take a fairly light approach to interaction and communication, providing simple wrappers for the common communication patterns, e.g. HTTP requests, and a list of options for communication between components (e.g. [44]). Uselets on the other hand require that all communication happens via message passing, making it a somewhat more unified interface for developers and abstracting away more details of the communication aspect. This way, the developer does not have to choose from a myriad of communication patterns. The greater control of distinct communication methods is pushed towards the translation from the different sources to messages in a Uselet system. Since the proxy-actors, that perform this translation, can be imported or customized, the control is not lost, but merely hidden away.

For facilitating user-interaction the majority of tools uses templating languages (e.g. [44, 36] etc.), where function-calls are bound to events in, usually, some HTML-like syntax.

This is effectively the same mechanism as Links' '`l:`'-attributes, so an interesting aspect for synergy with Uselets: The application logic and structure are defined with Uselets and then the interaction, i.e. the message sending, is bound to DOM elements in one of these templating languages. It remains to see in future work, how easily Uselets can be adapted to use pre-existing external templating systems.

Similar to binding events, those templating languages use data-binding to bring the values from the application logic into the UI. Links, and with that the Uselet implementation uni-directionally uses this mechanism too. Data-binding makes it easy to understand the relationship between underlying data and actual UI. However, two-way data-binding i.e. when user interaction is propagated to the application logic via data-binding as well, often makes the application more complex since the source of change is not immediately apparent.

An alternative approach alleviating that, namely updating a virtual DOM and then patching the actual one, as also used by Uselets, has been popularized by React [36]. Reacts flux architecture [42] also uses a Dispatcher to enforce unidirectional dataflow to get the benefits described above (in section 4.1). In the flux architecture, the data updated as the result of a message from the dispatcher is in so called "Stores" though, which are separate from the actual UI components. This has the benefit that multiple components can share the same *Store* but imposes a certain overhead since components and *Stores* have to be connected.

By removing the writing of HTML or HTML-like syntax from everything except `xmlUselets`, Uselets aim to prevent developers from adapting the application code in a way that eases presentation or vice versa. This enforces the separation of logic and presentation.

Packing data-model, data-update, and UI-generation into one package is used most prominently by Elm [29], a web-language transpiled into JavaScript. Here an application is roughly like a single Uselet, with data-model, update-, and view-function. In fact, it served as one of the major inspirations for Uselets. In Elm the model is one large piece of data for the whole application though. Composability is delegated to "reusable views" [27] which are a kind of helper function that extracts certain aspects of the view function. This however means that composability is exclusive to the view. The model and update function still have to be written and updated whenever the view is altered, manually or by composition. This also means that some knowledge of the internals of the reused view is required, although the presence of a static type system does help. The theoretical composability for the data-model is impaired by the fact, that the whole application always has only one model, so every sub-model has to be integrated at top-level. It then also has to be de-structured again to properly update it and to pass it to the correct part of the view-function. Again, knowledge of the whole data-model is required.

Being built for web-apps, and thus on top of the single-threaded JavaScript, does make it challenging to translate Uselets to Elm. In the opposite direction, with the similarity between their internal logic though it is reasonable to see a Uselet system as the composition of many small Elm UIs that communicate via message passing.

Evidently Uselets, each of the frameworks and tools outlined above, and many more, have different focus. For Uselets the focus is very much on a component-based approach

with little overhead. And as with many tools and abstractions, this comes with certain trade-offs, for Uselets most notably the fact that modelling the UI strictly as the Actor system, it prevents the developer from arbitrarily adding parts that might make certain visualisations easier.

This of course is a deliberate design choice to enforce strict separation of logic and presentation. But that means that it is important to consciously choose what tool to use, and to be aware of the trade-offs of all of these.

Uselets in particular, with message-passing as single, unified communication interface, and the simple composability, should be considered for larger systems, that heavily rely on communication, internal and external, and that have a lot of re-use potential.

9 Conclusion and Future Work

9.1 Future Work

Based on the implementations outlined above, there are a number of considerations to further improve Uselets.

Life-cycle hooks (cf. [45]) for e.g. initialization, removal etc. could give developers the tools to perform additional initialization or clean-up tasks. To maintain message-passing as the only source of change in the system, this would mean that at those points in a Uselets life-cycle it would receive messages to indicate e.g. initialization or removal, using the standard mechanism to react to them.

The current implementation of the message passing requires every component to potentially accept every message. Composing Uselets consequently quickly leads to very large, polluted types. If it were possible to know at runtime whether a Uselet accepts a message, e.g. via dynamic typing and pattern matching on the type or some form of a "canReceive" function (similar to "canEqual" [73]), it would be possible to send messages exactly and only to those that can deal with them (cf. [54]). This would greatly reduce the type pollution mentioned above and the number of messages sent.

Because this would mean that the type of a Uselet now holds only exactly the messages the Uselet accepts and sends, it would pose the, foremost philosophical, question, whether the Uselet that facilitates user interaction is the entity that sends the resulting message, or whether the user is a separate entity with its own type. The second view of course poses the challenge of assigning a type to the User, an abstract entity outside the system.

This can be even more challenging, when describing Uselet interaction including the user interaction with a type system like session types. This of course would give greater control over interactions, their order and interplay, to the developer. But assigning a session type to a user, one of the prime sources of non-determinism, would realistically degenerate into a large choice of all possible interactions, negating most of the benefit of session types. The benefit for inter-uselet and external communication remains though: Messages would only be sent to communication partners that *currently* know how to handle them.

These are all improvements for the concrete implementations. Uselets are first and foremost an abstract model though. To help with abstract modelling, tool support should be improved. Specification tools for creating Uselets independent of every underlying language are necessary, especially if they could be based on existing modelling languages like the UML [72].

Algorithms for transforming other models, e.g. FSMs into Uselets, and then tools that can apply such transformations, would also help, since those creating UIs would not need to learn a new formalism or language and could instead rely on the models they already know. Vice versa, algorithms and tools for transforming Uselets into other models can also help embedding Uselets into the context of model-driven development and the eco-system of existing standards, models, and meta-models.

Other applications

While web apps are popular, the web certainly is not the only platform out there. While there is much discussion, whether mobile, web, desktop etc. will dominate the future application landscape [1, 10, 11, 15, 64] it certainly is a good idea to support other platforms as well. This means porting the Uselet abstraction to more languages and other types of UI representation, such as other XML based ones, object-oriented UIs etc., or even to completely different UI concepts like 3D environments.

Other XML based user interfaces representations, e.g. FXML, XAML or SVG can be created just like the HTML-based Uselets described in this thesis. Some aspects will have to be changed, e.g. the virtual DOM based update mechanism. In many cases instead of creating a diff and making only the necessary changes to bring the UI in the desired shape, replacing the elements is sufficient. Removing and inserting UI elements programmatically is functionality that should be available for all those XML based UI representations.

Another application that is also often XML based are scene graphs, which are the foundation for many 3D applications. Here it is common practice to frequently redraw the scene to show changes in the graph. A restructuring of the Uselet hierarchy and thus the graph as corresponding representation due to a message and data-update is therefore rendered anyway, without any additional work. Using Uselets for 3D application therefore seems feasible enabling even more exotic user interfaces.

Ideally this leads to the point where tool support exists for either migrating Uselets from one UI representation to another and from one language to another or for generating a platform-independent model and generating the concrete implementations from that. This would greatly reduce the work required for developing one application for multiple platforms. It would also help with maintenance of such cross-platform applications, since changes would only need to be made to the one original version which could then propagate them to its derivates.

Verification

In the context of Uselets it might yet be possible to make certain assertions about User interaction: All change in a Uselet system, including all user interaction, is the result of a trace of messages. With that it may very well be possible to transform a Uselet- and thus Actor-based UI into a format where standard model-checking techniques can be applied ([5, 13]) to make assertions about the UI. Transformation from Actor-based concurrency to model-checking-compatible format have been described in the literature [32, 33, 55, 61] and Uselets sit merely on top of an Actor system.

Properties of interest for the versification can be about the underlying data and the messages, giving other developers using a Uselet some guarantees about the behaviour. Alternatively, with an appropriate way to make propositions about visual elements, they could make assertions about actual UI elements, interactions, and thus UX properties, e.g. for navigation “The user always has a way to transition back to the start-screen” ($AG\ EF\ screen = start$). Since the user in a Uselet is considered just another Actor that sends a finite set of messages to the system, we could — in a way — model check the user

interaction with the system.

Additionally, as described above, Uselet behaviour can be described and modelled state machine. Integration with other models, like navigation structure and business processes, with appropriate model transformation [58], gives a more comprehensive view of the whole application, which can, again, be model checked.

9.2 Conclusion

In this thesis I have introduced Uselets, an abstract model for web-based user interfaces based on the Actor model. Uselets describe easily composable user interface components. This eases development of large applications by breaking them down into smaller parts. Such components could potentially be available off-the-shelf in repositories, reducing development effort to finding fitting ones and composing them.

The simple composability is the result of each component being self-contained consisting of the underlying data, a way to update the data, and to produce the structure of the user interface.

The UI structure is inferred from the composition of Uselets, removing implementation- and rendering-details of the UI representation from the model. The resulting UI is equivalent to the Uselets in structure, meaning that each Uselet is responsible for exactly one part of the actual UI. This has benefits for the updating algorithm, which in the provided implementations is virtual DOM based. The elements inspected for diffing the actual DOM to the target state are contained exactly within the elements the Uselet corresponds to, reducing the usually high overhead of this approach. Additionally, since there is no overlap in responsibility, the diffing can happen concurrently.

All communication, inter-Uselet, client-server, user-interaction, etc., happens by message-passing, and message-passing only, giving a single unified interface throughout the whole system. This also simplifies dataflow and assists with testing and debugging.

Not only are Uselets the conceptual model of the interface, but to minimize the gap between model and implementation, Uselets also provide the framework for building the UI. I describe two different implementations of Uselets, one in Links, a functional language, and one in JavaScript as proof-of-concept. The differences between those implementations showcase some of the challenges of implementing Uselets but also provide a point of reference for future implementations in other languages.

The introducing example, two case-studies and some methodological notes demonstrate how to use Uselets for modelling and implementing different kinds of applications.

An evaluation of Uselets with criteria for the quality of software models from the literature as well as comparison with some existing frameworks for web-development, especially component-based, puts Uselets in context. Finally, I suggested a number of future improvements to Uselets, like life-cycle events or improved messaging, as well as some alternative applications besides web-apps.

List of Figures

3.1 The todo-application built with Uselets 19

4.1 A Uselet hierarchy is built on top of an equivalent Actor hierarchy. In structure, the resulting UI is equivalent to both. 22
4.2 The update cycle of Uselets: When a message is received (1), the internal data (2) is updated (3) and rendered as the Uselet substructure (4) a kind of blueprint for the UI. Using virtual DOM update (5), the real UI is modified to reflect the changes. 25
4.3 The UML state machine for the ATM example, as created in [48]. 30
4.4 Visual notation for Uselets 34

5.1 Exemplary comparison of user code for the Links and JavaScript implementation. 41
5.2 The debugger overlay with the message history on top of the TodoMVC application. 46

6.1 The `router-outlet` component for single-page routing 48
6.2 The overall hierarchy of the Uselets in the kanban application. 50
6.3 The kanban application after styling. 51
6.4 The Uselet hierarchy for the TodoMVC application. 52
6.5 Comparison of the TodoMVC implementation using Uselets with the original TodoMVC version. 53

List of Listings

2.1 A Formlet for entering a date. It is composed from three inputs for integers that are bound to the names *day*, *month*, and *year*, respectively. The `yields` clause describes how the submitted data should be processed, in this case passed to the `Date` type-constructor. 10
2.2 Rendering of a Formlet, using a simple handler function. Once rendered the returned initial XML can be embedded into a page. 11
2.3 Actor instantiation. `actorHandler` recursively defines the Actor behaviour for each message. `spawn` spawns a process on the server, `spawnClient` on the client. 11
2.4 A simple distributed calculator example. 12
2.5 Links functions annotated as client- and server-only. 13

3.1 The todo-app, composed of a header and the actual list. 15

3.2 Type signatures of the Uselet combinators: <+> for composing Uselets as siblings, +> to embed a Uselet in a piece of XML. Effect types parameterizing the Uselet type and the function-arrow are omitted for clarity from here on. 15
3.3 The signatures of the Uselet constructor-functions.
`textUselet` to construct static text
`xmlUselet` to construct static XML
`uselet` to construct dynamic Uselets
Effect types are omitted for clarity. 16
3.4 A static Uselet for the header. 16
3.5 The type of a single todo-item: Its textual content and the checked-status. . 16
3.6 The creation of the `todoList`. 17
3.7 The update-function for the todo-list. 17
3.8 The view function of the todo-list. 18
3.9 The markup for a single list-item. 18
3.10 Embedding of Uselets into a page in Links. 19

4.1 A Uselet is a function that takes a $Namer$ and produces the initial Xml, the updated namer $Namer$ and *spawns* the underlying Actor as a side-effect. Effect types are omitted for clarity. 23
4.2 Combining two Uselets yields a new Uselet, i.e. a function that passes the namer through the two originals and yields their combined HTML representation. 23
4.3 Uselets with the option to add context are functions that takes a $Namer$ and the additional context information *Ctx* and produces the initial Xml and the updated Namer, while spawning the Actor as a side-effect. The additional context has to be passed to all Uselets when combining them with the combinator, just as the namer has to be passed along. 26
4.4 The ATM types, model, and update function. 31
4.5 The ATM view function. 32

5.1 The `uselet` factory function creates dynamic Uselets from their name, their underlying data, an update-function and a view-function. Types are simplified, disregarding effect-types. label 36
5.2 `textUselet` and `xmlUselet` both produce Uselets that ignore additional context and pass the namer through. 36
5.3 Message passing in JavaScript via local closures bound to the `this`-reference (upper) and Links via the context (lower). 37
5.4 Named (upper) and anonymous (lower) Uselets. 38
5.5 Uselet composition using `asSibling` for same-level composition and `childOf` for embedding a Uselet in a HTML node. 39

5.6 A test case for the `InPlaceEdit`-Uselet using the rudimentary JavaScript testing framework. `describe` and `it` describe the test cases, as done by some popular testing frameworks. Access to a Uselets model and receive-function is restricted to testing-mode, as is the stand-alone initialization with `initUselet`. 43

List of Tables

6.1 Number of source lines of code to build the TodoMVC application for plain JavaScript, some popular frameworks, and some transpiled languages. . . . 54

7.1 Meyers “seven sins of the specifier” for formal specification [65] and how they apply to Uselets or how the model inherently prevents them. 56
7.2 Some Quality criteria for software models [82] and how Uselets comply with them. 57
7.3 Common criticism of models and their tools, as summarized in [87] 58

Bibliography

[1] Robbie Abed. *Hybrid vs Native Mobile Apps — The answer is clear*. Retrieved January 30, 2018. 2016. URL: https://ymedialabs.com/hybrid-vs-native-mobile-apps-the-answer-is-clear/.

[2] Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990. ISBN: 978-0-262-01092-4.

[3] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. "A Foundation for Actor Computation". In: *J. Funct. Program.* 7.1 (1997), pp. 1–72.

[4] Joe Armstrong. *Concurrency oriented programming in Erlang*. Invited talk, FFG. 2003.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[6] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. Tech. rep. 2015, pp. 1–96. DOI: 10.17487/RFC7540. URL: https://doi.org/10.17487/RFC7540.

[7] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. *Hypertext Transfer Protocol - HTTP/1.0*. Tech. rep. 1996, pp. 1–60. DOI: 10.17487/RFC1945. URL: https://doi.org/10.17487/RFC1945.

[8] Sam Birch and Alex Russell. *Progressive Web Apps: Great Experiences Everywhere*. At Google I/O 2017. 2017.

[9] Michael R. Blaha and James E. Rumbaugh. *Object-oriented modeling and design with UML, Second Edition*. Pearson Education, 2005. ISBN: 978-0-13-196859-2.

[10] Jack Calder. *HTML5 vs Native Apps: What's best for 2016?* Retrieved January 30, 2018. 2016. URL: http://justcreative.com/2016/03/17/html5-vs-native-apps-whats-best-for-2016/.

[11] Fred Cavazza. *Mobile Web App vs. Native App? It's Complicated*. Retrieved January 30, 2018. 2011. URL: https://www.forbes.com/sites/fredcavazza/2011/09/27/mobile-web-app-vs-native-app-its-complicated.

[12] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Trans. Comput. Syst.* 3.1 (1985), pp. 63–75. DOI: 10.1145/214451.214456.

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001. ISBN: 978-0-262-03270-4.

[14] William Douglas Clinger. *Foundations of Actor Semantics*. Tech. rep. Cambridge, MA, USA, 1981.

[15] Mathilde Collin. *Why desktop apps are making a comeback*. Retrieved January 30, 2018. 2015. URL: https://medium.com/@collinmathilde/why-desktop-apps-are-making-a-comeback-5b4eb0427647.

[16] World Wide Web Consortium. *Custom Elements. W3C Working Draft 13 October 2016.* Tech. rep. World Wide Web Consortium, 2016.

[17] World Wide Web Consortium. *HTML 5.3. Editor's Draft, 24 January 2018.* Tech. rep. World Wide Web Consortium, 2018.

[18] World Wide Web Consortium. *HTML Imports. W3C Working Draft 25 February 2016.* Tech. rep. World Wide Web Consortium, 2016.

[19] World Wide Web Consortium. *HTML Templates. W3C Working Group Note 18 March 2014.* Tech. rep. World Wide Web Consortium, 2014.

[20] World Wide Web Consortium. *Shadow DOM. W3C Working Draft 05 September 2017.* Tech. rep. World Wide Web Consortium, 2017.

[21] Dominic Cooney. *Introduction to Web Components. W3C Working Group Note 24 July 2014.* Retrieved January 30, 2018. 2014. URL: https://www.w3.org/TR/components-intro/.

[22] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. *An idiom's guide to formlets.* Tech. rep. LFCS, University of Edinburgh, 2008.

[23] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. "Links: Web Programming Without Tiers". In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures.* Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 4709. Lecture Notes in Computer Science. Springer, 2006, pp. 266–296. ISBN: 978-3-540-74791-8. DOI: 10.1007/978-3-540-74792-5_12. URL: https://doi.org/10.1007/978-3-540-74792-5_12.

[24] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. "The Essence of Form Abstraction". In: *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings.* Ed. by G. Ramalingam. Vol. 5356. Lecture Notes in Computer Science. Springer, 2008, pp. 205–220. ISBN: 978-3-540-89329-5. DOI: 10.1007/978-3-540-89330-1_15. URL: https://doi.org/10.1007/978-3-540-89330-1_15.

[25] Douglas Crockford. *JavaScript: The world's most misunderstood programming language.* Retrieved January 30, 2018. 2001. URL: http://www.crockford.com/javascript/javascript.html.

[26] Evan Czaplicki. *Elm's Time Traveling Debugger. Pause, rewind, and replay programs. Debug by changing history.* Retrieved January 30, 2018. 2014. URL: http://debug.elm-lang.org.

[27] Evan Czaplicki. *Scaling The Elm Architecture. An Introduction to Elm.* Retrieved January 30, 2018. 2012. URL: https://guide.elm-lang.org/reuse/.

[28] Evan Czaplicki. *The Perfect Bug Report. Debugging with Elm 0.18.* Retrieved January 30, 2018. 2016. URL: http://elm-lang.org/blog/the-perfect-bug-report.

[29] Evan Czaplicki and Stephen Chong. "Asynchronous functional reactive programming for GUIs". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 411–422. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2462156.2462161. URL: http://dl.acm.org/citation.cfm?id=2491956.

[30] Leslie Daigle. *WHOIS Protocol Specification*. Tech. rep. 2004, pp. 1–4. DOI: 10.17487/RFC3912. URL: https://doi.org/10.17487/RFC3912.

[31] Sébastien Doeraene. *Scala.js: Type-directed interoperability with dynamically typed languages*. Tech. rep. Ecole polytechnique federale de Lausanne, 2013.

[32] Emanuele D'Osualdo, Jonathan Kochems, and C.-H. Luke Ong. "Automatic Verification of Erlang-Style Concurrency". In: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 454–476. ISBN: 978-3-642-38855-2. DOI: 10.1007/978-3-642-38856-9_24. URL: https://doi.org/10.1007/978-3-642-38856-9_24.

[33] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. "Statistical Model Checking for Composite Actor Systems". In: *Recent Trends in Algebraic Development Techniques, 21st International Workshop, WADT 2012, Salamanca, Spain, June 7-10, 2012, Revised Selected Papers*. Ed. by Narciso Martı-Oliet and Miguel Palomino. Vol. 7841. Lecture Notes in Computer Science. Springer, 2012, pp. 143–160. ISBN: 978-3-642-37634-4. DOI: 10.1007/978-3-642-37635-1_9. URL: https://doi.org/10.1007/978-3-642-37635-1_9.

[34] Ralf Engelschall. *ComponentJS*. Retrieved January 30, 2018. 2009. URL: http://componentjs.com.

[35] Facebook Inc. *Flux. Application Architecture for Building User Interfaces*. Retrieved January 30, 2018. 2014. URL: https://facebook.github.io/flux/docs/in-depth-overview.html.

[36] Facebook Inc. *React. A JavaScript Library for building User Interfaces*. Retrieved January 30, 2018. 2017. URL: https://facebook.github.io/react/.

[37] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol - HTTP/1.1*. Tech. rep. 1999, pp. 1–176. DOI: 10.17487/RFC2616. URL: https://doi.org/10.17487/RFC2616.

[38] Roy T. Fielding, Mark Nottingham, and Julian F. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Caching*. Tech. rep. 2014, pp. 1–43. DOI: 10.17487/RFC7234. URL: https://doi.org/10.17487/RFC7234.

[39] Roy T. Fielding and Julian F. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Authentication*. Tech. rep. 2014, pp. 1–19. DOI: 10.17487/RFC7235. URL: https://doi.org/10.17487/RFC7235.

[40] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000.

[41] Simon Fowler, Sam Lindley, and Philip Wadler. "Mixing Metaphors: Actors as Channels and Channels as Actors". In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. Ed. by Peter Müller. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 11:1–11:28. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.11. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2017.11.

[42] Cory Gackenheimer. "Introducing flux: An application architecture for react". In: *Introduction to React*. Apress, Berkeley, CA, 2015, pp. 87–106.

[43] Jesse James Garrett. *Ajax: A new approach to web applications*. Retrieved January 30, 2018. 2005. URL: http://adaptivepath.org/ideas/ajax-new-approach-web-applications/.

[44] Google. *Angular*. Retrieved January 30, 2018. 2016. URL: https://angular.io.

[45] Google. *Angular. Lifecycle Hooks*. Retrieved January 30, 2018. 2016. URL: https://angular.io/guide/lifecycle-hooks.

[46] Irene Greif. "Semantics of communicating parallel processes". PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[47] W3C HTML working group et al. *XHTML 1.0: The extensible hypertext markup language*. Tech. rep. World Wide Web Consortium, 2002.

[48] Rolf Hennicker and Alexander Knapp. "Activity-Driven Synthesis of State Machines". In: *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*. Ed. by Matthew B. Dwyer and Antónia Lopes. Vol. 4422. Lecture Notes in Computer Science. Springer, 2007, pp. 87–101. ISBN: 978-3-540-71288-6. DOI: 10.1007/978-3-540-71289-3_8. URL: https://doi.org/10.1007/978-3-540-71289-3_8.

[49] Carl Hewitt. *Actors are intended to model *all* digital computation*. Note by Carl Hewitt on a discussion on Implementing the communication semantics of actors. 2015.

[50] Carl Hewitt. "Viewing Control Structures as Patterns of Passing Messages". In: *Artif. Intell.* 8.3 (1977), pp. 323–364. DOI: 10.1016/0004-3702(77)90033-9. URL: https://doi.org/10.1016/0004-3702(77)90033-9.

[51] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*. Ed. by Nils J. Nilsson. William Kaufmann, 1973, pp. 235–245. URL: http://ijcai.org/proceedings/1973.

[52] Ian Hickson, R Berjon, S Faulkner, T Leithead, ED Navara, E O'Connor, and S Pfeiffer. *HTML5 specification*. Tech. rep. World Wide Web Consortium, 2014.

[53] C. A. R. Hoare. “Communicating Sequential Processes”. In: *Commun. ACM* 21.8 (1978), pp. 666–677. DOI: 10.1145/359576.359585.

[54] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. “Type-Safe Eventful Sessions in Java”. In: *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*. Ed. by Theo D’Hondt. Vol. 6183. Lecture Notes in Computer Science. Springer, 2010, pp. 329–353. ISBN: 978-3-642-14106-5. DOI: 10.1007/978-3-642-14107-2_16. URL: https://doi.org/10.1007/978-3-642-14107-2_16.

[55] Mohammad Mahdi Jaghoori, Ali Movaghar, and Marjan Sirjani. “Modere: the model-checking engine of Rebeca”. In: *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*. Ed. by Hisham Haddad. ACM, 2006, pp. 1810–1815. ISBN: 1-59593-108-2. DOI: 10.1145/1141277.1141704. URL: http://doi.acm.org/10.1145/1141277.

[56] JS Foundation. *Dojo Toolkit*. Retrieved January 30, 2018. URL: https://dojotoolkit.org.

[57] Rajesh K. Karmani, Amin Shali, and Gul Agha. “Actor frameworks for the JVM platform: a comparative analysis”. In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009, Calgary, Alberta, Canada, August 27-28, 2009*. Ed. by Ben Stephenson and Christian W. Probst. ACM, 2009, pp. 11–20. ISBN: 978-1-60558-598-7. DOI: 10.1145/1596655.1596658.

[58] Alexander Knapp and Gefei Zhang. “Model Transformations for Integrating and Validating Web Application Models”. In: *Modellierung 2006, 22.-24. März 2006, Innsbruck, Tirol, Austria, Proceedings*. Ed. by Heinrich C. Mayr and Ruth Breu. Vol. 82. LNI. GI, 2006, pp. 115–128. ISBN: 3-88579-176-5.

[59] Ten-Hwang Lai and Tao H. Yang. “On Distributed Snapshots”. In: *Inf. Process. Lett.* 25.3 (1987), pp. 153–158. DOI: 10.1016/0020-0190(87)90125-6. URL: https://doi.org/10.1016/0020-0190(87)90125-6.

[60] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Prentice Hall, 2005.

[61] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul A. Agha. “A Framework for State-Space Exploration of Java-Based Actor Programs”. In: *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 2009, pp. 468–479. ISBN: 978-0-7695-3891-4. DOI: 10.1109/ASE.2009.88. URL: https://doi.org/10.1109/ASE.2009.88.

[62] Adam Lella and Andrew Lipsman. “The U.S. Mobile App Report”. In: (2014). Retrieved January 30, 2018. URL: https://www.comscore.com/Insights/Presentations-and-Whitepapers/2014/The-US-Mobile-App-Report.

[63] Lightbend Inc. *Akka*. Retrieved January 30, 2018. 2011. URL: https://akka.io.

[64] Ryan Matzner. *Why Web Apps Will Crush Native Apps*. Retrieved January 30, 2018. 2012. URL: http://mashable.com/2012/09/12/web-vs-native-apps/.

[65] Bertrand Meyer. “On Formalism in Specifications”. In: *IEEE Software* 2.1 (1985), pp. 6–26. DOI: 10.1109/MS.1985.229776. URL: https://doi.org/10.1109/MS.1985.229776.

[66] Microsoft. *TypeScript - JavaScript that scales.* Retrieved January 30, 2018. 2012. URL: http://www.typescriptlang.org.

[67] Robin Milner. *A Calculus of Communicating Systems.* Vol. 92. Lecture Notes in Computer Science. Springer, 1980. ISBN: 3-540-10235-3. DOI: 10.1007/3-540-10235-3. URL: https://doi.org/10.1007/3-540-10235-3.

[68] Robin Milner. *Communicating and mobile systems - the Pi-calculus.* Cambridge University Press, 1999. ISBN: 978-0-521-65869-0.

[69] Miguel Mota. *JavaScript idiosyncrasies.* Retrieved January 30, 2018. 2017. URL: https://github.com/miguelmota/javascript-idiosyncrasies.

[70] Brad A. Myers, Scott E. Hudson, and Randy F. Pausch. “Past, present, and future of user interface software tools”. In: *ACM Trans. Comput.-Hum. Interact.* 7.1 (2000), pp. 3–28. DOI: 10.1145/344949.344959.

[71] Node.js Foundation. *Node.js.* Retrieved January 30, 2018. URL: https://nodejs.org/en/.

[72] Object Management Group®, Inc. *What is UML. Introduction to OMG's Unified Modeling Language™(UML ®).* Retrieved January 30, 2018. 2005. URL: http://www.uml.org/what-is-uml.htm.

[73] Martin Odersky, Lex Spoon, and Bill Venners. *Object equality.* Retrieved January 30, 2018. Artima Inc., 2008. URL: http://www.artima.com/pins1ed/object-equality.html.

[74] Tim O'reilly. *What is web 2.0.* Retrieved January 30, 2018. 2005. URL: http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.htm%20l.

[75] Addy Osmani, Sindre Sorhus, Pascal Hartig, Stephen Sawchuk, Colin Eberhardt, Arthur Verschaeve, and Sam Saccone. *TodoMVC.* Retrieved January 30, 2018. 2011. URL: http://todomvc.com.

[76] Polymer Authors. *Polymer Project.* Retrieved January 30, 2018. 2016. URL: https://www.polymer-project.org.

[77] Q-Success. *Usage of JavaScript for websites.* Retrieved January 30, 2018. 2018. URL: https://w3techs.com/technologies/details/cp-javascript/all/all.

[78] Dave Raggett, Arnaud Le Hors, Ian Jacobs, et al. *HTML 4.01 Specification.* Tech. rep. World Wide Web Consortium, 1999.

[79] Raghu Rajkumar, Nate Foster, Sam Lindley, and James Cheney. “Dr. Formlens, Or: How I Learned to Stop Worrying and Love Monoidal Functors”. In: (2012).

[80] Jeff Rothenberg. *The nature of modeling.* John Wiley Sons, Inc., 1989, pp. 75–92.

[81] Douglas C. Schmidt. “Guest Editor's Introduction: Model-Driven Engineering”. In: *IEEE Computer* 39.2 (2006), pp. 25–31. DOI: 10.1109/MC.2006.58. URL: https://doi.org/10.1109/MC.2006.58.

[82] Bran Selic. “The Pragmatics of Model-Driven Development”. In: *IEEE Software* 20.5 (2003), pp. 19–25. DOI: 10.1109/MS.2003.1231146. URL: https://doi.org/10.1109/MS.2003.1231146.

[83] Jonathan Strickland. *How Web 2.0 Works*. Retrieved January 30, 2018. 2007. URL: https://computer.howstuffworks.com/web-20.htm.

[84] Pedro A. Szekely. “Retrospective and Challenges for Model-Based Interface Development”. In: *Design, Specification and Verification of Interactive Systems’96, Proceedings of the Third International Eurographics Workshop, June 5-7, 1996, Namur, Belgium*. Ed. by François Bodart and Jean Vanderdonckt. Springer, 1996, pp. 1–27. ISBN: 3-211-82900-8.

[85] The jQuery Foundation. *jQuery*. Retrieved January 30, 2018. URL: http://jquery.com.

[86] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. “Synthesis of Behavioral Models from Scenarios”. In: *IEEE Trans. Software Eng.* 29.2 (2003), pp. 99–115. DOI: 10.1109/TSE.2003.1178048. URL: https://doi.org/10.1109/TSE.2003.1178048.

[87] Jean Vanderdonckt. “Model-Driven Engineering of User Interfaces: Promises, Successes, Failures, and Challenges”. In: *Proceedings of 5th Annual Romanian Conference on Human-Computer Interaction ROCHI’2008*. Matrix ROM (Bucharest), 2008.

[88] Steve Vinoski. “Concurrency with Erlang”. In: *IEEE Internet Computing* 11.5 (2007), pp. 90–93. DOI: 10.1109/MIC.2007.104. URL: https://doi.org/10.1109/MIC.2007.104.

[89] WHATWG. *Living Standard — Last Updated 29 January 2018*. Tech. rep. WHATWG, 2018.

[90] Jon Whittle and Johann Schumann. “Generating statechart designs from scenarios”. In: *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*. Ed. by Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf. ACM, 2000, pp. 314–323. ISBN: 1-58113-206-9. DOI: 10.1145/337180.337217. URL: http://dl.acm.org/citation.cfm?id=336512.

[91] Jim R Wilson and Jacquelyn Carter. *Node. js the right way: Practical, server-side javascript that scales*. Pragmatic Bookshelf, 2013.