A Study on a Ecosystem Approach for Model-Based Software Engineering

Morten Hartvigsen
Aalborg University, dept. Computer Science
Aalborg, Denmark
mhartv16@student.aau.dk
m.hartvigsen@gmail.com

Mikkel Pedersen
Aalborg University, dept. Computer Science
Aalborg, Denmark
mipede16@student.aau.dk
mikkel.pedersen96@live.dk

ABSTRACT

Utilizing Model-Based Software Engineering (MBSE) can ensure more sound software solutions because of model checking and finding issues earlier in the development cycle.

We present an ecosystem approach for using MBSE to develop modern web applications and present a case study on our proof of concept framework, highlighting the viability of using MBSE in a low-code environment.

We investigate eight hypotheses in a qualitative study of the benefits and usability of MBSE.

We conclude that the ecosystem would enable cross-compatibility between many different MBSE tools, allowing for easier adaptation of MBSE in the industry. Furthermore, we conclude that it is hard for developers to transition to an MBSE workflow, but it would be advantageous for them to do so. Lastly, we conclude that the friction in the adaptation of MBSE is due to the lack of maturity in the current MBSE tools available.

1 INTRODUCTION

When developing modern web applications, companies have many different types of people who need to work together to create something the customer wants. Either they can use technologies such as Bootstrap [1] to enable developers to use prebuilt HTML, CSS, and JavaScript components to get the desired behavior with minimal effort or Headless UI [2], where developers have prebuilt JavaScript and only need to add the wanted styling. Alternatively, they can utilize low-code platforms [3, 4], which allow people with different backgrounds and programming knowledge to create software solutions within a drag-and-drop environment, with minimal code writing.

Using a low-code platform, the users create abstract models that can be transformed into executable code. Because of this higher level of abstraction, it might be beneficial to introduce model checking into the modern web development world to ensure more sound programs using an MBSE approach.

It is often claimed that an MBSE approach will allow developers to make software faster than the traditional methods [5, 6]. Although we believe the findings of these sources, many of them only have a statistical analysis of the speed increase. We believe that to spread the use of MBSE; we need to understand how and why it is faster to use. Along with the costs of this speed increase, how does it affect the software projects'? Does it bring some adverse side effects such as developers missing opportunities for:

- A deep understanding of the problem domain?
- Exploring alternative solutions?
- Does it lead to a carelessness mentality?

From all of these concerns, we present an ecosystem that is a low-code platform, where the developers can sit with the customer and define the behavior of specific components to ensure that the components are correctly defined as the customer wants it. Furthermore, the developer can also model check the given component to ensure that the code does not contain specific errors. Our ecosystem consists of the following parts, which can also be seen in Figure 1:

- The Visual Formalism
- A system specification format for representing the visual formalism
- A compiler that can take the system specification format as input
- An "Abstract Behaviour Tree"(ABT) that will be the result from the compiler.

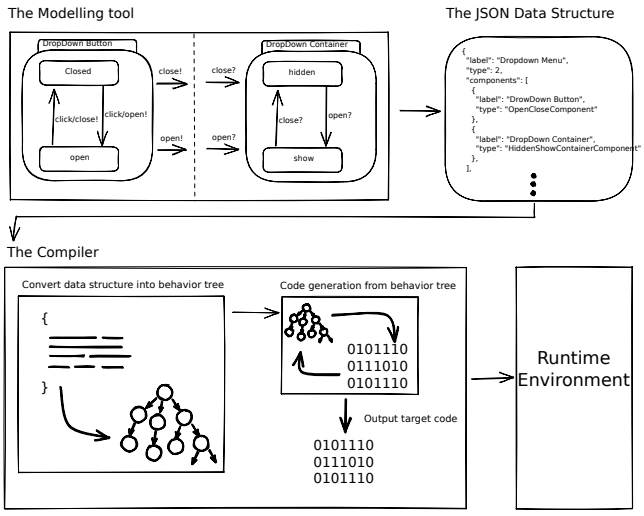

Figure 1: An overview of the ecosystem

These parts would allow for a couple of custom tools, such as:

- A Modelling Tool for drawing the behavior models
- Integration with any Model Checking Engine (by utilizing the compiler to translate to any Model Checking Engine format from the ABT)
- Allow for translating to any target code platform (by implementing a code generator for the "Abstract Behaviour Tree")

To understand the viability of MBSE and low-code platforms, we built a prototype of all parts of this ecosystem and held multiple workshops with different types of participants. From our wonderings, we have created eight hypotheses that we would like to answer:

- **H1:** Using MBSE, developers can create correct behaving software solutions faster.
- **H2:** The use of MBSE provides better communication between developers and designers.
- **H3:** The use of MBSE provides better communication between developers and stakeholders.
- **H4:** MBSE makes it easier for developers to create complex systems.
- **H5:** People without experience or training in software development principles can utilize an MBSE tool to create software systems.
- **H6:** MBSE is difficult to adopt for people without formal training in the discipline.
- **H7:** Traditional developers will have reservations and feel some friction in adopting an MBSE approach.
- **H8:** MBSE gives developers more confidence in their work.

The main contribution is a proof of concept framework to allow users to create visual components in web applications using our visual formalism. Furthermore, we have conducted numerous workshops to collect qualitative data regarding the usage of MBSE and low-code platforms. Finally, we have looked at which challenges need to be solved to create a software solution utilizing the MBSE approach.

The rest of the paper is organized as follows. Section 2 looks at related work and Section 3 will highlight the vision we have for the final system. Section 4 focuses on preliminary work for this paper, and Section 5 outlines the compiler that has been built for this ecosystem and shortly describes how it works. Section 6 will highlight some of the benefits of MBSE, and Section 7 will highlight the case studies we have been conducting, and finally section 8 and will conclude our findings.

2 RELATED WORK

The related work is divided into two distinct parts.

2.1 Tools/frameworks

The use of low-code platforms is growing and becoming more and more popular [7], as the demand for more complex systems arises. Systems such as OutSystems [3] and Mendix [4] are examples where the companies can have different types of people working on the system. However, because of the growth of platforms, it can be hard to determine which platform is the best, where the work of [8] compares some of the more powerful low-code platforms. Furthermore, [8] also looks at what these platforms have in common, such as their overall structures, and gives the reader an insight into the pros and cons of each platform.

The work on XState [9] should also be mentioned as they used the idea of statechart [10] and have implemented a way of making state machines in React, which the developers can use. However, their work still requires that the users understanding and are able to write JavaScript code.

2.2 Model-Based Software Engineering

Highlighted in [5], the idea of MBSE should be more widely adopted because of the way that we can use the models in terms of generation of code and model checking.

We believe that MBSE should be more used. However, we also see some of the problems and challenges that come with it, and some research only focuses on that MBSE is faster than traditional methods. However, looking at [5, 11], many problems need to be solved before broader adoption of MBSE can be achieved. One of the problems mentioned in [5] is the usability of MBSE, and the infrastructure around such tools is more complex than traditional methods. As mentioned in [11], the lack of standards and benchmarks are somewhat of a problem, as we have no way of telling if one approach is better than another.

3 VISION OF ECOSYSTEM

This section presents the grand vision for the ecosystem that we propose. We believe that a visual tool using our ecosystem for web application development should have three main areas; The Modelling Tool, The Structure Builder, and lastly, The Styling Tool.

The modeling tool is where the user creates the behaviors of their models, where they can verify the given model using a model checker engine to see if the model is sound. Furthermore, if an error occurs, the user can use the simulator to know why the error occurred and fix the error.

The structure builder is where the user can piece together how their web application should look, from the structure aspect, not the styling. Here, the user can drag different HTML elements and their custom components and see how it looks in a live preview window.

The styling tool allows the user to style the different elements they have in their web application. Here we envision that one would use TailwindCSS [12] to ensure a more uniform styling across the entire web project. The idea is that the user can click on an HTML element, and then they get suggested which properties they can change, e.g., add a margin or padding to that element. Then, just as in the structure builder, the user should see how the given element looks in the different viewports.

3.1 System Specification Format

To ensure a uniform way of specifying our components as data, we have created multiple JSON schemas representing our System Specification Format, which can validate a JSON object as a component. Furthermore, these JSON Schemas are all extendable to allow for new features in the future. We have three different JSON schemas, one for each component type in our formalism, and Listing 1 shows some of the schemas for our Simple Component. From the schema, one can see which data is required to have in a Simple Component. For example, we have selected that the *label* key should have the type *string*, and the pattern, which is a regex, should allow every character. To see the rest of these schemas, see Appendix A.

Listing 1: Snippet of the simple component schema

```
{
  ...
  "title": "1",
  "type": "object",
  "required": [
```

```
    "label",
    "initialState",
    "states",
    "events",
    "inputs",
    "outputs",
    "transitions",
    "positions"
  ],
  "properties": {
    "label": {
      "$id": "#root/label",
      "title": "Label",
      "type": "string",
      "default": "",
      "pattern": "^.*$"
    },
      ...
  }
}
```

To validate if these JSON objects are correct and live up to the JSON schema (System Specification Format), we have utilized *ajv* [13] to validate the JSON data against the schema. Validating the JSON data ensures that when the modeling tool reads a file from disc, it has the correct format and knows how to parse it into a drawing, amongst other things.

4 PRELIMINARY WORK

This work is a continuation of [14], a ninth-semester project by the same authors as this paper. In [14], we proposed a visual formalism based on Statecharts [10] and Interface Automata [15] for modelling web-based user interface behaviour. Along with that, we built a proof-of-concept modelling tool that integrates with the Spin Model Checker engine [16]. The concept was to develop a formalism that would allow developers to model a front end's behaviour and utilize a model checking engine to check for different behaviours.

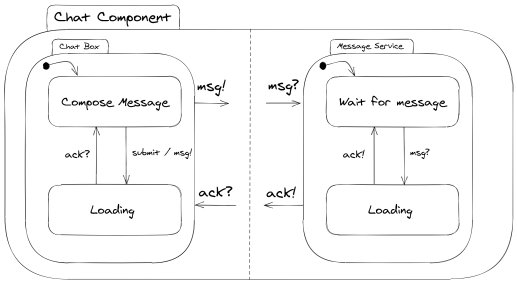

Figure 2: Example of the Robust UI Formalism

Figure 2 shows an example of our formalism, where we have adapted the AND-superstate from statecharts [10], and we use Interface Automata [15] to make communications between super states. However, instead of the name super state, we refer to them as components. Table 1 shows the different types of components we have and what they compare to in statecharts [10].

In our formalism	In statecharts
Simple component	XOR super state
Composite component	AND super state
Selective component	XOR super state with selection entry

Table 1: Shows what our components compare to in the statechart formalism

Another thing to note is that we have limited our formalism compared to Statecharts [10]. Hence, features such as History State, transitions across State boundaries (Component boundaries in our vocabulary), actions and activities, and others are not present in our formalism.

The important takeaway from our formalism is in the manner we handle Component Communication. Communication has to happen via the Interface Automata [15] formalism, meaning a Component can only communicate to another via a message channel. Forcing communication via message channels disallows transitions across component boundaries, leading to a more modular formalism that can allow for a component from one project to be added to another project.

5 THE COMPILER

The actual development of the compiler is not the main focus of this paper. Nevertheless, we find it advantageous to quickly discuss how we chose to build the compiler and the primary consideration in developing the compiler.

It was essential for us that the compiler could run on the most widely adopted operating systems, i.e., Windows, macOS, and Linux. Therefore we chose to utilize the JVM as our runtime environment, and we decided to use the programming language Kotlin. One uses the compiler from the terminal by specifying a target language, a folder path to search for dependencies, and the main file of the compilation.

The compiler is constructed of three modules:

- The Tokenizer
- The Parser
- The Code Generator

5.1 The Tokenizer

Since this compiler takes a pre-defined data structure, which is JSON-based, as opposed to traditional text-based source code, we chose to omit the Lexical Analyzer and Syntax Analyzer and instead use an off the shelf JSON parser to read the source files and convert the data structure of the formalism into a stream of tokens.

The Tokenizer can construct the following tokens:

- Composite Component Token
- Selective Component Token
- Simple Component Token
- State Token
- Transition Token

The **Composite Component Token** contains its name, a list of child components, a list of input events, and a list of output events.

The **Selective Component Token** contains its name, the name of the default case, a list of the cases, and the input stream.

The **Simple Component Token** contains its name, the name of the initial state, a list of states, a list of browser events, a list of input events, a list of output events, and a list of transitions. Next, the **State Token** contains its name, and finally, the **Transition Token** contains the name of the origin state, the label of the transition, and the destination state.

5.2 The Parser

After the tokenizer has converted the data structures into a stream of tokens, the parser will convert the tokens into a tree structure representation that allows the code generator to traverse the tree and generate the correct semantics' target code.

The different nodes that the Tree Structure can consist of is:

- Module Node
- Composite Component Node
- Selective Component Node
- Simple Component Node
- State Node
- Transition Node
- Stream Node
- Identifier Node
- Guard Node
- Case Node

5.2.1 Module Node. The head of the tree will always be a **Module Node** and acts as an indicator of the start of a component, either Simple, Composite, or Selective component. The idea behind such an indicator is for target codes that would like to utilize reusable code, where they can check if the module is already generated and then reference it instead of building it. Furthermore, the Module Node contains a Stream Node for the inputs, a Stream Node for the outputs, and a Stream Node for the browser events. The idea behind having this information on the module is based on our practical experience with different configurations, where we found it easier to have all streams in the module node as appose to putting it into the component node itself. Finally, the body property of a Module Node contains either a Simple, Composite, or Selective Component.

Listing 2: ModuleNode in pseudocode

```
class ModuleNode {
  var name: String;
  var type = "ModuleNode";
  var intputStreamNode: StreamNode;
  var outputStreamNode: StreamNode;
  var eventStreamNode: StreamNode;
  var body: SimpleComponentNode | CompositeComponentNode |
        SelectiveComponentNode;
  var typeLookUpTable: Map<String, String>;
}
```

5.2.2 Simple Component Node. The Simple Component Node act as the bottom of the component hierarchy. It is the only component that is allowed to contain actual states. The property *children* contains a list of all the State Nodes included in the component. Besides that, it includes a reference to its parent, i.e., the Module Node. This reference has shown through empirical testing to be an excellent property to have for some code generators.

Listing 3: SimpleComponentNode in pseudocode

```
class SelectiveComponentNode {
  var name: String;
  var type = "SimpleComponentNode";
  var children: List<StateNode>;
  var typeLookUpTable: Map<String, String>;
  var parent: ModuleNode;
}
```

The remaining nodes can be seen in Appendix B.

Figure 3 shows a visual representation of the ABT of the Chat Component shown in Figure 2.

5.3 Code Generator

The Code Generator is where third-party developers would extend the compiler to support any desired target platform. Currently, the compiler has four code generators, one for QTree (a latex tree-drawing plugin), one for TreantJS [17], a JavaScript library for interactive tree-structures in the browser, one for Promela for the model checker Spin [16] and lastly, one for the RobustUI Framework in TypeScript.

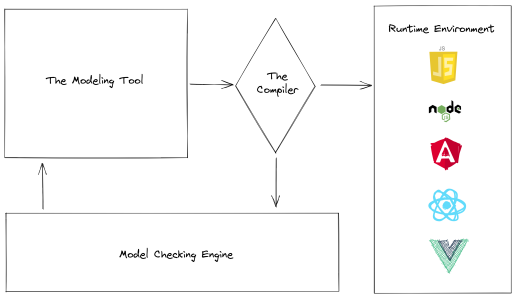

Figure 4: An overview of how one can use the ecosystem

The RobustUI Framework is our proof-of-concept runtime framework for running the constructed machines in the browser and allowing real-world behavior to be translated from visual drawings into browser interactions.

These four Code Generators use a Visitor Pattern [18] inspired method to convert the ABT into target code. Since all names of the nodes are prefixed with a namespace, helping the code generator correctly identify the nesting of components, they also have different helper methods for dealing with the namespacing. We would recommend any third-party developer who wishes to extend the compiler to study these four already existing generators to get an idea of converting the ABT into target code.

6 BENEFITS OF MODEL BASED SOFTWARE ENGINEERING

In this section we will present the supposed benefits of an MBSE approach to software development. These supposed benefits acts as the ground for our case studies and we will try to validate the claims in an qualitative manner, but more about that in Section 7.

6.1 Makes Development Faster

One of the advantages often agreed-upon with an MBSE approach is that it makes developing sound software solutions faster [6]. It also seems logical when considering the power of modeling formalism combined with model-checking engines that can verify specific

Figure 3: A visual example of the ABT for Figure 2

properties. However, we allow ourselves to be skeptical to accept this as a fact since software engineering is about more than shipping the first version.

One example is how software innovation is becoming more and more critical for the competitiveness of software companies [19]. How would the supposed speed of MBSE affect a project that requires software innovation? Would engineers have enough time to realize opportunities for pivoting, exploitation of existing technologies and so on? Would an MBSE approach even fit in, in an innovative project using a process such as Essence [19]?

Another example is the software lifetime. For a long-lived software project, how would the project's maintainability be affected by using an MBSE approach? Would it be possible to extend or change the behavior of the software in a reasonable manner with regards to deployment pipelines, modularity, and scalability both in terms of computing scalability and personal scalability?

There are also soft-value considerations to take into account, how does MBSE affect the individual engineer's commitment to the project. Would the supposed speed harm their professionalism, would it affect their sense of quality and result in a carelessness among the developers? On the other hand, would it have a positive effect in terms of relieving burnout and stress among engineers? Would it give engineers more confidence in their solutions and help engineers trust the quality of their work?

6.2 Easier to Change

It is a reasonable assumption that using an MBSE approach would lead to a process that will allow engineers to make changes to the system easier and safer, this assumption is shown somewhat correct in Model-Driven Development [20]. We believe that this assumption comes from the fact that all developers with experience have tried the dreadful situation of changing a small part of a traditional developed system and thereby mystically breaking the entire system. Whereas on the surface of MBSE, it looks pretty manageable to make a small change to a sub-model in a bigger model and utilizing the power of both the visual modelling tool and perhaps a model checking engine to verify the model to ensure that nothing broke.

However manageable it seems in theory, how would it work in practice for an entire system? Would the engineers be able to comprehend the effect of their changes throughout the model of the system? Do visual modeling tools offer enough encapsulation to allow for minor changes in a reasonable scope? How would changing a software built with MBSE compare to changing a well-designed software built with practices such as Test-Driven Development?

6.3 Ease of Collaboration

In chapter 10, "Practical experience and implementation", of [10], David Harel talks about how using a visual modelling language helped with the communication across different stakeholders and contributors of varying disciplines. But David Harel's scenario is one where the development was internally in the company, with engineers and other domain experts representing the same company.

How does it hold up in a scenario where there is a customer and supplier dynamic, where a customer hires the software developers. Would the customer invest the time to learn how to decode the visual models? Would they be able to understand it (assuming that the customer has little to no technical expertise)? Would they recognize the value-proposition of MBSE? Or would they instead prefer a hands-off approach and get the final product delivered with little to no involvement? Does a visual model allow engineers to communicate hard to understand technical details in a condensed and understandable fashion for non-technical stakeholders?

7 WORKSHOP

We held three workshops with different participants. However, due to COVID-19, it was more challenging to get the number of participants that we would like to have. Table 2 shows how many participants we had in each workshop and what their background are. For the other disciplines, one of them works with different technologies in product line management and understands how workflows work. The other understands prototyping and engineering for different kinds of machines and has a little programming knowledge, but not enough to be classified as a developer in our study.

	Developers	Designers	Other disciplines
Workshop 1	2	1	0
Workshop 2	0	0	1
Workshop 3	0	0	1

Table 2: Classifications of participants in each workshop

We have two primary wishes for the workshops in regards to what we aim to achieve. The first one is that we want a qualitative discussion regarding the use of MBSE and to try and answer some of the hypotheses presented in Section 1. Secondly, we want to discuss our proof-of-concept tool to understand what requirements such a tool should have for future development.

The workshop is split into three parts. The first part introduces the participants to what MBSE is, then introduces our visual formalism and presents our proof of concept tool. The second part consisted of four different tasks that the participants should solve using our tool. Finally, the last part is an open discussion where we first talked about the tasks they had to solve and how it went, and then spoke about MBSE.

7.1 Tasks

The four tasks will be increasing in complexity, and the participants should reuse what they had learned in the previous task to solve the next one. We created a small TypeScript project, per task, where the HTML and CSS code needed to solve the task is pre-defined. First, the participants were to make the correct model, where each task had a description regarding the functionality that the given component should have. After making the model, the participants would then generate the TypeScript code and copy it into the TypeScript project for that particular task. The last thing they should do was to open the project in the browser and validate that they had met the task's requirements.

An example of task one can be seen in Listing 4. The rest of the task can be seen in Appendix C

Listing 4: SimpleComponentNode in pseudocode

```
In this task you should create a "simple" component which can do
    the following

* It has two states which are "on" and "off"
* If the component is in the state "on" and the user presses it,
    then the component should go to the state "off" and vice
    versa

You have got the needed HTML and CSS to solve the task. Your task
    is to model this component and paste it into the given
    project and get everything to work with the functionalities
    mentioned above
```

7.2 Results from workshops

This section will highlight some of the key points from our workshops.

7.2.1 Visual programming as a concept. To start our discussion, we wanted to get feedback on our proof of concept to understand if visual programming is useful for developing software systems. Here we have an interesting discovery between the developers and non-developers. The developers argued that a visual system would make it harder for them to develop software, with arguments such as

- *Developers have, through many years, learned to make their keyboard the primary tool, so drawing with the mouse is very unnatural for us.*
- *Developers are generally not of the visual thinking types. We think more in abstract constructs, so it is hard to translate the thoughts to visually representations*

The non-developers have opposing views, where they find the visual way of doing it beneficial and have arguments such as:

- *You can use it[our tool] to make prototypes of software and can easily get developers to further develop on the system at a later stage of the development stage*
- *It is possible to relieve some of the work from the developers, which enables non-developers to influence the project in the right direction and get a uniform language of communication with the developer.*

This disagreement between developers and non-developers is something we expected, but what is interesting is the developers' arguments, which are not as serious as they might seem. The argument regarding the keyboard being their primary tool is valid in our perspective, as the keyboard is the only tool, which traditional development allows. However, this is a question about the historical necessity and not necessarily a desired feature. We also find the argument that *developers are not visual thinking* to be wrong, as most developers can understand and create diagrams such as UML diagrams to communicate ideas. Furthermore, we assume that the problem is that using a visual tool is a radical change in discipline than the traditional way. It would require many hours to convert a traditional developer to be comfortable with a visual tool. Our opinion is that this issue is similar when developers work in one programming paradigm and then need to work in another paradigm. There is a learning curve where the new paradigm would be unnatural. However, we see developers switch from one paradigm to another multiple times in their careers, so why would it not be the same with MBSE? We believe that Model-Based development is still rather new and misses the validation through usage that the other programming paradigms have.

Regarding the arguments from the non-developers, we have certain reservations regarding these arguments, such as the non-developers can create a prototype and then give it to the developers at a later stage. We acknowledge that MBSE would allow non-developers to create a prototype. However, we see a problem when

the developer should work further on the prototype. There would be a massive overhead and assumptions such as the developer should understand MBSE and be comfortable with such a tool. Alternatively, it could be understood as the generated code would be sent to the developers. However, this highlights another assumption: the generated code has been created so that a traditional developer can understand it and work further on it. The last way we see how it can be understood is that the developers should start from scratch and use the prototype as a requirements specification. We find this point the most realistic and an important point regarding if MBSE could be used to, as an example, creating software for microcontrollers that could test a potential marked quickly and later send the prototype and the models to developers, which would make the actual product.

Regarding the argument that the non-developers could take some of the work from the developers, we are not sure that this would be "allowed," as a company might have in-house developers, which jobs are creating such software. However, we agree that the visual program allows non-developers to get a better overview of the software to find issues in the early stages. We also agree that a visual tool would allow better communication between developers and non-developers if both parties understand the visual formalism.

7.2.2 Reliability on MBSE. We asked the participants regarding their confidence in the models they exported into the project. The developers have arguments such as:

- *It can be hard to trust the auto-generated code, as you do not have any feelings for it. You do not know if the developers behind the auto-generated code have been thorough or if there are edge cases that destroy the behavior*
- *Developers will have little to no trust in the auto-generated code, whereas designers and non-developers will trust more on their models if it 'works on their computer'*
- *It is hard to evaluate if the model is good or not, compared to if it has any errors for the given use case, but also if the modeling will give unnecessary overhead in the generated code, contra to have modeled it differently*

For the non-developers, they have a different view on the reliability for MBSE, where they had arguments such as:

- *You have confidence in the solution if the program tells you that there are no errors, which is a big help and gives confidence in your own work*

From Section 7.2.1, we see the same thing in regards to developers vs. non-developers. Again, there is a clear line between the developers, who have more experience and understanding of what happens in an auto-generated code environment and know that mistakes can happen. The non-developers do not consider that the auto-generated code can affect the end product. We got the intuition that the non-developers see the code to have two main properties: Either it works or it does not. The developers have more properties, which they consider such as, scalability, maintainability, and edge cases.

We want to point out that the question about the quality of models can be hard to assess, compared to traditional development, where principles such as SOLID [21] can help one assess the quality of the code. The missing principles in MBSE are a problem. However, it should be possible to introduce similar principles. The same is true, for example, for software patterns, as we see no problem in translating these patterns into modeling patterns. We even see many software patterns communicated as UML diagrams already [18], which is a form of modeling.

We will recognize that the developers' worries are valid and that the developers has to lay great confidence in a tool that creates auto-generated code before they will accept using it. However, the truth is that almost every developer already does it today. The developers do not concern about the quality of the assembly code, which their compiler generates, so we see this more like a "fear." It could be beneficial for MBSE to position itself differently. Instead of positioning MBSE as a means of code generation, it could take the position as a new programming language in the declarative paradigm, for example. That way, MBSE will possibly not seem so frightening to traditional programmers.

7.2.3 Other points for MBSE. This section will cover some of the points and arguments, we got from the discussion which was not planned. The following points have been captured here are:

- Possibility of expression compared to traditional programming
- Speed of MBSE
- Mental burden in MBSE
- Communication benefit

It quickly got discussed by the developers how the expressiveness in the models are a problem, with arguments:

- *It is a problem that there are things in traditional programming that you are being forced to consider, e.g., naming of details, separation of concern, etc, which are being removed in a model. It is often through these details that we, as developers, can signal our intent to other developers and make it easier for others to understand and maintain our systems*
- *The principle about communication from and to different components should go through message channels is simple to understand. However, it is hard to evaluate if it is easy in the real world. We have different communication techniques in traditional programming, such as message queues, method invoking, getters, and setters. They have all proven to solve different communications demands in a system*

The non-developers found that it was non-intuitive that you should define browser events on every state, with arguments:

- *When you sit down and make behavior for a button, then it is weird that you should define a click event on all states since it is known that a button can be clicked. It disturbs the process that you need to define a standard behavior*

We find all of these arguments interesting, and we will first look at the arguments for the developers. The problem regarding a higher level of abstraction, thereby removing details, we do not see as a problem with MBSE but more a problem with the developers' missing experience in decoding models. However, there might be a valid argument in separations of concern, which we have tried to solve by our component-based formalism. However, it would be necessary to build a larger system to evaluate whether it is still a problem or not.

Regarding the non-developers argument, we believe that this is a question about the missing experience in MBSE. The whole argument is based on false facts that a button can always be pressed. However, this is not true in the real world. If non-developers got more experience, they would understand why they need to add a click event for multiple transitions.

After we had discussed which challenges the participants feels that there are in using our system contra their daily tools, we talked about the speed in MBSE, and the participants have arguments such as:

- *I believe [the use of MBSE] would give a big lead at the start of a new project, but there will be some cut-off point, where it would begin to be a hindrance in the development, as the models will be too big or depend too much on each other and it will be more difficult to introduce new features to the system over time*
- *Once you have modeled your component and exported it, I can imagine that you will often end up fighting against whatever framework you actually use. It is always a sign of bad crafts when you end up fighting against the framework. Likewise, I believe that no matter what formalism you use, you will end up fighting the MBSE tool every time you hits a use case that is outside of the norm*

However, the developers could also see when MBSE would be beneficial to use, saying:

- *If you use it to make small prototypes to validate the understanding of a problem domain, MBSE will make it much easier. However, you will probably take the models and hand implement them when you go away from the prototype phase*
- *MBSE will be good to use when you sketch your ideas among developers where you would use a tool that could be an executable whiteboard sketch and get validation immediately*

The developers feel that MBSE's biggest benefit is in starting a system project or developing smaller atomic systems. We mean that this is a good insight into what the developers feel about the useability of an MBSE tool. The developers have an understanding that MBSE means a lower option for maintenance and expanding of systems. Furthermore, this can indicate that MBSE still suffers from immature tools compared to tools developers are used to in traditional development. Finally, this shows that the developers care about other details than getting the software to work. From all of this, we believe that in our proof of concept, it is a fair assumption that there would be a cut-off point where the tool will become a hindrance. However, we think that this might not be true in general for MBSE approaches, but just an indication that our proof of concept is not mature enough for real-life projects. It would be advantageous to create an experiment with a bigger project where one would try to identify which features a tool should have to create a big project.

Regarding the argument to fight against the framework to get the auto-generated code to work, we believe that our proof of concept tool has introduced this problem. However, we mean that an MBSE tool should not integrate with other frameworks after the code generation. Instead, it should export final projects, which we already see, such as in OutSystems [3].

Regarding fighting the MBSE tool, if you try to implement something outside of the formalism, we think that this is the price you need to pay for early adopters of such technology. Furthermore, this also happens in traditional development, which has multiple years of practical uses, and we still see new languages that try to fix the problem with the "perfect language." We believe that we would be able to improve formalisms for modeling iteratively, just as programming languages, and then slowly removing the pain points. However, before we can do this, we need developers to use MBSE to find these pain points and then solve them.

From all of this, we find it interesting that both developers and non-developers recognize that MBSE will allow for making prototypes and communicating about solutions faster than traditional techniques. However, they cannot see the possibility that the speed will be faster in system development. Here we see two causes which we suspect are the reason why, 1) MBSE tools are not mature enough at this time to get traditional developers to take these tools seriously, 2) developers' general distrust of auto-generated code.

The participants discussed how MBSE affects their mental burden under the development of systems with arguments:

- *There is missing a way to define an unknown component that should have a certain interface, as we see it with dependency injection in traditional programming. If I model something which should use another component, then I do not care about how the component works, I am only interested in the interface, so my only option is to create a new component to get going, which is a big mental break, compared to get the overview of a vertical slice*
- *It quickly gets unmanageable in which dependencies a given component has. I am missing a better hierarchy structure, where it would be more simple and obvious*

We want to dive deeper into these arguments and understand how MBSE affects the mental burden for developers. The first argument about missing a construct for dependency injection, we see as a major missing feature, since we could observe that doing the tasks, the developers had issues starting a new task, as they had to work in a top-down manner. As a result, the developers needed to decide important details early on, which can have undesired effects on later stages of the model. Therefore, we see it beneficial to introduce a construct for handling dependencies in our formalism. Such as dependency construct could be introduced by specifying a placeholder interface automata that could be "swapped" with a real model later. This idea is possible, but our visual tool and visual formalism do not have any constructs to allow this. Because of this, we see this as an important feature to introduce in the future, which would enable developers to work more encapsulated.

Regarding the unmanageable hierarchy structure issue, we recognize that our visual tool did not help developers. For example, none of the developers realized that they could zoom in on a composite component to see the details inside. Therefore when building a visual tool, we recommend researching if other visual hierarchical methods are available other than zooming or make it intuitive and robust to utilize the zoom effect. A visual tool must have a good hierarchy system to manage complex models.

The participants also came with a point regarding the communication between components, with the argument:

- *It was not easy to get an overview of a component when it needed to communicate with another. The communication was made through the naming convention. It would be better if you could connect which outputs should go to which input and vice versa*

We agree that this was something that is confusing in our system. The only reason why we did not find a better solution to this is that our tool is just a proof of concept. It would be beneficial if the users could drag and drop which outputs should talk with which inputs or open up the possibility that when creating an input, the user should select which outputs it is connected to and which component.

The last point we will highlight from the discussion is whether MBSE would make it easier to include the clients into the development process, which is wished in an agile work process. The developers said this:

- *It depends on the client, compared to how much time they want to invest in learning and to understand the concepts, some might like it, whereas others do not want to use their time on it*

This point is somewhat worrying as [22] shows that the clients' involvement in a software project has a big effect, and one of the benefits which are often highlighted in modeling is that it makes it easier to communicate with the clients. Therefore we believe that it should be examined if using a visual modeling tool would help to involve the clients more.

8 CONCLUSION

When building web-based user interfaces, companies utilize libraries or frameworks to get pre-built components, allowing the companies to build software faster. Furthermore, utilizing low-code platforms can allow even faster development cycles and even enable non-developers to create end-to-end software solutions.

Customization can be hard when using libraries, frameworks, or low-code platforms. Sometimes even resulting in developers fighting against the tools to realize their requirements. Therefore we introduce a general use ecosystem for building web-based user interfaces that allow developers to create their tool-chain. This ecosystem contains a visual formalism and a compiler to translate the graphical models to an Abstract Behaviour Tree. Developers can then traverse the Abstract Behaviour Tree to generate the desired platform-specific code.

Furthermore, the ecosystem allows an open-source community to grow around it with many different visual modeling tools, code generators, model checking engine extensions, etc. The point is that using an ecosystem, as presented in this paper, will allow cross-compatibility between many tools, which we believe is the first step to achieve general adoption of Model-Based Software Engineering.

On evaluating our qualitative study, we first would like to point out that it would have been beneficial to have more participants. We originally aimed for 15-20 participants, but most participants could not attend due to COVID-19 restrictions. Unfortunately for our study, but we still believe that the few participants we had provided valuable insights.

Therefore we would like to conclude our hypothesis from the discussion in Section 7.

H1: *Using MBSE, developers can create correct behaving software solutions faster.* Per our participants, this is only true in a limited scope for developing prototypes, small isolated systems, or for defining requirements. So, yes, MBSE will allow developers to be faster. But for larger systems or long-lived systems, it is suspected that it would make the process slower. But slower does not mean worse, and we suspect that given enough time, it would be possible to create a tool that could mitigate the decrease in speed.

H2: *The use of MBSE provides better communication between developers and designers.* It is hard to derive a clear answer from our study. On the one hand, the designer expressed difficulties understanding the visual formalism, especially when talking about transitions. However, on the other hand, throughout the four tasks, we observed that the designer was quite involved in the process. We had chosen to pair the designer with a developer and have them work together on the tasks. The level of involvement from the designer gave us an indication that they were able to understand and work with the developer to define the behavior. Also, we felt that the designer could envision the different visual "states" of the component. However, through the discussion, it did not seem like the designer shared our insights. Therefore we are hesitant to conclude a definitive answer. We believe, however, that through training, the hypothesis might show to be true.

H3: *The use of MBSE provides better communication between developers and stakeholders.* Our participants did not necessarily agree with this hypothesis. They express concerns about whether clients would invest the time needed to understand the visual formalism and review them with the developers. However, the participants believe that as an internal tool, it would strengthen the communication. Therefore we believe that this is true for internal stakeholders, just like Harel [10] observed in his work.

H4: *MBSE makes it easier for developers to create complex systems.* With the discussion from our study, we have to assert this hypothesis as false. However, we suspect that this might be due to our specific proof-of-concept tool and not necessarily true for any MBSE tool. There might be value in conducting a study with multiple modern MBSE tools to see how our findings compare with more mature tools.

H5: *People without experience or training in software development principles can utilize an MBSE tool to create software systems.* The consensus of the participants was that this might be true, as long as the desired software system is a prototype or has a very small scope. With the tasks that we had the participants do, all non-developers could achieve the desired behavior, but it is unclear how they would fair building a larger system.

H6: *MBSE is difficult to adopt for people without formal training in the discipline.* Through the study, we observed a significant difference in the intuition between the people that had some experience with modeling and those that had no experience. Although the people without experience could create the correct models, they never understood why the model worked, and they tended to trial and error their way into the correct solution. In comparison, the people with experience were more methodic in their approach. The difference between experienced and inexperienced indicates that

although people without training in the discipline can utilize an MBSE tool, it would never become their tool of choice without some form of training.

H7: *Traditional developers will have reservations and feel some friction in adopting an MBSE approach.* Throughout our discussion in Section 7, it is clear that the traditional developers have many reservations and friction when it comes to using an MBSE approach. Therefore it is obvious to us that this hypothesis is true.

H8: *MBSE gives developers more confidence in their work.* On face value, it seems from the discussion that this hypothesis might be false. However, an interesting detail is that many expressions of lack of confidence were not about the developers' work but rather about the generated code and so forth. Furthermore, doing the tasks, we did not observe the developers ever having to question why something did not work. As soon as they came to generating code, they all integrated it into the javascript project without any major issues. So, we believe that the developers had great confidence in their work, but they were unsure of the quality of the tool that they used. The only detail that cannot be overlooked is that they expressed a lack of quality measurement when creating a model. They felt that they did not have any way to ensure the model itself was good enough. So we believe that a mature MBSE tool with some quality measures built-in would give developers more confidence in their work.

We conclude that traditional developers experience friction when using an MBSE approach from the answers to our hypothesis. We suspect this friction is due to; the lack of mature MBSE tools, and the lack of big project examples using MBSE showcasing the capabilities and advantages. A majority of the issues expressed by the participants can be attributed to our proof-of-concept tool. Due to this, we still believe that MBSE could deliver on many if not all of the supposed benefits given more effort in building a mature tool. Therefore, we conclude that it would be beneficial for developers to adopt an MBSE approach if a mature tool were to exist.

9 FUTURE WORK

There are a couple of areas that we find of interest for future work. Firstly, the current visual formalism has no defined semantic behind it. We find it essential for any formalism to have a clearly defined semantic. Therefore it would be valuable to look into defining the semantics of our visual formalism. On the visual formalism, we are also in doubt regarding the feature set currently included. Therefore, it would be beneficial to study our visual formalism to identify and develop missing features or generally improve it. Also, our formalism has no notion of time. Time is a crucial part of modern web development, so it would be important to introduce time into our formalism.

The current state of the compiler might not be to an industry quality. Therefore we also want to work on improving the compiler. One key aspect would be introducing a plug-in system that would allow third-party plug-ins to be added at runtime instead of at compile time.

Late into the project, we realized that the Spin [16] model checker engine might not be a natural choice for our formalism. Therefore, it would benefit from integrating or even developing a new model checker engine with a more natural fit. On the notion of a model checker engine, it is currently a major lack that we do not facilitate any language for formal verification of properties in a model. Therefore we need to implement the support for a query language such as Computation Tree Logic (CTL), Linear Temporal Logic (LTL), or something else.

In order for such an ecosystem, as presented in this paper, to gain any adoption, it is important to have a strong toolset surrounding the ecosystem. Therefore, we want to develop a tool-chain with mature and robust tools for Modeling behavior, structuring HTML elements, and styling those elements, alongside tools for model checking and code generators for vanilla JavaScript alongside the major JavaScript frameworks.

Lastly, due to the impact COVID-19 had on our case study, we would like to repeat the case study with more participants in the future. We would also like to have a long-running case study with one or more companies to get more in-depth insights into how MBSE would affect a development workflow.

REFERENCES

[1] Bootstrap homepage. https://getbootstrap.com/.
[2] Headless ui homepage. https://headlessui.dev/.
[3] Outsystems homepage. https://www.outsystems.com/.
[4] Mendix homepage. https://www.mendix.com/.
[5] Bran Selic. Personal reflections on automation, programming culture, and model-based software engineering. *Automated Software Engineering*, 15(3):379–391, Dec 2008.
[6] T. Weigert and F. Weil. Practical experiences in using model-driven engineering to develop trustworthy computing systems. In *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)*, volume 1, pages 8 pp.–, 2006.
[7] Gartner forecasts worldwide low-code development technologies market to grow 23% in 2021. https://www.gartner.com/en/newsroom/press-releases/2021-02-15-gartner-forecasts-worldwide-low-code-development-technologies-market-to-grow-23-percent-in-2021, 2021.
[8] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178, 2020.
[9] Xstate homepage. https://xstate.js.org/.
[10] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
[11] Ragnhild Van Der Straeten, Tom Mens, and Stefan Van Baelen. Challenges in model-driven software engineering. *Models in Software Engineering*, pages 35–47, 2009.
[12] Tailwindcss homepage. https://tailwindcss.com/.
[13] Ajv json schema validator homepage. https://ajv.js.org/.
[14] Mikkel Pedersen and Morten Hartvigsen. Robust ui - when the ui becomes wonky, robustui makes it funky. https://projekter.aau.dk/projekter/files/402371631/SV902E20.pdf.
[15] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, September 2001.
[16] Spin homepage. http://spinroot.com/spin/whatispin.html.
[17] Treantjs - javascipt library for visualization of tree diagrams. https://fperucic.github.io/treant-js/.
[18] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.
[19] Ivan Aaen. Essence: facilitating software innovation. *European journal of information systems*, 17(5):543–553, 2008.
[20] Wojciech J. Dzidek, Erik Arisholm, and Lionel C. Briand. A realistic empirical evaluation of the costs and benefits of uml in software maintenance. *IEEE Transactions on Software Engineering*, 34(3):407–432, 2008.
[21] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson, 2008.
[22] Rashina Hoda, James Noble, and Stuart Marshall. The impact of inadequate customer collaboration on self-organizing agile teams. *Information and Software Technology*, 53(5):521–534, 2011. Special Section on Best Papers from XP2010.

A JSON SCHEMA

Listing 5: JSON schema for a Simple component

```
{
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://example.com/object1612866524.json",
  "title": "1",
  "type": "object",
  "required": [
    "label",
    "initialState",
    "states",
    "events",
    "inputs",
    "outputs",
    "transitions",
    "positions"
  ],
  "properties": {
    "type": {
      "$id": "#root/type",
      "title": "Type",
      "type": "number",
      "default": "",
      "pattern": "^.*$"
    },
    "type": {
      "$id": "#root/type",
      "title": "Type",
      "type": "number",
      "default": ""
    },
    "label": {
      "$id": "#root/label",
      "title": "Label",
      "type": "string",
      "default": "",
      "pattern": "^.*$"
    },
    "initialState": {
      "$id": "#root/initialState",
      "title": "Initialstate",
      "type": "string",
      "default": "",
      "pattern": "^.*$"
    },
    "states": {
      "$id": "#root/states",
      "title": "States",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/states/items",
        "title": "Items",
        "type": "object",
        "required": [
          "label",
          "type"
        ],
        "properties": {
          "label": {
            "$id": "#root/states/items/label",
            "title": "Label",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          },
          "type": {
            "$id": "#root/states/items/type",
            "title": "Type",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          }
        }
      }

    },
    "events": {
      "$id": "#root/events",
      "title": "Events",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/events/items",
        "title": "Items",
        "type": "string",
        "default": "",
        "pattern": "^.*$"
      }
    },
    "inputs": {
      "$id": "#root/inputs",
      "title": "Inputs",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/inputs/items",
        "title": "Items",
        "type": "string",
        "default": "",
        "pattern": "^.*$"
      }
    },
    "outputs": {
      "$id": "#root/outputs",
      "title": "Outputs",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/outputs/items",
        "title": "Items",
        "type": "string",
        "default": "",
        "pattern": "^.*$"
      }
    },
    "transitions": {
      "$id": "#root/transitions",
      "title": "Transitions",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/transitions/items",
        "title": "Items",
        "type": "object",
        "required": [
          "from",
          "label",
          "to"
        ],
        "properties": {
          "anchorPoint":{
            "$id": "#root/transitions/items/anchorPoint",
            "title": "anchorPoint",
            "type": "object",
            "required": [
              "x",
              "y"
            ],
            "properties": {
              "x": {
                "$id": "#root/transitions/items/x",
                "title": "x",
                "type": "number",
                "default": "",
                "pattern": "^.*$"
              },
              "y": {
                "$id": "#root/transitions/items/y",
                "title": "y",
                "type": "number",
                "default": "",
                "pattern": "^.*$"
              }
            }
          },
          "from": {
            "$id": "#root/transitions/items/from",
            "title": "From",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          },
          "label": {
            "$id": "#root/transitions/items/label",
            "title": "Label",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          },
          "to": {
            "$id": "#root/transitions/items/to",
            "title": "To",
            "type": "string",
```

```
            "default": "",
            "pattern": "^.*$"
          }
        }
      }

    },
    "positions": {
      "$id": "#root/positions",
      "title": "Positions",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/positions/items",
        "title": "Items",
        "type": "object",
        "required": [
          "label",
          "x",
          "y",
          "width"
        ],
        "properties": {
          "label": {
            "$id": "#root/positions/items/label",
            "title": "Label",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          },
          "x": {
            "$id": "#root/positions/items/x",
            "title": "X",
            "type": "number",
            "default": 0
          },
          "y": {
            "$id": "#root/positions/items/y",
            "title": "Y",
            "type": "number",
            "default": 0
          },
          "width": {
            "$id": "#root/positions/items/width",
            "title": "Width",
            "type": "integer",
            "default": 0
          }
        }
      }

    }
  }
}
```

Listing 6: JSON schema for a Composite component

```
{
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://example.com/object1613469890.json",
  "title": "2",
  "type": "object",
  "required": [
    "label",
    "type",
    "components",
    "inputs",
    "outputs",
    "positions"
  ],
  "properties": {
    "label": {
      "$id": "#root/label",
      "title": "Label",
      "type": "string",
      "default": "",
      "pattern": "^.*$"
    },
    "type": {
      "$id": "#root/type",
      "title": "Type",
      "type": "integer",
      "default": 0
    },
    "components": {
      "$id": "#root/components",
      "title": "Components",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/components/items",
        "title": "Items",
        "type": "object",
        "required": [
          "label",
          "type"
        ],
        "properties": {
          "label": {
            "$id": "#root/components/items/label",
            "title": "Label",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          },
          "type": {
            "$id": "#root/components/items/type",
            "title": "Type",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          }
        }
      }

    },
    "inputs": {
      "$id": "#root/inputs",
      "title": "Inputs",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/inputs/items",
        "title": "Items",
        "type": "string",
        "default": "",
        "pattern": "^.*$"
      }
    },
    "outputs": {
      "$id": "#root/outputs",
      "title": "Outputs",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/outputs/items",
        "title": "Items",
        "type": "string",
        "default": "",
        "pattern": "^.*$"
      }
    },
    "positions": {
      "$id": "#root/positions",
      "title": "Positions",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/positions/items",
        "title": "Items",
        "type": "object",
        "required": [
          "label",
          "x",
          "y",
          "width"
        ],
        "properties": {
          "label": {
            "$id": "#root/positions/items/label",
            "title": "Label",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          },
          "x": {
            "$id": "#root/positions/items/x",
            "title": "X",
            "type": "number",
            "default": 0
          },
          "y": {
            "$id": "#root/positions/items/y",
            "title": "Y",
            "type": "number",
```

```
            "default": 0
          },
          "width": {
            "$id": "#root/positions/items/width",
            "title": "Width",
            "type": "number",
            "default": 0
          }
        }
      }

    }
  }
}
```

Listing 7: JSON schema for a Selective component

```
{
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://example.com/object1614245601.json",
  "title": "3",
  "type": "object",
  "required": [
    "label",
    "type",
    "initialCase",
    "observer",
    "cases",
    "inputs",
    "outputs",
    "positions"
  ],
  "properties": {
    "label": {
      "$id": "#root/label",
      "title": "Label",
      "type": "string",
      "default": "",
      "pattern": "^.*$"
    },
    "type": {
      "$id": "#root/type",
      "title": "Type",
      "type": "integer",
      "default": 0
    },
    "initialCase": {
      "$id": "#root/initialCase",
      "title": "initial Case",
      "type": "string",
      "default": "",
      "pattern": "^.*$"
    },
    "observer": {
      "$id": "#root/observer",
      "title": "Observer",
      "type": "object",
      "required": [
        "input",
        "dataType"
      ],
      "properties": {
        "input": {
          "$id": "#root/observer/input",
          "title": "Input",
          "type": "string",
          "default": "",
          "pattern": "^.*$"
        },
        "dateType": {
          "$id": "#root/observer/dataType",
          "title": "Datatype",
          "type": "string",
          "default": "",
          "pattern": "^.*$"
        }
      }
    }
  ,
    "cases": {
      "$id": "#root/cases",
      "title": "Cases",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/cases/items",
        "title": "Items",
        "type": "object",
        "required": [
          "guard",
          "type"
        ],
        "properties": {
          "guard": {
            "$id": "#root/cases/items/guard",
            "title": "Guard",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          },
          "label": {
            "$id": "#root/cases/items/label",
            "title": "Label",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          },
          "type": {
            "$id": "#root/cases/items/type",
            "title": "Type",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          }
        }
      }

    },
    "inputs": {
      "$id": "#root/inputs",
      "title": "Inputs",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/inputs/items",
        "title": "Items",
        "type": "string",
        "default": "",
        "pattern": "^.*$"
      }
    },
    "outputs": {
      "$id": "#root/outputs",
      "title": "Outputs",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/outputs/items",
        "title": "Items",
        "type": "string",
        "default": "",
        "pattern": "^.*$"
      }
    },
    "positions": {
      "$id": "#root/positions",
      "title": "Positions",
      "type": "array",
      "default": [],
      "items":{
        "$id": "#root/positions/items",
        "title": "Items",
        "type": "object",
        "required": [
          "label",
          "x",
          "y",
          "width"
        ],
        "properties": {
          "label": {
            "$id": "#root/positions/items/label",
            "title": "Label",
            "type": "string",
            "default": "",
            "pattern": "^.*$"
          },
          "x": {
            "$id": "#root/positions/items/x",
            "title": "X",
            "type": "number",
            "default": 0
          },
          "y": {
            "$id": "#root/positions/items/y",
```

```
            "title": "Y",
            "type": "number",
            "default": 0
          },
          "width": {
            "$id": "#root/positions/items/width",
            "title": "Width",
            "type": "number",
            "default": 0
          }
        }
      }

    }
  }
}
```

B COMPILER NODES

B.1 Composite Component Node

The Composite Component node contains a list of children which are all the modules that are to run orthogonal inside this component. Besides that, it includes a reference to its parent, i.e. the Module Node. This reference has shown through empirical testing to be an excellent property to have for some code generators.

Listing 8: CompositeComponentNode in pseudocode

```
class CompositeComponentNode {
  var name: String;
  var type = "CompositeComponentNode";
  var children: List<ModuleNode>;
  var typeLookUpTable: Map<String, String>;
  var parent: ModuleNode;
}
```

B.2 Selective Component Node

The Selective Component node contains a list of children which are all the cases that the selective entry should consider. Besides that, it includes a reference to its parent, i.e. the Module Node. This reference has shown through empirical testing to be an excellent property to have for some code generators.

Listing 9: SelectiveComponentNode in pseudocode

```
class SelectiveComponentNode {
  var name: String;
  var type = "SelectiveComponentNode";
  var children: List<CaseNode>;
  var typeLookUpTable: Map<String, String>;
  var parent: ModuleNode;
}
```

B.3 Case Node

The Case Node represents a case in the Selective Component. It contains a component property that holds a ModuleNode and means which component should activate if the guard is true. The Guard Node is in the guard property. Besides this, it also contains an initial property that is true if this particular case should be considered the default case, either at bootup or if no other guard is true.

Listing 10: CaseNode in pseudocode

```
class CaseNode {
  var name: String;
  var type = "CaseNode";
  var guard: GuardNode;
  var component: ModuleNode;
  var typeLookUpTable: Map<String, String>;
  var parent: ModuleNode;
  var initial: Boolean;
}
```

B.4 Guard Node

The Guard Node contains the information necessary to evaluate whether a given case should be active or not. The property stream contains the name of the input message channel that includes the data to assess. The property guard contains the expression that should use to evaluate the boolean result.

Listing 11: GuardNode in pseudocode

```
class GuardNode {
  var name: String;
  var type = "GuardNode";
  var stream: String;
  var guard: Expression;
  var typeLookUpTable: Map<String, String>;
  var parent:CaseNode;
}
```

B.5 State Node

The State Node represent a specific state that a simple component can be in. The property children contain all the allowed transition away from the particular state.

Listing 12: StateNode in pseudocode

```
class StateNode {
  var name: String;
  var type = "StateNode";
  var children: List<TransitionNode>;
  var typeLookUpTable: Map<String, String>;
  var parent: SimpleComponentNode;
}
```

B.6 Transition Node

The Transition Node contains all the information to perform a transition internally in a Simple Component. The property "from" contains the name of the source state, the label is the event that should happen for activation, and the "to" property has the name of the target state.

Listing 13: TransitionNode in pseudocode

```
class TransitionNode {
  var name: String;
  var type = "TransitionNode";
  var from: String;
  var label: String;
  var to: String;
  var typeLookUpTable: Map<String, String>;
  var parent: SimpleComponentNode;
}
```

B.7 Stream Node

The Stream Node symbolizes a communication "direction", either input, output or environment. In our use case, the environment is the stream of browser events that a Module might need for transitions inside the component of its body. The property children contain a list of identifiers that are the name of the streams.

Listing 14: StreamNode in pseudocode

```
class StreamNode {
  var name: String;
  var type = "StreamNode";
  var children: List<IdentifierNode>;
  var typeLookUpTable: Map<String, String>;
  var parent: ModuleNode;
}
```

B.8 Identifier Node

The Identifier Node simply holds an identifier name in the name property. This node is helpful since the identifiers might be have meaning outside the generated code.

Listing 15: IdentifierNode in pseudocode

```
class IdentifierNode {
  var name: String;
  var type = "IdentifierNode";
  var typeLookUpTable: Map<String, String>;
  var parent: StreamNode;
}
```

C TASKS

Listing 16: Task two

```
In this task, you should create a "composite" component where you
      should make a dropdown button with the following
      functionalities:

* When you hover one of the options, then the option should be
      highlighted
* When you press the button, then the options should be visible
* If you press the dropdown menu button, then the options should
      disappear or if you press outside of the dropdown button
* If you press one of the options, then the menu should disappear,
       and the selected option should be logged

You have got the needed HTML and CSS to solve the task. Your task
      is to model this component, paste it into the given project,
      and bring everything to work with the functionalities
      mentioned above.
```

Listing 17: Task three

```
In this task, you should create a "selective" component with the
      following functionalities:

* A input field where the user can write some text
* If you write "1", then there should be shown some text, and if
      there is nothing or you write anything else than "1", then
      another piece of text should be shown

You have got the needed HTML and CSS to solve the task. Your task
      is to model this component, paste it into the given project,
      and bring everything to work with the functionalities
      mentioned above.
```

Listing 18: Task four

```
This task is optional

In this task, you should use everything you have learned in the
      previous tasks. The task is about showing two components
      depending on what the user writes. It should have the
      following functionalities:

* A text input field where if you write "1", then the toggle
      component(Task 1) should be shown, and if there is nothing or
       you write anything else than "1", then the dropdown
      component(Task 2) should be shown
* The Toggle component should work as described in Task 1
* The dropdown component should work as described in Task 2

You have got the needed HTML and CSS to solve the task. Your task
      is to model this component, paste it into the given project,
      and bring everything to work with the functionalities
      mentioned above.
```