COMMUNICATING STATECHARTS (CSC)

COMMUNICATING STATECHARTS (CSC)

By SHEIDA EMDADI, BEng

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Science - Computer Science

McMaster University

MASTER OF SCIENCE - COMPUTER SCIENCE (2024)

Hamilton, Ontario, Canada (Dept of Computing and Software)

TITLE:	Communicating Statecharts (CSC)
AUTHOR:	Sheida Emdadi BEng (Computer Engineering), Islamic Azad University, Tehran, Iran
SUPERVISOR:	Dr. Spencer Smith Dr. Christopher Anand
NUMBER OF PAGES:	xxv, 179

Abstract

Concurrency is increasingly gaining importance due to the rapid development of networked applications. However, concurrency comes with some complexities, including handling race conditions or deadlocks. Therefore, learning this concept and understanding its correct implementation is challenging, even for experienced programmers. This problem arises from the current practices of teaching concurrency because of the focus on confusing details instead of necessary concepts.

We propose a new concurrency paradigm called Communicating StateCharts (CSC) to simplify the teaching of concurrency to beginner programmers. CSC preserves five main principles, aiming to make concurrency easier to learn and use for novices: software visualization, Model-Driven Development (MDD), pure functions, separation of concerns, and raising abstraction levels. In this regard, CSC adapts features from existing concurrency models that aligned with our principles, namely process calculi, the actor model, and Harel's statecharts. This synthesis led to CSC's atomic statecharts, communicating through messages transmitted via channels.

To make CSC accessible for beginners, a visual MDD tool called CSCDraw is designed and developed. The main requirements that guided the design of CSCDraw include enforcing CSC principles, considering beginner-friendly features, ensuring faithful code generation, supporting conditional branches, and channel cardinality.

We also present the design of a pilot study that investigates the most effective way of teaching CSC to beginning programmers. This study serves as a prelude to a more rigorous experiment to compare the effectiveness of CSC with the existing paradigms.

Dedication

I extend my heartfelt gratitude to my supervisors, Dr. Spencer Smith and Dr. Christopher Anand. Dr. Smith's dedication, insightful guidance, steadfast support, and encouragement have continually motivated me to aim high, even when I was struggling to believe in myself. Dr. Anand's unwavering support, motivating advice, and constant belief in my capabilities and academic potential gave me the strength to persevere, even during my lowest moments. Thank you both for being such dreamy, amazing supervisors.

A special thanks to my friends Narges, her husband Omid, and Tina for their wonderful friendships, and to my lifelong friends Bahar and Mahshid for always being there for me and providing motivation when it felt impossible to continue.

I also want to express my deepest appreciation to my parents for their love and support, especially to my Father, without whose unbounded support, I could never dream so ambitiously.

Finally, I am sincerely grateful to my sweet little sister, Bahar, whose love and presence in my life have always warmed my heart.

This thesis is a reflection of the love and support that I received from each of you. Love you all!

Table of Contents

List of Figures

List of Tables

Notation

Q	A finite set of states, usually represented by circles labelled with unique strings.
Σ	A finite set of inputs that trigger a transition.
δ	A transition function $\delta : \Sigma \times Q \to Q$.
q_0	The initial state, $q_0 \in Q$.
ε	Element indicating that no synchronizing message is to be sent.
G	A countable set of global states.
Γ	A countable set of global messages that trigger a transition, $\varepsilon \notin \Gamma$.
$\delta_G : \Lambda \times L \times G \to G \times (\Lambda_U \cup \{\varepsilon\})$	A transition function also called the "global update" function.
$g_0 \in G$	The initial global state.
L	A countable set of local states.
Λ	A countable set of local messages that trigger transitions.

Λ_G The subset of messages which can be sent by δ_G.

Λ_U The subset of messages which can be triggered by a user action.

$\delta_L : \Lambda \times G \times L \to L \times (\Gamma \cup \{\varepsilon\})$**:**

A transition function also called the "local update" function.

$\nu : L \to 2^{\Lambda_U}$ A function that for each local state determines which local messages could be generated by the user.

$l_0 \in L$ The initial local state.

Definitions

Beginner/Novice Programmer:

We define the term *beginner/Novice Programmer* as a subject who

- has completed no more than one programming course at the university level, and is unable to write a program with at least 100 lines of code in a text-based programming language, or
- has completed no programming course at the university level, and is unable to write a program with at least 100 lines of code in a text-based programming language, or
- would take a first-year programming course in the next academic year, and is unable to write a program with at least 100 lines of code in a text-based programming language, or
- is unable to write a program with at least 100 lines of code in a text-based programming language.

Concurrency Paradigm:

We define the term *Concurrency Paradigm* as a high-level abstract definition of a model that describes the structure of a concurrent system and the execution of its components. In other words, a concurrency paradigm covers the definition of the interaction between the components of a concurrent system, the actions that shape the executions of different operations, and the mechanisms for synchronization. This high-level definition captures the level of concurrency models, such as the actor model, process calculi, or Petri nets.

Abbreviations

CCS	Communicating Systems
CSC	Communicating Statecharts
CSP	Communicating Sequential Processes
DbC	Design by Contract
EDP	Event-Driven Programming
FSM	Finite State Machine
GUI	Graphical User Interface
IDE	Integrated Development Environment
LG-MVU	Local-Global Model-View-Update
MDD	Model-Driven Development
MP	Multi-Player
MVU	Model-View-Update
MVC	Model-View-Controller

OO	Object-Oriented
OOP	Object-Oriented Programming
SC	StateChart
SCOOP	Simple Concurrent Object-Oriented Programming
SoC	Separation of Concerns
SOA	Service-Oriented Architecture

Declaration of Academic Achievement

I, Sheida Emdadi, confirm that the work presented in this thesis is mine. Wherever information has been derived from other sources, I have cited those sources accordingly.

Chapter 1

Introduction

In today's increasingly networked world, almost every application depends on concurrency. However, using concurrency requires handling complexities such as race conditions or deadlocks. This introduces challenges in learning this concept and implementing it correctly, even for experienced programmers [68]. The Turing Award winner, Lamport [46], says "I have worked with a number of computer engineers (both hardware and software engineers), and I have seen what they knew and what they didn't know that I felt they should have. [...] I can't claim to know the best way to teach computer engineers how to cope with concurrency. I do know that what they seem to be learning now is not helping them very much." According to Lamport, concurrency should not be as hard as it currently appears to computer engineers [46]. This is because they are focusing on the details of the programming language instead of the important concepts. This problem motivated us to reduce the barrier for beginner programmers. Therefore, in this thesis, we propose a new visual paradigm, called Communicating StateCharts (CSC), that aims to simplify concurrency to the level that even novice programmers can understand and use it.

CSC tries to achieve its goal of helping beginners by considering principles that contribute to reducing complexities, including raising the abstraction levels, visualizing the system, and generating purely functional, event-driven skeleton code based on the designed model. Event-Driven Programming (EDP) offers easier concurrency by increasing abstraction through using events instead of threads, and allows programmers to make interactive applications such as games, which is a strategy used to motivate beginners [50]. To further increase the engagement of EDP, and exploit its success in making concurrency more straightforward, we put the focus of CSC on allowing the visual design of event-driven multi-user, distributed applications including multi-player games, and generate skeleton code from that graphical representation. We believe that visualization will help students with concurrency concepts analogously to how visualization has helped students correct misconceptions about object-oriented programming [85]. We anticipate that visualization will contribute to smoothening the path toward learning concurrency for novice programmers by reducing the cognitive load required to understand the behaviour of the concurrent systems.

In this chapter, we describe how Lamport's [46] criticism of the existing methods of teaching concurrency inspired us to simplify concurrency for beginners. Then, we introduce the three research questions that are used to structure and motivate the research. The first question focuses on identifying the five principles of a concurrency paradigm suitable for beginners. We show that the existing tools and frameworks do not satisfy our paradigm's principles. Then, we develop a new paradigm, called CSC, that answers the five principles. CSC combines existing models of concurrency (i.e.,

process calculi and the actor model) in a visual language including modified state-charts communicating asynchronously through channels, to satisfy those principles.

The second research question motivates the visual MDD tool, called CSCDraw. This new tool aims to support CSC by considering the requirements necessary for making it straightforward for beginning programmers to learn and use it.

The third research question inspires the design of a pilot study, which works as a prelude to evaluating whether CSC really simplifies concurrency contrasted to the existing frameworks. More specifically, this pilot study answers the question of finding the best way to teach CSC, which is critical for us to learn before performing a rigorous experiment. After that, we describe the contributions made through this work.

1.1 Motivation

Concurrency is one of the necessary concepts that software engineers and computer scientists need to master. It is increasingly important due to the development of multi-core architectures and the proliferation of networked applications that require concurrent programming in the software development process. However, concurrency can be challenging to learn. As Nanz et al. [68] state "Concurrent programming is, notoriously hard even for expert programmers". There is a long history of attempts to simplify concurrency—usually by raising the level of abstraction. In particular, concurrency models such as *Java Threads* and *SCOOP* were designed for Object-Oriented (OO) languages, Java and Eiffel, respectively [58, 70]. Also, there are several attempts through functional programming languages, including Haskell [36, 51]. Although these approaches were successful in making concurrency easier, concurrency is

still a challenge, and, not surprisingly, a particular challenge for beginners. Our goal in this thesis is to lower the barrier to understanding and using concurrency to the extent that it is feasible for beginners. As mentioned before, Lamport [46] believes this complicacy can be handled by changing the way that concurrency is being taught. This inspired us to move the focus from language details to higher-level concepts that are necessary for effectively learning and using concurrency.

1.2 Research Questions

In this section, we present three research questions guiding the study of this thesis. They will be referred to throughout the thesis by their RQ numbers.

RQ1

What does a beginner-friendly paradigm for distributed user-interface programs look like?

This research question explores the features of a concurrency paradigm that is suitable for novice programmers to learn and use. This paradigm focuses on distributed user-interface programs, e.g., multi-player games, since these interactive apps are motivating for beginners to learn. As mentioned in the previous section, the existing frameworks improve the teaching of concurrency, but not to the extent that is easy for beginners to learn.

RQ2

How best to implement a design tool for the paradigm from RQ1 to make it accessible to beginners?

RQ3

How does the proposed paradigm from RQ1 compare to traditional paradigms for teaching beginners?

RQ3.1

How to effectively teach the proposed paradigm?

RQ3.2

How does the effectiveness of the proposed paradigm compare to Java Threads with respect to teaching beginners?

RQ3 seeks to evaluate whether the proposed paradigm is more successful in removing the complexities of concurrency in contrast with the previous works. This question leads to two sub-questions: (1) identifying the best way to teach the paradigm, which prepares us for conducting the evaluation; and, (2) comparing our paradigm with one of the most commonly used concurrency frameworks, Java Threads, which builds the bones for reasoning about the bigger question, RQ3. The evaluation suggested by RQ3.2 is outside the scope of the work completed in this thesis.

1.3 Principles for a Beginner-Friendly Concurrency Paradigm

To answer RQ1, we defined five principles to guide the design of a concurrency paradigm suitable for beginners. Figure 1.1 illustrates these principles, shown in squares, together with their consequent choices, outlined by circles. The first two

principles, *Visualization* and *Model-Driven Development (MDD)*, are tied together to maximize the benefits out of their combination. Additionally, each consequent choice takes into account the previous principles and their implications. In this section, we first describe the principles and then, their consequent choices. They will be referred to throughout the thesis by their P and C numbers, respectively.

1.3.1 Principles

We now present the principles shaping our paradigm, shown in squares in Figure 1.1.

P1. Use Software Visualization: Many researchers have been studying the effectiveness of visualization in supporting teaching software-related concepts. For instance, in surveying the Literature on Event-Driven Programming (EDP), Lukkarinen et al. [50] proposed a question: "How could software visualization support learning EDP-related concepts? What concepts should be visualized and how?" Reinforcing that question, Carro et al. [16] says: "Students often face interaction and temporal issues for the first time in a concurrency course, and many of them find it difficult to visualize concurrent execution." This inspired us to exploit software visualization to support our goal of simplifying concurrency for beginners.

P2. Use Model-Driven Development (MDD): Multiple authors have found MDD to bring practical advantages [56, 79]. E. V. and Samuel [25] explains this succinctly: "The concept of model compilation can reduce the effort we put into coding and testing, and in turn, we can improve the quality of the software. It can reduce the bugs in the developed products. It helps us to refine the requirement specification."

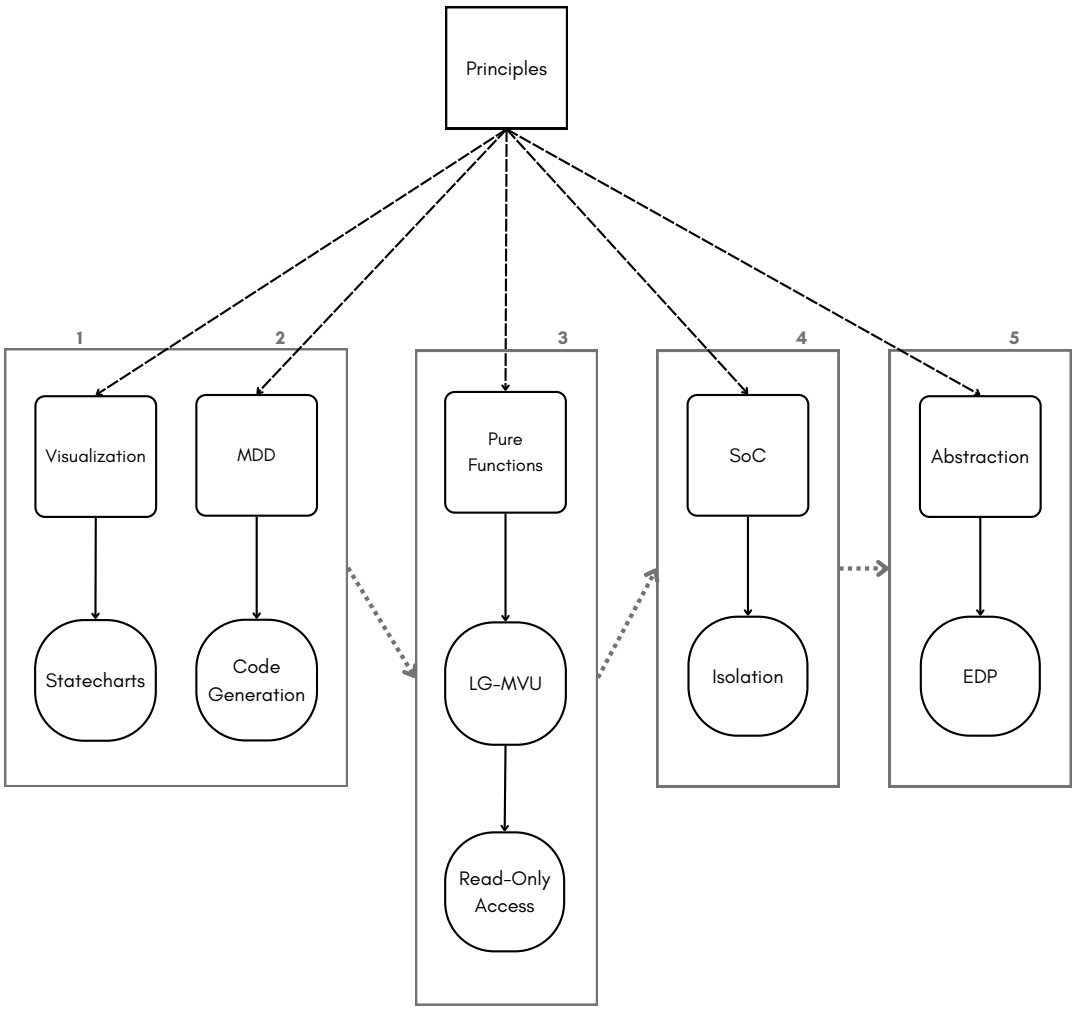


Figure 1.1: Principles for shaping our new paradigm. The first and second principles are tightly coupled. The dashed and solid arrows illustrate the *is a* and *implies* relationships, respectively. The dotted arrows, connecting the grey boxes, display that each principle considers the previous implications while picking a consequent choice.

Our experience in teaching a first-year computer science course at McMaster, Introduction to Software Design Using Web Programming, supports this. Particularly, we used the visual MDD tool, SDDraw [72] (see Section 2.10) to teach students to make single-user games using EDP concepts. As observed by the instructor, almost no students would start coding without first using SDDraw to design the overall structure of their application through state diagrams, including projects with over 10K lines of code.

We should, therefore, tie the Principle P1, *Visualization*, to the Principle P2, *MDD*, resulting in taking the proposed visual model one step further by using it for MDD.

P3. Use Pure Functions: Pure functions are mathematical functions, in which outputs depend only on inputs. Functional programming languages are based on this guarantee, and further benefit by treating functions as values, so they can be both inputs and outputs of other functions. This allows patterns to be explicitly programmed once [42]. A purely functional language is much easier to reason about and design with, by virtue of eliminating side effects. Therefore, the proposed concurrency paradigm should use pure functions.

P4. Separation of Concerns (SoC): The principle of Separation of Concerns (SoC), as described in Section 2.12, is to divide the problem into separate subproblems that can be reasoned about and designed independently one at a time. This principle is useful for all developers, but especially beginners. An example of SoC would be modularization. Tarr et al. [81] says "separation of concerns can provide many software engineering benefits, including reduced complexity, improved reusability, and simpler evolution." The proposed paradigm should

take advantage of SoC in handling the complexity of concurrency.

P5. Use Abstraction to Handle Complexities: As mentioned in the beginning of this chapter, Lamport [46] thinks focussing on confusing details in teaching concurrency has caused difficulties for computer engineers to learn and use this concept. Inspired by his criticism of teaching methods of concurrency, our paradigm should raise the abstraction levels and allow beginners to think above the confusing details.

1.3.2 Principle Consequences

We now describe the consequences of each principle, shown in circles in Figure 1.1.

C1. Visualization through Statecharts: Addressing Principle P1, *Use Software Visualization*, requires detecting an appropriate modelling language. Agreeing with Pérez et al. [73] that "state machine specifications (including UML state machines, finite state machines and Harel statecharts) are considered the most widely used method to specify the dynamic behaviour of reactive systems," our proposed visual paradigm for concurrency, should be based on statecharts.

C2. Code Generation from Visual Models: The principle P2, *Use Model-Driven Development (MDD)*, comes with practically desirable code generation, first used by the Executable UML [56]. Code generation not only relieves beginners from coding drudgery but also protects them against making flawed systems, preventing early frustration. As mentioned before, the Principle P1, *visualization*, and P2, *MDD*, should be combined in our proposed paradigm to maximize the benefits they can offer. This can be achieved by allowing the users

to design the structure and behaviour of the systems by drawing diagrams and then generating code based on their models. Furthermore, a highly trustworthy code generation is possible by using pure functions due to the absence of side effects. This reliability is ensured in our proposed paradigm due to Principle P3, *Use Pure Functions*.

C3. Read-Only Access to Models through Pure Functions: Multi-user concurrency involving Principle P3, *Use Pure Functions*, can be achieved through the newly introduced architecture, LG-MVU, described in Section 2.11, which separates the program into local and global portions, all without losing the transparency afforded by pure functions. A convenient feature of LG-MVU is that the local `view` and `update` functions take both local and global models as inputs. This gives the local statechart read-only access to the state of the global statechart, as shown in Figure 1.2.

C4. Enforcing Isolation of Models to Achieve SoC: In our proposed concurrency paradigm, the Principle P4, Separation of Concerns (SoC), can be achieved by the *isolation* of statecharts. In other words, statecharts should not be able to modify each other directly. They must communicate with each other via messages transmitted through channels. Figure 1.3 illustrates this isolation of the statecharts, through an example. In this figure, the transition highlighted in red is an instance of communication not allowed among our statecharts.

C5. Using Event-Driven Programming (EDP) to Raise Abstraction: To some extent, concurrency is involved in every interactive program, although it plays a much larger role in multi-user applications. Event-Driven Programming

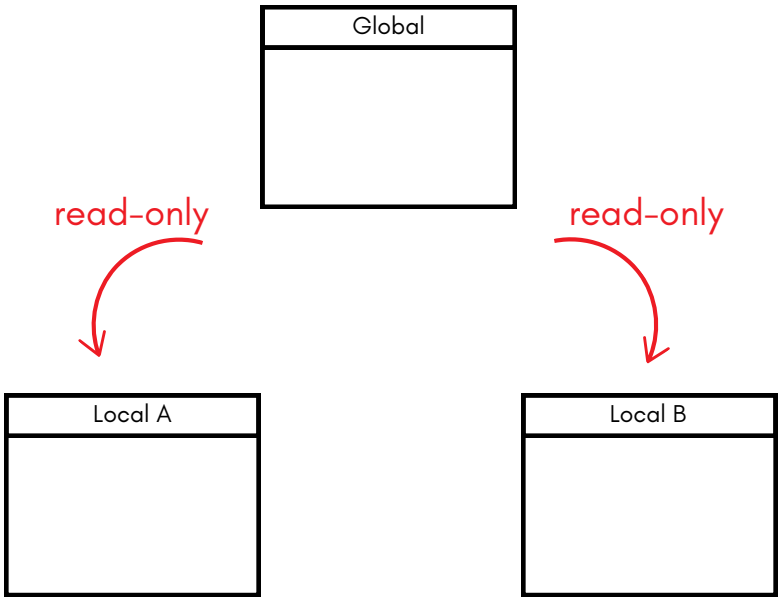


Figure 1.2: The principle P3, Pure Functions, leads to Consequence C3, using LG-MVU which enforces a read-only access of the global statechart to the local statecharts.

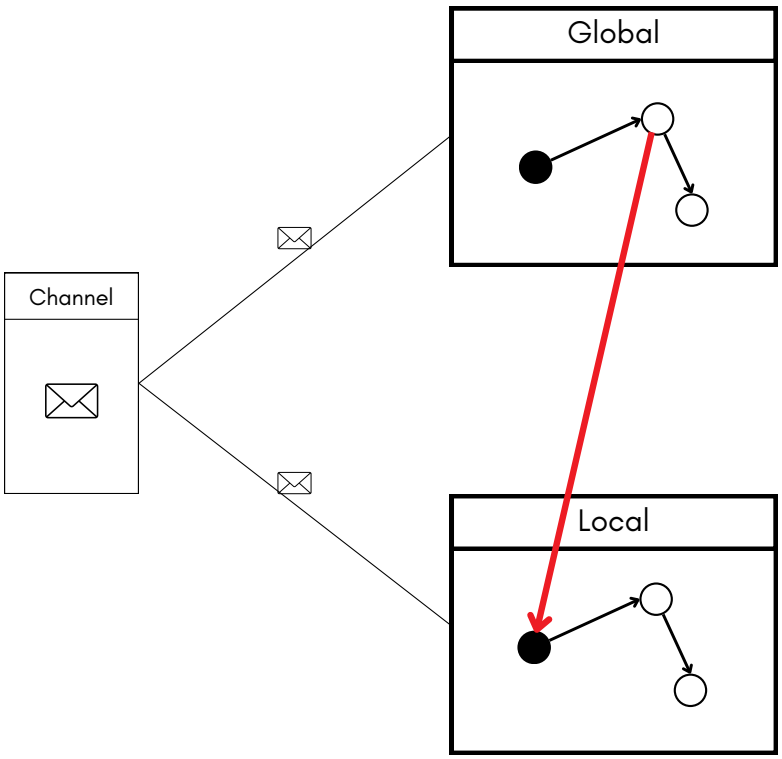


Figure 1.3: The *isolation* of statecharts in our proposed paradigm, as Consequence C4 of the Principle P4 (SoC). This isolation ensures that statecharts cannot modify each other directly. The violating transition is highlighted in red. Instead of direct modification, communication between statecharts is through message-passing via channels.

(EDP) is a paradigm for programming interactive applications that was developed to simplify the process, by using events instead of threads. Dabek et al. [21] says: "Programmers find programming with threads difficult, however, and as a result, produce buggy software. [...] Event-based programming can provide a convenient programming model, that it is naturally robust." This makes it very relevant to our goal. In particular, EDP handles the complexity of concurrency by abstracting the confusing details such as threads. See Section 2.1. This abstraction contributes to addressing our Principle P5, *Use Abstraction to Handle Complexities*. Therefore, our paradigm should allow the design of *event-driven* multi-user applications.

1.4 Limitation of Existing Tools

To evaluate whether existing tools can support our principles, we implemented a multi-user application using TEASync (described in Section 2.11), a text-based MDD framework, for LG-MVU, implemented in a purely functional language called Elm, supporting Event-Driven Programming (EDP). The above-mentioned features of TEASync help us adhere to the principles P3, P4, P5, *Pure Functions*, *Separation of Concerns (SoC)*, and *Abstraction*, respectively. The application we made is a Login System for a virtual class, in which a teacher creates usernames and passwords allowing students to subsequently log in.

The one principle not addressed by this approach was principle P1, *Visualization*. In fact, the development process was more confusing than anticipated, with many questions about the behaviour of the system being difficult to answer.

Subsequently, we looked into existing tools for visualization and tried to make

a graphical representation of our system using elements from UML statecharts as supported by Microsoft Visio, resulting in Figure 1.4. This statechart, however, required adding elements not included in the UML specification, namely:

- the lines highlighted in cyan, representing the Consequence C3, *Read-Only Access*, of Principle P3, *Pure Functions*.
- the arrows highlighted in yellow, arising from a transition and entering another transition in the other statechart. These arrows illustrate the messages that need to be sent from one statechart to the other, as a communication means among them. These direct arrows, however, violate the Consequence C4, *isolation*, enforced by Principle P4, *SoC*.

This model did not meet any of our principles nor entirely following the rules of previously known modelling languages, such as the UML or Harel's statecharts. We attribute these failures to four issues:

1. **Object-Orientation:** UML statecharts are *object-based*. This is in conflict with our Principle P3, *Pure Functions*, and its Consequence C3, *LG-MVU*. This fundamental difference caused difficulties in aligning the LG-MVU model with object-oriented conventions.
2. **Hierarchy:** Other models enforce a *hierarchy* incompatible with LG-MVU and its Consequence C3. Figure 1.5 compares the conventional hierarchy versus the LG-MVU conventions. In particular, in our approach, if we consider the local model as the sub-state of the global model, in terms of hierarchy, the flow of access is opposite to the typical ones. In other words, the local statechart must have read-only access to the global statechart. This flow, however, is reversed in

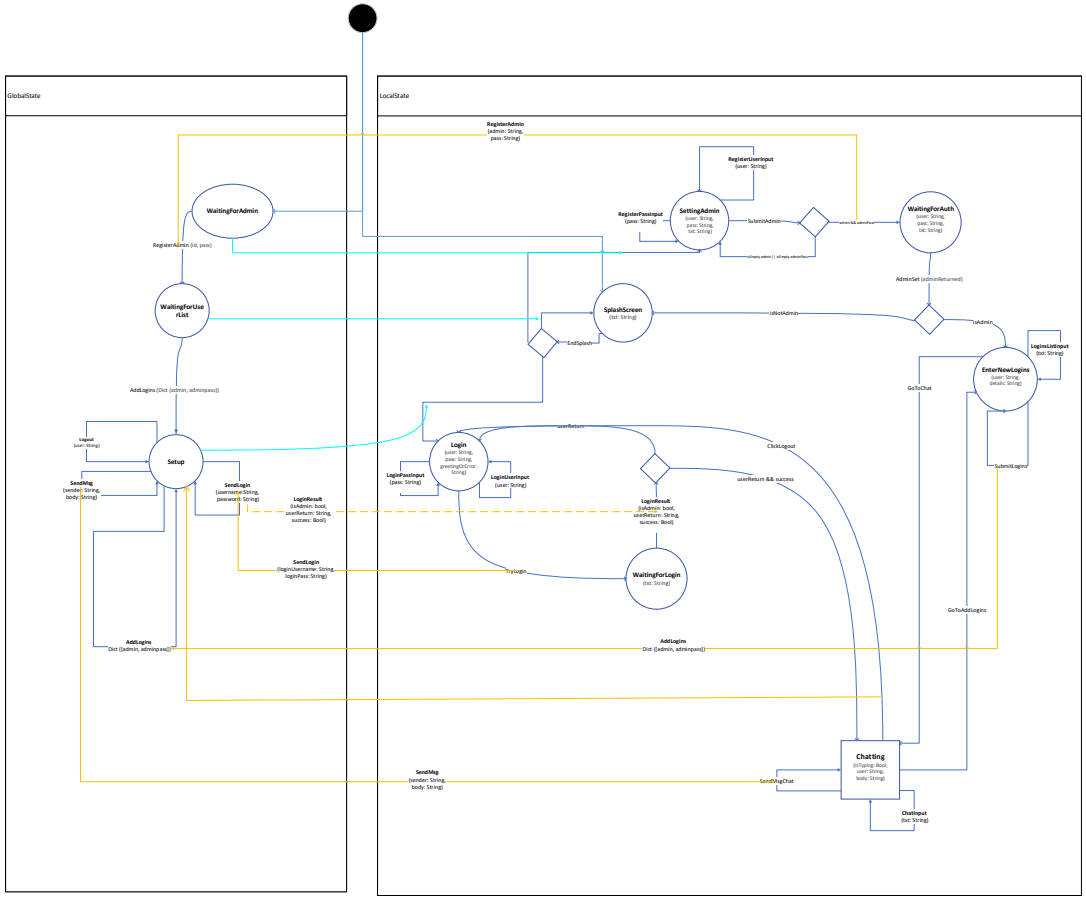


Figure 1.4: The statechart of the Login System multi-user app generated using TEASync, drawn in Microsoft Visio. This representation does not satisfy any of our principles, and violates other models' (UML or Harel statecharts) conventions.

other models that allow the parent statechart to access *and manipulate* children statecharts.

3. **Lack of Isolation:** Our principle P4, Separation of Concerns (SoC), necessitates the *isolation* of the models, which is absent in other modelling languages. Figure 1.6 shows the lack of isolation in the existing modelling languages. Specifically, UML and Harel's statecharts permit direct transitions between the states of different models. In contrast, our model demands interactions through message-passing via channels.

4. **Unrestricted Modelling:** Existing tools, such as Microsoft Visio, allow designers to create models without enforcing a specific modelling language's conventions. While this flexibility is highly desirable for designers, it can introduce challenges depending on the user's expertise level. Notably, this lack of guided constraints can cause difficulties for beginners who may get discouraged by errors arising from the flaws embedded in their models.

1.5 Communicating Statecharts (CSC)

Since we could not find an existing paradigm that satisfies our principles, we address the research question RQ1, by proposing a new beginner-friendly paradigm, called Communicating Statecharts (CSC). CSC aims to simplify concurrency to a level that is easy for novice programmers to learn and use. To find the answer to the principles that shape CSC, described in Section 1.3, we surveyed the literature on concurrency, and found partial solutions. Therefore, we borrowed the features of each model that best aligned with our principles. Then, we synthesized those features

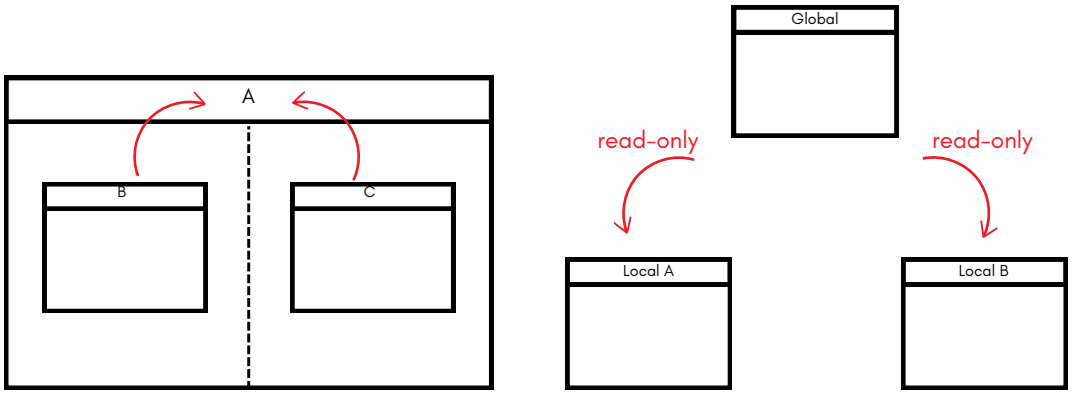
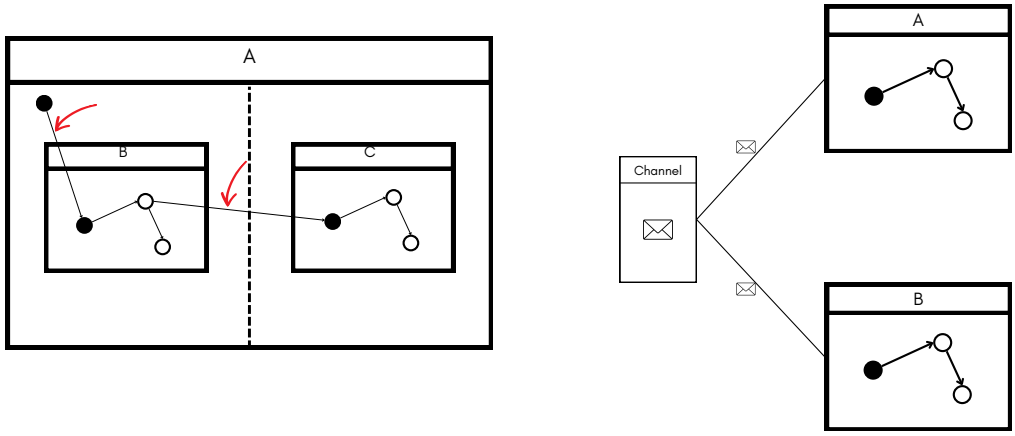


Figure 1.5: Our proposed paradigm's principle P3, *Pure Functions*, and its Consequence C3, *Read-Only Access*, enforces the children statecharts to have read-only access to the parent statechart. This flow is reversed in conventional hierarchy flows (parents accessing children).



A. Lack of isolation in other models

B. Isolation in our Paradigm

Figure 1.6: Lack of isolation in other modelling languages conflicts with the CSC principle P4, *Separation of Concerns (SoC)*, and its Consequence C4, *Isolation*. Particularly, the interaction among statecharts in our paradigm should be through message-passing via channels.

into our paradigm, CSC. See Chapter 2 for details on those concurrency models.

To address Principle P1, *visualization*, we began with Harel's original statecharts [34]. However, Harel's statecharts contain two types of decomposition: concurrent components, and substates, together with unrestricted inter-component transitions breaking our Principle P4, *SoC*. Harel's decompositions correspond to sum and product datatypes, see Table 1.1. CSC does not contain this type of decomposition because (1) they break SoC; (2) they introduce "concurrency" for all product types unnecessarily, making it harder to reason about necessary concurrency; and, (3) together the resulting statecharts are harder to understand than they need to be. Necessary concurrency includes user action, and network communication. Unnecessary concurrency would be the decomposition of statecharts, mentioned above, which requires supporting nested statecharts that execute concurrently. This level of detail doesn't belong to the high-level specification that CSC should capture due to the Principle P5, *Use Abstraction to Handle Complexities*. CSC's focus is limited to the necessary concurrency to minimize the cognitive load required to understand where concurrency is not needed. For example, in a login form with two fields for `ID` and `PASS`, If we allow the decomposition of statecharts, a possible design would be dividing the Local statechart into two concurrent sub-statecharts (one for `ID` and the other for `PASS`. Figure 1.7 illustrates this unnecessary concurrency, which translates into the product type as follows:

```
type alias LocalState =
    {
        id : ID
        pass : PASS
    }
```

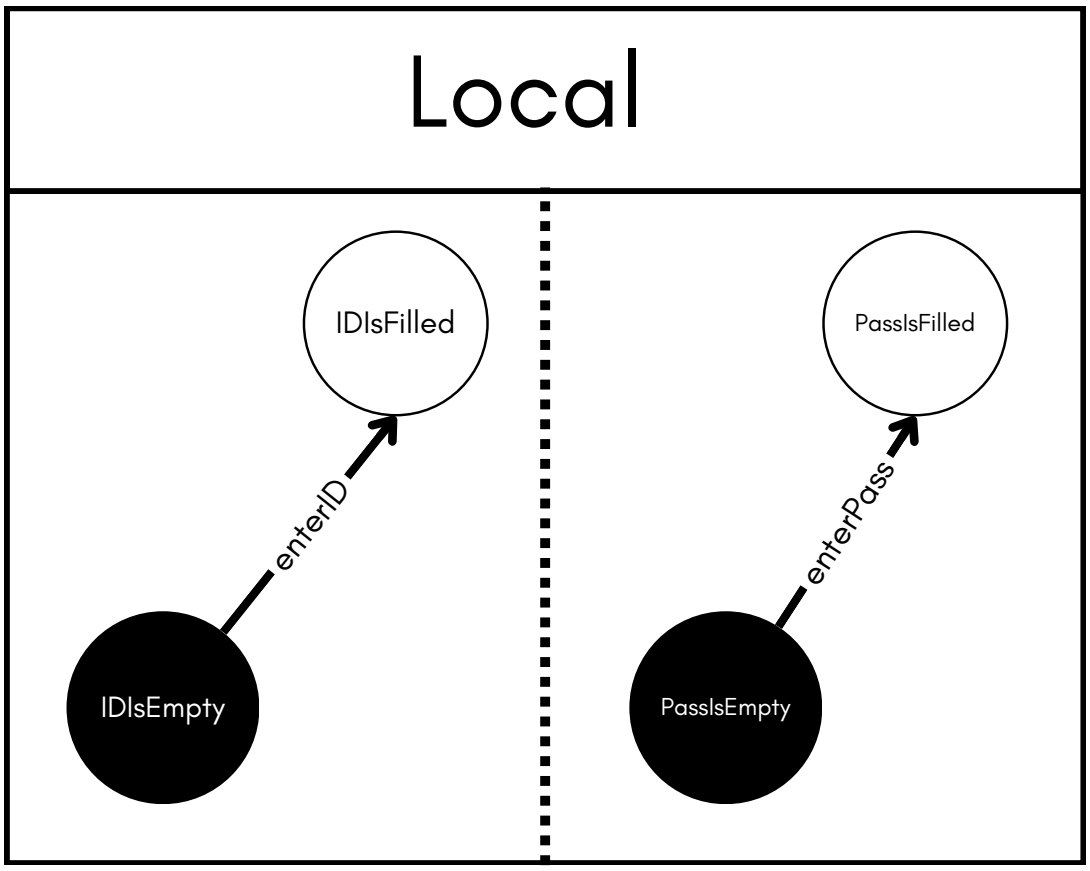


Figure 1.7: The example of introducing concurrency unnecessarily with the decomposition of (nested) statecharts. This design is not allowed in CSC due to Principle P5 (Abstraction).

```
6
7  type ID = IDIsFilled
8         | IDIsEmpty
9
10 type Pass = PassIsFilled
11          | PassIsEmpty
```

However, these fields do not need to be concurrent because the user first fills in

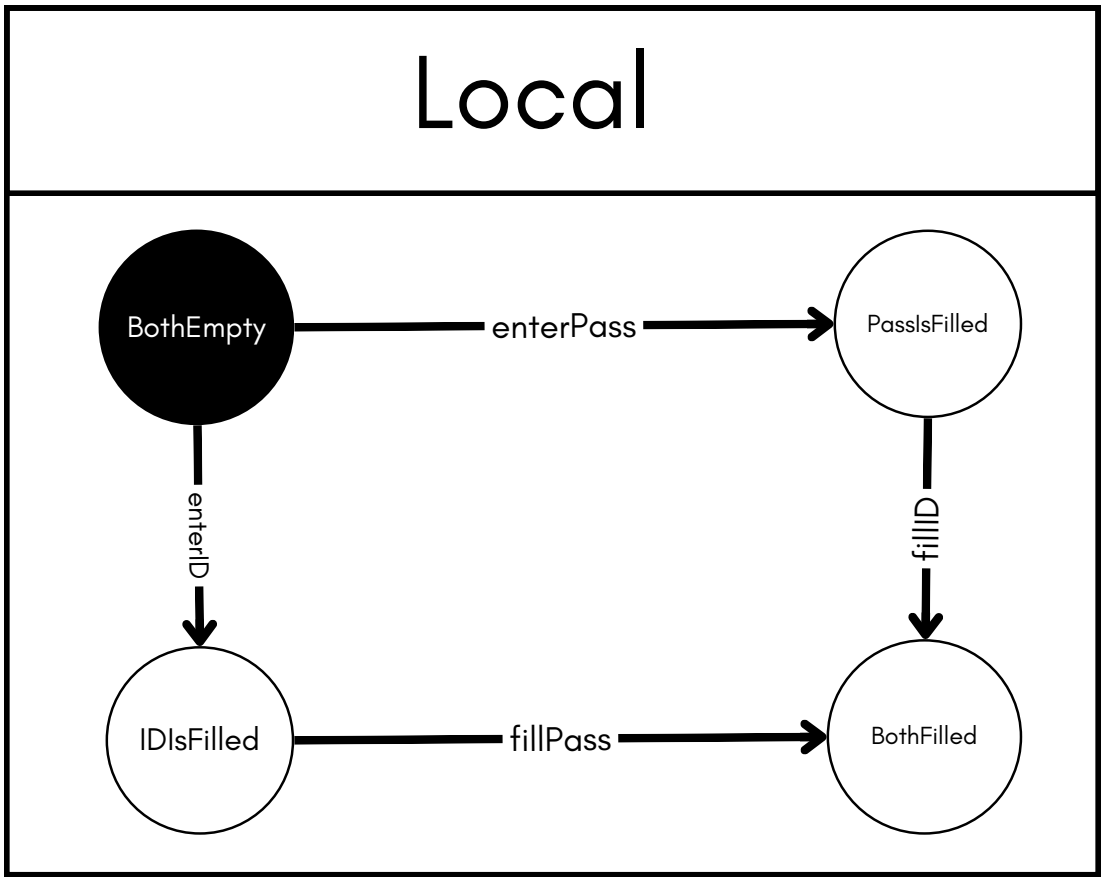


Figure 1.8: The example design of a login system without introducing concurrency unnecessarily.

the ID and then the PASS. This can be designed as illustrated in Figure 1.8, which translates into the SUM type as follows:

```
type LocalState = IDIsFilled
                | PassIsFilled
                | BothEmpty
                | BothFilled
```

Focusing on necessary concurrency prevents novices from early frustrations and

Algebraic Datatypes	**State Diagrams**	**Harel's Statecharts**
sum types	OR composition	substates
product types	AND composition	concurrent components

Table 1.1: Equivalence of decompositions. The natural translations between representations preserve compositions.

prepares them for learning more advanced concurrency concepts in courses such as Operating Systems.

To address Principle P2, *Model-Driven Development (MDD)*, which is tied to the Principle P1, *Visualization*, we need a visual MDD tool, which we present in the following sections (Sections 1.6 and 1.7).

To address Principle P3, *pure functions*, we adopt pseudo-states from recent UML standardizations, because they model case expressions, including pattern matching in functional languages. To address P3's Consequence C3, *Read-Only Access*, we model Local-Global Model-View-Update (LG-MVU) using global and local statecharts with one channel connecting them and a second channel connecting the local statechart to the user. As described before, LG-MVU, is an architectural pattern that separates the program into shared and private portions, allowing the creation of multi-user applications through pure functions. See Section 2.11 for more details. Perhaps unique to our model, we will allow statecharts to include in their scope other statecharts as read-only data, since this is a feature of LG-MVU.

To address Principle P4, *Separation of Concerns (SoC)*, we adapted channels from process calculi [14], explained in Section 2.3, connecting to statechart interfaces. This makes it possible to understand one statechart on its own. For similar reasons, from the actor model, introduced in Section 2.2, we adopt asynchronous messages and their principle of atomicity (which says that actors can only modify their own

states). Sending messages through channels is the only way actions of one statechart can modify the state of another. Table 1.2 summarizes the above-mentioned features borrowed from the existing concurrency models to shape our Communicating Statecharts. This table doesn't cover other concurrency models that we did not use in our synthesis. Specifically, Petri Nets are absent in our model since they allow the system to be in multiple states simultaneously [76]. This feature results in the system being non-deterministic, which makes it harder to reason about, while statecharts allow the system to be in only one state at a time.

Chapter 2 further describes these concurrency models and explains how CSC adopts their features.

Model	**Atomicity**	**Channels**	**Async. message-passing**
Actor Model	✓	-	✓
CSP	-	✓	✓
CCS	-	-	✓
π-Calculus	-	✓	-
CSC	✓	✓	✓

Table 1.2: CSC adopts features from multiple concurrency models.

To address Principle P5, *Abstraction*, CSC takes a different approach from the previous works, by using two languages, a visual language for high-level design, including concurrency, and a textual language for low-level implementation, by using event-driven concepts instead of threads, as well as, abstracting complications in code including threads and locks to a visual higher level, i.e. channels and messages.

The development process using CSC proceeds from high-level to low-level, with concurrency dealt with at the high level, as follows:

1. Designing local and global statecharts.
2. Choosing which messages can be sent in which channels.

3. Generating skeleton code using MDD.
4. Completing the skeleton by adding low-level implementation code.

1.6 Visual MDD Tool Requirements

To answer our research question RQ2, and support CSC, a visual MDD tool should be developed. Figure 1.9 illustrates the requirements for such a tool, shown in squares, and the implementation decisions derived from those requirements, outlined by circles. In this section, we describe our tool's requirements followed by the implementation decisions derived from them. They will be referred to throughout the thesis by their R and I numbers, respectively.

1.6.1 Tool Requirements

We now describe the requirements for our visual tool, shown in squares in Figure 1.9.

R1. Enforcing CSC Principles: The tool should prevent the designer from drawing configurations that conflict with CSC principles, as described in Section 1.3.

R2. Beginner-Friendly: The tool should be easy to learn and use for beginners. (Measurement and verification of this requirement are out of the scope of this thesis.)

R3. Code Generator: To address Principle P2, *MDD*, the tool should support code generation. The code generator should generate a skeleton program using the LG-MVU framework to address the Principle P3, *Pure Functions*.

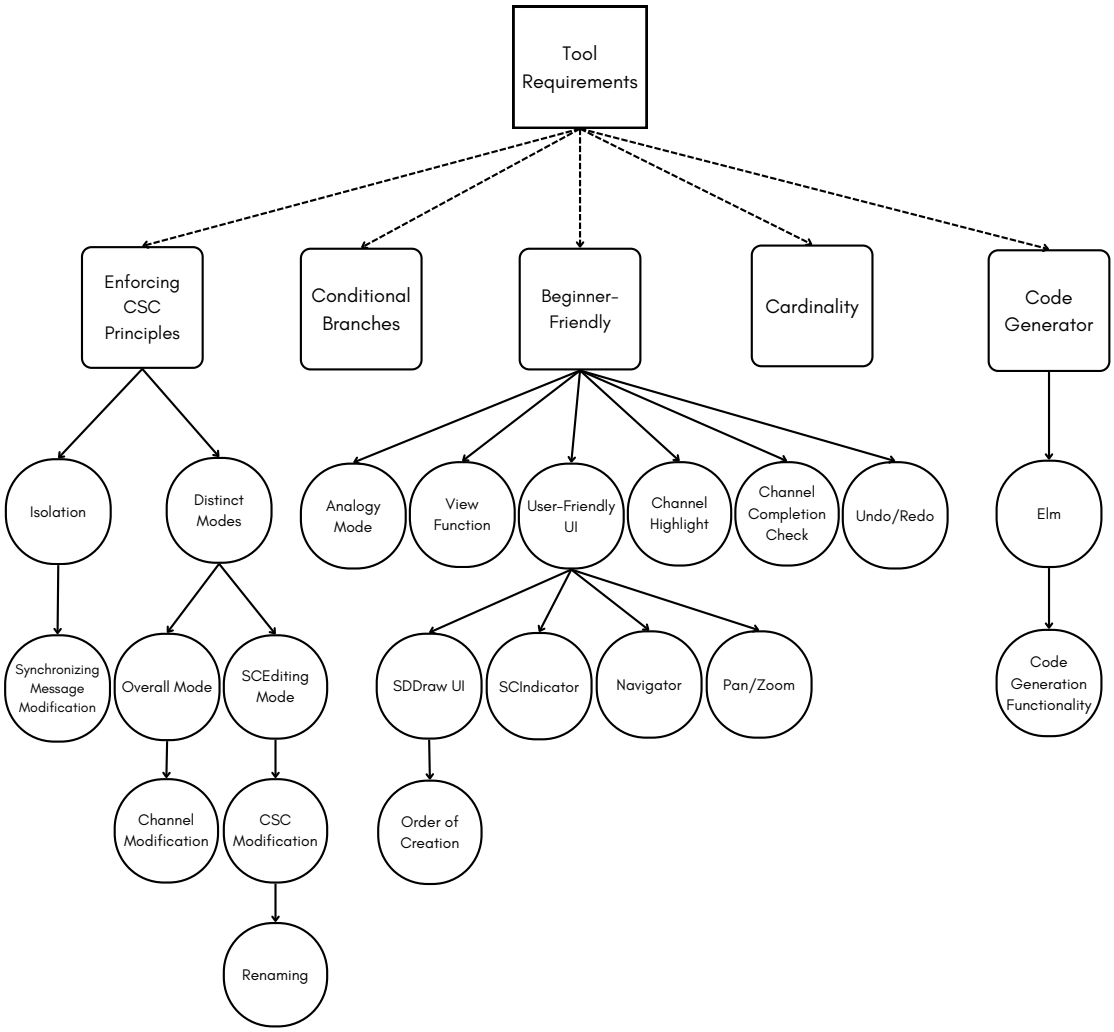


Figure 1.9: The requirements and Implementation Decisions of a visual MDD tool to support our proposed paradigm, CSC, outlined by squares and circles, respectively. The dashed and solid arrows represent the *is-a* and the *implies* relationships, respectively.

R4. Conditional Branches: Conditional branches are supported in recent UML State Machine Diagrams. They are equivalent to guarded transitions, but there are practical differences. This is essentially the same as the use of guards in Haskell, in addition to if-then-else and case expressions, versus the exclusive support for case expressions in Elm, which enables the compiler to check that case expressions—and therefore functions—are total. It is possible to express complex control flow more compactly with guards, but it is easier to understand that a case expression is total. Our goal is that conditional branches in CSC will translate directly to total case expressions, and result in the same increased transparency by gathering all the possibilities in one place, including the possibility of returning to the originating state.

R5. Cardinality: In CSC, statecharts connected by channels have cardinality relationships, i.e., many:one for local:global, and one:one for local:user. The tool should represent this cardinality analogously to how cardinality is represented on entity-relationship diagrams used in database design.

1.6.2 Tool Implementation Decisions

We now describe the implementation decisions derived from the gathered requirements, shown in circles in Figure 1.9.

I1. Isolation: Direct transitions between different statecharts should not be allowed. This implementation decision supports Requirement R1, *Enforcing CSC Principles*, protecting Principle P4, *SoC* and its consequence C4, *Isolation*.

I2. Synchronizing Message Modification: In CSC, communication among statecharts must be possible through the transmission of synchronizing messages, as a result of Principle P4, *SoC*. Therefore, the tool should support the addition and deletion of synchronizing messages. This implementation decision supports the above decision, I1, *isolation*.

I3. Distinct Modes: The tool should have distinct modes to visually and functionally enforce the separation of concerns, Principle P4. This implementation decision supports Requirement R1, *Enforcing CSC Principles*.

I4. Overall Mode: The tool should have an overall view where all of the statecharts with their details are visible, to allow defining channel contents. This implementation decision supports the above decision, I3, *Distinct Modes*.

I5. Channel Modification: The tool should allow the messages to be dragged to the channels, in the overall view, where statecharts with their contents are visible. This implementation decision supports the above decision,I4, *Overall Mode*.

I6. SCEditing Mode: The tool should have an editing mode where only one statechart should be visible, together with its interface. In this mode, details of the statechart should be editable. This implementation decision supports the above decision, I3, *Distinct Modes*.

I7. CSC Modification: The tool should allow the addition and deletion of states, transitions, and branches. This implementation decision supports the above decision, I6, *SCEditing Mode*.

I8. Renaming: The tool should allow the renaming of the created states, transitions, and branches. This implementation decision supports the above decision, I7, *CSC Modification*.

I9. Analogy Mode: The tool should provide an Analogy Mode which illustrates the CSC concepts as real-world objects to help users better understand the paradigm. This decision supports Requirement R2, the tool being *beginner-friendly*.

I10. View Function: The tool should clearly represent the flow of the data from the system to the user. In other words, it should graphically picture how the user receives the app's current state through the view function instead of receiving messages as a result of using *LG-MVU* from Principle P3. This decision supports Requirement R2, the tool being *beginner-friendly*.

I11. User-Friendly UI: The tool should have an easy-to-learn and use user interface. This is necessary to achieve Requirement R2, the tool being *beginner-friendly*.

I12. SDDraw UI: SDDraw [72] has an interface successfully used by novice programmers as observed by the instructor of a first-year computer science course. CSCDraw should adopt the visual appearance and conventions of SDDraw as much as possible. Novices use SDDraw to explore and develop the structure of their applications, rather than to encode applications for which they have a design. Supporting this exploratory behaviour will be one measure of success for CSC. This implementation decision supports the above decision, I11, *User-Friendly UI*.

I13. Order of Creation: In SDDraw, states must be drawn before transitions, and therefore the states must exist before the messages that label the transitions. Taking into account the previous decision, I12, *SDDraw UI*, the new tool should support the creation of synchronizing messages after the connected states have been created. Once messages/transition labels are created, users can decide which messages to add to which channels.

I14. SCIndicator: In the case of editing a statechart, the tool should have an indicator that gives information on which statechart is being edited. This implementation decision supports the above decision, I11, *User-Friendly UI*.

I15. Navigator: The tool should include a navigator button that allows the navigation between different views in the tool. This implementation decision supports the above decision, I11, *User-Friendly UI*.

I16. Pan/Zoom: Pan and zoom mechanisms should be provided in CSCDraw to enable drawing of large statecharts. This implementation decision supports the above decision, I11, *User-Friendly UI*.

I17. Channel Highlight: To prevent the user from designing an impossible path for messages, the possible routes for the transmission of the message should be highlighted when dragging a message to a channel. This decision supports Requirement R2, the tool being *beginner-friendly*, because it prevents designing infeasible communications.

I18. Channel Completion Check: To make sure the users' design is completely covering every possible case, code generation should not be allowed until every message is connected to a channel. This decision supports Requirement R2, the

tool being *beginner-friendly*, because it prevents unexpected behaviour of the system due to uncovered cases.

I19. Undo/Redo: Undo and redo should also be provided to offer a straightforward experience of using the tool. This decision supports Requirement R2, the tool being *beginner-friendly*, because it allows reverting the design without worrying about making mistakes.

I20. Elm: The Elm programming language, described in Section 2.7.4, promises purity together with immutable data types which means once a value is defined, it will not be allowed to change. Elm also has an intentionally restricted type system that ensures the minimization of type errors. Its static type-checking compiler promises the absence of run-time errors while providing easy-to-understand error messages. All these features recommend Elm as a suitable language for beginners. Therefore, the tool should use Elm for its code generation. This supports Requirement R3, *Code Generator*, and subsequently, Principle P2, *MDD*, and P3, *Pure Functions*.

I21. Code Generation Functionality: All aspects of CSC including states, transitions, Elm Types and subscriptions, and synchronizing messages should be functional. However, for conditionals, a default code should be generated, allowing the skeleton to compile, while the branch labels are included as comments in the generated code. This decision supports Requirement R3, *Code Generator*.

Chapter 4 describes how our tool addresses the above-mentioned requirements and their implementation decisions.

1.7 CSCDraw

To answer RQ2, an MDD tool, called CSCDraw, has been designed and implemented. CSCDraw considers the requirements, mentioned in Section 1.6, supporting CSC in reaching its goal of simplifying concurrency for beginners and addresses those requirements by applying the implementation decisions, described in that section.

A detailed introduction to the tool's UI and a specific description of how each CSCDraw element addresses those requirements are covered in Chapter 4. Additionally, an example of the development process through CSCDraw using CSC concepts is provided in this Demo[1].

In a nutshell, CSCDraw has two modes:

1. The *overall mode*, as shown in Figure 1.10, illustrates the channels and statecharts they connect, with their contents including states, transitions, and synchronizing messages. This view displays both statecharts with their details but only allows modification of channels, and code generation.

2. The *SCEditing mode*, shown in Figure 1.11, in which the states and transitions within a statechart can be edited.

1.8 Evaluation

To answer RQ3, a rigorous experiment should be conducted that measures how successful CSC and its tool are in simplifying concurrency compared to the existing paradigms. This study can be done by comparing CSC and CSCDraw to one of the

[1] `https://youtu.be/wPMjWU8C-x8`

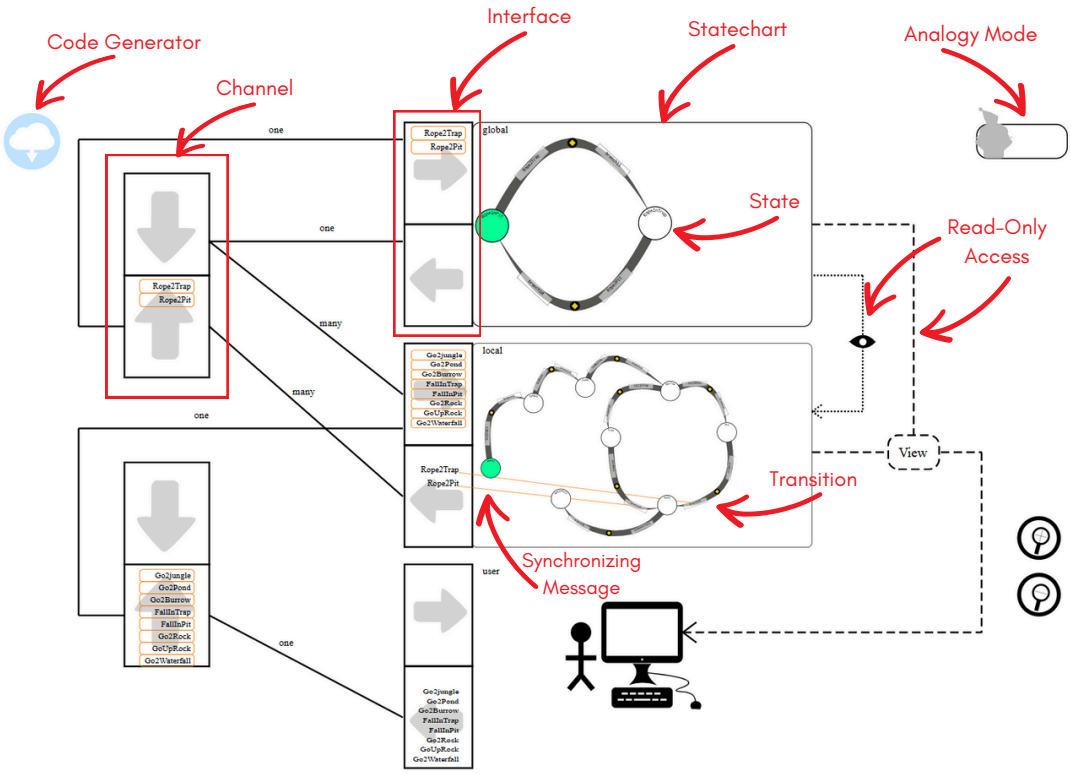


Figure 1.10: The Overall Mode of CSCDraw, rendering models/statecharts and their interfaces, as well as channels. Solid lines connect the channels to the statecharts. In this view, the contents of the channels are editable, while states, transitions, and synchronizing messages inside statecharts are visible. Dotted and dashed lines indicate read-only access. The user appears as a peer of the statecharts in this overall view. Code generation and switching to the Analogy Mode, where concepts are visualized in real-world representations, is possible through this view.

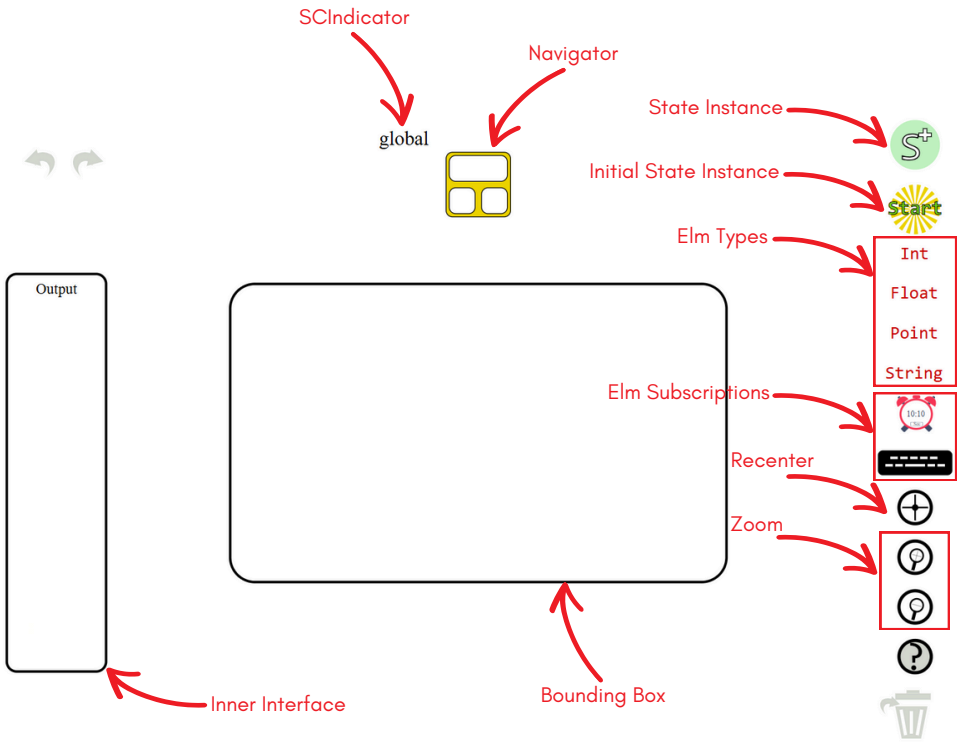


Figure 1.11: The SCEditing Mode of CSCDraw, which is reachable by clicking on a statechart in the overall mode. This mode allows the modification of each statechart, namely, addition, deletion, and renaming. State instances can be dragged and dropped to add a state or make a state the initial state. Elm types and subscriptions can be dragged and dropped into the states and transitions. Synchronizing messages can be connected to the corresponding messages through the inner interface. A navigator button allows navigation between the two modes while SCIndicator shows the name of the current statechart being edited.

most commonly used existing frameworks used to teach concurrency, Java Threads. However, before doing such a rigorous evaluation, we need to learn how to effectively teach concurrency using CSC to users with no prior knowledge. Therefore, we designed a pilot study, described in detail in Chapter 5, to prepare us for the later experiment comparing CSC and Java Threads. This pilot study aims to accomplish Bloom's revised Taxonomy's third level [5], illustrated in Figure 1.12, ordered from simple to complex; from *remember* to *create*. Bloom's Taxonomy is a framework that is used for classifying educational goals into six groups that may be expected to be learned by students in return for instructing them [11].

In this regard, the hoped-for learning outcomes of the designed pilot study are:

1. learning and *remembering* CSC concepts using CSCDraw,
2. *understanding* the translation of a multi-user application into a CSC and vice versa,
3. and finally *applying* the concurrency concepts to make a multi-player game.

In the current work we plan the pilot study (RQ3.1), along with the comparison between CSC and Java Threads for teaching effectiveness (RQ3.2).

1.9 Contributions

In this thesis, we made two main contributions aiming to simplify concurrency to a level that is easy for beginners to understand and use.

1. **CSC and CSCDraw:** We have proposed a new paradigm, called Communicating Statecharts (CSC), together with its visual MDD tool, CSCDraw. CSC

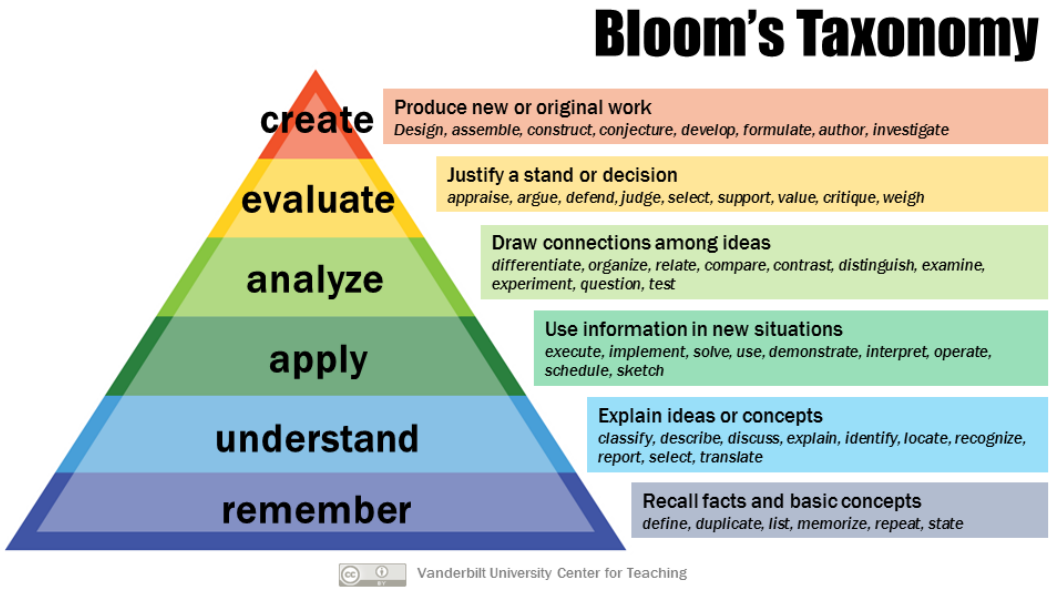


Figure 1.12: The revised version of Bloom's Taxonomy [5], a framework to categorize the objective learning outcomes. The levels are ordered from simple to complex, starting from *remember*.

aims to simplify concurrency by considering five main principles, such as raising the abstraction levels from complications in code including threads and locks to a visual higher level, i.e. channels and messages. In this regard, we adapted features of the existing concurrency models that aligned well with our principles. Also, to support the principles of visualization and MDD, we designed and implemented CSCDraw. Furthermore, we described our proposed paradigm's semantics and outlined how CSCDraw's code generator preserves those semantics. Then, we provided a detailed description of CSCDraw's user interface based on the gathered requirements.

2. **Design of a Pilot Study:** We designed a pilot study aiming to identify areas of confusion and possible improvement before planning an evaluation experiment. This pilot study helps us to detect the best way to teach CSC through CSCDraw,

as a prelude to measuring how successful CSC is in simplifying concurrency compared to the existing paradigms.

1.10 Thesis Structure

The rest of this thesis is organized as follows:

- Chapter 2 outlines a literature review, discussing different concurrency models, Event-Driven Programming, Programming languages paradigm, Architectural Patterns, Model-Driven Development, and Separation of Concerns.
- Chapter 3 defines the CSC semantics and explains how our code generator preserves CSC Semantics.
- Chapter 4 describes the user interface of CSCDraw, and explains how it meets the tool's requirements.
- Chapter 5 describes a pilot study designed to prepare us for future evaluations.
- Chapter 6 concludes the thesis, reviews the research question, and outlines potential future research.

Chapter 2

Background

To simplify concurrency for novice programmers, we established five principles, described in Section 1.3. These principles are the basis of our proposed meta-model for multi-client applications, CSC. In this chapter, we review the rich concurrency literature to borrow useful features that align best with these principles. In particular, we describe Event-Driven Programming (EDP) in Section 2.1, which provides a model for programming interactive applications. EDP's success in simplifying the concurrency arising from single-user interaction by raising the abstraction levels through using events instead of threads, which addresses our Principle P5, *Abstraction*. Additionally, several parallel research threads explore practical and theoretical aspects of multi-client (i.e., distributed) concurrency. In the middle sits the actor model (Section 2.2), which is the basis for industrially important frameworks like the Erlang run-time. Atomicity in the Actor Model, helps us to achieve *isolation*, derived from our Principle P4, *Separation of Concerns (SoC)*. On the theoretical side, we look into process calculi (Section 2.3) which promises to facilitate reasoning about concurrent and distributed programs. By exploiting features from process calculi,

including channels and asynchronous message-passing, we try to achieve the *isolation*, and subsequently, address our Principle P4, *Separation of Concerns (SoC)*. We also outline the key characteristics of a concurrency model called Simple Concurrent Object-Oriented Programming (SCOOP) in Section 2.4, because SCOOP was an attempt to make concurrency easier by raising the level of abstraction, which is in line with our Principle P5, *Abstraction*. From each of these, we take inspiration to shape the answer to our research question RQ1, exploring the features of a beginner-friendly concurrency paradigm. Furthermore, since statecharts are the most common visual modelling language in the case of event-driven systems, they will be adapted to address our Principle P1, *visualization*. Therefore, we look into the original model introduced by Harel in Section 2.5, followed by the semantics of statecharts, state machines, and finite automata in Section 2.6.

The use of Model-Driven Development (MDD) as Principle P2 of CSC, includes code generation. Code generation requires an understanding of the programming language paradigm and architectural pattern, which we review in Sections 2.7 and 2.8, respectively. In particular, we describe Object-Oriented Programming (OOP) and Functional Programming (Principle P3) and explain our choice of the Elm programming language and its architecture, Model-View-Update (MVU), for user interaction, and SDDraw and TEASync for multi-user interaction. This is explained in Sections 2.9, 2.10, and 2.11, for MDD, SDDraw and TEASync, respectively. In addition, Separation of Concerns (SoC), Principle P4 to shape CSC, is described in Section 2.12.

This background chapter does not cover concepts from concurrency and advanced programs that are not needed for the CSC paradigm or for the implementation of

CSCDraw. Specifically, we do not cover such topics as Petri nets, linear logic and related programming-language advances, or monads in functional programming.

2.1 Event-Driven Programming

Event-Driven Programming (EDP) is a programming paradigm that is based on receiving, processing, and reacting to events. In particular, the *event* refers to an occurrence in the software or hardware, e.g., clicking a button, or sensor outputs. The key feature of EDP is that the corresponding subprograms, called *event handler*s consistently listen to events and automatically react to them when they are received. Therefore, unlike sequential programs, EDP can be used where concurrency matters. Dabek et al. [21] describes EDP as a more convenient way to make interactive programs by using events instead of threads in managing concurrent IO. Concurrency is achieved in threaded systems by pausing the current thread when it is blocked on IO operations and switching to another thread. These systems rely on locks to protect shared data structures. This is hard for programmers to handle, which results in the production of error-prone systems. In contrast, Event-based programs offer an easier programming experience by using a loop, the central event handler, that as mentioned above, continuously listens for events and makes sure that it gets processed indivisibly. When the system arrives at a point where it should wait for an event, the central loop calls a new sub-event-handler for it. Consequently, using EDP leads naturally to a more robust software system while relieving programmers from difficulties such as handling threads. These benefits have caused the use of EDP to become ubiquitous in the software realm, from Graphical User Interfaces (GUI)s to operating systems and embedded systems.

Their popularity has made learning EDP necessary for software developers. However, the EDP concept appears complicated to novice programmers. For instance, they have the problem of determining what to put inside the event handlers, and at a higher level, how to design handlers in a way that they can work together. This may need an understanding of design patterns such as MVC, or MVU [50]. These design patterns will be described in Section 2.8. Another difficulty students have with EDP is testing the program, particularly, the use of both events and states, which also requires tools to observe the state of events [33].

To exploit the ease of concurrent programming in EDP over threaded systems, CSC uses EDP concepts such as events instead of threads, which addresses our Principle P5, *Abstraction*. CSC also naturally addresses the need for learning a design pattern, by using LG-MVU derived from its Principle P3, *Pure Functions*.

2.2 Actor Model

Hewitt et al. proposed a mathematical model for concurrency, in which actors can interact with each other through asynchronous, one-way message passing, and respond to the messages, concurrently [38]. This model offers the philosophy of "All physically possible computations can be directly implemented using Actors," as claimed by the creator of this model. This is similar to the philosophy of "everything is an object" in OO languages. In addition to physics, the Actor model was influenced by programming languages such as Simula, Smalltalk, and Lisp [44, 53, 69].

Later on, packet-switching also influenced the Actor model, and addresses were adopted to allow Actors to establish communications [7, 37].

Locality is another important feature of the Actor model. This ensures that

firstly, actors can only send messages to the addresses they know, and secondly, no simultaneous changes occur in multiple locations, unlike some other concurrency models, including Petri Nets [37].

Many programming languages are influenced by this model, namely Dart, Elixir, and Erlang [13, 15, 83].

CSC reserves isolation of its statecharts, derived from the Principle P4, *SoC*, inspired by the atomicity in the actor model. This feature ensures that actors in the actor model can only modify their own states. The direct communication is replaced by asynchronous message passing. However, unlike the actor model, in CSC, this communication happens through channels.

2.3 Process Calculi

Process Calculi, also known as Process Algebra, is a family of mathematical theories of concurrency [6, 9]. The key examples of this algebra include Communicating Sequential Processes (CSP), Calculus of Communicating Systems (CCS), and most recently π-calculus. Processes can be thought of as physical computers or threads of execution. Particularly, processes are the formal abstraction of a computational entity that can work concurrently and the interaction between them is possible through message-passing. Although "process" is in the name, they are not unique to process calculi, because actors are essentially processes. It is channels which distinguish these models. Channels can be used to add a level of abstraction to inter-processes communication, that can be visualized as wires connecting computers, or addresses of mailboxes. In other words, they are the abstractions of communication links between processes that allow processes to interact by sending and receiving messages.

Therefore, the use of channels in CSP and π-calculus, and asynchronous message passing in CSP and CCS, helped us to ensure the *isolation* of our statecharts and address our Principle P4, *SoC*.

2.3.1 Communicating Sequential Processes (CSP)

Communicating Sequential Processes (CSP) was introduced by Hoare as a concurrent programming language that only allowed programs to be written as a parallel composition of a fixed number of sequential processes [39, 40]. A noteworthy feature of the original CSP is that processes' communications were strictly synchronized and linked by named channels [14]. CSP uses named channels [14], in contrast to the Actor Model which has named processes. Theoretically, either system can simulate the other, but named channels can be thought of as offering an abstract interface. Behind the channel, the process handling the messages could be replaced by a new process or group of processes. Languages influenced by CSP include Erlang, Go, and Crystal [4, 17, 24].

2.3.2 Calculus of Communicating Systems (CCS)

Milner designed Calculus of Communicating Systems (CCS) to be the λ-calculus of concurrency [61]. Expressions in CCS describe labelled transition systems. Dijkstra's work on guarded commands influenced both Hoare and Milner to support conditional execution in CSP and CCS, respectively [6, 22, 40]. In CCS, communications are atomic between sending and receiving processes (which could be the same) [60].

Some languages inspired by CCS are the Java Orchestration Language Interpreter Engine (Jolie), and Language Of Temporal Ordering Specification (LOTOS) [48, 67].

2.3.3 π-Calculus

According to Milner's Turing Award lecture, some works on the development of the CCS, including the studies conducted by Kennaway and Sleep and also Engberg and Nielsen's work, *CCS label-passing*, led him to come up with the π-calculus [26, 45, 63, 65]. He also argued that the development of π-calculus [66] was with the goal of unifying values and processes in actors. This model's basic action is to communicate across an interface with a handshake, which enforces synchronization among participants [62]. The Actor Model allows processes to be passed as values in communication; while π-calculus instead allows references to processes, i.e., links, to be communicated [63, 66], and computation is represented purely as the communication of names across links. π-calculus has influenced some programming languages such as JoCaml. JoCaml further implements join-calculus which is a family member of π-calculus. The benefit of this adoption over using π-calculus is the provision of multi-way join patterns. This allows matching against messages from multiple channels, simultaneously [29, 30]. Also, two building blocks of JoCaml, processes and expressions, are executed asynchronously, and evaluated in a synchronous manner, respectively, as reported by Louis Mandel [49].

2.4 SCOOP

Eiffel programming language's creator, Bertrand Meyer, designed a concurrency model for this language, called SCOOP (Simple Concurrent Object Oriented Programming). The key characteristic of this model is the use of the principles of Design by Contract (DbC). Similar to a human contract, which consists of obligations and benefits for

both parties, in a software design the assertions include preconditions, postconditions, and invariants to specify the relationship between the client and the supplier, as part of the SCOOP strategy for synchronizing access to shared separate resources [57, 58]. Many languages utilize DbC to improve the reliability of their software, namely Kotlin, Scala, and SPARK ADA [8, 19, 28].

SCOOP attempts to make concurrency easier by raising the abstraction levels which is in line with our Principle P5, *Abstraction*. Particularly, SCOOP introduces the concept of separate objects, which are objects accessed by different threads. When the programmer defines an object to be separate, SCOOP automatically handles the synchronization required to ensure "safe access". This protects the programmer against low-level details such as locks for handling shared resources or condition variables.

2.5 Statecharts

A Finite State Machine (FSM) is an abstract, mathematical model of computation that consists of events, states, and transitions. State Diagrams, first introduced to visualize digital circuits, are a graphical representation in the form of a directed graph, with nodes denoting states, and arrows denoting transitions [54]. Statecharts are an extension of the formalism of FSM and state diagrams, and are used to visualize complex systems. In particular, statecharts can describe concurrent state machines. Transitions in statecharts are labelled with events and may be guarded by conditions. Specifically, the event triggers the transition from one state to another, if the corresponding condition evaluates to true.

This graphical model was first introduced by Harel [34], and many design and

development tools were built upon his visual language. The most commonly used extension is UML's version of statecharts, which is an object-based variant of the original work. Recent versions of UML statechart specifications include conditional pseudo-states which have similar semantics and look like conditional (diamond) bubbles in flowcharts [43].

Since statecharts and state diagrams are the most common modelling language, in the case of interactive systems [73], CSC bases its Principle P1, *Visualization*, on statecharts.

2.6 Semantics of Statecharts, UML State Machines, and Finite Automata

After Harel introduced his visual language for statecharts, and other researchers introduced variations, an effort was made to specify their semantics. Due to drawbacks in other approaches and other issues arising, Harel and Naamad [35] updated the official semantics for statecharts, bringing them up-to-date with their tool STATEMATE. The authors described "STATEMATE is a commercial tool, designed for the specification and design of real-life complex systems, coming from a variety of disciplines. Hence, the semantics are rich enough to support different models and to generate useful hardware and software code out of those models." [35].

Harel's statecharts can be considered as a hierarchy of cross-functional state diagrams. In particular, in Harel's model, multiple state diagrams can be contained in a super-state and execute transitions independently.

The semantics of Harel and Naamad [35], however, were not readily formalizable.

On the other hand, the semantics of Deterministic Finite Automata (DFAs) are easier to formalize and will be the basis for our design, following Shannon and Weaver [80] and Booth [12]. These semantics are as follows:

State Diagrams are the graphical representations of a finite automata. A finite automaton is a tuple (Q, Σ, δ, q_0) where:

Q**:** A finite set of states, usually represented by circles labelled with unique strings.

Σ**:** A finite set of inputs that trigger a transition.

δ**:** A transition function $\delta : \Sigma \times Q \to Q$.

q_0**:** The initial state, $q_0 \in Q$.

Typical interpretations include finite state machines that recognize strings of symbols (Σ) in a language, and states (Q) of an application that change based on user-generated events (Σ).

2.7 Programming Languages and Paradigms

The high-level organization of programs can be classified according to *paradigms*. There are multiple criteria used to identify paradigms, including program structure, execution model, syntax and grammar. Programming languages can support one or more paradigms, but most are known for one paradigm. The most popular paradigms include Object-Oriented Programming (OOP), Functional Programming, Imperative Programming, Logic Programming, Generic Programming, Structured Programming, and Procedural Programming.

OOP appears most often in the concurrency models that we looked into, as well as the most common modelling languages, such as UML statecharts. However, since functional programming promises the absence of side effects, and ensures immutability, CSC builds its Principle P3, *Pure Functions*, on functional programming languages. This choice can also help EDP to be implemented more conveniently. In this section, we go through OOP, and Functional Programming, as well as pattern matching, which is an important feature of functional languages including Elm, the programming language used for code generation in CSC.

2.7.1 Object-Oriented Programming (OOP)

OOP is a programming paradigm that follows the philosophy of *everything is an object*. In OOP, software objects interact much like physical objects interact in the physical world. Particularly, in OOP, a program consists of objects and classes, where the *object* refers to an abstract datatype or a collection of features and methods that share a state, and a *class* refers to the template from which objects can be created [84]. In other words, an object is an instance of a class.

Encapsulation ensures that the state of an object is not visible to the outside world, and the objects can interact with each other solely through calling each other's methods and those methods can access the state by references to the object's instance variables.

A fundamental concept in OOP is *inheritance*. Inheritance is a form of abstraction that supports the management of classes by organizing them into hierarchies. The base class is called a superclass (also known as a parent class), and the subclass

(or the child class) implements all the methods of the superclass and then may customize them by adding new operations or new instance variables. Inheritance allows separation of concerns, code reusability, and maintainability.

The Simula programming language is known as the first object-oriented (OO) language that used objects, classes, and inheritance [69]. Influenced by the Simula, Alan Kay developed the Smalltalk programming language [44] that allowed the dynamic creation and modification of classes.

Nowadays, many widely-used languages implement the OO paradigm to a significant extent. Pure OO languages include Ruby, Scala, Smalltalk, and Eiffel. Other languages designed mainly for OO programming are Java, Python, C++, and C#.

2.7.2 Functional Programming

Functional programming is a programming paradigm that uses functions entirely to construct a program. The key characteristic of this paradigm is the ability for the functions to be bound to names, passed as arguments, and returned from other functions. This is due to the functions being considered as first-class citizens. In functional programming, the main program itself is a function that receives the program's inputs and returns the result of the program.

Purely functional programming can be considered as a subset of functional programming, in which all of the functions are pure functions or mathematical functions. Purely functional programming offers less error-prone and easier to debug and test programs. This is because a pure function with the same inputs always returns the same result and no mutable state or other side effects can affect it. Also, assignment statements are not supported in functional programming. So, the value of a variable

never changes once defined, which eliminates side effects. Thus, functional programs are referentially transparent [42]. This also prevents the programmer from having to keep track of the execution order. Due to the absence of side effects, nothing can change the values, therefore, an expression can be evaluated at any time.

John McCarthy developed Lisp as the first high-level functional programming language, influenced by Church's lambda calculus [18, 52]. In particular, Lisp's functions extended the lambda calculus notation with labels, to support recursion which also allows iteration (looping). Later on, Robin Milner created ML, which eventually developed into several dialects. The most common of them are now OCaml and Standard ML [64].

Some of the most popular functional languages include Haskell, Miranda, and Elm, which promise purity, while Erlang and Elixir do not offer purity. Also, many other well-known, non-functional languages have adopted functional characteristics, namely Kotlin, JavaScript, GO, Rust, Python, and Dart.

2.7.3 Pattern Matching

Pattern matching is one of the most appreciated features of modern functional programming languages, such as Erlang, Scala, ML, Haskell, and Elm. Algebraic datatypes, supported in such programming languages, can be destructured by pattern matching. Pattern matching can be applied through case expressions, where some branches containing free variables are defined, and the first branch that matches the expression being matched against will be chosen. Once a branch is taken, the value of the expression will be bound to the free variable in that branch [41]. Similar to case

expressions are switch statements in OO languages, but switch statements are imperative, whereas case expressions are declarative. In some languages, a run-time error will arise if no matching pattern is found. However, other languages support exhaustive pattern matching, meaning that the compiler will not accept code that does not cover every possible pattern in the process of pattern matching. Therefore, no run-time error will be caused by this process. However, this exhaustiveness may also introduce difficulties in large projects where the number of branches to be handled may grow dramatically. One possible solution is using wildcards that will serve as the default branch and will capture any case that is not handled. The ease offered by this solution may come at a price. If the programmer forgets to handle a case, since the wildcard will catch that, no compile-time errors will be flagged, and therefore, there will be no clues to track down the unexpected behaviour of the system.

There have been some attempts to address this issue. For example, Eremondi [27] did an analysis and used Set Constraints to ensure that "missing branches of pattern matches are always unreachable." The author, as a possible solution proposes using constraint-based pattern matching, meaning that the functions restrict their inputs and hence, reject the impossible types.

Pattern matching in CSC is the translation of conditionals after code generation. In other words, the conditional branches drawn in CSCDraw turn into a pattern in the generated case expressions.

2.7.4 Elm

Elm is a purely functional programming language suitable for front-end web-based applications, developed by Czaplicki [20]. There is only one implementation of Elm,

and it is almost exclusively run in a browser with a standard run-time library that implements MVU. Elm uses a static type-checking compiler, which together with its purity, promises the absence of run-time errors and side effects. Its powerful type system helps programmers avoid unexpected behaviour of the system due to type errors. Elm also preserves immutability, therefore, once a variable is defined its value cannot be changed.

Elm belongs to the ML family, following its syntactic conventions. For example, Elm's standard library includes the forward pipe `|>`, as defined in the F# programming language, inspired by Unix pipes. Particularly, `|>` is an operator for function applications, feeding the output of one function as the input to the function on the right-hand side of the pipe.

Elm also supports *Algebraic Data Types*, which are essentially the composition of other types. Two main classes of algebraic data types are:

1. **Sum types** consist of at least one constructor that can contain several fields of different types. For example, in the following Elm algebraic data type

```
type LocalState = SantaHouse
                | Stable
                | CityCenter
                | NeighborHood
                | Mall
```

   the LocalState is defined by five constructors.

2. **Product types** can be constructed in three ways: as tuples, as data associated with a constructor, or as records. Records are most suitable for large product types, because fields have labels, and they will be familiar to programmers who

have used structures or objects. For instance, in the following Elm algebraic data type

```
type alias LocalModel =
    { time : Float
    , state : LocalState
    , hungerPoint : Int
    , remainedGifts : Int
    }
```

the LocalModel is defined by a record with four fields of different types. Note that all of the fields are themselves sum types.

General Algebraic Data Types nest sums and products inside of each other. For example,

```
type Colour = Red
            | Purple
            | Orange
            | RGB Float Float Float
```

Elm also supports one of the most appreciated features of modern functional programming languages, called *Pattern Matching* (Section 2.7.3). Algebraic datatypes can be destructured by pattern matching. Pattern matching can be applied through case expressions, where some branches containing free variables are defined, and the first branch that matches will be chosen. Once a branch is taken, the value of the expression will be bound to the free variable in that branch. For example, the following Elm code uses nested pattern matching, where the algebraic data type `msg` is being destructured at the outer level, followed by the `globalModel.state` at the

inner level.

```
globalUpdate msg globalModel =
    case msg of
        PlayMusic  ->
            case globalModel.state of
                MusicOFF ->
                        ({ globalModel | state = MusicON }, ↩
                           Cmd.none, Cmd.none )
                otherwise ->
                      ( globalModel, Cmd.none, Cmd.none )
```

In the case of this example, if the system is in `MusicOff` state, and it receives the `PlayMusic` message, the first branches in both levels will be taken.

2.8 Architectural Patterns

The fundamental features and the behaviour of an application can be defined through a concept called Architectural patterns. The use of various architectural patterns should be decided based on the project requirements and goals. "For example, some architecture patterns naturally lend themselves toward highly scalable applications, whereas other architecture patterns naturally lend themselves toward applications that are highly agile." [77].

The term *pattern*, is inspired by the work of Christopher Alexander on the concept in the architecture of buildings [3]. Alexander and his students documented hundreds of common design ideas for everything from communities to furniture. Their goal was to enable people and communities to design their own built environment. The

goal for software patterns is to promote software maintainability, and scalability [77], and improve the development process, in general, by making it easier to communicate about design. Various architectures may implement the same pattern and use common characteristics.

There are several popular architectural patterns, namely Layered Architecture (N-Tier Architecture), Microservice Architecture, Event-Driven Architecture, Service-Oriented Architecture (SOA), Model-View-Controller (MVC), and Model-View-Update (MVU).

In this section, we will explain the latter two architectural patterns in detail, as they are most relevant to our study since : (1) MVC is a popular architecture in the case of interactive systems, and (2) MVU is the architecture used by Elm which is used by CSC due to its Principle P3, *Pure Functions*.

2.8.1 Model-View-Controller (MVC)

MVC is a well-known architectural pattern highly suitable for interactive software systems, particularly web applications [47].

The key idea is to separate software constructs based on their responsibilities; i.e., separating the User Interface (UI) from the information that is rendered by the user interface.

The MVC architectural pattern divides the program into three components:

- **Model** is both the logic part of the software and the information that is rendered by the View.
- **View** is the part of the software that displays the information to the user,

- **Controller** is the part of the software that links the model and the view, by processing the user's interaction. The controller receives and processes the events caused by the user's interactions, and turns them into a Model or View command.

MVC was created by Trygve Reenskaug while he was working in Alan Kay's lab on Smalltalk-79. According to Reenskaug's notes, his main motivation was to create a pattern for large-scale projects that are made of complex tasks with a massive number of interdependent details. The initial version of MVC consisted of four components: Thing, Model, View, and Editor. Later, by consultation with other Smalltalk developers, Reenskaug changed the pattern to MVC [75].

Using the MVC design pattern boosts the software's maintainability and ease of development. For example, by making the data structure independent of the UI.

Many popular languages have MVC frameworks, such as ASP.NET in C#, Laravel in PHP, Django in Python, Angular in JavaScript, and Ruby on Rails in Ruby.

2.8.2 Model-View-Update (MVU)

MVU architecture is an architectural pattern for GUI development that plays to the strengths of functional programming. MVU is also known as The Elm Architecture as this pattern owes its fame to the Elm functional programming language [20].

The MVU model contains four components:

- **Model** is a data type for encoding the application state.
- **Msg** is a data type for encoding transition labels.

- **View** is a pure function taking a model value and outputting a concrete rendering of the application.
- **Update** is a pure function which takes model and message values and produces a new model value.

Technologies such as Redux[1] that are used by the widely-used ReactJS framework [2], have been inspired by MVU [31].

2.9 Model-Driven Development (MDD)

Mellor et al. [55] describe MDD, the CSC Principle P2, as "the notion that we can construct a model of a system that we can then transform into the real thing" In other words, in MDD the source code can be generated through model transformations, to simplify and formalize (to allow automation of) complicated systems [32]. The model can be generated through modelling languages such as UML through semantics like statecharts for reactive systems and class diagrams for OO static design.

The business applications of MDD are generalizations of the Object Management Group (OMG)'s Model-Driven Architecture (MDA) initiative [10]. Using MDD can improve software's maintainability and reusability by increasing the level of abstraction. As their name suggests, the OMG generated OO code, and this is still true of the vast majority of MDD tools today.

2.10 SDDraw

For several years, our research team has used a web-based MDD tool, SDDraw[1], to introduce interaction [72]. In their first example, we have students create the map of an adventure game and then generate Model-View-Update (MVU) code to render the interface with a separate screen for each state and buttons for transitions to another state. MVU can also be visualized in the time-data-flow diagram in Fig. 2.1.

The Elm runtime stores the model and handles the events (messages) and other impure logic, allowing the user to write their code entirely as types and pure functions, leading to code that is much easier to test and reason about. Although SDDraw generates code from the diagram, saving them time, we still expect the students to understand the code. For this reason, we teach them to map each state to an enumerated type with one constructor for each state, and similarly, map each transition label to a constructor in an enumerated "message" type. This contrasts with the unstructured way we have observed beginner programmers adding global variables to try to capture components of the state, without having an overall design, resulting in unexpected feature interactions (bugs). To reinforce this expectation, we included Fig. 2.2 as part of a midterm test in a first-year computer science course, introduction to Software Design Using Web Programming, in which students were asked to add a hydration (health) system to the code generated from the state diagram, which had a set of rules for the cost of different transitions and a method for replenishing hydration points. Implementation required many simple changes, which would be challenging to implement by searching through the code without being able to use the state diagram to understand the overall structure.

[1] https://sddraw.STaBL.Rocks/

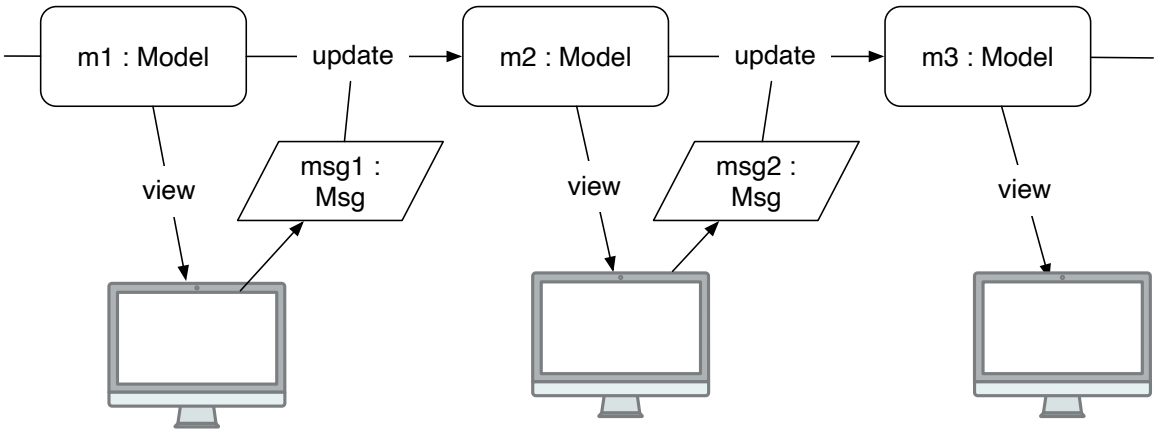


Figure 2.1: Time-data-flow diagram for Model-View-Update (MVU). The model is used by the pure view function to render the application in the browser, and messages (events sent by user interactions, e.g. mouse clicks) are passed into the update function to produce new models, changing the application state.

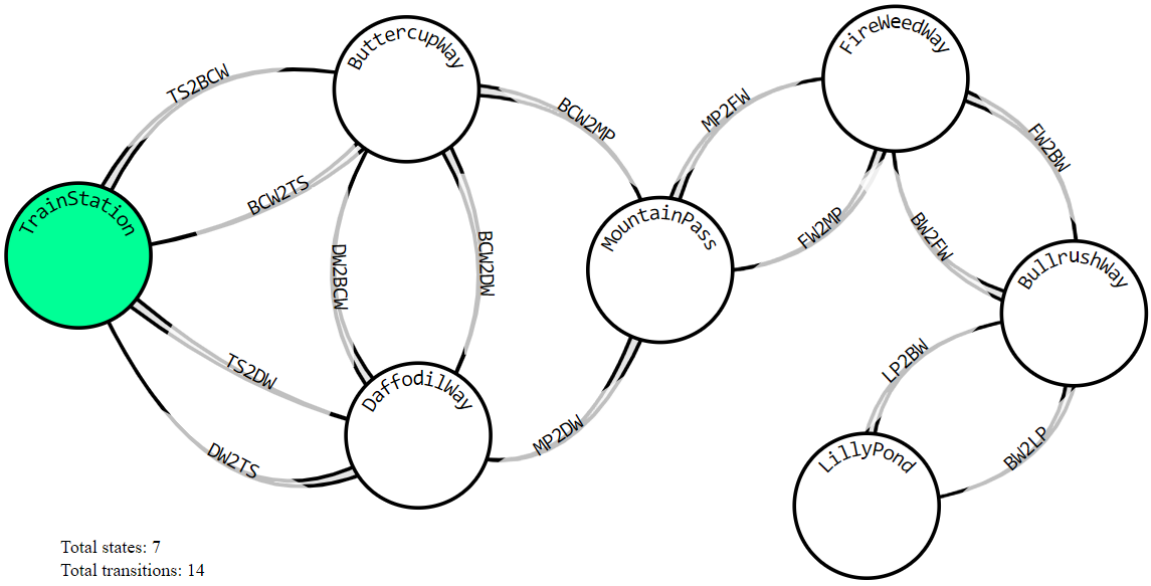


Figure 2.2: The state diagram, drawn with the MDD tool, SDDraw, and used to generate code students were given as a starting point with the above diagram on the exam. Transitions are narrow at the target, and wider at the origin. The initial state is green.

2.11 TEASync

TEASync is a novel framework for developing concurrent web applications [78]. It is an extension of the Elm's Model-View-Update (MVU) architecture. Because it uses two models, one local and one global, it is called *Local-Global MVU* (LG-MVU). It is an extension in the sense that any MVU application is also a purely local LG-MVU. The update in MVU has the signature

$$\text{update} : \text{Msg} \rightarrow \text{Model} \rightarrow \text{Model} \tag{2.1}$$

which can be extended to a function

$$\begin{aligned} \text{localUpdate} : \text{LocalMsg} &\rightarrow \text{LocalModel} \rightarrow \text{GlobalModel} \\ &\rightarrow (\ \text{LocalModel, Cmd LocalMsg, Cmd GlobalMsg}\ ) \end{aligned} \tag{2.2}$$

by ignoring the GlobalModel input, which can be taken to be the unit type. Without many changes, an Elm application can also be made into a purely global TEASync application, in which all connected clients can interact with the application at the same time. In general, the *local* model is private to each client, and the *local* update has read-only access to the global model. The local model can be used for state that does not need to be shared (e.g. highlighting moused-over buttons). The *global* portion is shared amongst all clients, which allows programmers to make multi-user applications entirely in the frontend language Elm. Figure 2.3 illustrates the dataflow/timeline of two devices connected to a server in the LG-MVU architecture.

Specifically, one global model is shared amongst every client connected to the server, while each client has its private MVU and read-only access to the global

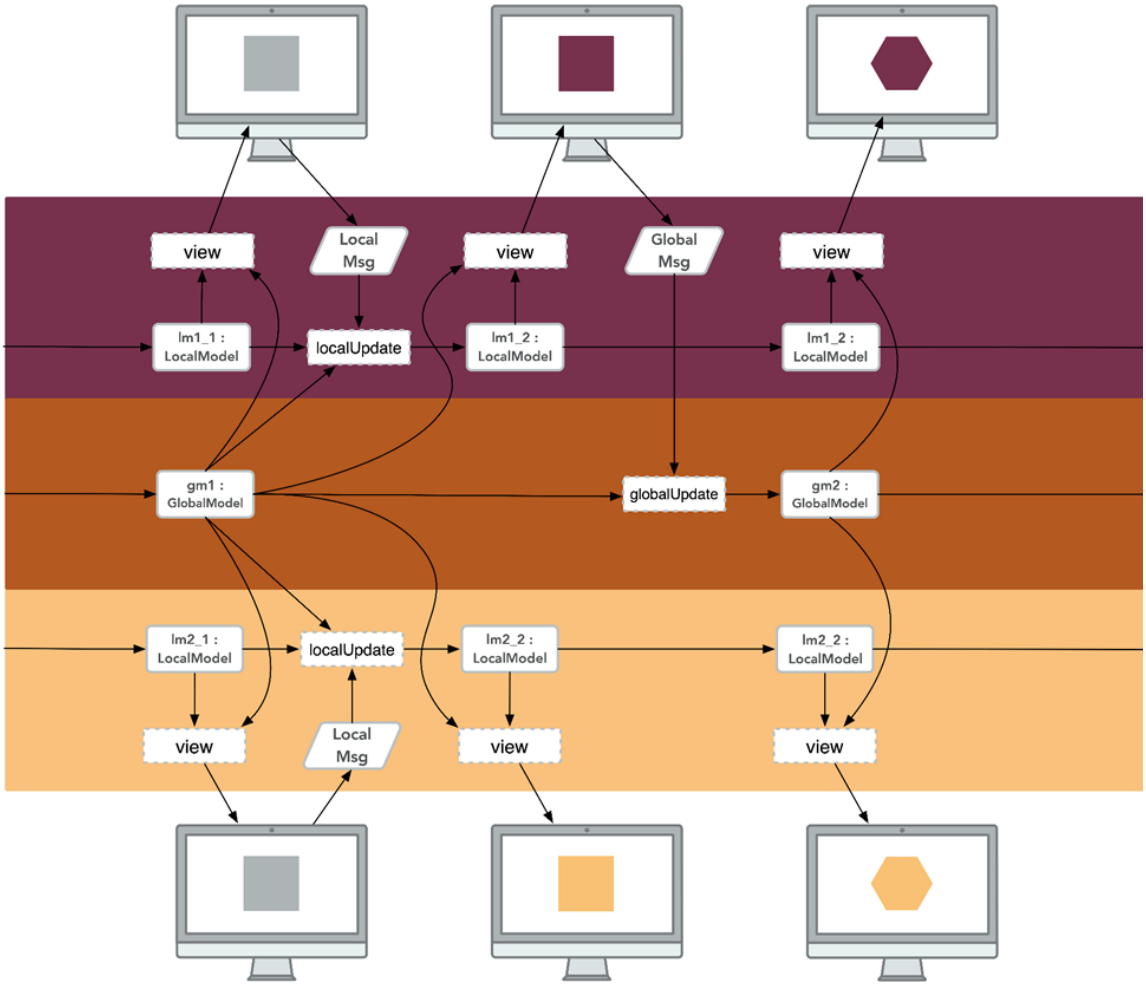


Figure 2.3: The dataflow/timeline of two devices connected to a server in LG-MVU architecture [78].

model. The Global Update function takes a Global Message and the Global model, and updates the current model based on that message, and returns a new Global Model. The view function will then render the user's screen based on the updated Global Model and the Local Model.

Although distributed applications require communication between the client and the server, TEASync handles this for the programmer, from writing encoders and decoders to establishing the connections. In other words, it allows so-called serverless development for complete beginners. The TEASync server handles ordering and broadcasting global messages, keeping all clients synchronized. Programmers can also run their applications in an "offline mode" where virtual clients can be spawned to test their programs with one or more clients without running a real server. This is an MDD approach to distributed computation, in which the data structures are the model. The system parses the data structures and generates server code and a shell that wraps around the user's client code.

Our team has used an online Integrated Development Environment (IDE), including a collaborative project environment, called STaBL.Rocks to support coding in Elm. TEASync adds code generation to this project system.

The above-mentioned features of TEASync made it a suitable framework for us to implement a multi-client application, helping us to evaluate whether existing tools can support our principles, as described in Chapter 1.4. In particular, by using LG-MVU, it helps us to adhere to our Principles P3, P4, P5, *Pure Functions*, *Separation of Concerns (SoC)*, and *Abstraction*, respectively.

2.12 Separation of Concerns (SoC)

Separation of Concerns (SoC) refers to the ability to first identify the concerns, which may be considered as an interest or purpose, then, divide the program into distinguished sections based on those concerns, and finally, encapsulate each section. In other words, SoC can be considered as a form of abstraction. Therefore, due to its promise of handling complexities, SoC constructs the Principle P4 of CSC.

Parnas [71] introduced the idea of SoC, with suggestions for how to do the separation, but Dijkstra [23] either originated the term SoC, or led to its popularization. By the time of Reade [74], the term Separation of Concerns was an accepted part of software design.

Different programming languages offer different ways of implementing SoC. For instance, object-oriented programming languages such as Java and C++ can separate concerns into objects. Procedural programming languages including C can separate concerns into procedures or functions. Also, design patterns like MVC provide SoC by separating the logic, from the interface.

Concerns can be identified from different aspects which are called *dimension*s of concern. In addition to the mentioned example, concern in object-oriented programming can be the data or the class; each concern in this dimension is a data type defined and encapsulated by a class. The program can be decomposed according to one or more dimensions.

Chapter 3

Design of CSC

In this chapter, we define our CSC semantics, based on the state diagram semantics, described in Section 2.6, since the semantics of Harel and Naamad [35], were not readily formalizable, and our CSC also consists of State Diagrams in the first level. Finally, we explain how CSCDraw's code generator preserves the semantics of CSC.

3.1 CSC Semantics

Our Communicating Statecharts (CSC) consist of the tuple:

$$(G, \Gamma, \delta_G, g_0, L, \Lambda, \nu, \delta_L, l_0, \varepsilon) \tag{3.1}$$

$$(\Lambda = \Lambda_G \cup \Lambda_U) \tag{3.2}$$

where

G**:** A countable set of global states, usually represented by circles labelled with unique

strings.

Γ**:** A countable set of global messages that trigger a transition, $\varepsilon \notin \Gamma$.

$\delta_G : \Gamma \times G \to G \times (\Lambda_G \cup \{\varepsilon\})$**:** A transition function also called the "globalUpdate" function. Note that ε indicates that no synchronizing messages should be sent.

$g_0 \in G$**:** The initial global state.

L**:** A countable set of local states, usually represented by circles labelled with unique strings.

Λ**:** A countable set of local messages that trigger a transition. Λ_G is the subset of messages that can be sent by δ_G. Λ_U is the subset of messages that can be triggered by a user action. $\varepsilon \notin \Lambda$.

$\delta_L : \Lambda \times G \times L \to L \times (\Gamma \cup \{\varepsilon\})$**:** A transition function also called the "localUpdate" function.

$\nu : L \to 2^{\Lambda_U}$**:** A function that for each local state determines which local messages could be generated by the user.

$l_0 \in L$**:** The initial local state.

ε**:** an element not in any set listed above, indicating that no synchronizing messages should be sent.

The current state of the system is the combination of a global model G, a queue of global messages $[\Gamma]$, together with N copies of the local model L and a queue of local messages $[\Lambda]$, corresponding to the number of connected clients N.

Therefore, the current state of the system is

$$(n, g, q_G, l_1, q_1, l_2, q_2, ..., l_n, q_n) \in \mathbb{N} \times G \times [\Gamma] \times \Pi_{i \in \{1,2,...,n\}} (L \times [\Lambda]) \tag{3.3}$$

where the left-hand side of the notation defines the elements that appear in the right-hand side sets. Also, $[\Theta]$ indicates a queue of elements of the set Θ, [ ] indicates an empty queue, and we will use the function $\alpha : \Theta \times [\Theta] \rightarrow [\Theta]$ to append an element to the end of a queue.

The initial state of the system is

$$(0, g_0, [\,]) \tag{3.4}$$

which indicates that no clients have joined, and the empty message queue shows no messages have been created so far.

The system state evolves through one of the following changes:

Process global message: If the global message queue q_G is not empty, it has a first element γ and a remainder q'_G. Let $(g', \mu) = \delta_G(\gamma, g)$ be the result of the update function. If the triggered transition causes a synchronizing message to be sent, then that synchronizing message will be added to every local message queue, resulting in the new system state

$$(n, g', q'_G, l_1, \alpha(\mu, q_1), l_2, \alpha(\mu, q_2), ..., l_n, \alpha(\mu, q_n)) \tag{3.5}$$

On the other hand, if $\mu = \varepsilon$ then no synchronizing message will be sent and the

new state is

$$(n, g', q'_G, l_1, q_1, l_2, q_2, ..., l_n, q_n) \tag{3.6}$$

Process local message: If the local message queue q_i nonempty for some $1 \leq i \leq n$, it has a first element λ and a remainder q'_i. Let $(l'_i, \mu) = \delta_L(\lambda, l_i)$ be the result of the update function. If the triggered transition causes a synchronizing message to be sent, that synchronizing message will be added to the global message queue. Therefore, if $\mu = \varepsilon$ then the new state is

$$(n, g, q_G, l_1, q_1, ..., l'_i, q'_i, ..., l_n, q_n) \tag{3.7}$$

otherwise, if a synchronizing message is created

$$(n, g, \alpha(\mu, q_G), l_1, q_1, ..., l'_i, q'_i, ..., l_n, q_n) \tag{3.8}$$

Accept user message: If there exists $\nu(l_i)$ nonempty for some $1 \leq i \leq n$, the user can send a message $\lambda \in \nu(l_i)$, and the new state is

$$(n, g, q_G, l_1, q_1, ..., l_i, \alpha(\lambda, q_i), ..., l_n, q_n) \tag{3.9}$$

Disconnection: If a user, $1 \leq i \leq n$, loses connection to the server or intentionally disconnects, that user's state and message queue will be discarded, resulting in the new state

$$(n-1, g, q_G, l_1, q_1, ..., l_{i-1}, q_{i-1}, l_{i+1}, q_{i+1}, ..., l_n, q_n) \tag{3.10}$$

Connection: When a new client connects to the system, the initial state of the local model and an empty local message queue will be inserted into the system state:

$$(n+1, g, q_G, l_1, q_1, l_2, q_2, ..., l_n, q_n, l_0, [\,]) \tag{3.11}$$

Note that client number has no meaning in the system, and is not available for use by the statecharts.

3.2 Code Generation Preserves CSC Semantics

Code generation is simplified due to the Principle P3, *Pure Functions*. In particular, implementing code generation to preserve CSC semantics in an impure language with side effects would be possible, but user additions to the generated skeleton could accidentally break the semantics.

Figure 3.1 illustrates the CSC for a basic multi-user application. In this example, a mystery button is inside of a MysteryRoom. The first user that presses the button will cause music to be played for everybody. Other button clicks will not make a change to the music being played. This example will be used in the following sections to describe how CSCDraw's code generator maps the CSC into the Elm code. The implementation decision of using Elm, I20, is explained in Section 1.6.

3.2.1 Generating Algebraic Data Types

In the Elm code, both local and global models, as well as messages and states, are represented as algebraic data types. CSCDraw's code generator preserves this structure by translating each state into a constructor within the corresponding algebraic data

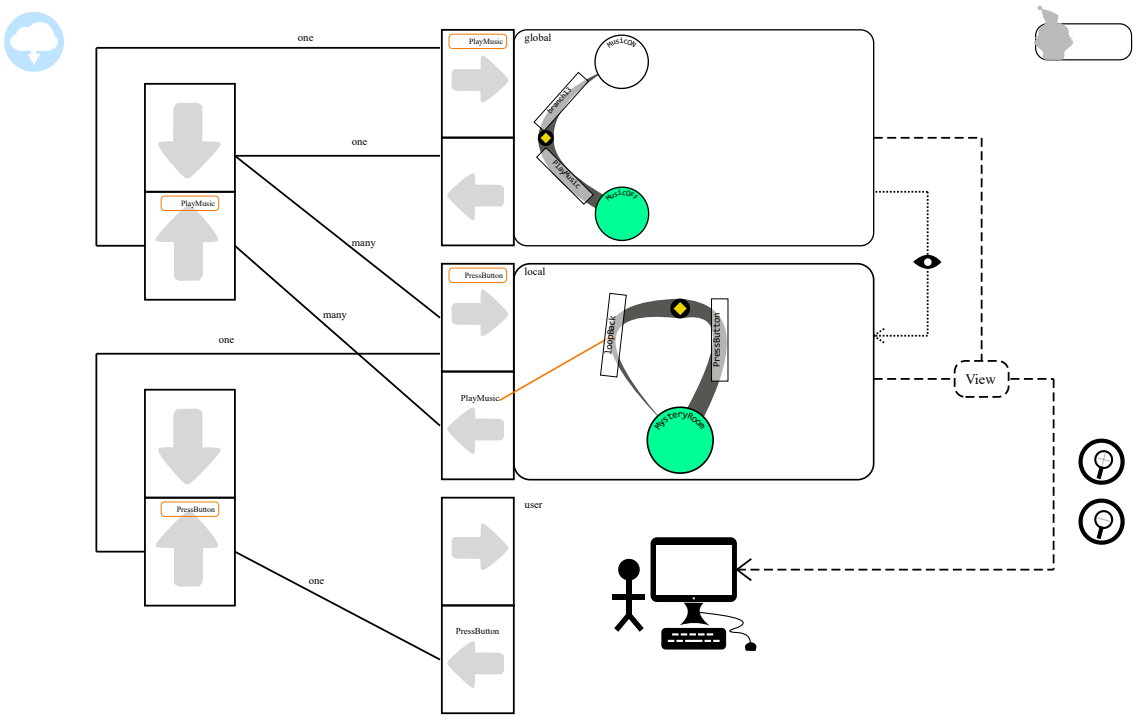


Figure 3.1: CSC of a basic multi-user application. The local statechart contains one state: *MysteryRoom*. The *PressButton* transition will send a synchronizing message to the global statechart. That synchronizing message will trigger the *PlayMusic* transition in the global model, which changes the global state from *MusicOFF* to *MusicON*.

```
--Put this code into the Types module.
import GraphicSVG.EllieApp exposing(GetKeyState)

type LocalMsg = Tick Float GetKeyState
              | PressButton

type LocalState = MystryRoom

type alias LocalModel =
    { time : Float
    , state : LocalState
    }

type GlobalMsg =  PlayMusic

type GlobalState = MusicOFF
                 | MusicON

type alias GlobalModel =
    { state : GlobalState }
```

Figure 3.2: Algebraic Data Types generated from example Figure 3.1. The code generator has converted the drawn CSC states and transitions into the constructors of the corresponding Elm data types.

type, `LocalState` and `GlobalState`, respectively. Similarly, each transition label will be mapped to a constructor of the message data types, `LocalMsg` and `GlobalMsg`, respectively. Each statechart is embedded into a record to facilitate the extension of the code skeleton. In particular, the `LocalState` is embedded into `LocalModel` along with a field to keep track of `time : Float` for animation and other timing purposes. See Figure 3.2.

3.2.2 Generating Update Functions

Type signature 2.2 in Section 2.11 describes the purpose of the *localUpdate* function. It takes a local message and the local model, and returns a triple of the new local model and the synchronizing messages. Additionally, this function allows the local

model to have *read-only access* to the global model, addressing our Consequence C3 of Principle P3, *Pure Functions*. CSCDraw's code generator generates nested case expressions as the body of the update function. There are different ways to implement those nested case expressions. For example, pattern matching can be done by first destructuring the message data type, and, at the inner level, destructuring the state data type, or vice versa. Our code generation follows Elm conventions, which leads to the first scenario: pattern match on the message first. Furthermore, our code generation behaves as a recursive mapping function to generate the branches in the case expressions.

Moreover, in CSCDraw, when a synchronizing message, indicated by a ε in the semantics (Section 3.1), is connected to its originating branch, the code generator translates that as the output of the corresponding message together with the updated model. The code generation process for the global update function is similar, with the type

$$\begin{aligned} &\text{globalUpdate} : \text{GlobalMsg} \rightarrow \text{GlobalModel} \\ &\qquad\qquad \rightarrow (\text{GlobalModel, Cmd GlobalMsg, Cmd LocalMsg}) \end{aligned} \tag{3.12}$$

Figures 3.3 and 3.4 illustrate how the code generator has made nested case expressions, destructuring the message data type first and then the state data type. The structure is used for the *localupdate* and *globalUpdate* functions. Note that `Cmd.none` corresponds to the case where no synchronizing message is sent. Also, the synchronizing message *PlayMusic* originating from the *PressButton* transition is translated with default values as `Task.perform(` $\_ -> PlayMusic$`)(`$Task.succeed\ 0$`)`.

```
28 localUpdate msg localModel globalModel =
29     case msg of
30         Tick t _ ->
31             ({ localModel | time = t }, Cmd.none, Cmd.none )
32         PressButton  ->
33             case localModel.state of
34                 MystryRoom ->
35                       ({ localModel | state = MystryRoom }, Cmd.none, Task.perform ( \ _ ->PlayMusic) (Task.succeed 0) )
```

Figure 3.3: The generated code for the localUpdate function based on the local statechart. This code first destructures the message and then the state data type at the inner level. Synchronizing messages are also translated as the output of their originating transition.

```
43 globalUpdate msg globalModel =
44     case msg of
45         PlayMusic  ->
46             case globalModel.state of
47                 MusicOFF ->
48                       ({ globalModel | state = MusicON }, Cmd.none, Cmd.none )
49                 otherwise ->
50                      ( globalModel, Cmd.none, Cmd.none )
```

Figure 3.4: The generated code for the globalUpdate function based on the global statechart. When no synchronizing message is sent, `Cmd.none` will fill the corresponding field.

3.2.3 Generating the View Function

The generated *view* function will contain buttons sending local messages based on the transitions in the local statechart. Additionally, to provide information for debugging reasons, it will render the current global and local state of the app. Figure 3.5 shows the generated view function code, named as `myShapes`, ready to be compiled on our online IDE, STaBL.Rocks.

3.2.4 Generating the Init Function

Our code generator forms the init functions in the Elm code from the initial states in the CSC. Figures 3.6 and 3.7 illustrate the translated Elm code for the *localInit* and *globalInit* functions, respectively.

```
myShapes localModel globalModel =
    [text ("GlobalState: " ++ if globalModel.state == MusicOFF then "MusicOFF"
                           else "MusicON")
       |> centered
       |> filled black
       |> move(0, 20)]
    ++
    case localModel.state of
        MystryRoom  ->
            [ text ("LocalState: " ++ "MystryRoom")
                  |> centered
                  |> filled black
            , group
                  [
                       roundedRect 40 20 5
                            |> filled green
                  ,    text "PressButton"
                            |> centered
                            |> size 8
                            |> filled black
                            |> move(0, -3)
                  ]
                     |> move (0, -25)
                     |> notifyTap (LocalMsg <| PressButton)
            ]
```

Figure 3.5: The *view* function generated by the code generator. This function renders information on the current state of the app with buttons mapping to local transitions.

```
initLocal : (LocalModel, Cmd LocalMsg)
initLocal = ({ time = 0
       , state = MystryRoom
       }, Cmd.none)
```

Figure 3.6: The *localInit* generated by the CSCDraw's code generator based on the initial local state.

```
initGlobal : (GlobalModel, Cmd GlobalMsg)
initGlobal = ({ state = MusicOFF
       }, Cmd.none)
```

Figure 3.7: The *globalInit* generated by the CSCDraw's code generator based on the initial global state.

3.2.5 Generating Conditionals

Conditional branches in CSCDraw are specified as comments. To make the generated skeleton code compile, the default condition and patterns are filled with integers, and therefore the asynchronous `Task` operations succeed with `zero` to match the first branch of the case expression. After the code is generated, the programmer should change the case expressions, as well as the `Task` operations appropriately.

Figure 3.8 illustrates an example of the *Party* game. Each player has a budget of a hundred dollars. They can go around the town and buy things. When they arrive at the *BusStop*, if they have enough money left they can take a bus to the *PartyRoom*, otherwise they have to walk back *Home*. When at least three players arrive at the party room, the party will start. Figure 3.9 illustrates the generated conditional, filled with default values and containing comments. This skeleton code will compile but to make it work reasonably, the user should fill out the condition and branches appropriately. Figure 3.10 shows the modified condition. In the case of this example, in the local model, a local variable *budget* of type `Int` is defined which is initially set to 100. That *budget* variable is then used to complete the condition.

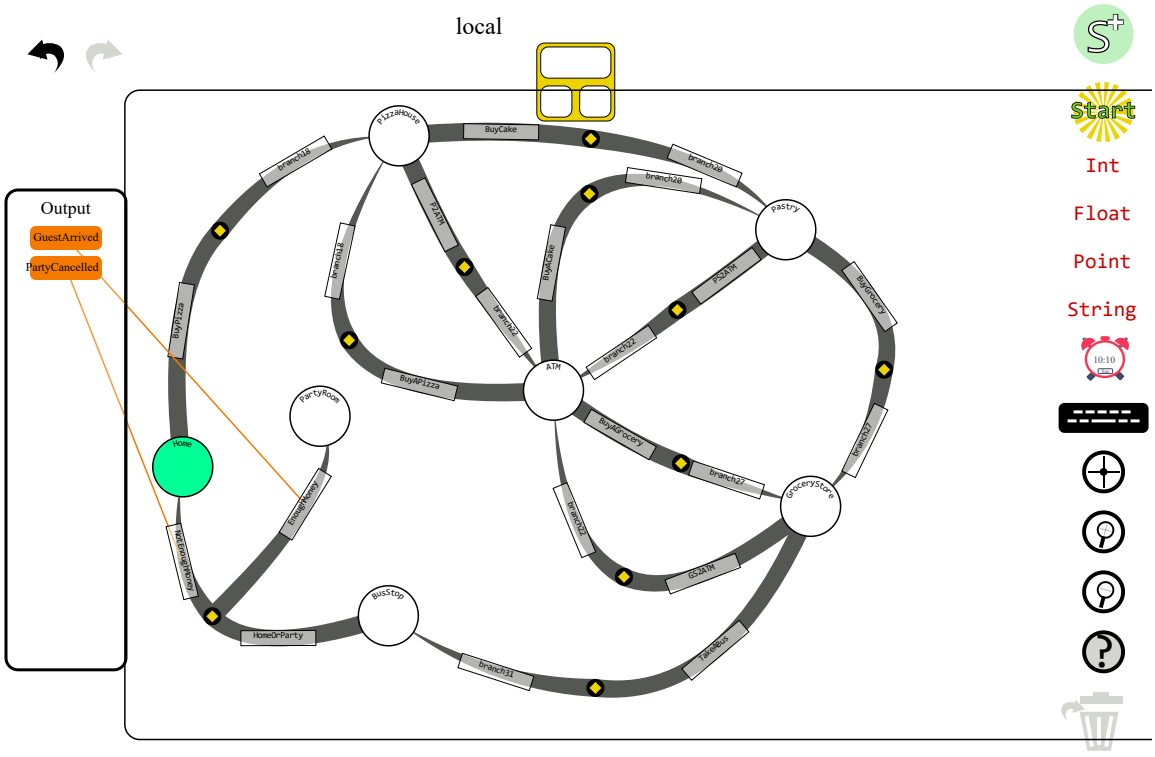


Figure 3.8: The local statechart of the *Party* game. A conditional is used when the user wants to take a bus that has a money requirement to reach the *partyRoom*.

```
235         HomeOrParty  ->
236             case localModel.state of
237                 BusStop  ->
238                     --TODO: Replace condition and branch names with actual ones.
239                       case 0 {-condition-} of
240                           0 {-lowBudget-} ->
241                                ({ localModel | state = Home }, Cmd.none, Task.perform ( \ _ ->FailParty) (Task.succeed 0) )
242
243                           _ {-enoughBudget-} ->
244                                ({ localModel | state = PartyRoom }, Cmd.none, Task.perform ( \ _ ->GuestArrived) (Task.succeed 0) )
245
246                 otherwise ->
247                       ( localModel, Cmd.none, Cmd.none )
```

Figure 3.9: The generated conditional code from the *Party* game. This code is filled with default values so the skeleton code will compile.

```
235         HomeOrParty  ->
236             case localModel.state of
237                 BusStop  ->
238                     --TODO: Replace condition and branch names with actual ones.
239                       case localModel.budget >= 0 of
240                           False {-lowBudget-} ->
241                                ({ localModel | state = Home }, Cmd.none, Task.perform ( \ _ ->FailParty) (Task.succeed 0) )
242
243                           True {-enoughBudget-} ->
244                                ({ localModel | state = PartyRoom }, Cmd.none, Task.perform ( \ _ ->GuestArrived) (Task.succeed 0) )
245
246                 otherwise ->
247                       ( localModel, Cmd.none, Cmd.none )
```

Figure 3.10: The modified generated conditional code from the *Party* game. This code is filled with appropriate condition and values.

Chapter 4

CSCDraw's UI

CSCDraw is a visual Model-Driven-Development (MDD) tool, developed to answer our RQ2, and subsequently, provide support for CSC. Therefore, the bulk of the interaction, in CSCDraw, has to do with visualizing and editing the diagram.

This chapter describes the specific features of CSCDraw that are designed and implemented to address the user-interface requirements and their implementation decisions mentioned in Section 1.6.

Table 4.1 summarizes how each CSCDraw feature addresses an implementation decision and consequently, a requirement.

Req#	**Requirement**	**Imp-Dec#**	**Implementation Decision Name**	**Feat#**	**FeatureName/ Description**
R1	Enforcing CSC Principles	I1	Isolation	E3	Read-Only Access
				-	Also addressed by OM1, OM2, OM3, and OM4.
		I2	Sync. Message Modification	KF2	Sync. Messages
				EM4	Inner Interface
		I3	Distinct Modes	-	Addressed by I4, I6.
		I4	Overall Mode	OM1	Models
				OM2	Sync. Messages
				OM3	Model Interfaces
				OM4	Channels
				E1	User
				E2	View Function
				E3	Read-Only Access
				E4	Cardinality
				E5	Download Button
				E7	Zoom Buttons
		I5	Channel Modification	OM4	Channels

		I6	SCEditing Mode	-	Addressed by I7, I8, I12, I14, I15, I16, I19.
		I7	CSCModification	M1	State Modifications
				M2	Transition Modifications
				M3	Branch Modifications
				EM5	State Instance
				EM6	Init. State Instance
				EM7	Elm Types
				EM8	Elm Subscriptions
				EM11	Trash Bin
		I8	Renaming	-	Renaming states, transitions, and branches.
R2	Beginner-Friendly	I9	Analogy Mode	E6	Analogy Mode
		I10	View Function	E2	View Function
		I11	User-Friendly UI	-	Addressed by I12, I13, I14, I15, I16.
		I12	SDDraw UI	M1	State Modifications
				M2	Transition Modifications
				EM5	State Instance

				EM6	Init. State Instance
				EM7	Elm Types
				EM8	Elm Subscriptions
				EM10	Help Button
				EM11	Trash Bin
		I13	Order of Creation	-	Addressed by I5 and I6.
		I14	SCIndicator	EM2	SCIndicator
		I15	Navigator	EM1	Navigator
		I16	Pan/Zoom	E7	(in Overall Mode)
				EM9	(in SCEditing Mode)
		I17	Channel Highlight	OM4	Possible directions will light up.
		I18	Channel Completion Check	E5	Download Button
		I19	Undo/Redo	EM12	Undo/Redo
R3	Code Generator	I20	Elm	E5	Download Button
		I21	Code Generation Functionality	E5	Download Button
R4	Conditional Branches	-	-	KF1	Conditional Branches
R5	Cardinality	-	-	E4	Cardinality

Table 4.1: Overview of the relationship between CSCDraw requirements and its implemented features.

To address the Implementation Decision I3, *Distinct Modes*, the editor provides two modes: (1) an *Overall mode*, showing the channels and statecharts they connect, and (2) an *SCEditing mode*, in which the states and transitions within a statechart can be edited. The next two sections describe these modes in detail.

4.1 Overall Mode

CSCDraw allows users to have a total view of their CSCs, as shown in Figure 4.1, addressing the Implementation Decision I4, *Overall mode*. In particular, the models/statecharts, interfaces, and channels would be visible in a unified view. In this view, the contents of channels are editable, addressing Implementation Decision I5, *Channel Modification*. Overall mode also addresses Implementation Decision I1, by its structure of models interacting through channels.

The following elements, described in the order shown in Figure 4.2, shape the Overall mode. These elements will be referred to throughout the thesis by their OM numbers.

OM1. Models: The two models, local and global, required by *Local-Global MVU (LG-MVU)*, as a consequence of Principle P3, are rendered in a unified view together with their contents, including states, transitions, and synchronizing messages. Clicking on a statechart results in the navigation to the editing state, as described in Section 4.2.

OM2. Synchronizing Messages: The orange lines indicate that a synchronizing message will be transmitted to the channel when the originating branch of the transition is taken.

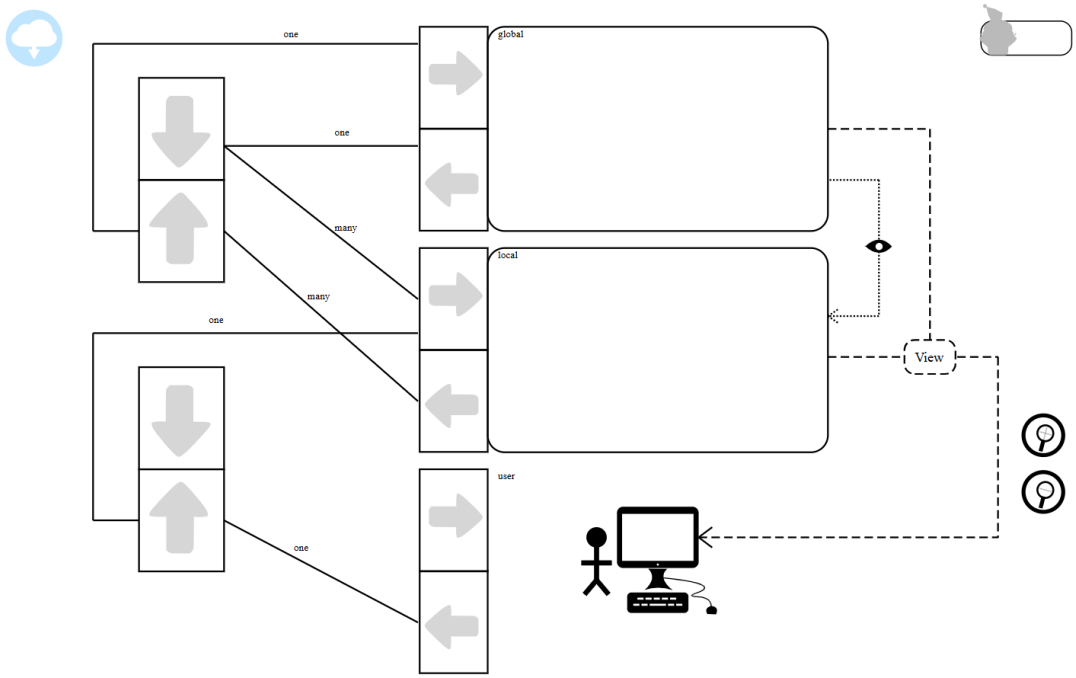


Figure 4.1: The Overall mode of CSCDraw, rendering models and their interfaces, as well as channels. Solid lines connect the channels with the statecharts. In this view, the contents of the channels are editable. Dotted and dashed lines indicate read-only access. The user appears as a peer of the statecharts in this overall view.

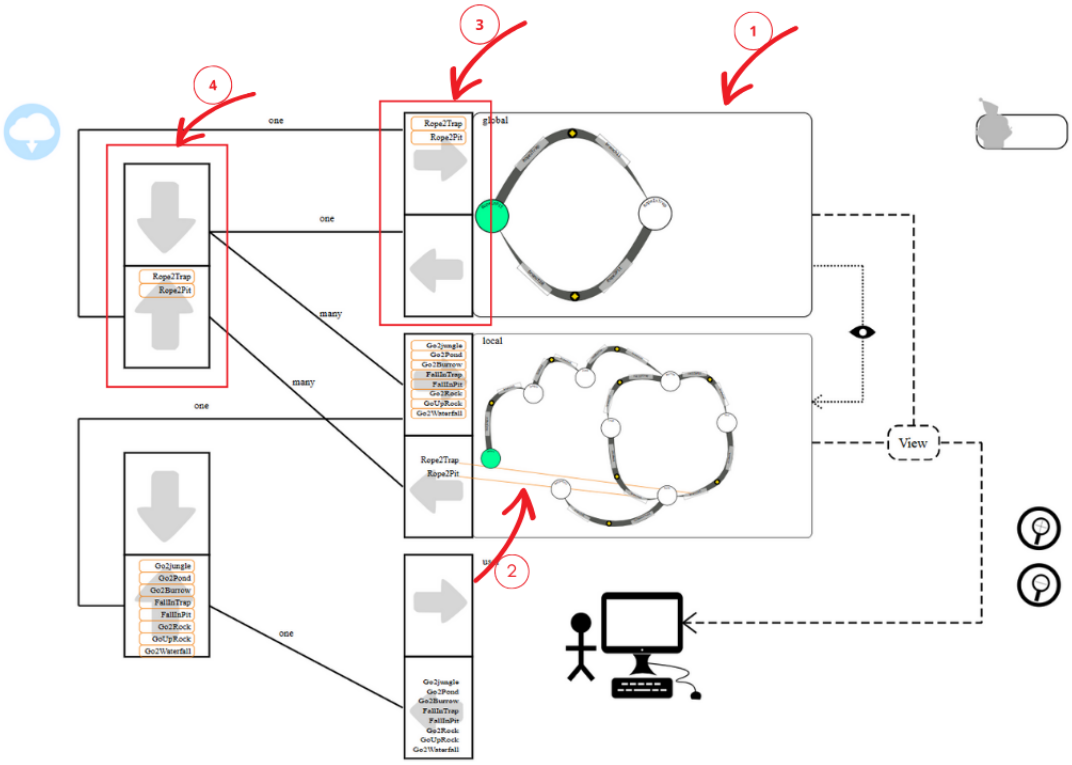


Figure 4.2: Active elements in the Overall mode of CSCDraw, including models/statecharts, synchronizing messages, interfaces, and channels.

OM3. Model Interfaces: Messages generated from transitions in a statechart, will wait in the source model's interface to be dragged to a potential channel, making them available as synchronizing messages to the connected statechart. Grey arrows represent the direction in which messages are transmitted from one statechart to the other through channels.

OM4. Channels: In CSC, interaction between statecharts is allowed through message-passing via channels. CSCDraw uses grey arrows to visualize the direction of the messages, for the channels, similar to the interfaces. Furthermore, when dragging a message to a channel, green highlighting indicates legal destinations within the channel, addressing Implementation Decision I17, *Channel Highlight*. Figure 4.3, illustrates an example in which the `Go2jungle` message is being dragged from the local statechart's input interface. Dropping the message to the lower highlighted channel means that the message can be generated through user interaction, while the upper highlighted channel would be used for synchronizing messages sent from the global statechart to the local statechart. This feature makes channel design possibilities clear, minimizing cognitive load.

Other elements of the Overall mode are described in the order shown in Figure 4.4. They will be referred to throughout the thesis by their E numbers.

E1. User: A user plays the role of a black box statechart in CSC. Messages created by user interactions, such as a button click, are represented as messages going from the user's interface to the local statechart.

E2. View Function: In CSC, due to using *LG-MVU* for Principle P3, *Pure Functions*, the update function will update the state of the application when a user

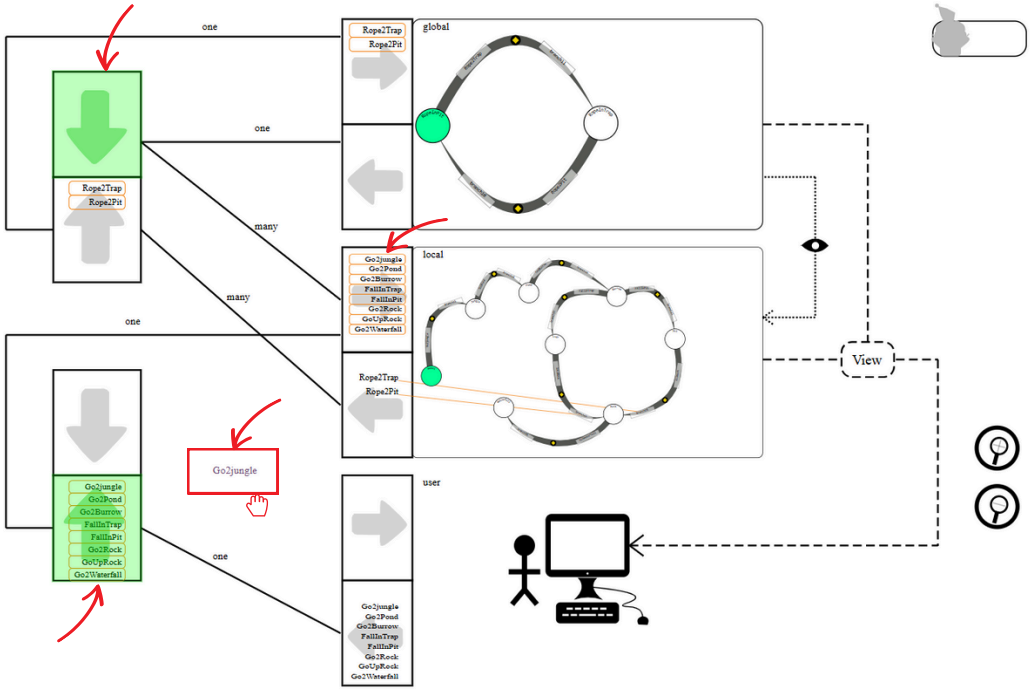


Figure 4.3: An instance of the state of a message being dragged to a channel in the Overall mode of the CSCDraw. In this example, the *Go2Jungle* message is being dragged from the local statechart's input interface and the potential paths in the channels are being highlighted.

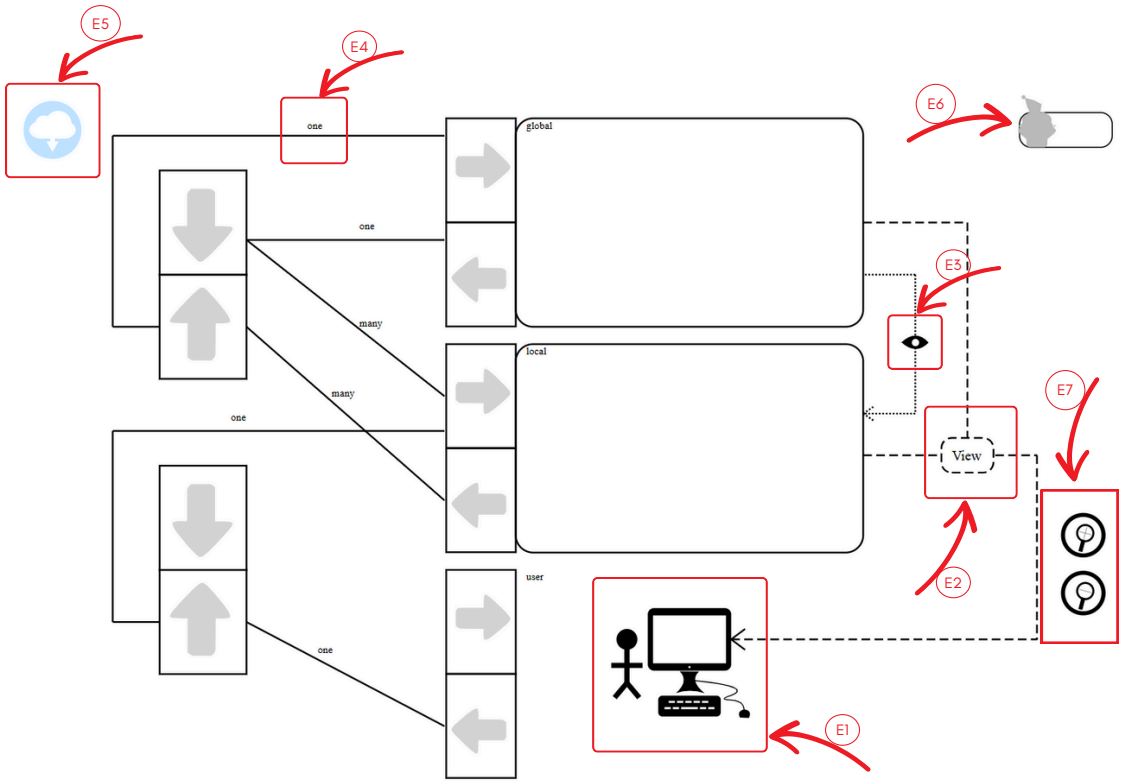


Figure 4.4: Elements shaping the Overall mode of CSCDraw.

interacts with the app elements. Then, the view function renders the updated model on the user's screen. In CSCDraw, as shown in Figure 4.4, the user sends messages to the local statechart while receiving no messages through its interface. Therefore, the connection between the user's input interface and the channel is eliminated. Instead, the current state of the app will be drawn on the user's screen by the view function. Element E2 addresses the Implementation Decision I10, *View Function*.

E3. Read-Only Access: In CSC, due to *Read-Only Access*, as Consequence C3 of Principle P3, *Pure Functions*, the local model has read-only access to the global model. In other words, the global statechart is in the scope of the local statechart. In CSCDraw, this access is visualized through a dotted line, showing the flow of the data with an arrowhead, and containing an eye icon as a symbol of read-only access. Element E3 contributes to the Implementation Decision I1, *Isolation*.

E4. Cardinality: In CSC, statecharts connected by channels have cardinality relationships, i.e., many:one for local:global, and one:one for local:user. CSCDraw visualizes this cardinality analogous to the entity-relationship diagrams used in database design. Element E4 addresses Requirement R5, *Cardinality*.

E5. Download Button: Once statecharts are drawn, synchronizing messages are connected and the the messages are dragged to the right channels, the code generator would be ready to generate the skeleton code. Clicking on the Download button leads it to check whether every message is connected to a channel. If channels completely cover the messages, CSCDraw generates the Elm skeleton

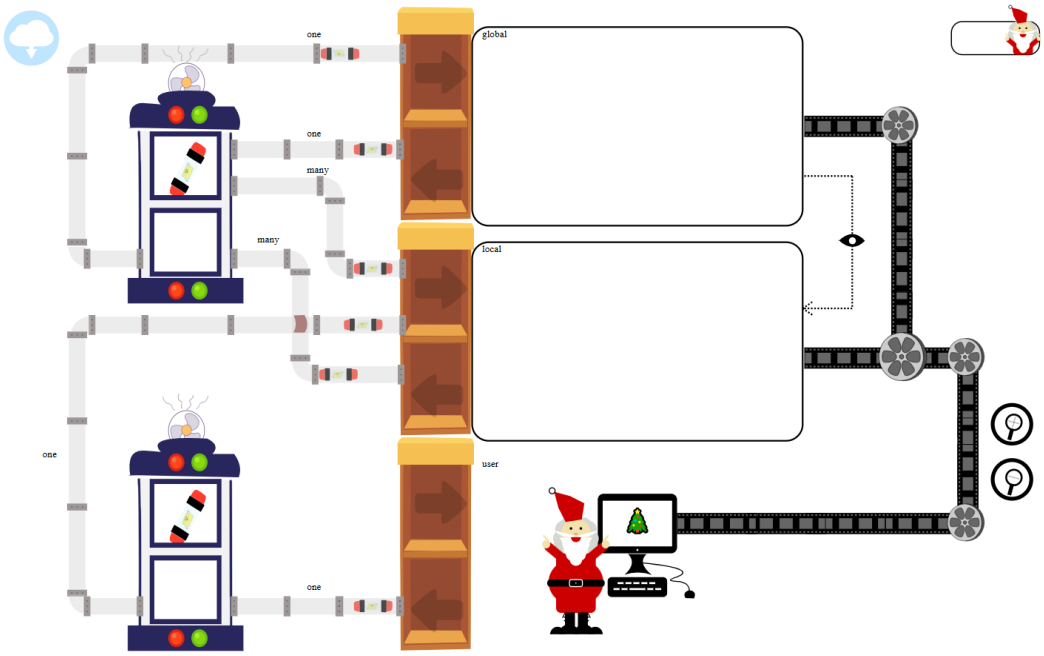


Figure 4.5: The Analogy mode of CSCDraw. Elements are conceptualized with real-world objects.

code, and gives access to it, either as a downloadable or copyable text. Element E5 addresses the Requirement R3, *Code Generator*, and the Implementation Decisions I18, I20, I21, *Channel Completion Check*, generate code in *Elm*, and *Functional Code with Comments*.

E6. Analogy Mode: To help explain CSC's message-passing concepts, Analogy mode (Figure 4.5) can be activated, in which concepts are represented as real-world objects. For example, messages are represented as envelopes and the message-passing is pictured as a pneumatic pipe system. Element E6 addresses the Implementation Decision I9, *Analogy mode*.

E7. Zoom Buttons: For easier usage, CSCDraw allows the screen to be zoomed in

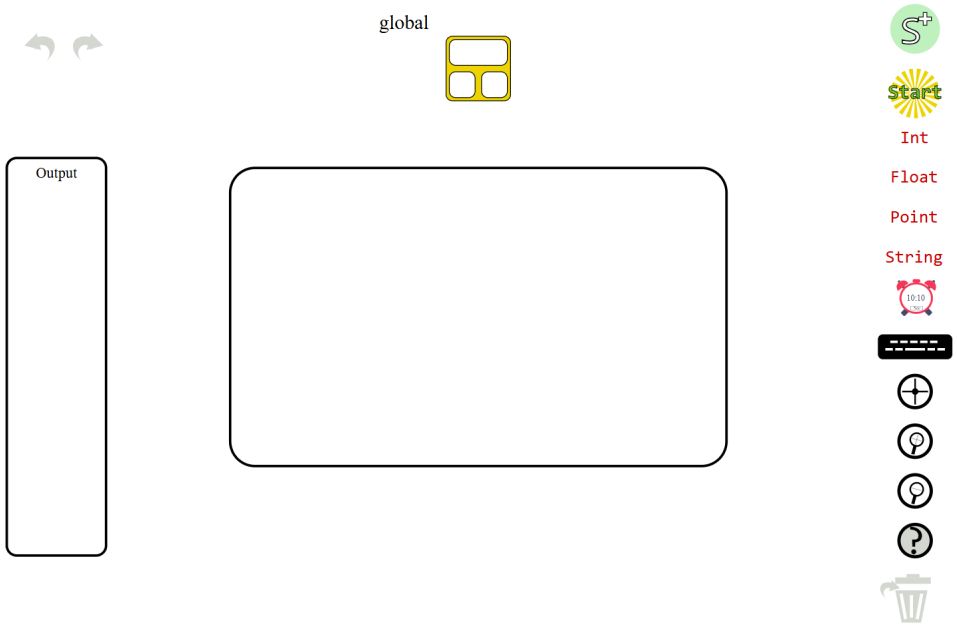


Figure 4.6: The SCEditing mode allows the modification of a statechart.

or out. Element E7 addresses the Implementation Decision I16, *Pan/Zoom*.

4.2 SCEditing Mode

Clicking on a statechart in the Overall mode results in the transition from the Overall mode to the SCEditing mode where model modification is possible, as shown in Figure 4.6. This mode addresses Implementation Decision I6, *SCEditing mode*.

In teaching a first-year computer science course, the instructor reported the eagerness of students in using SDDraw to build their projects' skeletons. Due to this success, CSCDraw follows the conventions established by SDDraw, addressing the Implementation Decision I12, *SDDraw UI*. This section describes the key features, elements, and possible modifications in this mode. They will be referred to throughout the thesis by their KF, EM, and M numbers, respectively.

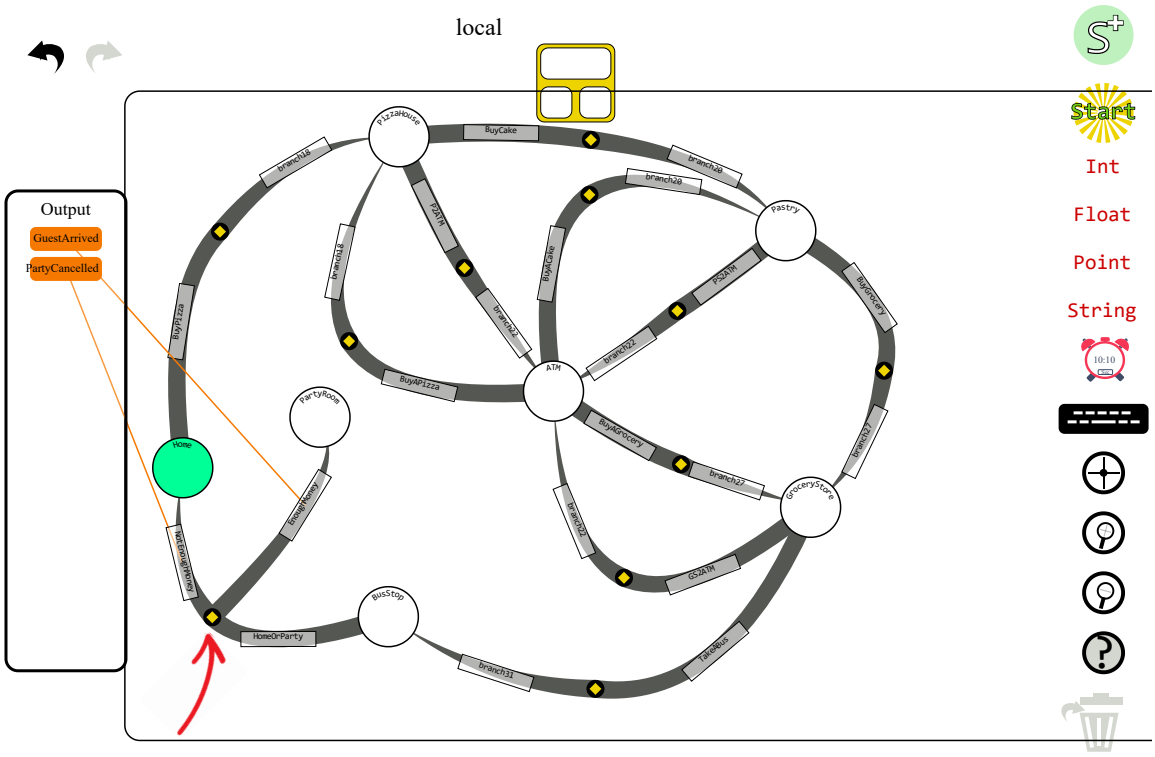


Figure 4.7: An example of using conditional branches choosing transitions to a state among a set of states based on meeting a condition.

KF1: Conditional Branches: Unlike the simple state diagrams, our statecharts have conditional branching, shown as circles with inscribed yellow diamonds from which multiple branches can lead to different states (in Figure 4.7). The yellow diamond is always present on a transition to allow the addition of multiple branches as needed. This models case expressions in pure functional languages. Figure 4.7 pictures an example of CSCDraw allowing the transitions to different states based on meeting a condition. In this example, the transition from `BusStop` state can lead to either the `PartyRoom` or the `Home`.

KF1 addresses Requirement R4, *Conditional Branches*.

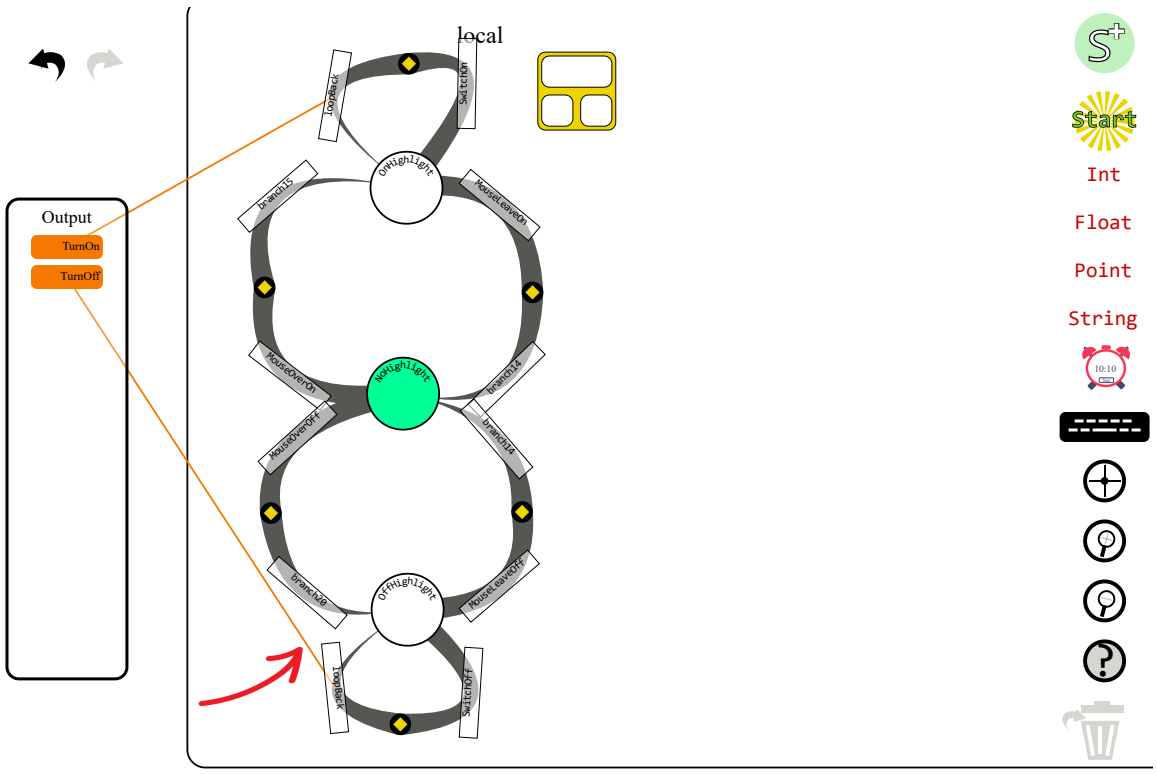


Figure 4.8: An example of the generation of a Synchronizing Message in the SCEditing mode.

KF2: Synchronizing Messages: CSC allows communication between different statecharts by transmitting Synchronizing Messages via channels. In CSCDraw, a Synchronizing Message can be sent by connecting the originating branch to the list of messages available in the inner interface. Figure 4.8 shows an instance of making a Synchronizing Message. Particularly, the `SwitchOff` transition causes the `TurnOff` synchronizing message to be sent.

KF2 addresses Implementation Decision I2, *Synchronizing Message Modification.*

We now describe each element of CSCDraw's SCEditing mode, as enumerated in

Figure 4.9.

EM1. Navigator: Changing the state of the CSCDraw from the SCEditing mode to the Overall mode is possible through the Navigator button. Element EM1 addresses the Implementation Decision I15, *Navigator*.

EM2. SCIndicator: This feature provides information on the statechart that is being edited. Element EM2 addresses the Implementation Decision I14, *SCIndicator*.

EM3. Bounding Box: A Bounding Box indicates the space that the statechart will occupy in the Overall mode.

EM4. Inner Interface: Synchronizing Messages are the transition labels of the other statechart dragged by the user to the channel. They are duplicated in the inner interface to define its originating branch. Figure 4.8 provides an example of a synchronizing message being connected to its source branch. Element EM4 supports the Implementation Decisions I1 and I2, *Isolation*, *Synchronizing Message Modification*.

EM5. State Instance: A state can be created by dragging the State Instance and dropping it on the screen. Element EM5 addresses Implementation Decisions I7 and I12, *CSC Modification* and *SDDraw UI*, respectively.

EM6. Initial State Instance: The first state that is created in a statechart is the initial state by default in CSCDraw. Changing the starting state is possible by dragging an Initial State Instance into the desired state. Element EM6 addresses the Implementation Decisions I7 and I12, *CSC Modification* and *SDDraw UI*, respectively.

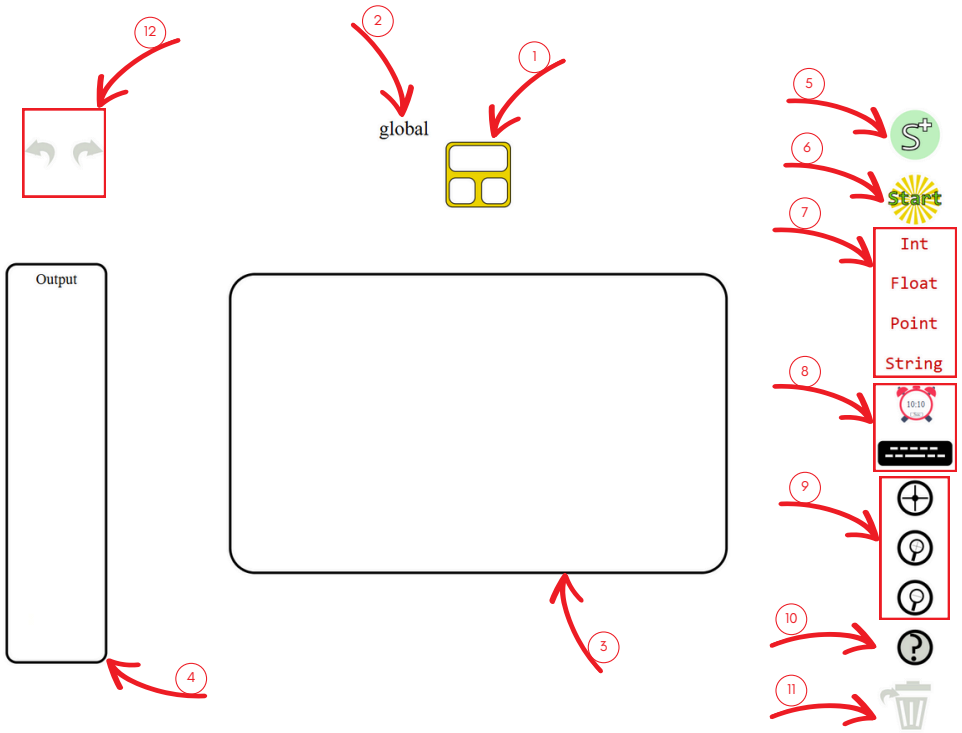


Figure 4.9: The UI elements active in SCEditing mode.

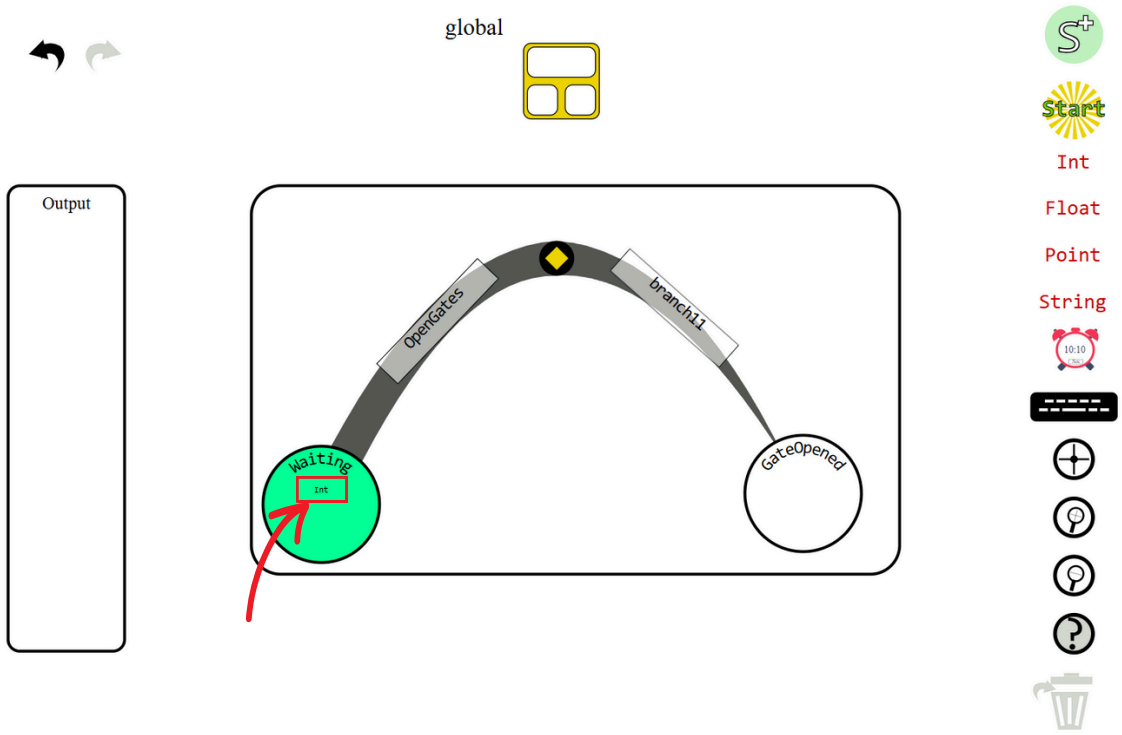


Figure 4.10: An example of a state storing a field of type `Int`.

EM7. Elm Types: States and Transitions are represented as constructors for Algebraic Datatypes in the generated code. Each constructor can have multiple fields of associated data. Dragging the data types to the states or transitions accomplishes this. Figure 4.10 illustrates an example where the `Waiting` state stores a field of type `Int`. Element EM7 addresses the Implementation Decisions I7 and I12, *CSC Modification* and *SDDraw UI*, respectively.

EM8. Elm Subscriptions: Elm subscriptions generate messages based on time intervals or keyboard actions. They can be attached to transitions, which translates into that transition's constructor of the corresponding Algebraic Datatype in the generated code. Element EM8 addresses the Implementation Decision I7,

CSC Modification.

EM9. Pan/Zoom: For easier usage, CSCDraw allows the screen to be zoomed or panned. Element EM9 addresses the Implementation Decision I16, *Pan/Zoom*.

EM10. Help: This button displays a help page. Element EM10 addresses the Implementation Decision I12, *SDDraw UI*.

EM11. Trash Bin: State, Transition, and Branch deletion is possible by dropping the item into the Trash Bin. To provide feedback, the Trash Bin will be highlighted in red and its lid will be opened when an item enters its region. Element EM11 addresses the Implementation Decisions I7 and I12, *CSC Modification* and *SDDraw UI*, respectively.

EM12. Undo/Redo: CSCDraw allows Undo and Redo actions while editing a statechart. Element EM12 addresses the Implementation Decision I19, *Undo/Redo*.

To address the Implementation Decision I7, *CSC Modification*, CSCDraw allows the addition and deletion of the statechart's key elements, i.e. states, transitions, and branches. The operations are as follows:

M1. State Modifications: CSCDraw allows State operations including addition, deletion, and setting the initial state. A state can be created by dragging a state instance, Element EM5 in Figure 4.9, and dropping it on the canvas. A state can also be set as an initial state by dragging and dropping the `Start` instance, Element EM6. Also, removing a state is possible by dragging the state to the trash bin, Elment EM11. Operation M1 also addresses I12, *SDDraw UI*.

M2. Transition Modifications: In CSCDraw, a transition can be created by clicking on a source state, pulling the transition, and letting go of the mouse in a destination state. When clicking on a state, an arrow would appear above the state, as shown in Figure 4.12, which can be pulled to generate the transition. Additionally, CSCDraw does not allow ambiguous design of statecharts, when more than one transition with the same label are fired by the same state. Particularly, in the case of drawing such conflicting transitions, the new transition will replace the previous one. The deletion of a transition works analogously. Operation M2 also addresses I12, *SDDraw UI*.

M3. Branch Modifications: In CSCDraw, a branch can be created by clicking on a conditional diamond, pulling the branch, and letting go of the mouse in a destination state. When clicking on a diamond, an arrow would appear above the diamond, as shown in Figure 4.11, which can be pulled to generate the transition. The deletion of a branch works analogously.

To address Implementation Decision I8, *Renaming*, CSCDraw allows the renaming of the statechart's key elements, i.e. States, Transitions, and Branches. Clicking on a state leads to the appearance of the cursor, ready to change the state name. Figure 4.12 shows an example of a state, i.e. `State12`, ready to be renamed. The renaming of a transition and a branch works analogously.

4.3 Validation

As mentioned in Section 1.4, the existing tools have limitations in supporting our CSC principles. Therefore, to answer our proposed RQ2, we gathered requirements

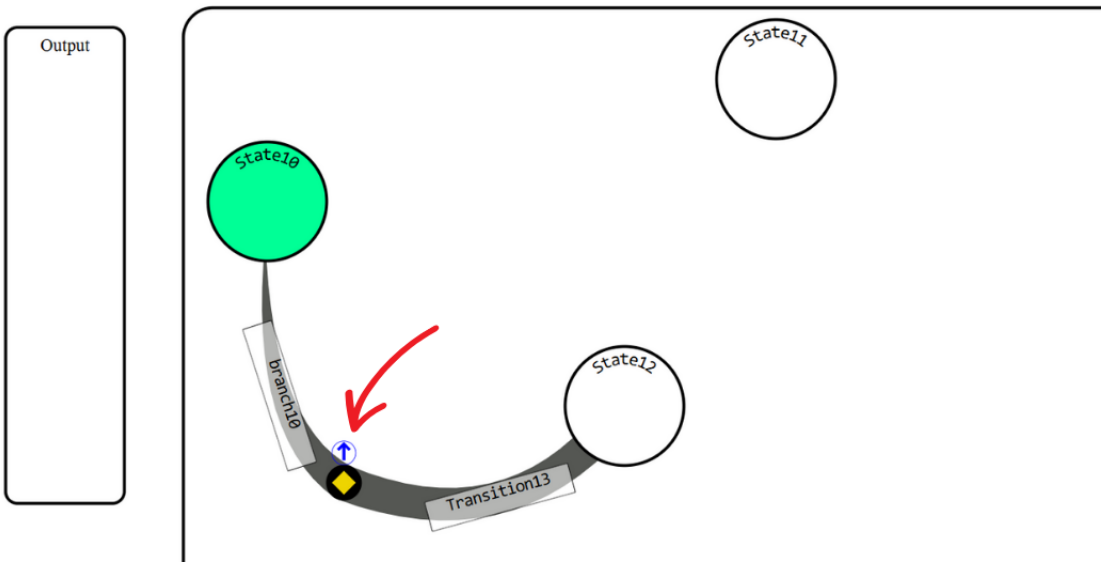


Figure 4.11: Making a branch for a transition is possible by clicking on a diamond and dragging the arrow onto the destination state.

for a suitable tool to support CSC, and implemented CSCDraw to satisfy those requirements. Then, to validate CSCDraw's support for those principles, we took the statechart drawn in Microsoft Visio, Figure 4.13, for the *Login System*, and redrew it in CSCDraw. Figure 4.14 illustrates the redrawn version.

The CSCDraw version of the diagram suggests six main benefits compared to the statechart drawn in Microsoft Visio:

1. **Separation of Concerns:** *Separation of Concerns (SoC)* constructs CSC's Principles P4, which aims to handle complexities by dividing a problem into subproblems and solving them one at a time. SoC in CSCDraw is achieved by:

   1.1. **Isolation:** CSC doesn't allow direct modifications between different statecharts, instead, they can communicate by sending synchronizing messages

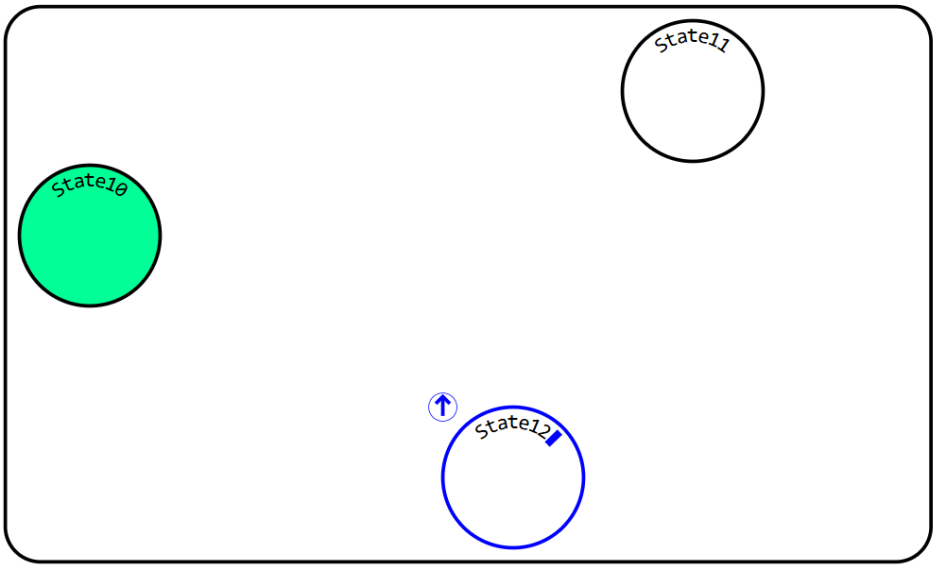


Figure 4.12: An example of CSCDraw in `StateRename` mode. In this mode, the state will be highlighted in blue and the cursor will be ready for renaming operations.

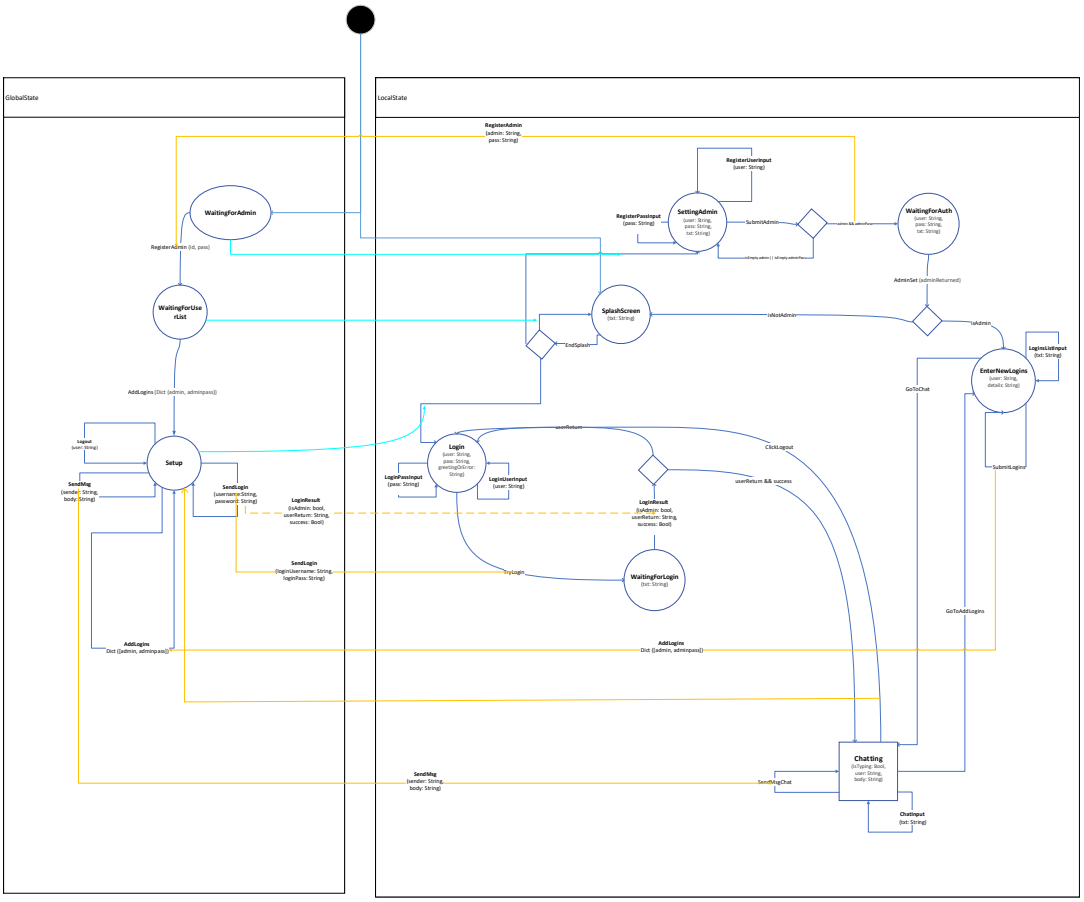


Figure 4.13: The statechart of the *Login System* written in TEASync, drawn in Microsoft Visio. The solid and dashed yellow lines correspond to the synchronizing messages to and from the global statechart, respectively. The light blue lines show the read-only access of the global statechart to the local statechart. This model does not satisfy our CSC principles.

through channels. Hence, it forces statecharts to be isolated. This isolation is obvious in CSCDraw's interface, while the Visio version does not offer such compartmentalization.

1.2. Distinct Modes: Two editing modes in CSCDraw reflect two types of consistent views of the system: (1) the overall view showing the contents and connections of channels, and (2) the SCEditing view, in which the statechart can be modified and its interaction with the rest of the system can be understood in terms of the interface. However, Visio does not offer such separation; There is only one view for editing both statecharts.

2. **Read-only Access:** CSC gives read-only access from the global statechart to the local statechart, as a result of using LG-MVU, due to Principle P3, *Pure Functions*. However, this read-only access has no analogue in existing languages such as UML state machines. Therefore, to represent the read-only access in Visio, we used colour—the cyan lines. But this is still not easy to follow, and conflicts with UML or Harel's statecharts conventions in terms of the hierarchy, where the parent statechart can see and modify the children statecharts. In CSCDraw, the global statechart being in the scope of the local statechart is visualized through the dashed line containing an eye icon as a symbol of read-only access. This way of representing the read-only access offers a more organized and easier-to-understand visualization compared to the Visio version.

3. **Code Generation:** CSCDraw supports code generation from the designed CSCs, which preserves Principle P2, *Model-Driven-Development (MDD)*. However, this code generation is not possible through Visio [59].

Additionally, CSCDraw's code generator has two features that distinguish it from other MDD tools;

3.1. Pure Functions: Due to Principle P3, *Pure Functions*, the skeleton code generated by CSCDraw promises the absence of side effects, resulting in the faithful translation of models into a code that preserves CSC semantics.

3.2. Abstraction: CSCDraw's code generator preserves the Principle P5, *Abstraction*, by using Event-Driven Programming (EDP) abstract concepts including events instead of confusing details such as threads.

4. **Enforcing CSC Principles:** CSCDraw only allows syntactically correct actions to be performed. In other words, in CSCDraw, there is no way to perform actions violating CSC principles. This restriction helps beginners design less error-prone systems and subsequently, prevent them from early frustrations. In contrast, Visio offers freedom to the designers which may cause complexities depending on the expertise level. For instance, in Visio a user can easily draw a direct transition from one statechart to another, which violates CSC's Principle 2.12, *SoC*, and its Consequence C4, *isolation*.

5. **Preventing Ambiguity in Statecharts:** CSCDraw does not allow multiple transitions with the same label to be sent from one source state to multiple destinations. This restriction, however, is absent in Visio, which can lead to contradictory designs and undesirable behaviour of the system.

6. **Message Organization:** In the Visio version, the difference between local messages and synchronizing messages is only noticeable by using different colours. In contrast, CSCDraw uses different shapes and colours for different types of

messages, as well as, interfaces to organize synchronizing messages both in SCEditing and Overall modes. In Overall mode, CSCDraw organizes a statechart's input and output messages into a bipartite interface containing grey arrows to visualize the flow of the information. Furthermore, when connecting messages to channels, the potential paths in channels will light up. This prevents programmers from designing impossible interactions by mistake. This message organization helps designers have an easier experience designing their multi-user systems by reducing the cognitive load. Visio does not offer such an organization due to the absence of interfaces and channels.

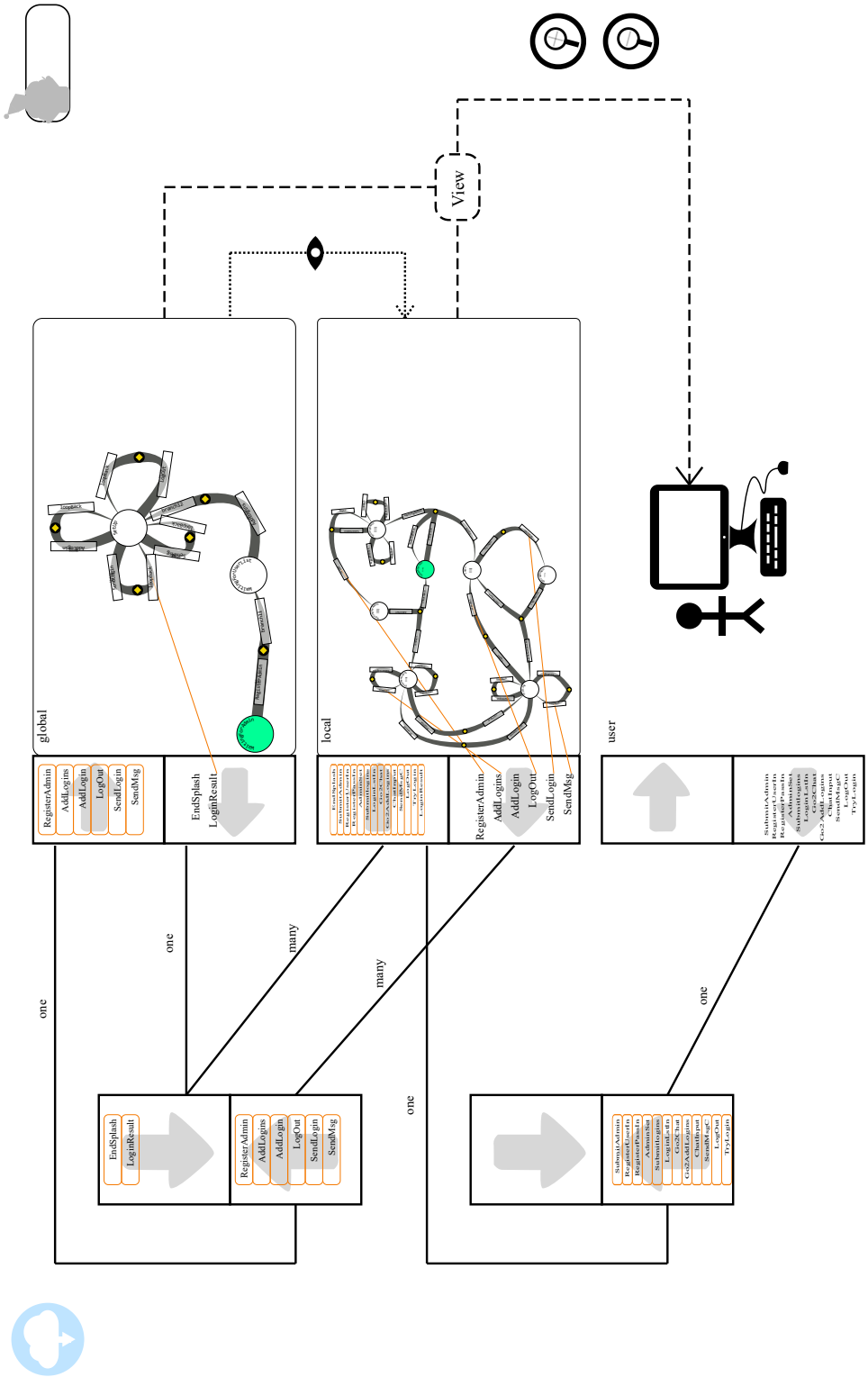


Figure 4.14: The *Login System* redrawn in CSCDraw from Figure 4.13. The isolation of statecharts is visible and the communication between them is visualized in terms of synchronizing messages transmitted by channels. Messages are organized into bipartite interfaces showing the flow of the information. Channels will point out the possible paths when dragging a synchronizing message. Code generation is also possible from the designed CSCs. This model successfully preserves CSC principles.

Chapter 5

Pilot Study

Evaluating CSC's success in achieving its goal of making concurrency easy to learn and use for beginners (answering RQ3) requires conducting a large-scale experiment. For instance, this rigorous experiment could measure how well students perform in making a multi-user application through CSCDraw (using CSC concepts) compared to Java Threads.

However, before conducting such an experiment, we need to answer RQ3.1, which requires investigating the best way of teaching CSC. Thabane et al. [82] defines a pilot study, also known as a feasibility study, as a brief study to determine the feasibility of a large-scale, confirmatory experiment. The main purpose of such a study is to prevent researchers from spending time and money on an investigation likely to fail due to insufficient data or time for assigned tasks, or other issues which are hard to estimate without any experience. Therefore, we present the design of a pilot study to prepare for the large-scale evaluation.

To pilot such a study, we can leverage our experience running summer camps and coding workshops for senior primary students. Our base assumption is that

participants have no prior knowledge in coding with text-based languages such as Elm. To ensure the study targets total beginners, data on participants' background in coding should be collected through a pre-quiz. Additionally, to prevent participants from feeling overwhelmed by teaching them everything at once, we distribute the pilot study into two main phases:

1. Teaching Event-Driven Programming (EDP) concepts through single-player adventure games with SDDraw (introduced in Section 2.10).

2. Teaching CSC concepts through multi-player adventure games with CSCDraw.

Each phase consists of five main steps. The overall perspective of steps one to five for the first and second phases are displayed in Table 5.1 and Table 5.2, respectively. The tables include information on each step's activities, game features, whether the games store parameters in their models (such as an integer variable in a particular state), and the desired learning outcomes for that step. Each phase includes two challenges (in the first and third steps) and a quiz (in the fifth) step consisting of multiple-choice questions. The participants' answers to the challenges/quizzes will be gathered anonymously through Google Forms. This means emails and identities should not be collected. Analysis on the collected data will tell us how well the students are learning EDP and CSC concepts. At the end of the pilot study, we may repeat it if needed.

In the rest of this chapter, we describe the details of the hoped-for learning outcomes and the activities done in each phase in Sections 5.1 and 5.2, respectively.

Step	StepName	Game	Parameterized	Features	Learning Outcomes
1	Introduction (Challenge)	Mood	No	Two states: Happy/Sad	Remember (SD, SDDraw, and Case Expressions)
2	Play and Design	Hiking	No	Seven states, SD includes only states and simple transitions.	Understand (How to design an SD.)
3	Play and Choose (Challenge)	Santa	Yes	Seven states, SD includes self-loops.	Understand (Translating a game to an SD.)
4	Read, Design, and Code	Rope	No	Seven states, SD includes one tricky transition.	Apply (The creation of an adventure game.)
5	Evaluation	Saving Swan	No	SD includes six states, transitions, and a self-loop.	Apply (Answer evaluation questions.)

Table 5.1: Overview of the steps used in the first phase of the pilot study using SDDraw. Learning Outcomes correspond to the first three levels of Bloom's Taxonomy (shown in Figure 1.12).

5.1 Learning Outcomes

The hoped-for learning outcome of this study is for participants to achieve the first three levels of Bloom's Taxonomy (*Remember*, *Understand*, and *Apply*). In particular, the relation between each Bloom's level and each study step is:

Remember: The first step tries to achieve the *Remember* level. Particularly, this step provides an introduction to each phase's required concepts. These concepts

Step	StepName	Game	Parameterized	Features	Learning Outcomes
1	Introduction (Challenge)	Lights	No	Two states: LightsOn/-LightsOff	Remember (CSC and CSCDraw.)
2	Play and Design	MP-Hiking	Yes	At least three players must arrive at the Terminal to open gates and return home	Understand (Translating a game to a CSC.)
3	Play and Choose (Challenge)	Party	Yes	At least three players must manage their money to start a party.	Understand (Translating a game to a CSC.)
4	Read, Design, and Code	MP-Rope	Yes	Only one rope is available to climb up both the Pit and the Trap.	Apply (The creation of an MP adventure game.)
5	Evaluation	Open the Cave	Yes	Players collaborate on collecting 150 sticks to start a fire.	Apply (Answer evaluation questions.)

Table 5.2: Overview of the steps used in the second phase of the pilot study using CSCDraw. Learning Outcomes correspond to the first three levels of Bloom's Taxonomy (shown in Figure 1.12).

include *state* and *transition* in state diagrams, as well as, *synchronizing messages*, *Local-Global portions*, and *channels* in CSC. The first step also includes a challenge to measure how well participants can *Remember* the concepts.

Understand: The second and third steps of the study aim to achieve the *Understand* level, by asking participants to play games and translate them into state diagrams or CSCs. In particular, the second step focuses on describing the concepts and discussions, while the third step evaluates how well they *Understood* the concepts.

Apply: The fourth and fifth steps try to accomplish the *Apply* level. Specifically, the fourth step requires the participants to interpret a scenario to the high-level visual representation (SD/CSC), and its low-level implementation in code. The fifth step includes a quiz that measures how well the *Apply* learning outcome is achieved.

5.2 Experiment Details

First Phase: The first phase focuses on teaching EDP through SDDraw. In this phase, the utilized apps are single-player games, and participants are expected to eventually be able to apply what they learned in implementing a game according to a given scenario. Figures and scenarios of the games used in this phase can be found in Appendix A.1.

Referring to the Table 5.1, the detailed steps are as follows:

Step 1. Introduction: Participants will be provided with an introduction to SDDraw's interface and code generation, as well as, state diagram's fundamental concepts such as state and transitions. This introduction uses a simple mini-game called *Mood*, which consists of two states: *Happy*/*Sad*

and two transitions: *Cry*/*CheerUp*. Figure 5.1 demonstrates the state diagram of this game. The graphical interface of the Mood game can be found in Appendix A.1.1.

To help participants better understand how to work with SDDraw, they will be asked to implement the same map as the Mood game's state diagram. Mentors will assist them with any potential problems that they might have with this introductory task.

Thereafter, a multiple-choice question should be used to evaluate whether they can define how the graphical representation (state diagram) translates into the low-level generated code. In this regard, the question can include the case expression of the app's view function that needs to be completed. In this case, the choices are the complete version of that case expression. This task will measure how well they can *remember* the basic concepts.

Step 2. Play and Design: To make sure participants have a good *understanding* of state diagrams, they will be asked to first play a game called *Hiking*, and then, draw the state diagram, accordingly. The Hiking game consists of seven states and eleven transitions. Figure 5.2 represents the map of this game. The player starts from the MainStreet and hikes through other states until they reach the Terminal. They should be able to go back and forth between different states, except for the Terminal. Once they arrive at the Terminal the game is over. This step focuses on improving their *understanding* of state diagrams and EDP-related concepts such as user interaction. Evaluation of how well this learning outcome is achieved will be done in Step 3. In this regard, after they finish their task of drawing

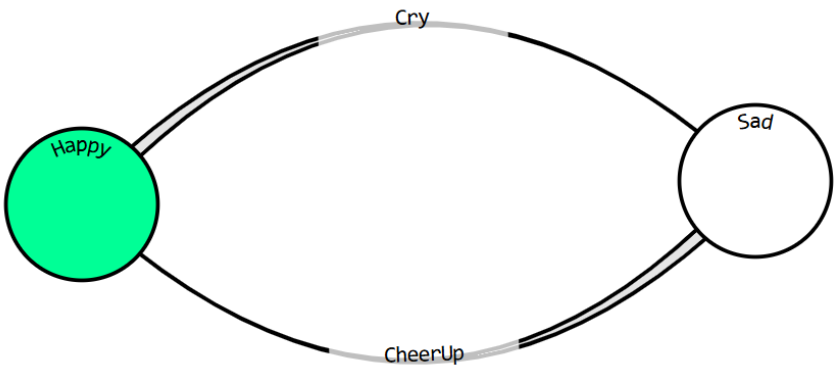


Figure 5.1: The state diagram of the Mood game, a mini-game to introduce SDDraw, used in Step 1 of the pilot study's first phase.

the state diagram, the correct solution (Figure 5.2) will be released and any misconceptions will be discussed with mentors.

Step 3. Play and Choose: To measure how well participants *understood* state diagrams and EDP-related concepts, they will be asked to translate a game (called *Santa*) into a state diagram. Particularly, after they played the Santa game, they will be provided with four different diagrams and be asked to choose the one matching that game (Figure 5.3). The story of the Santa Game and the figures illustrating its graphical interface can be found in Appendix A.1.2.

Step 4. Read, Design, and Code: Prior to the evaluation of how well participants can *apply* the materials they learned, in new situations, they will

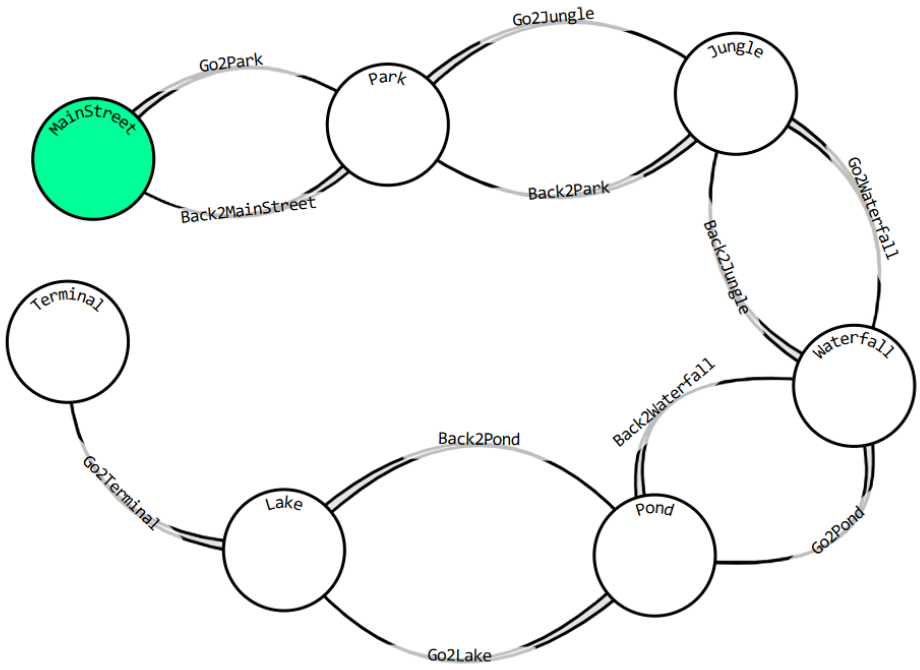


Figure 5.2: The state diagram/map of the Hiking game, a mini-game to introduce SDDraw, used in Step 2 of the first phase.

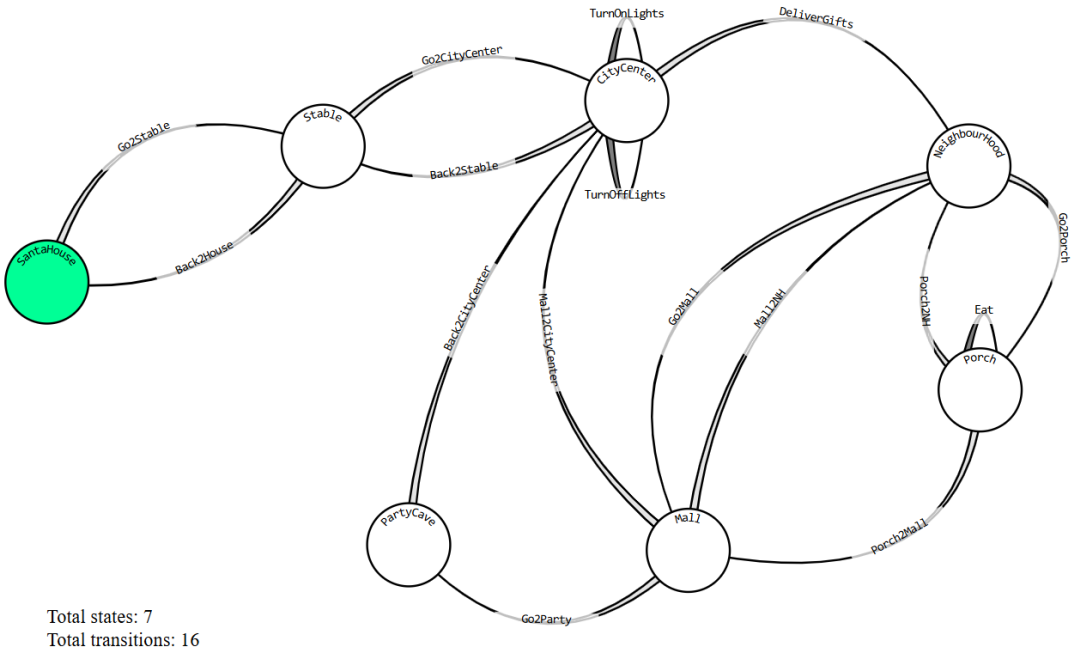


Figure 5.3: The state diagram solution to the challenge done in Step 3 of the first phase (Santa game).

have an opportunity to implement a game themselves. They will be provided with the scenario of a game called *Rope*. The scenario can be found in Appendix A.1.3. Then, they will be asked to interpret the English context with respect to state diagrams and EDP concepts. The interpretation consists of designing the state diagram of the game, comparing their design with the revealed solution (shown in Figure 5.4), discussing any potential questions with mentors, code generation, and compiling the code on the web-based IDE (STaBL.Rocks).

Step 5. Evaluation: To measure how well participants can use their learned information and *apply* them in solving new challenges, they will be asked to do a quiz including four multiple-choice questions. The first three questions evaluate the ability to design, after interpreting a scenario into a state

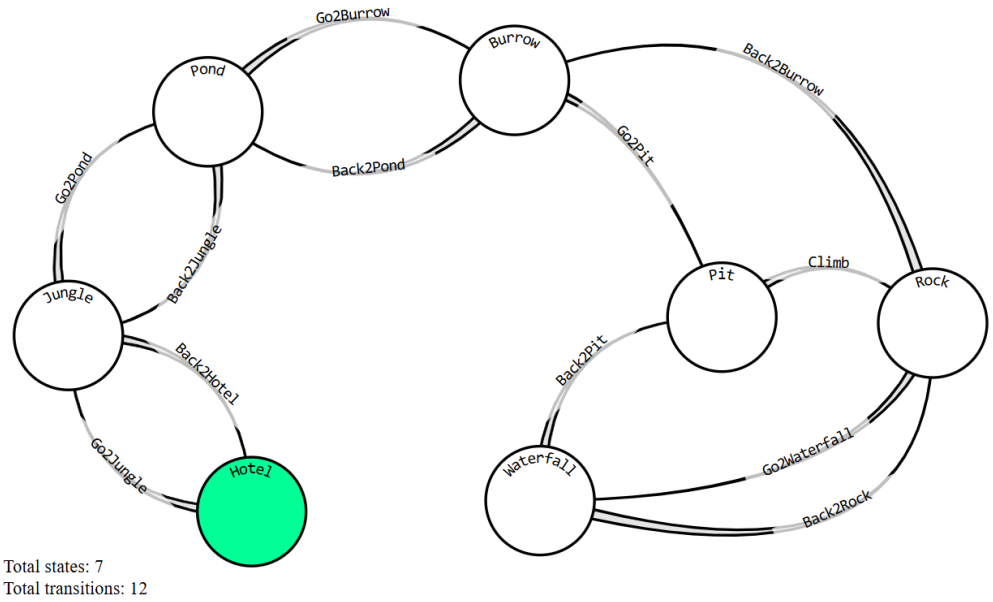


Figure 5.4: The solution state diagram to the challenge asked in Step 4 of the first phase (Rope game).

diagram. The last question measures their ability to apply changes in the low-level implementation code.

The questions and their answers (shown in bold font) are as follows:

> Read the scenario of an adventure game called *Saving Swan* and answer the questions.

A tundra swan has lost its family during migration. Right now, the swan is swimming in a pond, calling out in search of its group. Our team has tracked the location of the rest of the flock. Your mission is to transport the swan to its family safely. In this regard, you have first to visit our station to collect the necessary equipment. Then, travel past Teddy Trail and cross Lily Bridge to reach the pond. You should not have any trouble finding the swan, as it is making loud calls. Once you've taken and secured the swan, head north, where its family is waiting. You'll need to pass through Maple Jungle and find the Province's Big Lake. Once you arrive at the lake, release the swan so it can reunite with its group.

Good luck on your mission!

Q1. Consider the map (state diagram) of the *Saving Swan* Game. Which state is the initial state?

a) Lily Bridge

b) Pond

c) Saviors' Station

d) Province's Big Lake

Q2. What can be a potential button in the game when the Savior is in Teddy trail?

a) Go to Maple Jungle

b) **Go to Lily Bridge**

c) Take the Swan

d) Pond

Q3. In the state diagram, how can we represent the action of *taking the swan* in the Pond?

a) **A self-loop transition in the Pond state.**

b) A transition from Lily Bridge to the Pond.

c) Just the Pond state.

d) none of the above.

Q4. Imagine you want to draw a bear in the Teddy Trail. Which code snippet describes your implementation of the game?

a)
```
myShapes model =
  [
    ( case model.state of
        SaviorsStation  ->
            [ text "SaviorsStation"
                |> centered
                |> filled black
            , bear
            , group
                [
                    roundedRect 40 20 5
                        |> filled green
                ,   text "Go2TedT"
                        |> centered
```

```
                                |> size 8
                                |> filled black
                                |> move(0, -3)
                    ]
                        |> move (-25, -25)
                        |> notifyTap SS2TT
            ]
        (...)
```

b) (This is the answer.)

```
myShapes model =
  [
    ( case model.state of
        TeddyTrail  ->
          [ text "TeddyTrail"
                |> centered
                |> filled black
          , bear
          , group
                [
                     roundedRect 40 20 5
                          |> filled green
                ,    text "Go2LB"
                          |> centered
                          |> size 8
```

```
                                |> filled black
                                |> move(0, -3)
                    ]
                        |> move (-25, -25)
                        |> notifyTap TT2LB

            ]
        (...)
```

c)

```
myShapes model =
  [
    ( case model.state of
        LilyBridge  ->
          [ text "LilyBridge"
                  |> centered
                  |> filled black
          , bear
          , group
                  [
                       roundedRect 40 20 5
                            |> filled green
                  ,    text "Go2P"
                            |> centered
                            |> size 8
                            |> filled black
```

```
                                |> move(0, -3)
                ]
                    |> move (-25, -25)
                    |> notifyTap LB2P

            ]
        (...)
```

d)

```
myShapes model =
  [
    ( case model.state of
        MapleJungle  ->
          [ text "MapleJungle"
                |> centered
                |> filled black
            , bear
            , group
                [
                     roundedRect 40 20 5
                          |> filled green
                ,    text "Go2PL"
                          |> centered
                          |> size 8
                          |> filled black
                          |> move(0, -3)
```

```
18                  ]
19                      |> move (-25, -25)
20                      |> notifyTap MJ2PL
21
22              ]
23          (...)
```

We then, will move on to the next phase, where they go through the same steps with multi-player games.

Second Phase: The second phase focuses on teaching CSC concepts through CSC-Draw. In this regard, the activities used in this phase include multi-player games. By the end of this phase, we expect participants to be able to *apply* CSC concepts in solving new challenges. Figures and scenarios of the games used in this phase can be found in Appendix A.2.

Referring to the Table 5.2, the detailed steps are as follows:

Step 1. Introduction: In this step, participants will be provided with an introduction to the Local-Global Model-View-Update (LG-MVU). In this regard, a basic multi-player game, called *Lights* is used to conceptualize the private and shared portions of the system. Particularly, in this game, when a player hovers over a button it will light up only for them (local) (as shown in Figure 5.5), and when they click on a button, it will switch ON/OFF the lights for everybody connected to the server (global) (as shown in Figure 5.6). Therefore, this game consists of two global states: *LightsOFF* and *LightsON*, and three local states: *NoHighlight* (When no button is

being hovered over), *HighlightONBtn* (When the TurnONLights button is being hovered over), *HighlightOFFBtn* (When the TurnOFFLights button is being hovered over).

After introducing the concepts, the mentor will show the CSC of the Lights game (Figure 5.7), and will discuss the core ideas such as the separation of models into local and global and the synchronizing message causing the lights to turn ON/OFF for every user.

The mentor may also switch to the Analogy mode of the CSCDraw (Figure 5.8) to better conceptualize the materials. The analogy can be as follows:

> Santa is trying to turn the lights ON for everybody, so he clicks on the *TurnONLights* button, which causes a message to be sent from Santa's shelf (where he puts his messages) to the local statechart. Channels are here to help Santa transmit his messages. This message will trigger the *TurnONLights* transition in the local statecharts, which leads to a synchronizing message (SwitchON) being sent to the global statechart. Then, that synchronizing message will trigger the transition from the *LightsOFF* state to the *LightsON* state.

To measure how well participants can *remember* the CSC concepts, they will be asked to do a challenge which determines whether they are able to separate the program into local and global portions in a given scenario. The challenge and its solution (shown in bold) are as follows:

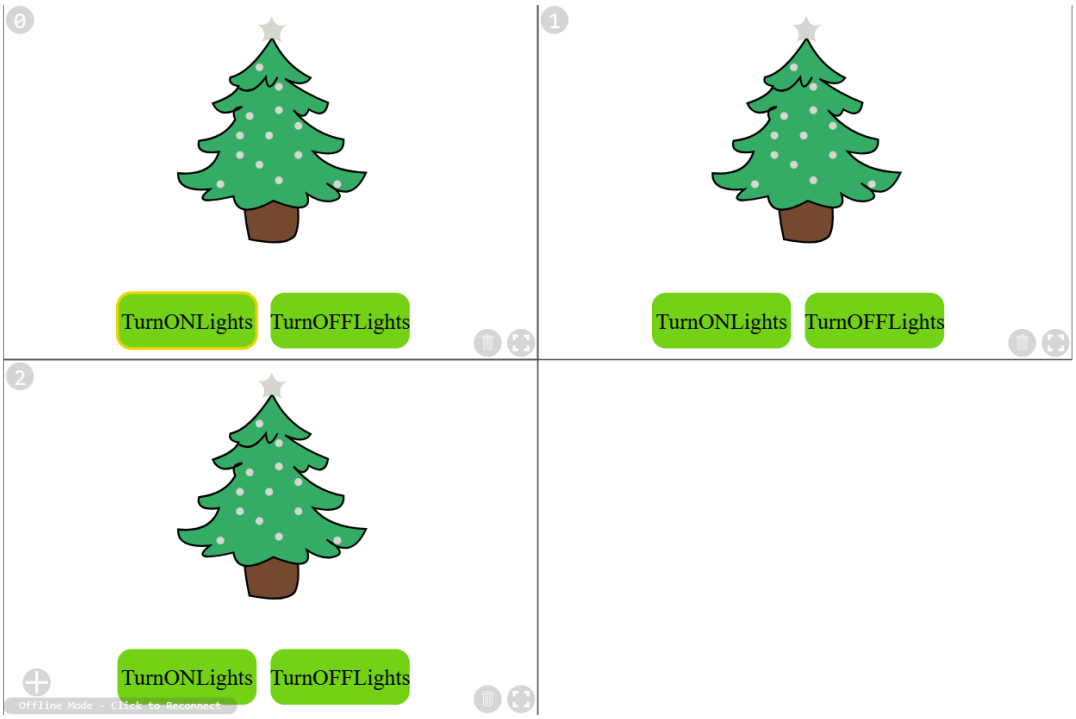


Figure 5.5: The state of the lights being OFF for every client (global), while the button highlight is private to each user (local), in the Lights Game. This game is used in Step 1 of the study's second phase.

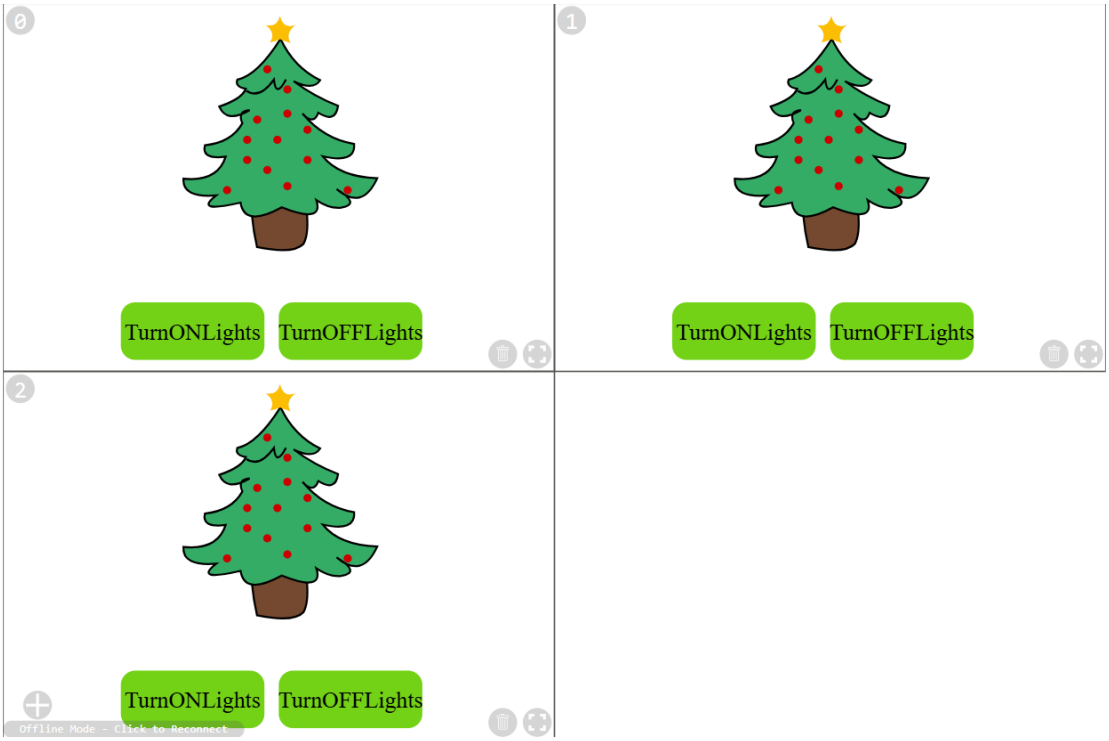


Figure 5.6: The state of the lights being ON for every client in the Lights Game. This game is used in Step 1 of the study's second phase.

Imagine we have a festival and we want to play the music for the whole town. The first facilitator who arrives at the concert hall can turn the speakers ON. Which option describes the scenario best in terms of a CSC?

a) The facilitator walking in town is the global portion and the music being played or not is the local portion.

b) The facilitator walking in town is the local portion and the music being played or not is the global portion.

c) The facilitator walking in town and the musing being played or not are both local.

d) The facilitator walking in town and the musing being played or not are both global.

Step 2. Play and Design: To allow participants to better *understand* the CSC concepts introduced in the previous step, they will be asked to design the CSC of the multi-player version of the *Hiking* game (used in Step 2 of the first phase) after playing it (Figure 5.9).

The *MP-Hiking* game's difference from the single-player one is the addition of the condition where the *Terminal* gates are closed until three players finish the hike and arrive at the *Terminal*. Therefore, the local statechart has the same states and transitions as the single-player version (Figure 5.10), and the global statechart has two new states: *Waiting* (for three players to arrive) and *GatesOpened*, (Figure 5.11). Also, a synchronizing message will be sent to the global statechart once a player arrives at the *Terminal*.

This step focuses on improving their *understanding*, and the measurement of how well this learning outcome is achieved will be done in Step 3. In this regard, after they finish their task of drawing the state diagram, the correct solution will be released and any misconceptions will be discussed with mentors.

Step 3. Play and Choose: In this step, participants will first play a game (called *Party*), and then, they will be asked to choose the exact mapping CSC of that game out of four different CSCs. This challenge measures how well participants *understood* CSC concepts, namely Local/Global models, synchronizing messages, conditionals, and interactive design. Next, the correct answer (shown in Figure 5.12) will be released and any misconceptions or questions will be discussed with mentors.

The *Party* game is a multiplayer game that requires players to manage their money to go to a party. In particular, the party will start only if at least three players successfully arrive at the *PartyRoom*. If a player fails to manage their money, then the party will be cancelled.

Figures 5.13 and 5.14 are provided for better visibility of the details. The detailed story of the game and its graphical figures can be found in Appendix A.2.1.

Step 4. Read, Design, and Code: Prior to the evaluation of how well participants can *apply* the covered CSC materials in new situations, they will have the opportunity to implement a multi-player game themselves. Therefore, they will be provided with a scenario of a game called *MP-Rope*. This scenario is the multi-player version of the game used in Step 4 of the

first phase. Then, they will be asked to interpret the English context with respect to CSC concepts. The scenario can be found in Appendix A.2.2. This task includes designing the CSC of the scenario they read, comparing their design with the revealed solution, discussing any potential questions with mentors, code generation, and compiling the code on the web-based IDE (STaBL.Rocks). Figure 5.15 illustrates the overall mode of the solution CSC, while Figures 5.16, and 5.17 are provided to make the details more visible.

Step 5. Evaluation: As the final step of the experiment, we evaluate how well participants can use their learned information and *apply* them in solving new challenges. In this regard, they will be asked to do a quiz including four multiple-choice questions. The quiz questions and answers can be found in Appendix A.2.3. In particular, the first two questions evaluate their ability to design the statecharts, as well as distinguish the local and global portions of the app, after interpreting a scenario into a CSC. The third question evaluates their ability to design the communication of statecharts through channels. The last question measures their ability to apply changes in the low-level implementation code.

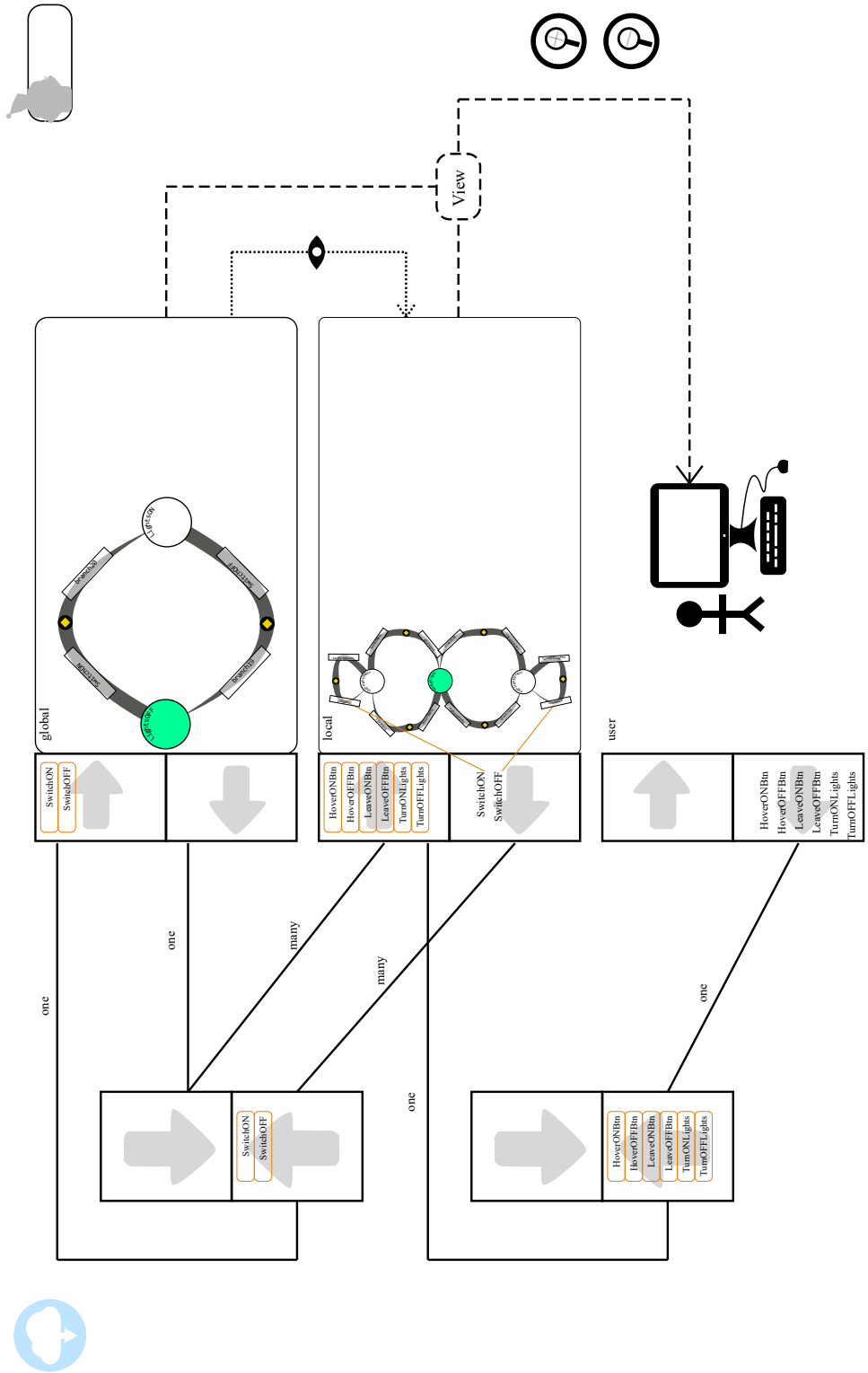


Figure 5.7: The Overall Mode of the CSCDraw for the Lights Game used in Step 1 of the second phase.

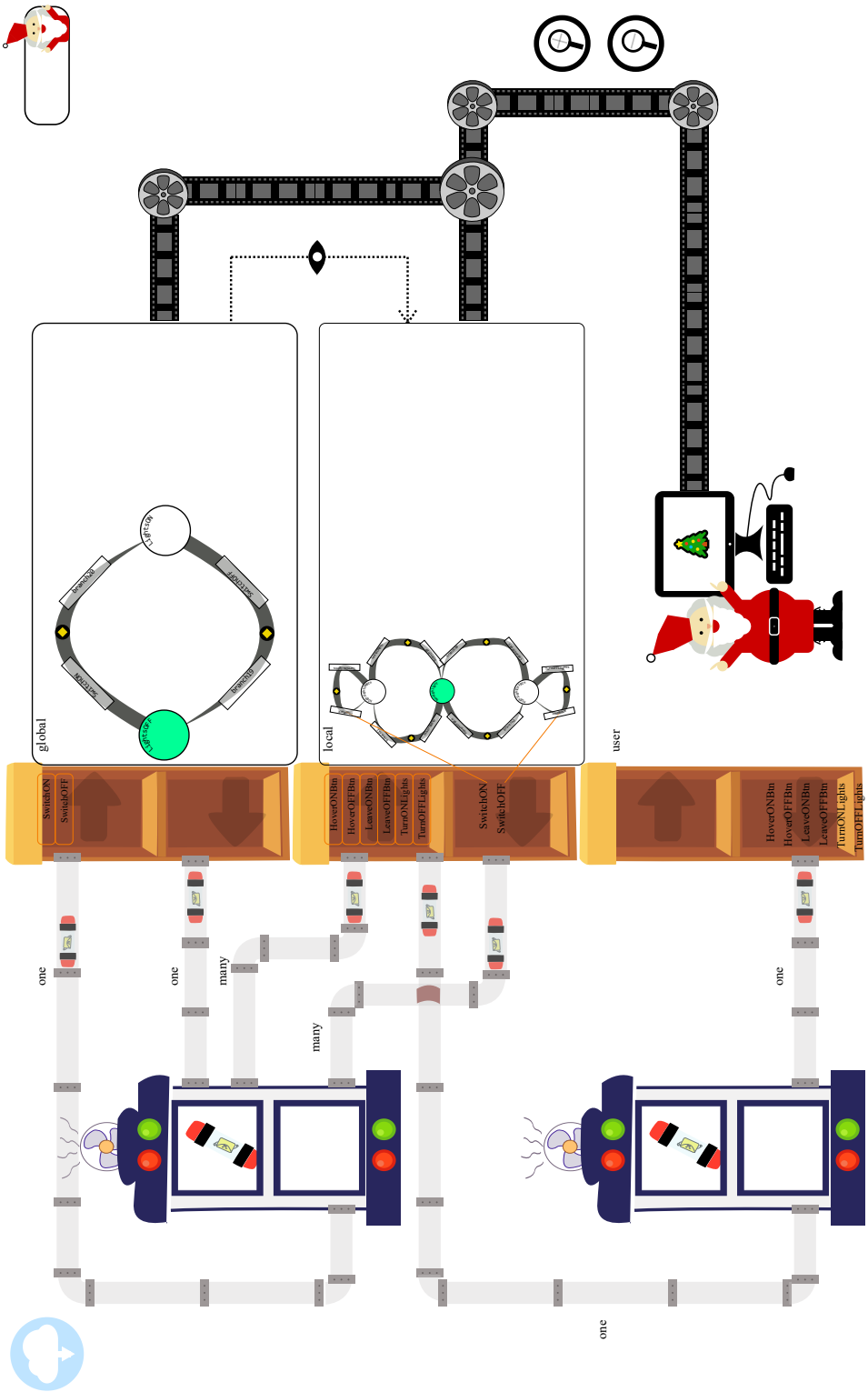


Figure 5.8: The Analogy Mode of the CSCDraw for the Lights Game used in Step 1 of the second phase.

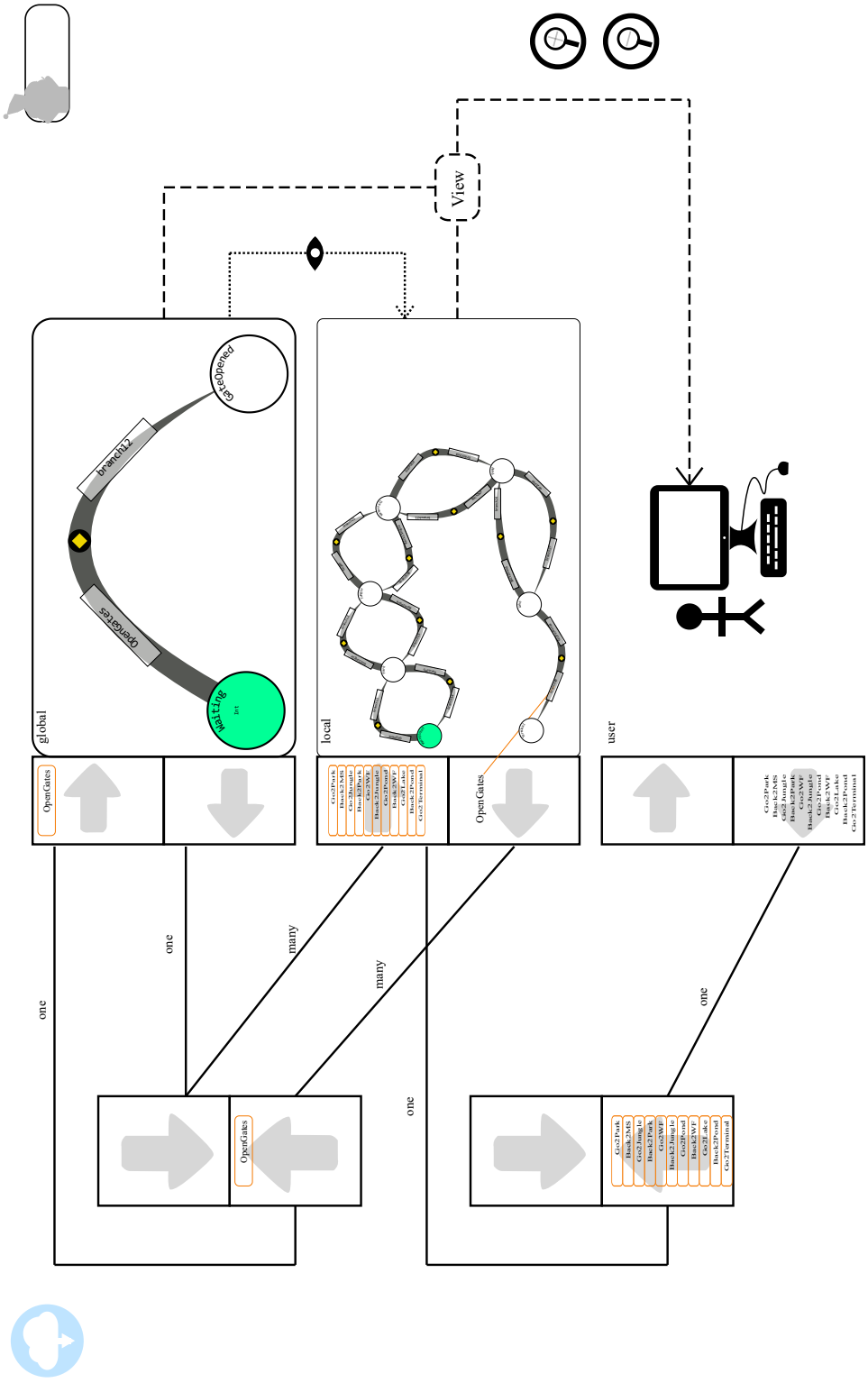


Figure 5.9: The Overall mode of the CSC solution to the MP-Hiking game challenge from Step 2 of the second phase.

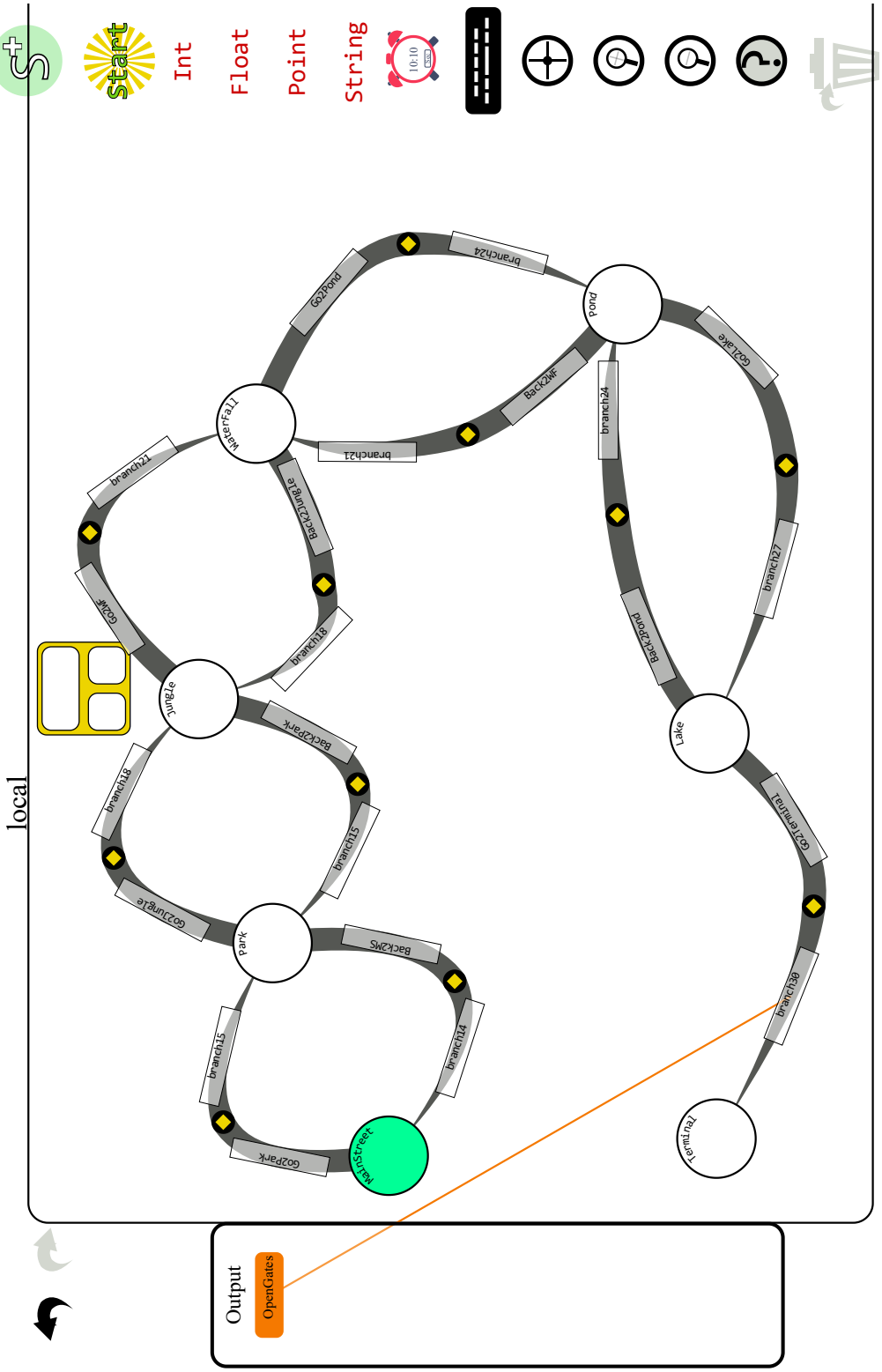


Figure 5.10: The local statechart of the CSC solution to the MP-Hiking game challenge done in Step 2 of the second phase. This figure is provided for better visibility of the details shown in the Overall mode.

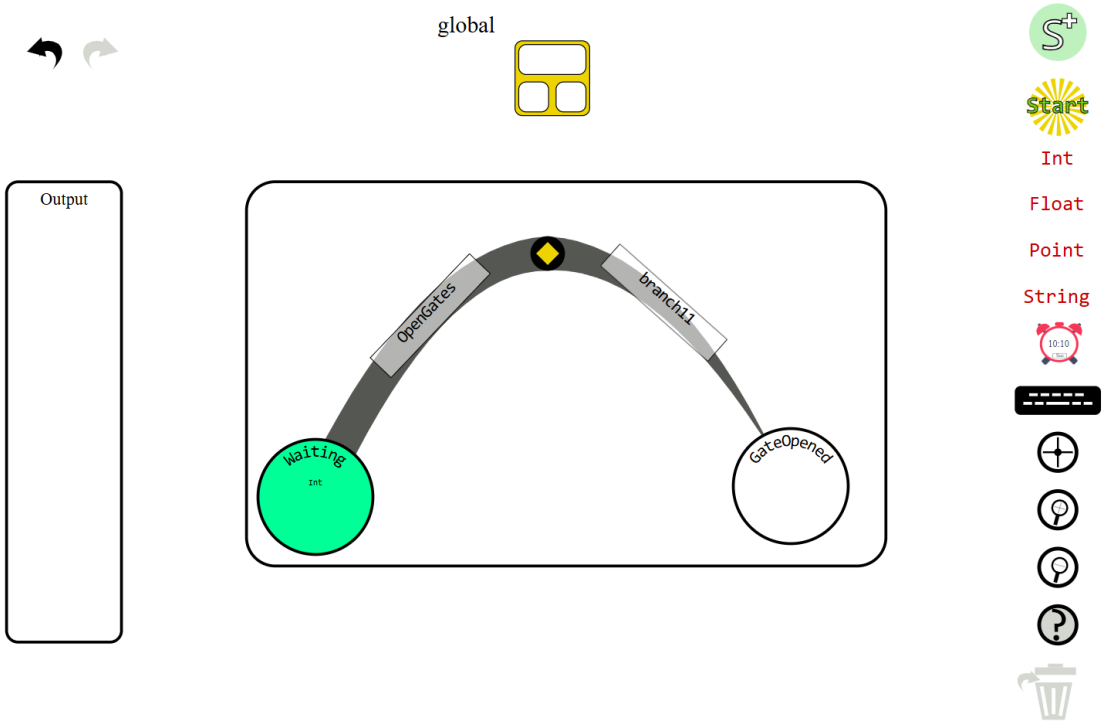


Figure 5.11: The global statechart of the CSC solution to the MP-Hiking game challenge from Step 2 of the second phase. This figure is provided for better visibility of the details shown in the Overall mode.

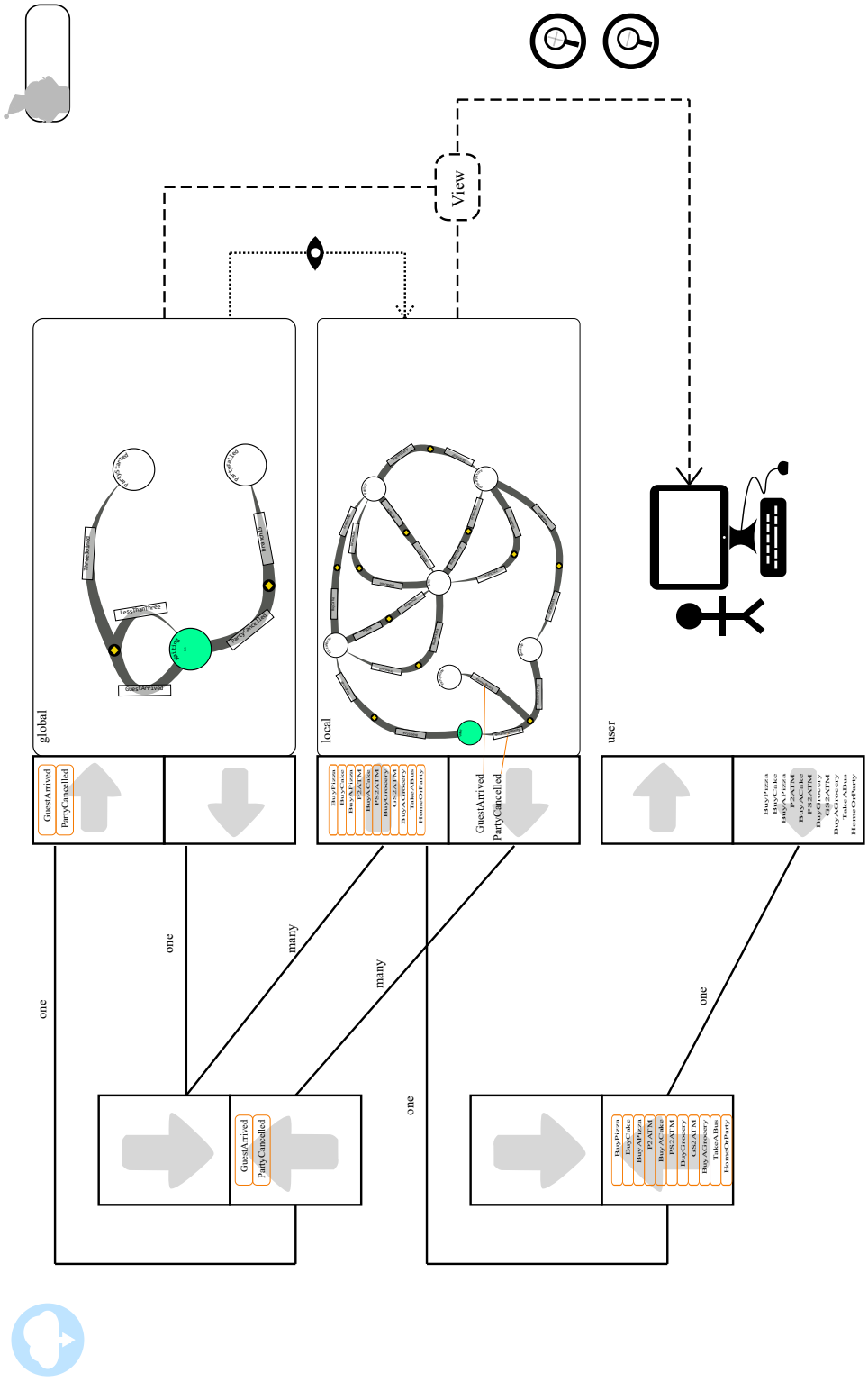


Figure 5.12: The Overall Mode of the solution CSC to the challenge done in Step 3 of the second phase (Party Game).

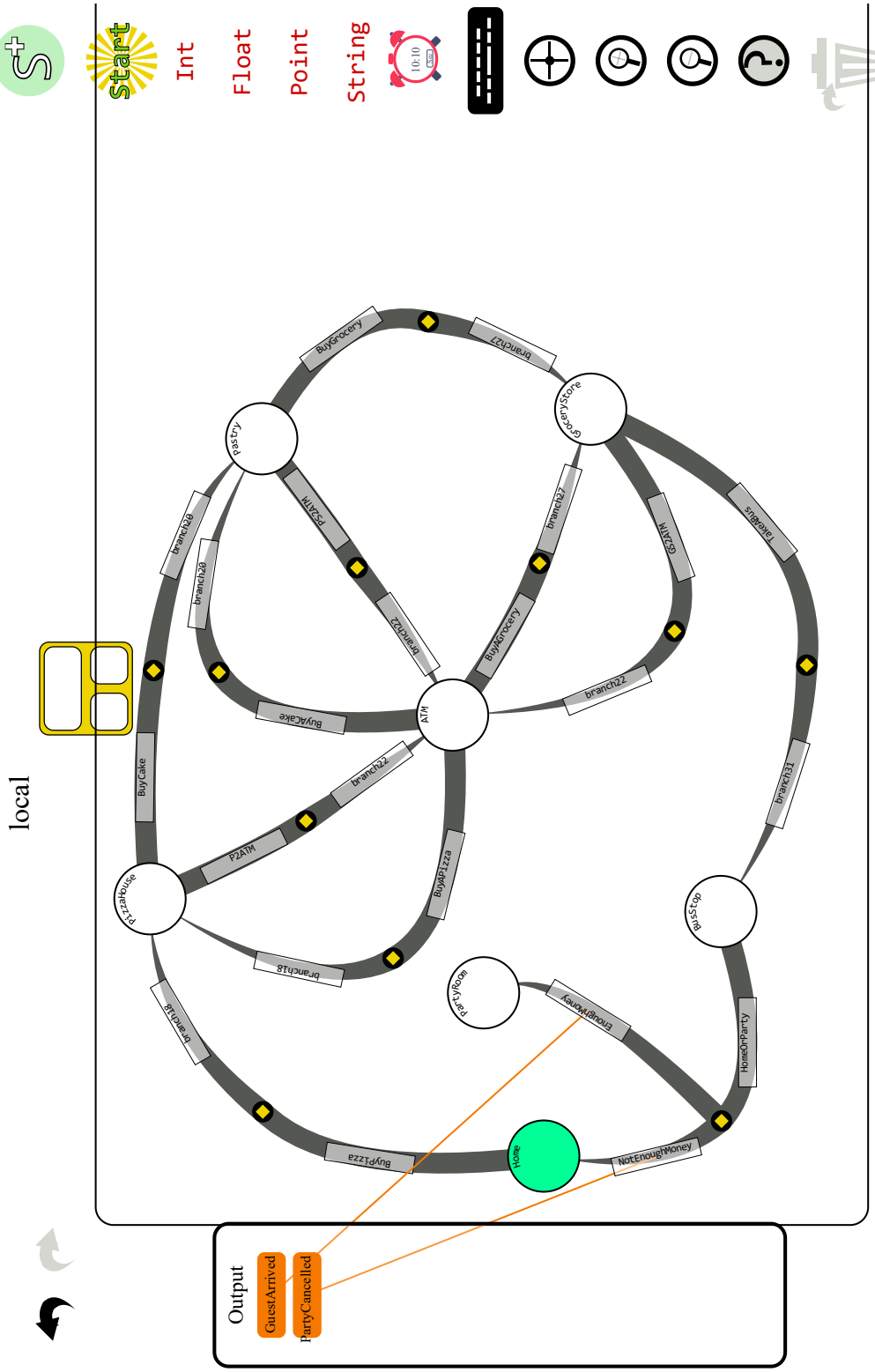


Figure 5.13: The local statechart of the solution CSC to the challenge done in Step 3 of the second phase (Party Game). This figure is provided for better visibility of the details shown in the Overall mode.

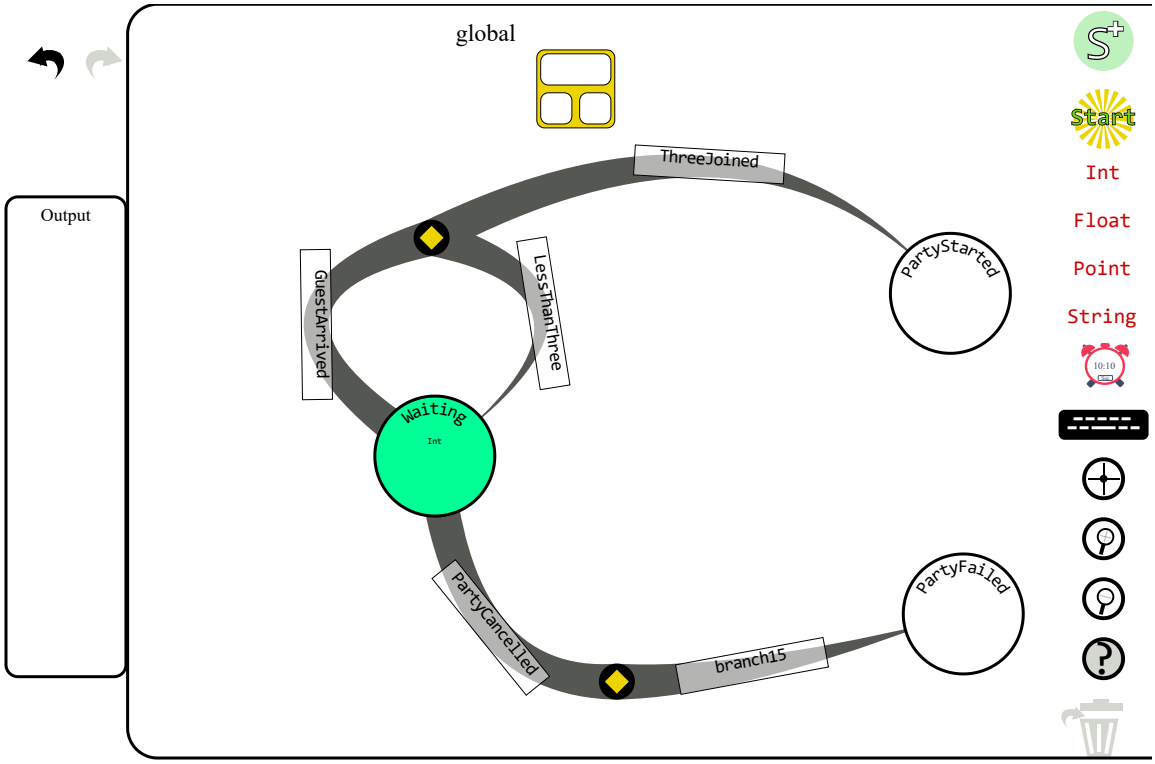


Figure 5.14: The global statechart of the solution CSC to the challenge done in Step 3 of the second phase (Party Game). This figure is provided for better visibility of the details shown in the Overall mode.

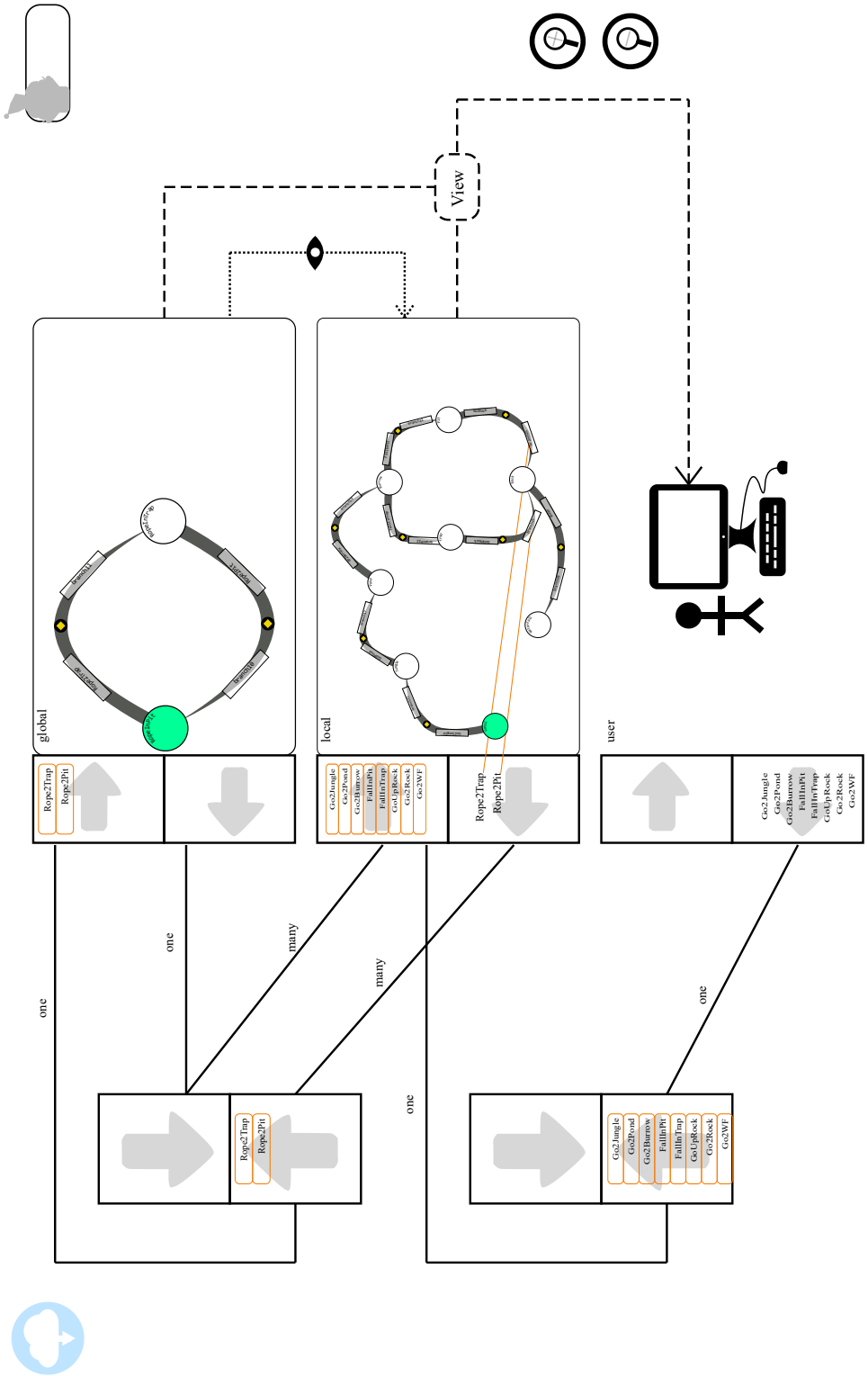


Figure 5.15: The Overall Mode of the solution CSC to the challenge done in Step 4 of the second phase (MP-Rope Game).

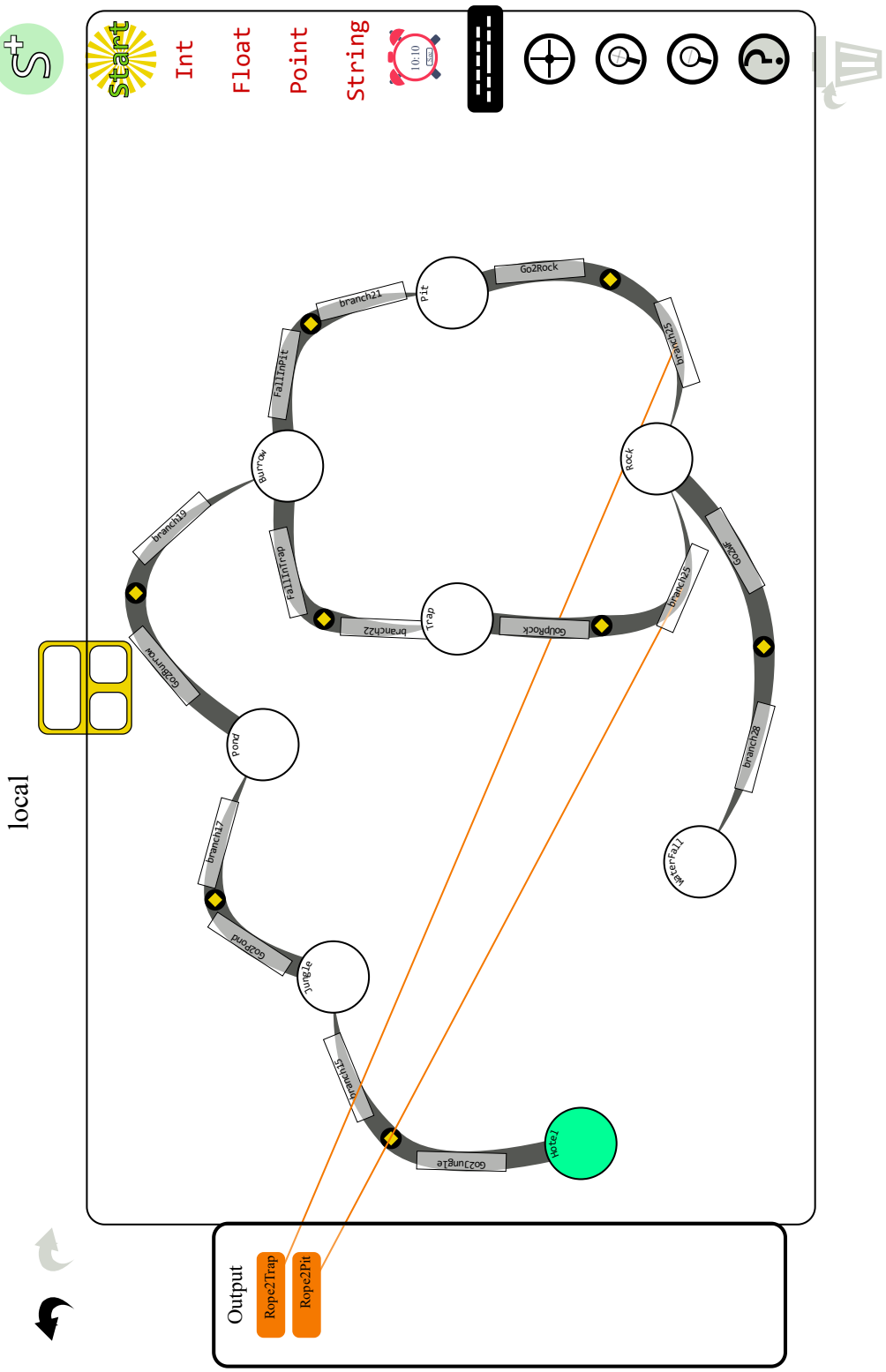


Figure 5.16: The local statechart of the solution CSC to the challenge done in Step 4 of the second phase (MP-Rope Game). This figure is provided for better visibility of the details shown in the Overall mode.

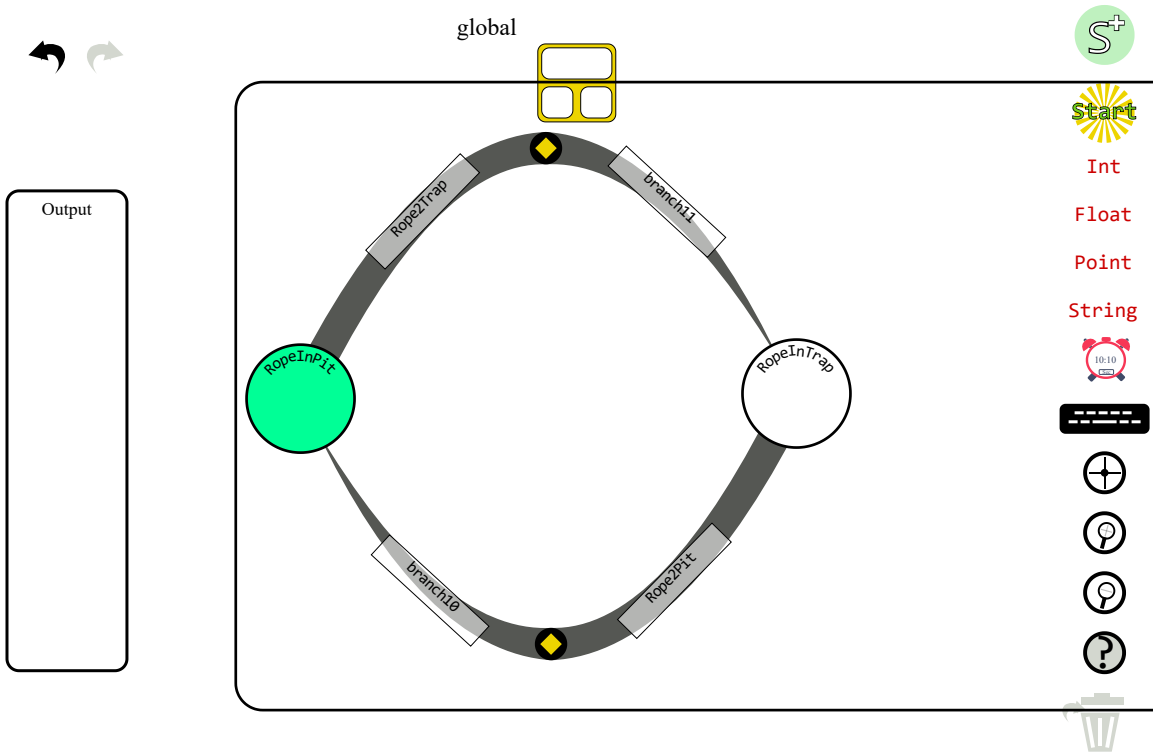


Figure 5.17: The global statechart of the solution CSC to the challenge done in Step 4 of the second phase (MP-Rope Game). This figure is provided for better visibility of the details shown in the Overall mode.

Chapter 6

Conclusion

This chapter provides the summary of the work presented in this thesis, discusses the answers to the research questions, suggests potential future directions, and explains potential threats to validity of this study.

6.1 Summary

The increasing importance of concurrency and the problems associated with learning and using this concept, requires rethinking the current practices of teaching it [46, 68]. This thesis aims to simplify concurrency even for beginners by making three main contributions:

1. Proposing a beginner-friendly concurrency paradigm called Communicating Statecharts (CSC).
2. Designing and developing CSCDraw, a visual Model-Driven Development (MDD) tool to support CSC.

3. Designing a pilot study to learn the most effective way of using CSCDraw to teach CSC to beginners.

CSC was designed based on five main principles chosen to reduce the barriers for beginners, including using software visualization and MDD, pure functions, Separation of Concerns (SoC), and raising the level of abstraction. To address these principles, we adopted features from existing concurrency models (i.e., process calculi and the actor model), resulting in CSC's atomic statecharts, communicating through channels.

CSCDraw supports CSC by addressing the requirements necessary for making this new paradigm accessible to beginners. In particular, in CSCDraw, beginners cannot violate CSC principles (unintentionally) by drawing conflicting configurations such as direct communication between statecharts. Other requirements include the tool being beginner-friendly (easy to learn and use for novices), supporting faithful code generation, conditional branches, and channel cardinality.

The development process in CSCDraw using the CSC paradigm proceeds from high level to low level; first, designing visual models/CSCs, then, defining the communication among statecharts through messages and channels, followed by generating the skeleton code from the designed models, and finally, completing the generated code. In this regard, beginners face concurrency at a high-level of abstraction, instead of dealing with it at the low-level while implementing code.

Validating the usability of this paradigm is out of the scope of this thesis. However, a pilot study is designed to identify the most effective way of teaching CSC

to novice programmers. This short study is preliminary to more rigorous experiments, and measures how well participants can achieve the hoped-for learning outcomes. These learning outcomes include the first three levels of Bloom's Taxonomy (Remember, Understand, and Apply), as illustrated in Figure 1.12.

6.2 Research Questions

In this section, we discuss the answers to the research questions presented in Section 1.2. The first two questions are answered, while answering the third question is out of the scope of this thesis and is suggested as a future step in Section 6.3.

6.2.1 RQ1. What does a beginner-friendly paradigm for distributed user-interface programs look like?

Section 1.3 of this thesis investigates five main principles that contribute to simplifying concurrency. Since we could not find any existing framework that addresses all of the identified principles, we propose a new concurrency paradigm, called Communicating Statecharts (CSC) (presented in Section 1.5). Particularly, CSC puts together the features from existing paradigms that met our principles as follows:

P1. *Visualizing* the system through adapting Harel's statecharts, the most commonly used modelling language in the case of interactive applications [34, 73]. We believe that visualization will contribute to simplification

by reducing the cognitive load, analogous to how it has helped students correct misconceptions about object-oriented programming [85].

P2. Using *Model-Driven Development (MDD)* to generate code from visual models. This code generation relieves beginners from repetitive coding and can protect them from many flaws caused by typos and incomplete understanding, preventing early frustration.

P3. Using *pure functions* and generating purely functional skeleton code based on the designed models. Distributed concurrency can be expressed using pure functions in the recently introduced architecture, Local-Global Model-View-Update (LG-MVU) [78]. LG-MVU separates the program into local (private to each user) and global (shared among every client connected to the server) portions, all without losing the transparency afforded by pure functions.

P4. *Separation of Concerns (SoC)* through isolation of models. In particular, CSC uses message passing via channels. This method of communication replaces direct transitions between statecharts. To achieve this isolation, CSC adapts channels from process calculi [14] and asynchronous messages with atomic models from the actor model [38].

P5. Raise *abstraction* to handle complexities by using Event-Driven Programming (EDP). Specifically, EDP offers easier concurrency by increasing abstraction through using events instead of threads, and allows programmers to make interactive applications such as games, which is a frequent strategy used to motivate beginners [50]. CSC puts its focus on the design of multi-player games, to further increase the engagement of EDP, and

exploit its success in making concurrency easier.

6.2.2 RQ2. How best to implement a design tool for the paradigm from RQ1 to make it accessible to beginners?

Section 1.4 explains how existing tools could not address our principles. Therefore, we gathered requirements for supporting CSC (described in Section 1.6). Section 1.7 presents CSCDraw, our visual MDD tool that is designed and developed to address those requirements.

Specifically, CSCDraw address the requirements by:

R1. Enforcing CSC principles by preventing novices from drawing models conflicting with those principles, such as direct transitions between different statecharts, which conflicts with Principle P3 (Pure Functions). Furthermore, to both visually and functionally enforce Principle P4 (SoC) CSCDraw provides two modes:

1. The *overall mode* illustrates the channels connected to a statecharts' interfaces, with the model contents including states, transitions, and synchronizing messages (as shown in Figure 1.10). This view allows message specification and code generation.
2. The *SCEditing mode*, that allows the modification of states and transitions within a statechart (as shown in Figure 1.11).

R2. The tool being **beginner-friendly** by providing a user-friendly UI, supporting modification guides, such as channel completion check, view function representation, and Analogy mode. (Measurement of this criteria is

out of the scope of this thesis.)

R3. Faithful **code generation** from the visual models (as proven in Chapter 3).

R4. Supporting **conditional branches** similar to recent UML state machine diagrams. These branches will translate into case expressions in the generated code.

R5. Visualizing **cardinality** of channels analogous to entity-relationship diagrams used in database design.

6.2.3 RQ3. How does the proposed paradigm from RQ1 compare to traditional paradigms for teaching beginners?

Answering this question requires a large-scale experiment that is planned to be conducted in the future. In this regard, two sub-questions can help us to prepare for this experiment:

RQ3.1: Chapter 5 presents the design of a pilot study that aims to answer RQ3.1. In other words, this study works as a prelude to a more rigorous experiment, suggested by RQ3.2, and targets identifying the best way of teaching CSC through CSCDraw to beginning programmers.

More specifically, the designed pilot study assumes participants have no prior coding knowledge in a text-based language including Elm. Therefore, to prevent beginners from facing overwhelming materials at once, the study is divided into two main phases. The first phase focuses on teaching

EDP-related concepts and state diagrams, while the second phase teaches distributed concurrency through CSC and CSCDraw.

This study helps participants to reach the third level of Bloom's Taxonomy (Apply), as illustrated in Figure 1.12. The evaluation of this study is based on measuring how well the expected learning outcomes are achieved.

RQ3.2: After learning the most effective way of teaching CSC by answering RQ3.1, a more rigorous experiment can be conducted to answer RQ3.2. Particularly, this bigger study can compare CSC's effectiveness in teaching beginners with one of the most commonly used concurrency frameworks, Java Threads.

6.3 Next Steps

In this section, we discuss the future research directions and areas of improvement.

6.3.1 Experiments on the Proposed Paradigm

The next step for this research is to measure CSC's effectiveness in simplifying concurrency for beginners which would be the answer to RQ3. This research question inspires the sequence of the next steps as follows:

1. Conducting a pilot study that answers RQ3.1, which investigates the most effective way of teaching CSC to novice programmers.

2. Conducting a more rigorous experiment that answers RQ3.2, which measures how effective CSC is in making concurrency easier to learn and use for beginners compared to one of the most commonly used concurrency frameworks, Java Threads.
3. Conducting a large-scale experiment that answers RQ3, which explores whether the proposed paradigm is more successful in removing barriers in concurrency contrasted to the previous works.
4. After validating the usability of the proposed paradigm, an interesting area of research would be measuring the engagement of novice programmers in implementing concurrent distributed applications using CSC and its tool.

6.3.2 Future Technical Steps

Once the efficacy of CSC and its tool is established through the experiments suggested above, more generalizations can be included in the paradigm. For example, different user types could appear as peers with different read-only access levels to different statecharts. For instance, in the case of designing an app for a virtual classroom, the messages a teacher can send to the statecharts would be different than the messages a student can generate. In this case, a database interface should be provided to store client/user information including their accessibility permissions.

Additionally, complex applications could be decomposed into more than two isolated statecharts, communicating through channels. The decomposition should still keep the paradigm easy to learn and use by considering CSC principles.

Furthermore, in future implementations, CSCDraw can be integrated into a text-based structure editor. The main benefit of this integration is preventing the graphical "documentation" from getting out-of-sync with the "implementation" code.

6.4 Threats to Validity

In this section, we first discuss the threats to the validity of the requirements gathered for the design of CSCDraw (discussed in Section 1.6), then, we go through the threats to the validity of the designed pilot study (presented in Chapter 5).

6.4.1 Threats to Validity of the Proposed Tool

CSCDraw builds its SCEditing mode's interface on the bones of our lab's MDD tool, SDDraw (introduced in Section 2.10). This implementation decision (I12. *SDDraw UI*), which is implied from Requirement R2 (the tool being beginner-friendly), is made based on the instructor's report of successfully teaching a first-year computer science course at McMaster University. The instructor mentioned that once the students found out about SDDraw, they would not stop using it to make the bones of their systems that were launched reaching over ten thousand lines of code. The validity of the decision of expanding the SDDraw UI may be threatened by:

External Validity: This observation is limited to the instructor's report of

teaching a first-year computer science course at McMaster University. Subsequently, the identified requirement might not apply to a broader student population.

Internal Validity: The students' eagerness to use SDDraw inspired us to expand the same UI. However, no data is collected to be analyzed and the validity of this success is limited to the instructor's observation. Additionally, this observation may have been biased since the instructor also leads the lab where SDDraw was created.

6.4.2 Threats to Validity of the Pilot Study

The designed pilot study involves two potential threats that may impact its validity.

External Validity: Since the participants are self-selected to the study, i.e. by registering themselves in a summer camp, there might be a bias on motivated students participating in the experiment. Subsequently, the results may not generalize to a broader beginner population. To mitigate this threat, the summer camps should be advertised to diverse schools (with different backgrounds), through online social media, and our outreach program that engages teachers who can motivate a wider range of students to participate.

Internal Validity: The gathered data might be affected by the challenge/quiz difficulty levels, instead of their actual learning outcomes. However, to mitigate this threat, we ensured that the designed evaluations were consistent

in complexity.

Appendix A

Pilot Study Instruments

This Appendix contains the scenarios and figures of the games and the evaluation questions used in the first (Appendix A.1) and second (Appendix A.1) phases of the pilot study, described in Chapter 5.

A.1 Pilot Study's First Phase Scenarios and Figures

A.1.1 First Step Figures

Figures A.1, and A.2 illustrate the game used in Step 1 of the first phase.

A.1.2 Third Step Scenario and Figures

The story of the Santa Game used in Step 3 of the first phase is:

Figure A.1: The Happy state of the Mood game, a mini-game to introduce SDDraw, used in Step 1 of the pilot study's first phase.

Figure A.2: The Sad state of the Mood game, a mini-game to introduce SDDraw, used in Step 1 of the pilot study's first phase.

The Christmas season has arrived and Santa has to deliver gifts to six children. Santa starts his journey from his house (Figure A.3). He first goes to the stable to get on his sleigh with the toys and flies to the city center with his six elves (Figure A.4). Once he arrives at the city center (Figure A.5), he turns on the Christmas tree's lights to be able to see the neighbourhoods (Figure A.6). Then, he flies to deliver the toys. Each time he delivers a toy, one of his elves can join the party. After delivering a toy, Santa can either go to the house's porch, to refresh his energy levels, or to the mall to take pictures with families during the day. Santa has to also take care of his health. He will lose 20 or 50 hunger points when he delivers a gift or meets people (Figure A.7). To get his full hunger points back he can go to the porch of the house after delivering a toy and eat the cookies and milk that the children left for him (Figure A.8). After refreshing, he can either go to the mall, or keep delivering gifts (Figure A.9). After meeting people in the mall, if Santa has some remaining toys, he will keep delivering them (Figure A.10). Otherwise, if he has no toys remaining and also his hunger points are not negative (Figure A.12), he can go to the party cave to take his elves back home (Figure A.13). Otherwise, if he is hungry, he can't visit the party cave and he has to leave the city without his elves (Figure A.11). Finally, he parks his reindeer in the stable and goes back to meet Mrs. Claus at his house (Figure A.14).

Figure A.3: The SantaHouse state of the Santa Game used in Step 3 of the pilot study's first phase.

Figure A.4: The Stable state of the Santa Game used in Step 3 of the pilot study's first phase.

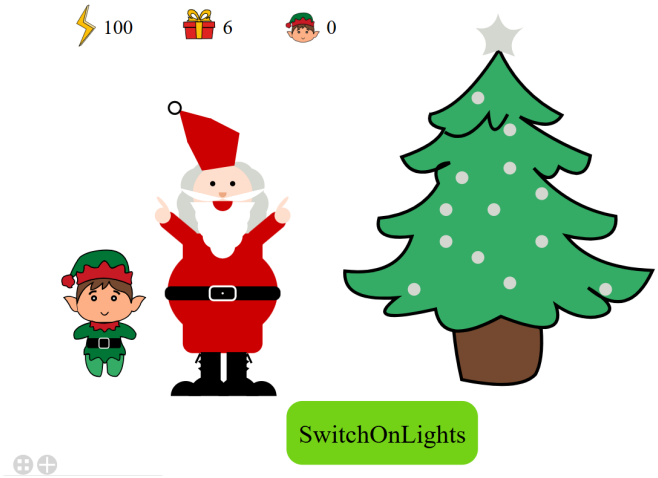


Figure A.5: The LightsOff state of the CityCenter of the Santa Game used in Step 3 of the pilot study's first phase.

Figure A.6: The LightsOn state of the CityCenter of the Santa Game used in Step 3 of the pilot study's first phase.

Figure A.7: The Neighbourhood state of the Santa Game used in Step 3 of the pilot study's first phase.

Figure A.8: The Porch state of the Santa Game used in Step 3 of the pilot study's first phase.

Figure A.9: Santa has full energy in the Porch state of the Santa Game, used in Step 3 of the pilot study's first phase.

Figure A.10: Santa has to deliver gifts after taking pictures with people in the Mall state of the Santa Game, used in Step 3 of the pilot study's first phase.

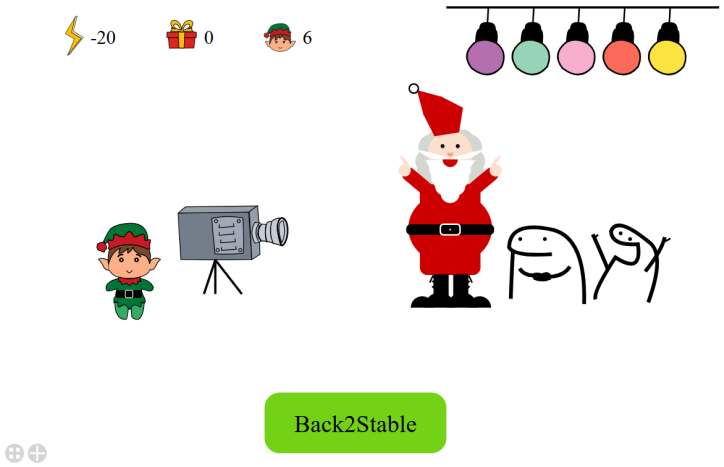


Figure A.11: Santa can't take his elves back due to low hunger points, so he has to leave without his elves, after taking pictures with people. This is the Mall state of the Santa Game, used in Step 3 of the pilot study's first phase.

Figure A.12: Santa has enough energy to take his elves from the party cave and leave the city after taking pictures with people. This is the Mall state of the Santa Game, used in Step 3 of the pilot study's first phase.

Figure A.13: PartyCave of the Santa Game used in Step 3 of the pilot study's first phase.

Figure A.14: Santa has successfully returned home with his elves. This is the SantaHouse state of the Santa Game, used in Step 3 of the pilot study's first phase.

A.1.3 Fourth Step Scenario

The scenario used in Step 4 of the first phase is as follows:

> We have a friend who has recently come back from an adventurous trip. She describes an incident that she managed to tackle as follows:
>
> From my hotel room, I had a breathtaking view of a lush jungle. Inspired by its beauty, I decided to dedicate one of my days to exploring it. One day, I was walking in the jungle, and taking pictures of the beautiful landscape there. I passed through a pond covered with eye-catching lilies. A rabbit jumped rapidly next to a tree that I was taking a snap of. When I moved closer, I saw a burrow where that rabbit jumped into. Just then, I heard a distant sound—something like rushing water. Intrigued, I moved forward, scanning my surroundings carefully. Before long, a breathtaking waterfall came into view. Excited, I hurried toward it, completely unaware of my surroundings. Suddenly, a hidden pit appeared in my path, and I fell into that. Luckily, there was a rope left in that hole, probably from people who had encountered this situation before. Using the rope, I climbed up and pulled myself onto a nearby rock. Finally, I accomplished taking a picture of that amazing scenery gifted by nature, the waterfall. As the sun began to set, I retraced my steps back through the jungle. This time, I was careful to avoid the pit, and I made it safely back to the hotel.

A.2 Pilot Study's Second Phase Scenarios and Figures

A.2.1 Third Step Scenario and Figures

The story of the Party Game used in Step 3 of the second phase is as follows:

> Each player starts from their home and initially has a hundred dollars (Figure A.15). Before going to the party, each player can buy pizza (Figure A.16), cake (Figure A.17), or groceries (Figure A.18), which will cost fifty, thirty-five, and twenty dollars, respectively. They can also access the ATM, Figure A.19, from wherever they are, before going to the BusStop. Each time they visit the ATM, they withdraw forty dollars. Going to the party requires taking a bus which will cost them fifteen dollars, so they won't be able to catch the bus unless they have that money when they arrive at BusStop (Figure A.20). If they don't have the required money they will have to walk back home (Figure A.21), and the party will be cancelled (Figure A.22). Once a player arrives at the party room, they will wait until two other people join them (Figure A.23). The party will start when the third player arrives at the party room (Figure A.24).

A.2.2 Fourth Step Scenario

The scenario used in the Step 4 of the second phase is as follows:

Figure A.15: The initial local state of the Party Game: Home state, used in Step 3 of the pilot study's second phase.

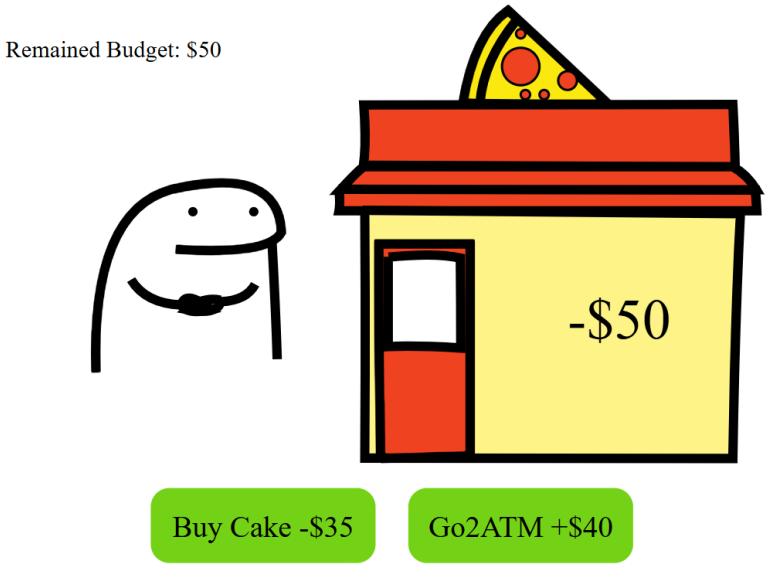


Figure A.16: The PizzaHouse state of the Party Game, used in Step 3 of the pilot study's second phase. Players will lose fifty dollars by buying a pizza.

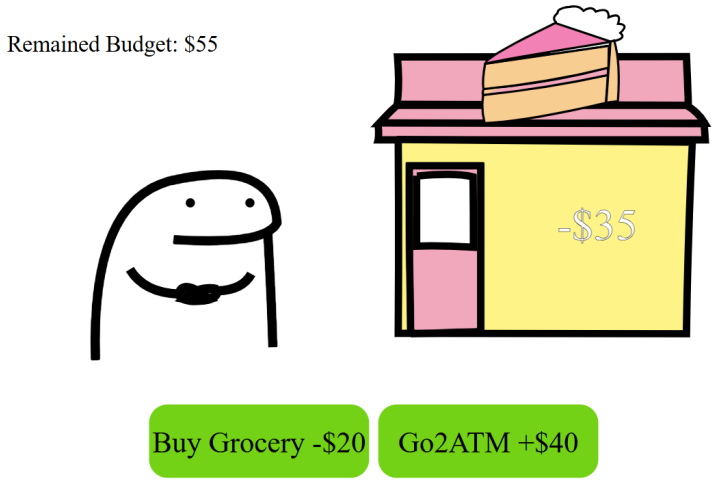


Figure A.17: The Cake state of the Party Game, used in Step 3 of the pilot study's second phase. Players will lose thirty-five dollars by buying a cake.

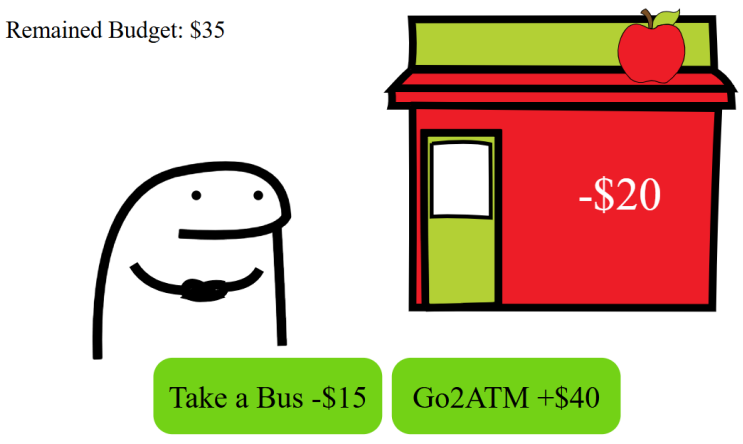


Figure A.18: The GroceryStore state of the Party Game, used in Step 3 of the pilot study's second phase. Players will lose twenty dollars by buying groceries.

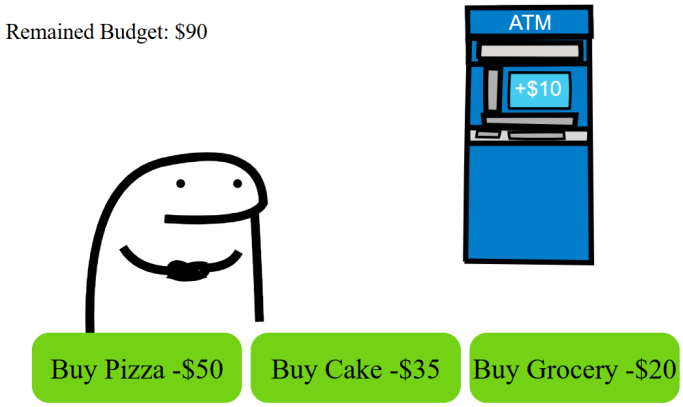


Figure A.19: The ATM state of the Party Game, used in Step 3 of the pilot study's second phase. Players can take forty dollars each time they visit the ATM.

Figure A.20: The BusStop state of the Party Game, used in Step 3 of the pilot study's second phase. Players will lose fifteen dollars to catch a bus.

Figure A.21: The BusStop state of the Party Game, used in Step 3 of the pilot study's second phase. Players will have to walk back home due to the low money. This case will lead to the party getting cancelled.

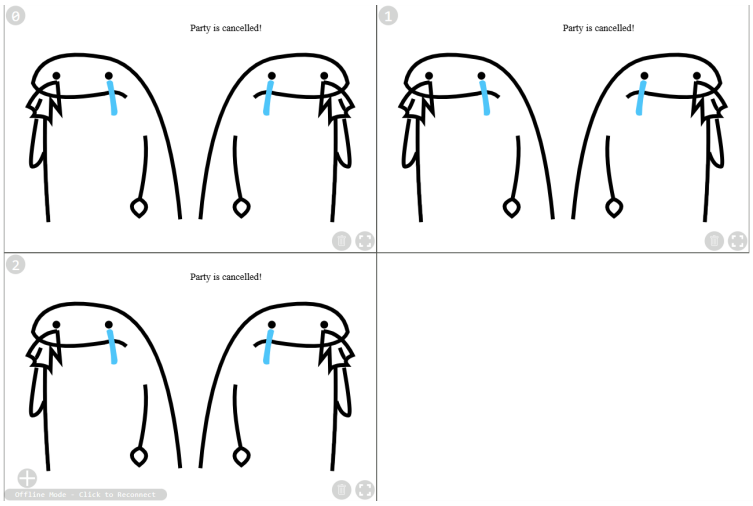


Figure A.22: The PartyCancelled state of the Party Game, used in Step 3 of the pilot study's second phase.

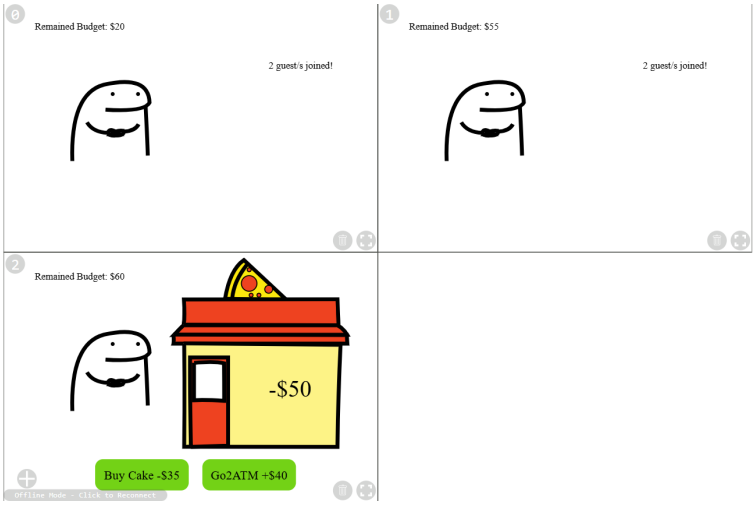


Figure A.23: The Waiting state of the Party Game, used in Step 3 of the pilot study's second phase. Players who made it to the PartyRoom have to wait until the number of guests reaches three.

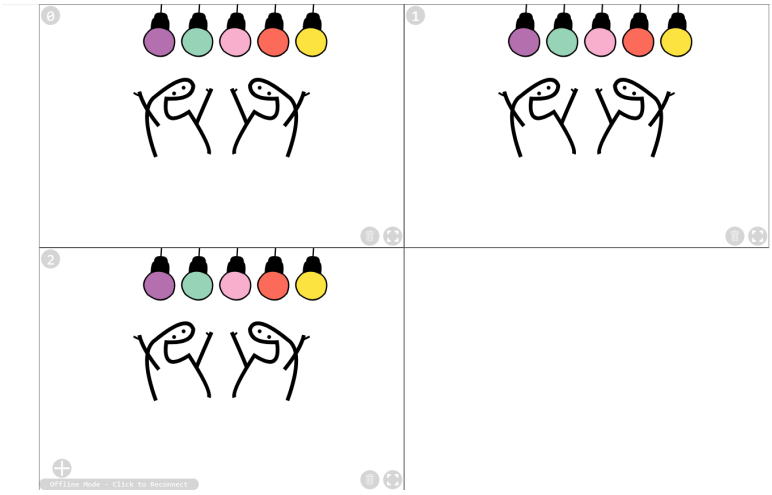


Figure A.24: The PartyStarted state of the Party Game, used in Step 3 of the pilot study's second phase. Reaching this state means at least three people have managed their money to make it to the party.

Remember our adventurous friend from the Rope game in the previous phase? A group of friends, inspired by her story, sought her advice before setting out to see the waterfall. She shared her guidance with them:

When you leave the hotel you will see the jungle right in front of you. As you enter, the first thing that will catch your attention is a beautiful pond. If you look carefully, you will spot a rabbit burrow beneath an old oak tree. That is the only oak tree there, so you will find it easily. By then, you will probably hear the sound of the waterfall. Be careful about the pit in your way, and then you can find the waterfall.

The group of friends started their journey and everything happened as planned, until they arrived at the burrow. They heard the sound of the waterfall, and they started approaching, carefully. They were walking gingerly, and finally, they saw the pit. They changed their path to avoid falling down, but there was a trap dug by hunters on the other side. Some of them fell into the trap and the others fell into the pit while they were panicking. Fortunately, the rope was still inside the pit, so they could climb up and stand on a big rock that was in between the trap and the pit. Then, they threw the rope into the pit to allow the others to get out of that trap. Thereafter, they moved forward to reach the waterfall.

A.2.3 Evaluation Quiz

The quiz used in Step 5 of the second phase consists of four multiple-choice questions.

The questions and their answers (shown in bold font) are as follows:

Read the scenario of a multi-player adventure game called *Open the Cave* and answer the questions.

> A cave's entrance is frozen because of the polar vortex, and our adventure pieces of equipment are trapped. We had to leave them behind, when we received the weather alert. Now, we need a team to gather wooden sticks to build a fire and melt the ice. At least 150 sticks are required to start the fire. The sticks can be collected from the jungle. The team must start from our station, cross the lake, river, and castle, and then reach the jungle. Each team member can carry up to 5 pieces of stick each time they visit the jungle. After collecting sticks, they must pass the old train station and the mountain, before reaching the cave. Once 150 sticks are gathered, the fire will start and melt the ice. Once the ice melts, we can retrieve our equipment pieces!

Q1. Consider the CSC of the *Open the Cave* Game. What are the global state(s)?

a) ***Cave Blocked* and *Cave Opened***

b) *Cave* and *Jungle*

c) Just *Jungle*

d) None of the above

Q2. Which transition can trigger a synchronizing message from the local statechart to the global statechart?

a) Collect Sticks

b) **Drop Sticks (in front of the cave)**

c) Go to Jungle

d) None of the above

Q3. Choose the correct sentence.

a) Starting the fire is a local transition.

b) Going to the Jungle is a global transition.

c) The number of collected sticks is local.

d) **The number of collected sticks is global.**

Q4. Which code snippet refers to a correct conditional state in the global statechart?

a)
```
case globalModel.state of
    CaveOpened  ->
          case globalModel.sticks >= 150 of
              False ->
                    ({ globalModel | state = ↩
                       CaveOpened }, Cmd.none, ↩
                       Cmd.none )

              True ->
                    ({ globalModel | state = ↩
                       CaveBlocked }, Cmd.none,↩
                        Cmd.none )
```

```
9
10      otherwise ->
11            ( globalModel , Cmd.none , Cmd.none )
```

b)

```
case globalModel.state of
    CaveBlocked  ->
          case localModel.sticks >= 150 of
              False ->
                    ({ globalModel | state = ↩
                       CaveBlocked }, Cmd.none ,↩
                        Cmd.none )

              True ->
                    ({ globalModel | state = ↩
                       CaveOpened }, Cmd.none , ↩
                       Cmd.none )

    otherwise ->
          ( globalModel , Cmd.none , Cmd.none )
```

c) (This is the answer.)

```
case globalModel.state of
    CaveBlocked  ->
          case globalModel.sticks >= 150 of
              False ->
```

```
5                    ({ globalModel | state = ↩
                        CaveBlocked }, Cmd.none ,↩
                         Cmd.none )
6
7               True ->
8                    ({ globalModel | state = ↩
                        CaveOpened }, Cmd.none , ↩
                        Cmd.none )
9
10    otherwise ->
11          ( globalModel , Cmd.none , Cmd.none )
```

d) None of the above.

Bibliography

[1] D. Abramov. Redux: a framework in JavaScript, 2015. URL `https://redux.js.org/`. [Online; accessed 26-July-2024].

[2] S. Aggarwal et al. Modern web-development using ReactJS. *International Journal of Recent Research Aspects*, 5(1):133–137, 2018.

[3] C. Alexander. The pattern of streets. *Journal of the American Institute of Planners*, 32(5):273–278, 1966. doi: 10.1080/01944366608978208.

[4] J. Armstrong. CSP and Erlang Concurrency Model, 2018. URL `https://elixirforum.com/t/does-earlier-erlang-concurrency-model-stem-from-csp/16905/5`. [Online; accessed 22-May-2024].

[5] P. Armstrong. Bloom's taxonomy. *Vanderbilt University Center for Teaching*, pages 1–3, 2010.

[6] J. C. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[7] P. Baran. On distributed communications networks. *IEEE Transactions on Communications Systems*, 12(1):1–9, 1964.

[8] H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *International Symposium on Formal Methods*, pages 57–72. Springer, 2011.

[9] J. A. Bergstra, A. Ponse, and S. A. Smolka. *Handbook of Process Algebra*. Elsevier, 2001.

[10] J. Bézivin and O. Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE, 2001.

[11] B. S. Bloom, M. D. Engelhart, E. J. Furst, W. H. Hill, and D. R. Krathwohl. *Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook 1: Cognitive Domain*. David McKay, New York, 1956.

[12] T. L. Booth. *Sequential machines and automata theory / [by] Taylor L. Booth.* New York : Wiley, 1967.

[13] G. Bracha. *The Dart programming language*. Addison-Wesley Professional, 2015.

[14] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *J. ACM*, 31(3):560–599, jun 1984. ISSN 0004-5411. doi: 10.1145/828.833. URL `https://doi.org/10.1145/828.833`.

[15] S. Callan. Concurrency in Elixir, 2021. URL `https://elixirschool.com/en/lessons/intermediate/concurrency`. [Online; accessed 22-May-2024].

[16] M. Carro, A. Herranz, and J. Marino. A model-driven approach to teaching concurrency. *ACM transactions on computing education*, 13(1):1–19, 2013. ISSN 1946-6226.

[17] M. C. Chen. Transformations of Parallel Programs in Crystal. In *IFIP Congress*, pages 455–462, 1986.

[18] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940. doi: 10.2307/2266170.

[19] D. Crocker. Safe object-oriented software: the verified design-by-contract paradigm. In *Practical Elements of Safety: Proceedings of the Twelfth Safety-critical Systems Symposium, Birmingham, UK, 17–19 February 2004*, pages 19–41. Springer, 2004.

[20] E. Czaplicki. Elm: Concurrent FRP for functional GUIs. *Senior thesis, Harvard University*, 30, 2012.

[21] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazieres, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189, 2002.

[22] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communication of the ACM*, 18(8):453–457, aug 1975. ISSN 0001-0782. doi: 10.1145/360933.360975. URL `https://doi.org/10.1145/360933.360975`.

[23] E. W. Dijkstra. *On the Role of Scientific Thought*, pages 60–66. Springer New York, New York, NY, 1982. ISBN 978-1-4612-5695-3. doi:

10.1007/978-1-4612-5695-3_12. URL https://doi.org/10.1007/978-1-4612-5695-3_12.

[24] N. Dilley and J. Lange. An Empirical Study of Messaging Passing Concurrency in Go Projects. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 377–387, 2019. doi: 10.1109/SANER.2019.8668036.

[25] S. E. V. and P. Samuel. Automatic Code Generation From UML State Chart Diagrams. *IEEE Access*, 7:8591–8608, 2019. doi: 10.1109/ACCESS.2018.2890791.

[26] U. Engberg and M. Nielsen. A Calculus of Communicating Systems with Label Passing. *DAIMI Report Series*, 15, 07 2000. doi: 10.7146/dpb.v15i208.7559.

[27] J. Eremondi. *Set Constraints, Pattern Match Analysis, and SMT*, pages 121–141. 05 2020. ISBN 978-3-030-47146-0. doi: 10.1007/978-3-030-47147-7_6.

[28] D. R. Ferreira, A. Mendes, and J. F. Ferreira. How are contracts used in android mobile applications? In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 400–401, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705021. doi: 10.1145/3639478.3643536. URL https://doi.org/10.1145/3639478.3643536.

[29] Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In

Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96, page 372–385, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917693. doi: 10.1145/237721.237805. URL https://doi.org/10.1145/237721.237805.

[30] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml: A language for concurrent distributed and mobile programming. In *International School on Advanced Functional Programming*, pages 129–158. Springer, 2002.

[31] S. Fowler. Model-view-update-communicate: Session types meet the Elm architecture. *arXiv preprint arXiv:1910.11108*, 2019.

[32] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM systems journal*, 45(3):451–461, 2006.

[33] S. Hansen and T. Fossum. Events not equal to GUIs. *ACM SIGCSE Bulletin*, 36(1):378–381, 2004.

[34] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, jun 1987. ISSN 0167-6423. doi: 10.1016/0167-6423(87)90035-9. URL https://doi.org/10.1016/0167-6423(87)90035-9.

[35] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4): 293–333, Oct. 1996. ISSN 1049-331X. doi: 10.1145/235321.235322. URL https://doi.org/10.1145/235321.235322.

[36] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.

[37] C. Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[38] C. Hewitt, P. B. Bishop, I. Greif, B. C. Smith, T. Matson, and R. Steiger. Actor Induction and Meta-Evaluation. In P. C. Fischer and J. D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 153–168. ACM Press, 1973. doi: 10.1145/512927.512942. URL `https://doi.org/10.1145/512927.512942`.

[39] C. A. R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8):666–677, aug 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL `https://doi.org/10.1145/359576.359585`.

[40] C. A. R. Hoare et al. *Communicating sequential processes*, volume 178. Prentice-hall Englewood Cliffs, 1985.

[41] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[42] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[43] Y. Jin, R. Esser, and J. W. Janneck. A method for describing the syntax

and semantics of UML statecharts. *Software & Systems Modeling*, 3:150–163, 2004.

[44] A. C. Kay. The early history of smalltalk. *SIGPLAN Not.*, 28(3):69–95, Mar 1993. ISSN 0362-1340. doi: 10.1145/155360.155364. URL `https://doi.org/10.1145/155360.155364`.

[45] R. Kennaway and M. R. Sleep. Syntax and informal semantics of dyne, a parallel language. In *The Analysis of Concurrent Systems*, page 222–230, Berlin, Heidelberg, 1983. Springer-Verlag. ISBN 3540160477.

[46] L. Lamport. Teaching concurrency. *SIGACT news*, 40(1):58–62, 2009. ISSN 0163-5700.

[47] A. Leff and J. Rayfield. Web-application Development using the Model/View/Controller Design Pattern. In *Proceedings fifth IEEE International Enterprise Distributed Object Computing Conference*, volume 2001, pages 118 – 127, 02 2001. ISBN 0-7695-1345-X. doi: 10.1109/EDOC.2001.950428.

[48] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: learning by examples. *Computer Networks and ISDN systems*, 23(5):325–342, 1992.

[49] Louis Mandel, Luc Maranget. JoCaml Manual, 2014. URL `http://jocaml.inria.fr/doc/concurrent.html`. [Online; accessed 02-May-2024].

[50] A. Lukkarinen, L. Malmi, and L. Haaranen. Event-driven programming in programming education: a mapping review. *ACM Transactions on Computing Education (TOCE)*, 21(1):1–31, 2021.

[51] S. Marlow. Parallel and concurrent programming in haskell. In *Central European Functional Programming School*, pages 339–401. Springer, 2011.

[52] J. McCarthy. *History of LISP*, page 173–185. Association for Computing Machinery, New York, NY, USA, 1978. ISBN 0127450408. URL https://doi.org/10.1145/800025.1198360.

[53] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 programmer's manual*. MIT press, 1962.

[54] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. ISSN 0005-8580.

[55] S. Mellor, T. Clark, and T. Futagami. Model-driven development - guest editor's introduction. *Software, IEEE*, 20:14– 18, 10 2003. doi: 10.1109/MS.2003.1231145.

[56] S. J. Mellor, M. Balcer, and I. Jacoboson. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002. ISBN 0201748045.

[57] B. Meyer. Applying 'Design by Contract'. *Computer*, 25(10):40–51, 1992. doi: 10.1109/2.161279.

[58] B. Meyer. Systematic Concurrent Object-Oriented Programming. *Communication of the ACM*, 36(9):56–80, sep 1993. ISSN 0001-0782. doi: 10.1145/162685.162705. URL https://doi.org/10.1145/162685.162705.

[59] Microsoft Agent. Microsoft Visio Code Generation Query, 2018. URL https://answers.microsoft.com/en-us/msoffice/forum/all/how-

do-i-generate-code-from-visio-class-diagrams/39e26ef9-3b94-4687-a18e-e01417f30b53. [Online; accessed 28-Jan-2025].

[60] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *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.

[61] R. Milner. Lectures on a calculus for communicating systems. In *International Conference on Concurrency*, pages 197–220. Springer, 1984.

[62] R. Milner. The Polyadic π-Calculus: a Tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-642-58041-3.

[63] R. Milner. Elements of interaction: Turing award lecture. *Communication of the ACM*, 36(1):78–89, Jan 1993. ISSN 0001-0782. doi: 10.1145/151233.151240. URL https://doi.org/10.1145/151233.151240.

[64] R. Milner. *The definition of standard ML: revised*. MIT press, 1997.

[65] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, USA, 1999. ISBN 0521658691.

[66] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and computation*, 100(1):1–40, 1992.

[67] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. JOLIE: a Java Orchestration Language Interpreter Engine. *Electronic Notes in Theoretical Computer Science*, 181:19–33, 06 2007. doi: 10.1016/j.entcs.2007.01.051.

[68] S. Nanz, F. Torshizi, M. Pedroni, and B. Meyer. Empirical assessment of languages for teaching concurrency: Methodology and application. In *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEET)*, pages 477–481. IEEE, 2011. ISBN 9781457703492.

[69] K. Nygaard and O.-J. Dahl. *The development of the SIMULA languages*, page 439–480. Association for Computing Machinery, New York, NY, USA, 1978. ISBN 0127450408. URL https://doi.org/10.1145/800025.1198392.

[70] S. Oaks and H. Wong. *Java Threads*. O'Reilly Media, Inc., 2004. ISBN 0596007825.

[71] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communication of the ACM*, 15(12):1053–1058, Dec 1972. ISSN 0001-0782. doi: 10.1145/361598.361623. URL https://doi.org/10.1145/361598.361623.

[72] P. Pasupathi, C. W. Schankula, N. DiVincenzo, S. Coker, and C. K. Anand. Teaching Interaction using State Diagrams. *Electronic Proceedings in Theoretical Computer Science*, 363:132–152, July 2022. ISSN 2075-2180. doi: 10.4204/eptcs.363.8. URL http://dx.doi.org/10.4204/EPTCS.363.8.

[73] B. Pérez, A. Rubio, and M. Zapata. A systematic review of code generation proposals from state machine specifications. *Information and Software Technology*, 54:1045–1066, 10 2012. doi: 10.1016/j.infsof.2012.04.008.

[74] C. Reade. *Elements of functional programming*. Addison-Wesley Longman Publishing Co., Inc., USA, 1989. ISBN 0201129159.

[75] T. Reenskaug. MVC XEROX PARC 1978-79, 1979. URL `https://folk.universitetetioslo.no/trygver/themes/mvc/mvc-index.html`. [Online; accessed 26-July-2024].

[76] W. Reisig. *Petri nets: an introduction*, volume 4. Springer Science & Business Media, 2012.

[77] M. Richards. *Software architecture patterns*, volume 4. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA, 2015.

[78] C. Schankula, S. Smith, and C. Anand. A functional event-driven framework for simplified concurrent applications. In *CASCON 2024*, November 2024.

[79] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003. doi: 10.1109/MS.2003.1231150.

[80] C. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, 1964.

[81] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119, 1999.

[82] L. Thabane, J. Ma, R. Chu, J. Cheng, A. Ismaila, L. P. Rios, R. Robson, M. Thabane, L. Giangregorio, and C. H. Goldsmith. A tutorial on pilot

studies: the what, why and how. *BMC medical research methodology*, 10: 1–10, 2010.

[83] R. Virding, C. Wikström, M. Williams, and J. Armstrong. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., GBR, 1996. ISBN 013508301X.

[84] P. Wegner. Concepts and paradigms of object-oriented programming. *ACM Sigplan Oops Messenger*, 1(1):7–87, 1990.

[85] I. Çetin. Visualization: a tool for enhancing students' concept images of basic object-oriented concepts. *Computer Science Education*, 23:1 – 23, 2013. doi: 10.1080/08993408.2012.760903.