SpecTacles: Supporting Control Flow Comprehension of Software Developers in TLA+

Daniel Stachnik[†], Tom Beckmann[*], Patrick Rein[*] and Robert Hirschfeld[*]
Hasso Plattner Institute, University of Potsdam, Germany
Email: * firstname.lastname@hpi.uni-potsdam.de, † hi@dast.xyz

***Abstract*—Formal specifications like TLA+ support software developers in formulating and verifying properties of their systems as state machines. Writing specifications helps software developers incrementally develop an understanding of their systems' domains. However, according to our formative study, while *reading* such specifications authored by others, software developers tend to struggle to build comprehension. To overcome the limitations of text, participants visualized specifications and manually constructed concrete example sequences of actions.**

We propose an approach that visualizes state-machine-based specifications to better support software developers in reading, exploring, and understanding them. Our tool, *SpecTacles*, generates and presents interactive visualizations derived from a specification's state space next to its original source text. Users can formulate example sequences of actions, which are synchronized between all visualizations.

To explore how and why such interactive visualizations help, we conducted a user study (n=12) with software developers who had no prior training in formal specifications, and explored how they use our tool in a set of comprehension tasks. Participants used the source text to obtain local and definitive information, for example, to validate an assumption, but they referred to the visualizations to gain an overview and to compensate for the lack of control flow information in the source text. Given the strengths of both visualizations and source text, participants found *SpecTacles* useful for their comprehension and indicated that they would like to use it in other software engineering contexts.

***Index Terms*—formal specifications, interactive visualizations, TLA+, reading comprehension**

I. Introduction

In his seminal paper, Peter Naur argued that the process of writing programs should be regarded as a process of building theories in the minds of programmers [1]. Theory-building suggests that, while programming, the programmer constructs a theory about how the program should be used, why each part of the program exists, and how modifications can be implemented to best conform to the theory of the program author. What if the idea of theory building also applied to formal specifications?

Applying the concept of theory-building to formal specifications suggests that comprehending a specification involves possessing a correct theory of it. This theory would enable the holder to understand how and why parts of the specifications work, and how it could be effectively modified. Just as programmers build their theories primarily through programming, authors of formal specifications also appear to gain comprehension in particular through the process of writing, as it forces writers to be rigorous and precise when putting their theories to text [11, 21, 24]. Other than writing, reading formal specifications could also provide an abstract, precise, and correct definition of an implementation for professional software developers [16, 3], who typically have little to no training in formal methods.

However, an implication of Naur's theory-building idea is that reading a program alone may be insufficient to reconstruct its theory; relevant implicit knowledge in the original writer's mind needs to be reconstructed to fully comprehend a program's theory. Likewise, this would imply that reading a formal specification alone may be insufficient as a means to fully comprehend it. Indeed, Parnas, for example, notes that sometimes formal specifications "are 'write only' because no one but the author can answer questions" about them [16]. Studies on the readability of formal specifications suggest that they are challenging for untrained, as well as trained, users to comprehend [10, 4]. In our formative study, we observed that participants (n=8) struggled to answer comprehension tasks about a formal specification. To aid their understanding, they instead manually constructed visualizations and example sequences of actions, thereby attempting to build their own theories from the abstract specification alone.

In theory, formal specifications contain all the information to generate the visualizations and example sequences that our participants were seeking. In practice, visualizations of specifications struggle with state space explosion, where even a small specification can result in a huge state space. One specification language commonly used in the industry is TLA$^+$ [32, 21, 29]. In TLA$^+$, authors can define distributed systems as state machines in concise and abstract specifications, that define initial states and *actions*, effectively the state machine transitions [14, 7]. A model checker can explore all possible sequences of *actions* of a specification until the complete state space has been explored, provided a finite state space. Many state-based model checkers use underlying representations akin to state machines [22]. Model-checking distributed systems often results in a state explosion due to the interleaving or out-of-order arrival of asynchronous messages, making visualizations of the complete state space impractical.

Based on the insight that readers of formal specifications seek visualizations and example sequences of actions, we built *SpecTacles*, a tool that enables users to explore meaningful slices of the state space of TLA$^+$ specifications exported by the TLC model checker [6]. *SpecTacles* mitigates state explosion by allowing interactive formulation of example sequences and

1943-6106/24/$31.00 ©2024 IEEE
DOI 10.1109/VL/HCC60511.2024.00028

by scoping visualizations to individual actors of the distributed system. We arrange four views of the specification side-by-side: a sequence diagram, a state diagram, a state table, and the specification's source text. In an exploratory user study (n=12), we investigated how software developers navigate between the views to satisfy different information needs during a set of comprehension tasks. In accordance with prior work [9, 31], we believe that formal specifications should be broadly readable by software developers who received formal education in computer science but had no extensive training in formal specifications, if they are to be used in practice. Consequently, all participating software developers only received limited (up to 30 minutes) training in state-based formal methods and had no prior experience in formal methods.

They prefer source text for its complete and precise nature when validating assumptions derived from visualizations. Overall, participants appreciated *SpecTacles* and indicated that they would like to use it in various software engineering contexts.

Our contributions in this paper are:

- a formative study assessing struggles of software developers when reading formal specifications,
- an approach for generating interactive visualizations from a state space dump that allows effective exploration of large state spaces, and
- a user study that demonstrates how software developers use multiple, interactive views in our approach to form a better understanding of control flow.

Next, we describe our formative study (section II). We then describe our tool (section III) and the user study evaluating it (section IV). Finally, we discuss our results (section V), present related work (section VI), and conclude the paper.

II. Mismatch Between Text and Information Needs

Prior work suggests that readers of formal specifications prefer working with non-textual state machine views [23]. Worse, in a study on the comprehension of formal specifications, including both beginners and experts in formal methods, 77% of participants struggled to identify inconsistencies in a given textual specification [33]. Mathematical training does not appear to address comprehension issues sufficiently [4]. Anecdotal experience by educators [31] and a quantitative study [13] suggest that visualizations of simple formal specifications can improve comprehension. Still, specification languages commonly used in the industry, such as TLA$^+$, are mostly textual and provide only few capabilities for visualization. To better understand what information needs software developers have while reading formal specifications, we conducted a formative study. Next, we describe the setup and the insights we gained.

A. Setup

We recruited 8 software developers unfamiliar with formal specifications (aged 24-40, 1 postdoc, 7 CS master's students; 1 female and 7 male). We refer to the participants as F1-F8[1].

[1]All quotes are translated from German.

Participants first answered demographic questions and indicated their familiarity with distributed systems, the chosen example protocol, formal specifications in general, and TLA$^+$ in particular. They then read one paragraph outlining the concept of TLA$^+$ specifications and another paragraph outlining the goal and basic assumptions of the example protocol.

As our participants were new to formal specifications, we translated the TLA$^+$ specification into JavaScript-like pseudo code (see Listing 1 for an example) and walked participants through two example actions of the modeled state machine. We offered participants to clarify any parts of the specification's notation they were unsure how to interpret.

We asked participants to think aloud while working on the tasks and observed their actions. We recorded audio during the session, transcribed relevant parts, and grouped statements by topic, combined with observations we made.

```
RMRcvCommitMsg(rm) ==
  (* Resource manager $rm$ is told  *)
  (* by the TM to commit.           *)
  if msgs.has({type: "Commit"}):
      rmState[rm] = "committed"
```

Listing 1. An example action from the specification used in our formative study. We translated the TLA$^+$ syntax to JavaScript-like pseudo code.

B. Task

We chose the Two-Phase-Commit (2PC) protocol as a real-life example protocol because there exists a simplified and commented specification of it for TLA$^+$ [12]. 2PC is a consensus protocol designed to ensure that a group of distributed resource managers all either commit or abort a transaction. First, the transaction manager (TM) sends a commit request to all resource managers (RM). The RMs either answer that they are now prepared to commit or need to abort the transaction. If each RM has promised to commit, the TM sends a commit message to all RMs; or an abort message, otherwise. One happy-case sequence is shown in Figure 2.

The tasks for our study were inspired by exams of distributed systems lectures that are publicly accessible. We also designed our questions such that they would require a wider range of reading comprehension, as recommended in prior work [28, 34]. We chose three tasks shown in Table I, designed to test the participants' comprehension of the specification through their ability to correctly paraphrase the workings of the specified systems and to isolate relevant behavior.

In addition, we provided participants with the state diagram for 2PC which can be automatically generated by TLC, the standard model checker of TLA$^+$ [25]. The diagram displayed the state space for 2PC with three resource managers in a state diagram, resulting in 1370 states for our example specification.

C. Results

Participants struggled to answer the tasks from text alone and instead relied on their own drawings or example sequences.

The specification's text was considered challenging, despite its familiar syntax. For F2, "code is not suitable to convey the idea [of the specification]." It was also initially unclear to F5

Nr.	Task
1	Provide a sequence of actions that ends with all resource managers being in the "committed" state.
2	Why can one resource manager never reach state "committed" while another resource manager is at state "aborted"?
3	How could you simplify the state machine? Find and name actions which can be safely removed, i.e. which maintain the safety property mentioned in the introduction.

Table I
TASKS IN OUR FORMATIVE STUDY.

and F6 how the textual definitions relate to the invariants and states of the 2PC state machine.

Four participants attempted to work with the provided state diagram but ultimately concluded that the information it provided was too dense to be helpful. Still, participants expressed wishes for various visualizations: F1 asked for a graph, F2 and F7 began sketching a state diagram, F3 asked for a linearized view of the specification that makes the temporal dependencies of actions clear, and F8 asked for a visualization to track variable changes. We provide some of the drawings of the participants [35]. Both F2 and F5 also expressed the wish to understand the actors of the system in isolation. F5 concluded: "I feel like I have to keep this all in my mind. [...] I am constructing a state diagram in my head."

Participants formulated example sequences of actions to answer the given tasks. F5 and F8 wished for exemplary message exchanges to illustrate relevant examples in 2PC. Similarly, F7 wished for "sequences of messages leading to this state." F1, F6, and F8 noted sequences of actions would help answer the tasks.

D. Conclusions of The Formative Study

Our observations align with insights gained in prior work: although the text provides all information sought by participants, they actively asked for representations beyond the text. The participants probed for information not readily available from the text. Instead, they switched views of the specifications, for example by drawing state diagrams, formulating example sequences of actions, or manually navigating between actions and their preconditions. However, the state diagram of the global state does not scale well with the complexity of real-life specifications either, as it overwhelmed all participants. This insight leads to our central design challenge:

> What tool design would make information available to users that is inadequately conveyed by text, without overwhelming them?

III. APPROACH

To approach the central design challenge, we designed a tool to cover information needs not readily available from textual specifications. Formal specifications are unique in that model checkers can statically enumerate the specification's state space as long as the state space is finite [6]. Given this enumerated state space, we can reconstruct every example sequence of actions possible. Thus, we have all the information that software developers are seeking—we just need to find ways to meaningfully present the information to users, which is challenging due to state explosion.

To make the presented information meaningful, our tool, *SpecTacles*, shows *multiple*, *interactive* views isolated by *actors*. This is in contrast to the visualization we tested during the formative study, where the entire state space is contained in a single, static visualization. Figuratively speaking, each of our tool's views can be seen as a pair of spectacles, providing a uniquely sliced view of the state space. This way, the user can focus on the aspects of the specification and its state space they want, at the level of information density they prefer. The code for *SpecTacles* is open-source[2] and a demo video is available [35].

A. Interactive Views

One way to reduce the visible state space is to only show one concrete sequence of actions instead of all possible sequences. For instance, any particular sequence diagram requires a single, non-branching sequence of actions[3].

As participants were asking for examples for various scenarios, *SpecTacles* allows users to interactively formulate an example sequence of actions, which is used by all views to adapt the slice of the state space they show. Views offer the user buttons to trigger the actions of the specification's state machine that are currently valid, based on the state machine's state for the formulated example sequence.

For instance, in 2PC, when the transaction manager receives information from all resource managers that they are prepared, the next possible actions of the transaction manager are to commit or abort the transaction. Users can then trigger one of the actions to explore its effect or go back via an undo button. As all views share a single example sequence of actions, users can switch between views without manually recreating any context. To shorten the feedback loop further, we show a preview of the changes when an action button is hovered.

B. Views Isolated by Actors

A common approach to visually reducing state machines is by using statecharts [2]. Statecharts allow hierarchical and orthogonal organization of the state space. Authors can use orthogonality by defining multiple nested state machines, which can concurrently change their states. They can use hierarchy by using superstates, each of which encapsulates parts of the state space as a more abstract state, while tracking the lower-level states internally.

TLA$^+$ supports neither hierarchical nor orthogonal state space organization. Instead of hierarchical state space organization, TLA$^+$ uses refinement mappings to specify that a state machine behaves like another, usually more abstract one. Information on orthogonality is not directly derivable from TLA$^+$ specifications or their state spaces because TLA$^+$ uses global variable scopes to simplify writing of properties [8].

[2]https://github.com/hpi-swa-lab/SpecTacles

[3]In general, the state space may contain infinitely many such sequences of actions due to loops in the state machine.

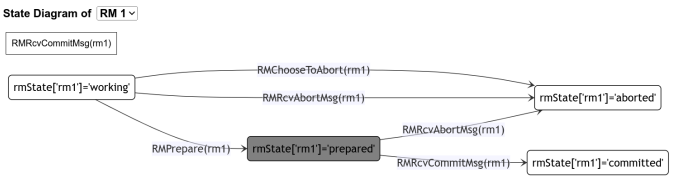

Figure 1. State diagram of the first resource manager. The active state according to the example sequence of actions is marked in gray. The user had triggered the $RMPrepare$ action before reaching this state. From this state, this resource manager can only receive two messages, signaled by the two outgoing transitions. The state diagram shows *all* possible states of an actor and *all* possible transitions between its states. Not each outgoing transition must be available from the current state, but there must be a sequence of actions leading to this transition.

While not using statecharts, we can still use the underlying mechanism that reduce the visual complexity of state machines, i.e., by projecting the entire state space on a smaller, filtered state space. This mechanism is also used to define projection diagrams [20]. As TLA$^+$ is mostly used for modeling distributed systems, we can create a projection of the state space on the *actors* of the system. For instance, in 2PC, some state pertains to the transaction manager, other states to each resource manager, and finally, a set of messages represents the networked communication.

To project the state space, authors provide mappings from the variables of the specification to the actor which conceptually owns that variable. For 2PC, the model checker returns one large global state space, which combines all actions of all actors into one. Exploring the state space of 2PC with three resource managers yields 1370 distinct states. If we project the state space on the actors, we obtain four separate state spaces for each defined actor. Now, the three projected diagrams for the resource managers have four states each, while the projected diagram for the transaction manager has 25 states.

C. Multiple Views of the State Space

We selected four views drawn from existing literature and participants' wishes expressed during our formative study. All four views can be opened side-by-side through checkboxes at the top of the tool. Our views include three visualizations, a state diagram (Figure 1), a sequence diagram (Figure 2), and a state table (Figure 3), as well as the source text of the specification. We include a video of the interface [35].

As the traditional mechanism of model-checking TLA$^+$ specifications is through explicit building of the modeled state machine, a *state diagram* (Figure 1) is a common choice for visualizing specifications. By isolating diagrams to a single actor instead of showing the full state diagram, we bring the number of states down to a manageable number. Users can select the actor for which they want to see the state diagram. The actor state diagram loses some information: side effects on the state of other actors are not directly visible and it is not always clear what actions of other actors are necessary to reach a specific state. However, this information can still be more easily derived from the other visualizations.

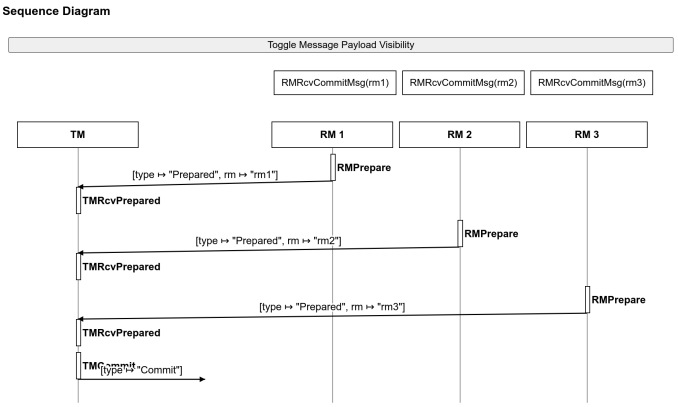

Figure 2. Sequence diagram for the user's configured example sequence of actions. The user had progressed the sequence of actions such that each resource manager prepared ($RMPrepare$) and the transaction manager received the respective prepared messages ($TMRcvPrepare$). Lastly, the user let the transaction manager commit the message. Each resource manager can now receive the message, as signaled by the buttons above the resource manager headers. The sequence diagram shows all actors and the messages passed between them in a vertical, logical timeline, but lacks information about the internal states of the actors. The arrows are labelled with the payload of the messages that actions trigger. The last action triggered the send of the commit message, but none of the resource managers have received it yet. This is indicated by the arrow that does not reach a destination. The state table in Figure 3 shows the identical sequence of actions.

Using the user-formulated example sequence of actions, we can derive a *sequence diagram* (Figure 2). Sequence diagrams, or closely related visualizations like timing diagrams or live sequence charts, are a popular way to visualize distributed systems [27, 23] and have been shown to help improve requirement comprehension [17]. The sequence diagram relies on the isolation of actors and shows the interaction between them. It uses *one* sequence of actions and shows the messages sent between *all* affected actors. In contrast, the state diagram shows *all* possible states of *one* actor.

As sequence diagrams do not present variable values, only the sequence of actions and messages, we added another view that provides a history of all variables in a *state table* (Figure 3). Authors of formal specifications generally try to keep the number of variables low to reduce the specification's complexity, which is beneficial for visualization. The table has a higher information density than the sequence diagram as it provides the actual values in the notation of TLA$^+$, but it is less clear how communication between actors occurs conceptually.

Finally, as the status quo for interacting with formal specifications, we include the read-only *textual source code* of the specification in TLA$^+$. As is common practice for TLA$^+$, our rendering of the spec displays mathematical symbols where applicable, e.g., turning `\forall` into ∀. The browser search can be used to find text occurrences over all views. To maintain the link to the other views, we also integrate the the user-formulated sequence of actions into the text view. Each action that the user can currently trigger is shown above the corresponding action definition in the TLA$^+$ specification. Users can preview the state of the variables used in the action, before and after a possible action.

Table Values

RMRcvCommitMsg(rm1) | RMRcvCommitMsg(rm2) | RMRcvCommitMsg(rm3)

msgs	rmState			tmPrepared	tmState	Action taken
	rm1	rm2	rm3			
{ }	"working"	"working"	"working"	{ }	"init"	
{ [type ↦ "Prepared", rm ↦ "rm1"] }	"prepared"	"working"	"working"	{ }	"init"	RMPrepare(rm1)
{ [type ↦ "Prepared", rm ↦ "rm1"] }	"prepared"	"working"	"working"	{ "rm1" }	"init"	TMRcvPrepared(rm1)
{ [type ↦ "Prepared", rm ↦ "rm1"], [type ↦ "Prepared", rm ↦ "rm2"] }	"prepared"	"prepared"	"working"	{ "rm1" }	"init"	RMPrepare(rm2)
{ [type ↦ "Prepared", rm ↦ "rm1"], [type ↦ "Prepared", rm ↦ "rm2"] }	"prepared"	"prepared"	"working"	{ "rm1", "rm2" }	"init"	TMRcvPrepared(rm2)
{ [type ↦ "Prepared", rm ↦ "rm1"], [type ↦ "Prepared", rm ↦ "rm2"], [type ↦ "Prepared", rm ↦ "rm3"] }	"prepared"	"prepared"	"prepared"	{ "rm1", "rm2" }	"init"	RMPrepare(rm3)
{ [type ↦ "Prepared", rm ↦ "rm1"], [type ↦ "Prepared", rm ↦ "rm2"], [type ↦ "Prepared", rm ↦ "rm3"] }	"prepared"	"prepared"	"prepared"	{ "rm1", "rm2", "rm3" }	"init"	TMRcvPrepared(rm3)
{ [type ↦ "Commit"], [type ↦ "Prepared", rm ↦ "rm1"], [type ↦ "Prepared", rm ↦ "rm2"], [type ↦ "Prepared", rm ↦ "rm3"] }	"prepared"	"prepared"	"prepared"	{ "rm1", "rm2", "rm3" }	"committed"	TMCommit

Figure 3. State table of the user's configured example sequence of actions. The user had progressed the state such that the transaction manager could commit ($TMCommit$) in the most recently selected action. The sequence diagram in Figure 2 shows the identical sequence of actions. The state table visualizes the historic values of all variables defined in the specification. The columns show the different variables, and the rows shows the evolution of the variable states. This representation shows *all* variables of the specification unmodified.

The resulting workflow allows users to show or hide views based on whether they think the view would currently be helpful. As they work to formulate assumptions to support their comprehension, they can make use of the example sequence to find evidence for or against their assumptions. If users realize a different view may help their comprehension better, they can easily switch with no overhead, as our tool keeps the example sequence they configure synchronized across all views.

D. Deriving Visualizations from Specifications

To show the different views, our tool only requires information that can be derived directly from the specifications. The main input for our tool is a dump of the state space resulting from the specification's model-checking process.

The state space dump contains a list of nodes and edges that form a directed graph. A node contains a snapshot of all variables' states. An edge is an action that has been taken by the model checker, going from one state to another.

To obtain a state space dump for TLA$^+$ specifications, we instrumented the model checker to collect the described information and save it to a file once model-checking terminates. We further collect information on which variables are read and written in each action, which we use to determine how messages are exchanged in the sequence diagram.

Information on how to isolate the actors cannot be derived from the specification alone. Consequently, authors of specifications need to map each state snapshot to the corresponding actor states. To do so, each actor is provided with a list of selectors of the state that form the actor's state, for example, in 2PC the actor *RM 1* has only one selector which selects the value of `rmState[rm1]`. Tagging the four actors of 2PC requires 11 such selectors in total.

The state space dump also forms an interface that state-machine-based formal languages other than TLA$^+$ can target, such that they can be used with our tool. Including data tagged by authors, the input our tool requires is as follows:

ID	Age	Sex (m/f/d)	Years SWE	CS Degree
P1	25	m	3	B.Sc.
P2	29	m	4	M.Sc.
P3	24	m	2	M.Sc.
P4	24	m	3	B.Sc.
P5	28	m	7	M.Sc.
P6	23	m	1	B.Sc.
P7	26	m	3	B.Sc.
P8	24	f	1	B.Sc.
P9	26	f	1	B.Sc.
P10	24	f	2	B.Sc.
P11	27	f	5	M.Sc.
P12	34	m	9	Vocational Training
	$\bar{x} = 26$	$m = 75\ \%$	$\bar{x} = 3.4$	

Table II
PARTICIPANT DEMOGRAPHICS.

1) The source code of the specification.
2) A dump of the state space graph including every read and write of variables in the edges of the graph.
3) A mapping from variables to individual actors manually provided by the author. For function variables, authors can map the particular inputs to actors.
4) Optionally, a mapping from variables to messages, and a format string for displaying state variables as text.

IV. Exploratory User Study

To investigate the tool's effect on readers of specifications, we conducted a user study. We formulate the following research questions to explore questions on our tool design:

RQ1 When and why do participants switch between text and visualizations?
RQ2 How do participants make use of interactivity?
RQ3 Is the mathematical TLA$^+$ syntax perceived as hindrance?
RQ4 How do participants perceive the tool's value as documentation?

A. Setup

We recruited 12 software developers for the study. Participants were compensated for their time. Demographic data and descriptive statistics are provided in Table II. None of the participants had prior experience in formal methods or distributed systems. We excluded persons with expertise in distributed systems because we wanted to see how participants would work with systems they did not already know.

To gain additional insights on obstacles that the TLA$^+$ syntax may present, this time, we did not to translate the source text to a JavaScript-like syntax. To assess perceived comprehension, we used the same questions as in our formative study, Table I. For RQ3, we wanted to assess the understanding of the textual TLA$^+$ syntax. Therefore, we added two questions specifically targeted at working with TLA$^+$ syntax, Table III.

We began the study with a short explanation of the study, followed by a demographics and skills questionnaire to select participants. Next, the participants read an introduction about

Nr.	Task
4	Write the safety property “no one resource manager commits a value while another aborts” as a TLA$^+$ formula.
5	What does the following action do? (Participants were given the definition of a *Decide* action that abstracts all behavior of the resource manager into one six-lines action.)

Table III
ADDITIONAL TASKS IN OUR EXPLORATORY USER STUDY.

formal specifications, followed by five pages of introduction to TLA$^+$. The document also contained a reference manual of commonly used TLA$^+$ syntax. For this study, we excluded more advanced topics regarding liveness properties and temporal logic, as these only make up a small (but challenging) part of specifications, and are rarely used in practice [7, 22]. Instead, we focused on the state machine concept, given by the initial state predicate and the state transitions provided by the actions, TLA$^+$ syntax, and safety properties. Participants were allowed to refer back to the document.

In addition, we offered to answer any questions while participants read the document. Reading it usually took around 15 minutes. After answering questions, we asked them to read a short paragraph explaining the goals of 2PC. Finally, we showed a five-minute, pre-recorded introduction video of the tool to ensure that all participants received the same introduction.

Each test run took approximately 2.5 hours to complete. Some participants in our formative study indicated that being watched over while interacting with code-like documents can be stressful. Thus, we told participants that we would not evaluate their answers and that our main concern was to explore what strategies they used when answering. After each question, participants were asked to explain the strategies used to answer the task. If the interviewer observed a wrong answer at this point, or if participants answered the questions quickly, they asked participants if they were sure about their results.

B. Results

To judge the success participants had working with the tool, we asked them how they felt about their understanding of the protocol after using our tool. While a majority indicated that the tool improved their understanding, four were unsure, as seen in Figure 7. This may have been due to a conflation with implementation details. For instance, P1 noted that he “understood the basics”, but felt that for “a proper understanding, I would need the protocol in written [code] form. What happens after an abort? Is there a rollback?” More generally, even though people were only asked to read specifications, four participants voiced concerns about the effort required to create and maintain a specification for use with *SpecTacles*.

RQ1: When and why do participants switch between text and visualizations?

We present the views that participants had opened in Figure 4. Overall, participants used all views, though variance within used views per participant is high.

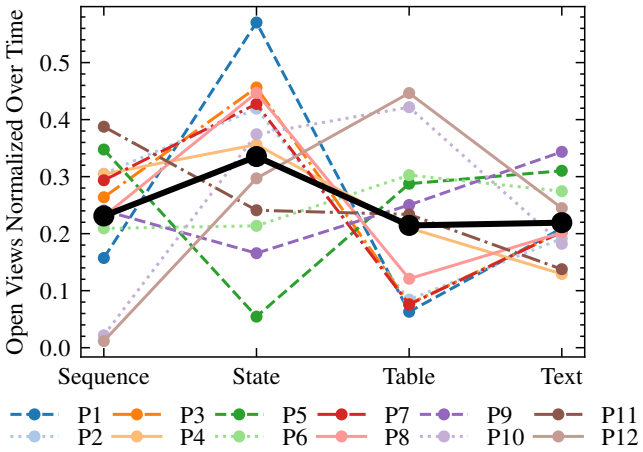

Figure 4. Usage of views per participant. Y axis shows the normalized time a view was open over all tasks. The black line shows the average across all participants.

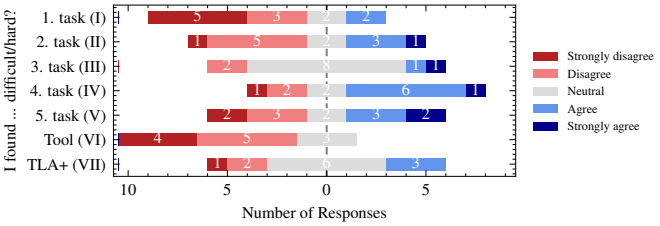

Figure 5. Responses of participants to statements: **(I)** I found the first task difficult to answer; **(II)** I found the second tasks difficult to answer; **(III)** I found the third task difficult to answer; **(IV)** I found the forth task difficult to answer; **(V)** I found the fifth task difficult to answer; **(VI)** This tool was challenging to use; **(VII)** I found the TLA$^+$ syntax hard to comprehend.

a) Participants use visualizations to gain an overview: All Participants initially wanted to gain an overview of the protocol. To do so, they were looking for condensed information of its behavior. Four participants (P5, P9, P10, P12) first attempted to skim over the text view, but quickly switched to other representations as the text view offered too much information. P10 explained that the text was “unsuitable for overview because you can’t grasp everything at once [...], but

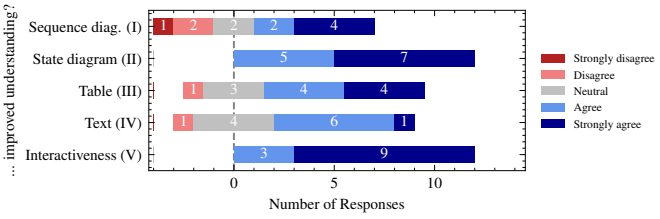

Figure 6. Responses of participants to statements: **(I)** The sequence diagram helped me to better understand the specification; **(II)** The state diagram helped me to better understand the specification; **(III)** The table helped me to better understand the specification; **(IV)** The textual specification helped me to better understand the two phase commit protocol; **(V)** The interactiveness helped me to better understand the specification.

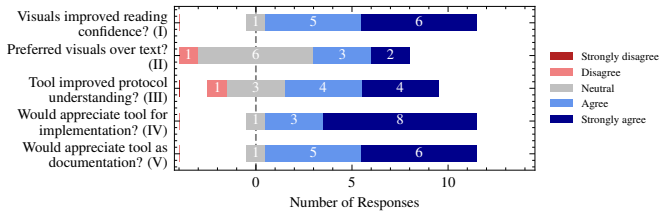

Figure 7. Responses of participants to statements: **(I)** The visual representations improved my confidence in reading the textual representation; **(II)** I preferred the visual representations over the textual specification; **(III)** This tool helped me to better understand how the Two-Phase commit protocol works; **(IV)** I would appreciate this tool when working with an actual implementation of the Two-Phase commit protocol; **(V)** I would appreciate this tool as documentation of system designs for existing projects.

must always understand and read the action, unlike with the state diagram." Similarly, P8 explained that the text view was "too complex" to gain an overview and overwhelmed her. Three participants (P7, P9, P11) also made explicit remarks that the state diagram view was overwhelming or close to being overwhelming for gaining an overview. The state diagram initially showed the state of the transaction manager, as seen in the accompanying video [35]. P7 added that the state diagram was unsuitable because it only showed the state of one actor, although he was interested in the actors' global interactions.

All participants mentioned that the visualizations generally provided a better overview. P4 stated that in the visualizations, he "could focus on the important things" and for P2 the "state diagram provided the big picture. It shows where the goal is and how to get there." As a tendency, participants preferred to work with the visualizations until an information or validation need led them to switch to text. P6 summarized his process as "always first use the visuals, then jump to the code." P2 said that working with the tool was a "continuous learning process. I start with the state diagram and work with it until I do not understand something." P12 found the table view "incredibly valuable [because] it gives a complete history of all the [variable] states in the system." All but one participant indicated that the visualizations improved their understanding of the text (Figure 7). For example, P8 wanted to validate her reasoning but could not fully understand the meaning of an action in text, so she switched to the sequence diagram "to check again how the interactions work."

b) Participants switch to text to prove observed theories: Especially for tasks 2 and 4, participants were looking for precise and unambiguous information, to allow them to generalize assumptions, i.e., to conclude that the observed behavior indeed had the theorized conditions. P7 mentioned that "it is important to look into the [spec's text], to see how [the actions] are written, to verify [and] see what actually happened." P11 said that she "read the text only to confirm what I concluded from the visualizations and only looked at [...] one concrete action [in the text]." P8 noted that the text "felt hard to understand just from skimming over it", and added: "when having a thesis or wanting to know something particular, I would just jump into [the corresponding text part]". Participants did not strongly favor visualizations over text, as seen in Figure 7, and argued in favor of both. P8, for example, said: "I would not leave out the [textual] specification. [...] I'm just not a fan of reading it." P11 further explained this sentiment: "Both visualizations and text are helpful but for different purposes. The visualizations give a feeling [for the protocol] but make it harder to say what exactly the rules are." P1 emphasized the definitive nature of the textual specification: "When someone asks me why things are how they are, I would always show them the text." P4 remarked that "text is for invariants. [...] For example, states that are forbidden can be read in the text," but not in the other views.

c) Participants find text obscures control flow: When asked why they found the text hard to read, participants explained that it was hard to see connections between the actions in the text. P7 explained that "text describes everything precisely, but [...] how everything interacts is not clear." Likewise, P8 found that "especially for text, it is very linear, and that makes it more difficult to understand interactions." P9 noted that the text's problem was that it was hard to tell "when can which transition happen?". P1 was missing the linear nature of imperative code, noting: "When programming, you have a script you can [read top to bottom]." Eight of the twelve participants explicitly made remarks similar to these (P1, P4, P6-10, P12). As further evidence, most participants only read small parts of the text when they wanted to understand something in detail. There was a shared sentiment between many participants that understanding the specification text without the accompanied visualizations would have been more challenging. P10 noted that "text is not suitable for gaining an overview because [...] I read one action at a time, unlike with the state diagram." P5 commented: "I want to see if I can better derive [in the state diagram], what I laboriously scrambled together in the code."

RQ2: How do participants make use of interactivity?

SpecTacles allows users to interactively explore the state space by formulating example sequences of actions (see section III). Throughout the study, we referred to the *interactiveness* of the tool as this construction of sequences. As seen in Figure 6, overall, participants strongly agreed that this form of interactivity helped their understanding.

d) Interactivity for exploration: As established in RQ1 a, participants initially sought an overview of the protocol's behavior. For this, most found the text view and some even the state diagram overwhelming. Instead, we observed that participants started exploring interactivity by simply clicking buttons and inspecting the effects, which participants found easier and requiring less mental load. For such exploration, they mostly used the table view or sequence diagram because they show an overview of all actors of the system, but only provided information explicitly asked for, i.e., what actions are possible at a specified state of the system. Over time, they inferred an understanding under what conditions specific actions become available. P12 used interactivity for "playing around, and clicking around." Both P5 and P8 likened the

interactivity to games, saying that it was "helpful to play with examples quickly," and that it provided a "playful feeling of experimenting," respectively. P9 noted "a low threshold for trying out paths." P8 said that, through interactivity, working with the specification had "felt like exploration, making it almost seem simulated."

e) Interactivity for instant feedback: All Participants used interactivity heavily to test their assumptions. P3 appreciated that he could "test each possible action, to check if I arrive in the goal state." P3 further noted: "I can see with one click what the action did. [...] If you only got the text [...] you might think 'I got it', but [using the visualizations], you can really check if it has the effects [you think it has]." Participants expressed that, through formulating example sequences of actions, they felt confident in their understanding. P6 compared the difference between interactivity and non-interactivity to the difference between "[hearing a] lecture versus doing it myself." P9 built custom examples in the sequence diagram, and stated that it "removes any doubt, and facilitates learning by doing." For P7, "feedback [from custom examples] helped to get a feeling, to internalize" the protocol.

f) Interactivity to understand relationships and dependencies: Two participants noted they used interactivity specifically to get a sense of relationships and dependencies. P7 liked interactivity "because it helped me a lot to understand how everything is connected, how single actions affect [the system]." Interactivity relieved P5 of "having to understand and retrace the constraints [of the protocol] myself." P5 added that without the interactivity, he would usually "simulate interactivity with his mouse or finger." P6 preferred the tool over "having to build the graph on my own in my head," referring to possible states of the system and their connection.

RQ3: Is textual TLA$^+$ syntax perceived as a hindrance?

No clear theme emerged regarding this research question, as participants had different opinions on the syntax of TLA$^+$. In the post-questionnaire, most participants were unsure whether they perceived TLA$^+$ syntax as hard to comprehend, as seen in Figure 5. Still, participants rated tasks concerning comprehension of TLA$^+$[4] as more difficult compared to the other tasks. In particular, the task of formulating a TLA$^+$ action was perceived as the most challenging of all tasks, on average. Only one participant, however, indicated that the textual specification did not improve their understanding (Figure 6).

A few explicit remarks were made in favor of TLA$^+$'s syntax, mostly noting the simplicity or familiarity with it. P2 noted that "the syntax on its own is not really complex," and P6 noted that the syntax "needs some getting used to, but is actually not hard to understand." Concerning writing of TLA$^+$, P1, for example, noted he was "familiar with algebra, thus I found this easy to formulate. [A TLA$^+$ formula] is really close to a mathematical formula." Likewise, P7 said that the syntax was not hard "because of the common mathematical constructs that I already know." P11 also attributed her neutral response to her background: "The syntax was ok: hard to write but somewhat easy to understand. You know the symbols from university classes." Two of our participants were not familiar with the mathematical notation; still, one mostly used the text and table views, while the other seldom used of the text.

Other participants noted a preference for a different formulation than then TLA$^+$ syntax. P4 said that he "always had trouble with predicate logic", and while "the [state machine] semantics are easy to comprehend, I never liked understanding syntax that only includes logic symbols." P9 said she felt intimidated by mathematical syntax, even though the syntax of TLA$^+$ was ultimately easy to understand for her. P8 expressed a preference for pseudo-code and P6 noted a preference for languages that use identifiers akin to natural language, such as spelling out "for all" as opposed to using a symbol.

RQ4: How do participants perceive the tool's value as documentation?

Through question (V) of Figure 7, we wanted to assess how participants would see our tool as a source of documentation. We asked participants to imagine that the relevant code base is already well documented through traditional means. This may include README files, static diagrams, or design documents. In the questionnaire, all but one participant agreed that our tool design would still provide a benefit. Three themes of how *SpecTacles* acts as documentation emerged from the participants' answers.

g) Abstraction: Overall, participants shared the sentiment that *SpecTacles* made "grasping the complexity" (P2) of a system easier. P2 also noted that the tool would help "better understanding [a system's] processes". P5 said that the tool "would be very helpful for system designs", for example, seeing the authentication states and flows, with similar notes from P3. P2 liked the state diagram in particular, noting that "if I worked on a software architecture and would see [the diagram], it would be super helpful." P8 appreciated the specification's abstraction, noting that "you don't have to think how exactly some service responds, but can simply see what happens."

h) Easy exploration of a system on their own terms: During the interview, participants appeared to value the tool as a means to easily explore a system on their own terms. Five Participants (P2, P4, P8, P11, P12) explicitly appreciated that the tool provides a simple way of exploring a model of a system. When P9 compared documentation in her job to *SpecTacles*, she noted "building example cases [in *Spectacles*] helps me more than just looking at [a single scenario]. [...] Usually, I want to try out multiple." P12 noted "I don't like simply reading documentation. [The tool] feels like interactive documentation where I can try things out." This sentiment was shared by P8 and P11, who valued the tool as a "safe space" (P8) for exploring states without having to "fiddle with a running system" (P11). P9 noted that when getting into a new code base, she either "would try things out explicitly in the code," or "write diagrams of my current understanding down on paper", both of which the tool would simplify.

[4]Questions (IV) and (V) of Figure 5.

i) Second source of truth: We asked participants to imagine working on a task to add a new feature or fix a bug in an actual implementation of the 2PC protocol. All but one participant agreed that they would appreciate our tool as an aid for implementation ((IV) of Figure 7). During the interview, nine participants remarked that the tool would support them while programming (P1-8, P12): participants imagined using *SpecTacles* to compare or verify the behavior of their implementation. For P6 and P11, the tool might allow checking that a state inspected during debugging was valid. P12 appreciated "the option of going through the states, to check transformations, and being able to compare it with code." P11 stated: "When debugging, I usually can't access everything, so I have to write fake events and so on." P12 and P8 noted that being able to instantly simulate sequences of communication might allow them to quickly verify if a bug exists in the protocol or the implementation. For P7 and P11, the tool promised to be helpful for deriving what tests might be interesting for their implementation, as it helped identify corner cases.

C. Threats to Validity

Our study was designed to capture qualitative statements, so insights might not generalize. To assess comprehension, we only relied on self-report questionnaires and our own observations, limiting ourselves to perceived comprehension of the participants. The professional software engineering experience of our participants is small, with an average of 3.4 years of experience. Prior formal education in mathematics could bias the results. To assess this bias in our study, we mapped the responses of the perceived difficulties of the tasks and TLA$^+$ syntax (Figure 5) to numerical values[5] and grouped them by the highest degree obtained. The differences in resulting mean averages of M.Sc.$= -0.05$, B.Sc.$= -0.01$, Training=0.10[6] appear too small to draw conclusions.

To counter the risk of demand characteristics, we did not answer questions referring to the development process or authors of the tool. When participants gave positive feedback on the tool, we asked participants to further outline their reasoning, to filter spontaneous reactions and instead obtain rationalisations of the participants' preferences.

Some of our user tests took place after a full work-day, whereas others were tested during the weekend. As we explored the formal specifications under the lens of software documentation, as stated in section I, these factors, however, provide a more accurate picture of real-life instances where software documentation is read.

V. Discussion and Future Work

Participants of both our studies relied on examples to build their comprehension (RQ2 f). These examples most commonly concerned either information local to one action (what actions are enabled by this action, and which enable it?) or information global to the relationship between actions (which sequences are possible?). This control flow information appears to be difficult to obtain from the textual specification we showed participants, creating a major hurdle to their perceived comprehension (RQ1 c). For example, most of our participants had no issue understanding this action of the 2PC specification[7]:

$RMRcvCommitMsg(rm) \triangleq$

Does set msgs contain a message of type commit?

$\land [type \mapsto \text{“Commit”}] \in msgs$

Then set state of given resource manager rm to committed.

$\land rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“committed”}]$

And leave remaining variables unchanged.

$\land \text{UNCHANGED } \langle tmState,\ tmPrepared,\ msgs \rangle$

Yet, it was difficult for participants to relate actions with each other. Actions do not provide information about what other actions they enable after running, or what actions are necessary to reach another. Instead, they are guarded by conditions, forcing the reader to check all actions to find outgoing and incoming connections. Contrast this to an imperative, typed language, where control flow is made statically explicit through function calls and directly accessible through an IDE.

The four interactive views in *SpecTacles* appear to have made more of the control flow information participants were seeking accessible. Participants used visualizations to gain an overview (RQ1 b), for example by identifying goal states in the state diagram and backtracking to a start state to comprehend control flow. Contrarily, participants of our formative study without our tool employed the same backtracking strategy but struggled to follow the specification. Still, as all three of our visualizations were deliberately leaving out details of the state space, participants used source text to verify assumptions drawn from visualizations (RQ1 a).

Naur's theory-building idea mentioned in section I implies that the specification alone could never document all required knowledge to fully comprehend the theory of a specification's original author. Conversely, to build a correct theory, programmers must work with the program source *and* consult with the program author(s). While *SpecTacles* cannot replace consultation with the authors, it may supports users in building their own theories[8] of the program. *SpecTacles* appears to do so by shortening the feedback loop, leading participants to iteratively hypothesize about and test the specification's behavior. Participants appeared to benefit from the familiarity of the diagrams, given the perceived ease of using the tool (Figure 5), which primarily consists of visualizations, and the absence of any questions on visualization. Future work could explore what additional insight the original author would provide to the reader if they were present.

In terms of the impact of the mathematical syntax of TLA$^+$, we cannot draw a clear conclusion. Many participants did wish for a language more akin to programming languages.

[5]-2 being "Strongly disagree" and +2 being "Strongly agree".

[6]The value for the single participant with vocational training is only given for completeness.

[7]Note that the version given to participants only had one comment at the top stating "Resource manager rm is told by the TM to commit."

[8]which may or may not match the theories of the original author(s).

Our studies suggest, however, that a simple replacement of syntax does not help.

Future work could investigate how far the findings of our study generalize. A quantitative study could provide proof for our qualitative observations. Further, it would be interesting to see how participants would benefit from the tool when combined with an actual implementation.

We tested *SpecTacles* on a range of other specifications, namely, with an education-focus like the dining philosopher problem or with an industry-focus like paxos [12]. While *SpecTacles* works with those specifications, we identify two directions for further improvements. First, *SpecTacles* relies on separation into different actors. Sometimes, as in the dining philosopher problem, the focus of the specifications is not on actors but on synchronization of shared resources. Other times, as in paxos, actors may change roles. Future work could explore other visualizations to better emphasize these.

Second, while *SpecTacles* generally scales well to complex specifications like paxos, complex specifications may require authors to define more elaborate mappings. In practice, formal specifications are usually scoped to specific error-prone aspects of a system [7, 8], limiting the complexity of the specification and the corresponding visualizations in *SpecTacles*. The information conveyed by the sequence diagram and the table view are even more constrained because they show only one specific sequence of actions, easily scaling to complex specifications. But the state diagram conveys the *complete* state space per actor, which means that complex actor state spaces might easily overwhelm users. For example, in 2PC, the actor state space of the transaction manager contains a set of all prepared RMs, amounting to $2^{|RM|}$ variations. To reduce the visual state space, more abstract projections can be used: in this example we could aggregate the number of prepared RMs, resulting in $|RM|$ variations. Future work could investigate these dynamic or user-specified mappings.

VI. Related Work

Our work demonstrates how text may be inadequate to answer some information needs and how interactive visualizations can augment text to support readers' comprehension. Various works have investigated using visualizations and adapting textual formal languages to improve reading comprehension.

Similar to our findings, an empirical study of the comprehension of textual, graphical, and tabular forms of state machines [9] found that participants preferred visualizations to gain an overview but struggled to derive details, such as trigger conditions. In terms of the relation to textual specifications, the authors state that participants found them "easy to read [but] very difficult to use" but do not go into further detail. To an extent, this appears to contradict our findings, where participants stated the utility of the source text when combined with the visualizations. In their work, the authors mitigated state explosion of visualizations through Statecharts [2], while we opted for the projection approach centered on actors.

A literature survey identified obstacles to teaching formal methods [26]. As mitigation, teachers make formal methods easier by using examples, by better engaging students with the material, by integrating formal methods into other parts of their curriculum, or by simplifying the material. Simplifying is commonly realized by switching to simpler or more familiar tools. Reducing feedback cycles and placing formal methods in programming contexts was also found to be beneficial [30].

In a discussion on the accessibility of formal methods [19], the author highlights the importance of textual formal notations for their precision but also the strength of visualizations to make formal methods approachable. The author advocates "graphical formal methods" as the combination of visualizations with textual formal notations, however noting that "underlying factors that contribute to superiority of graphical notation are not well understood". Our findings further support that both forms of views have unique strengths and highlight key advantages of visualizations when used together with text.

A common suggestion to improve the accessibility of formal methods is to change their language's syntax or structure, for example by using DSLs closer to the domain of the user [23, 15, 11]. An experience report on the use of a specification language by industry partners finds "that readability and reviewability by domain experts can be further enhanced by minimizing the semantic distance between the reviewer's mental model and the constructed models" [5]. Industry reports indicate that PlusCal, a DSL of TLA$^+$ more similar to programming languages, is more approachable for beginners and domain experts [24, 21]. Further, a study on best practices for writing formal specifications in the Z language found that some do indeed influence reading comprehension [18].

VII. Conclusion

Formal specifications benefit authors by forcing them to incrementally and precisely write down their theories to deepen comprehension. Through a formative study, we found that participants were struggling to comprehend formal specifications written by others. They expressed wishes for visualizations and concrete examples to facilitate their understanding.

We propose *SpecTacles*, a tool for the exploration of formal specifications using state dumps from model checkers. *SpecTacles* assists readers in building comprehension by providing four interactive views of the specification that incorporate visualizations and concrete example sequences of actions.

We learned that the source text of the specification appeared to lack control flow information. Through *SpecTacles*'s visualizations, participants appeared to gain access to different slices of the specification's state space, aiding their comprehension. Participants indicated that our tool design provides a valuable source of documentation and aid for implementation.

Acknowledgments

This work was supported by the HPI–MIT "Designing for Sustainability" research program[9], SAP, and Deutsche Forschungsgemeinschaft (DFG, 449591262).

[9]https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html

References

[1] Peter Naur. "Programming as theory building". In: *Microprocessing and microprogramming* 15.5 (1985). Publisher: Elsevier, pp. 253–261. URL: https://www.sciencedirect.com/science/article/pii/0165607485900328 (visited on 04/23/2024).

[2] David Harel. "Statecharts: A Visual Formalism for Complex Systems". In: *Sci. Comput. Program.* 8.3 (1987), pp. 231–274. DOI: 10.1016/0167-6423(87)90035-9. URL: https://doi.org/10.1016/0167-6423(87)90035-9.

[3] Jonathan P. Bowen and Michael G. Hinchey. "Ten Commandments of Formal Methods". In: *Computer* 28.4 (1995), pp. 56–63. DOI: 10.1109/2.375178. URL: https://doi.org/10.1109/2.375178.

[4] K. Finney. "Mathematical notation in formal specification: too difficult for the masses?" In: *IEEE Transactions on Software Engineering* 22.2 (Feb. 1996). Conference Name: IEEE Transactions on Software Engineering, pp. 158–159. ISSN: 1939-3520. DOI: 10.1109/32.485225. URL: https://ieeexplore.ieee.org/abstract/document/485225 (visited on 04/17/2024).

[5] Nancy G. Leveson, Mats Per Erik Heimdahl, and Jon Damon Reese. "Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future". In: *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*. Ed. by Oscar Nierstrasz and Michel Lemoine. Vol. 1687. Lecture Notes in Computer Science. Springer, 1999, pp. 127–145. DOI: 10.1007/3-540-48166-4_9. URL: https://doi.org/10.1007/3-540-48166-4%5C_9.

[6] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model Checking TLA+ Specifications". In: *Correct Hardware Design and Verification Methods*. Ed. by Laurence Pierre and Thomas Kropf. Red. by Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen. Vol. 1703. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66. ISBN: 978-3-540-66559-5 978-3-540-48153-9. DOI: 10.1007/3-540-48153-2_6. URL: http://link.springer.com/10.1007/3-540-48153-2_6 (visited on 04/21/2024).

[7] Brannon Batson and Leslie Lamport. "High-Level Specifications: Lessons from Industry". In: *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 2852. Lecture Notes in Computer Science. Springer, 2002, pp. 242–261. DOI: 10.1007/978-3-540-39656-7_10. URL: https://doi.org/10.1007/978-3-540-39656-7%5C_10.

[8] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN: 0-3211-4306-X. URL: http://research.microsoft.com/users/lamport/tla/book.html.

[9] Marc K. Zimmerman, Kristina Lundqvist, and Nancy G. Leveson. "Investigating the readability of state-based formal requirements specification languages". In: *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*. Ed. by Will Tracz, Michal Young, and Jeff Magee. ACM, 2002, pp. 33–43. DOI: 10.1145/581339.581347. URL: https://doi.org/10.1145/581339.581347.

[10] D. Carew, C. Exton, and J. Buckley. "An empirical investigation of the comprehensibility of requirements specifications". In: *2005 International Symposium on Empirical Software Engineering, 2005.* 2005 International Symposium on Empirical Software Engineering, 2005. Nov. 2005, 10 pp.–. DOI: 10.1109/ISESE.2005.1541834. URL: https://ieeexplore.ieee.org/abstract/document/1541834 (visited on 04/17/2024).

[11] Anthony Hall. "Realising the Benefits of Formal Methods". In: *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*. Ed. by Kung-Kiu Lau and Richard Banach. Vol. 3785. Lecture Notes in Computer Science. Springer, 2005, pp. 1–4. DOI: 10.1007/11576280_1. URL: https://doi.org/10.1007/11576280%5C_1.

[12] Jim Gray and Leslie Lamport. "Consensus on transaction commit". In: *ACM Trans. Database Syst.* 31.1 (2006), pp. 133–160. DOI: 10.1145/1132863.1132867. URL: https://doi.org/10.1145/1132863.1132867.

[13] Rozilawati Razali et al. "Experimental Comparison of the Comprehensibility of a UML-based Formal Specification versus a Textual One". In: *11th International Conference on Evaluation and Assessment in Software Engineering, EASE 2007, Keele University, UK, 2-3 April 2007*. Ed. by Barbara A. Kitchenham, Pearl Brereton, and Mark Turner. Workshops in Computing. BCS, 2007. URL: http://ewic.bcs.org/content/ConWebDoc/10663.

[14] Leslie Lamport. "Computation and state machines". In: (2008). URL: https://www.microsoft.com/en-us/research/publication/2016/12/Computation-and-State-Machines.pdf (visited on 04/23/2024).

[15] Andrei Lapets. *Improving the Accessibility of Lightweight Formal Verification Systems*. 2009. URL: https://open.bu.edu/handle/2144/1739.

[16] David Lorge Parnas. "Really Rethinking 'Formal Methods'". In: *Computer* 43.1 (2010), pp. 28–34. DOI: 10.1109/MC.2010.22. URL: https://doi.org/10.1109/MC.2010.22.

[17] Silvia Abrahão et al. “Assessing the Effectiveness of Sequence Diagrams in the Comprehension of Functional Requirements: Results from a Family of Five Experiments”. In: *IEEE Trans. Software Eng.* 39.3 (2013), pp. 327–342. DOI: 10.1109/TSE.2012.27. URL: https://doi.org/10.1109/TSE.2012.27.

[18] Andreas Bollin and Dominik Rauner-Reithmayer. “Formal specification comprehension: the art of reading and writing z”. In: *Proceedings of the 2nd FME Workshop on Formal Methods in Software Engineering, FormaliSE 2014, Hyderabad, India, June 3, 2014*. Ed. by Stefania Gnesi and Nico Plat. ACM, 2014, pp. 3–9. DOI: 10.1145/2593489.2593491. URL: https://doi.org/10.1145/2593489.2593491.

[19] R. Razali. “Understanding the Efficacy of Graphical Formal Methods-Empirical Assessment”. In: *World Applied Sciences Journal* 31.5 (2014), pp. 889–903.

[20] Lukas Ladenberger and Michael Leuschel. “Mastering the Visualization of Larger State Spaces with Projection Diagrams”. In: *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*. Ed. by Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi. Vol. 9407. Lecture Notes in Computer Science. Springer, 2015, pp. 153–169. DOI: 10.1007/978-3-319-25423-4_10. URL: https://doi.org/10.1007/978-3-319-25423-4%5C_10.

[21] Chris Newcombe et al. “How Amazon web services uses formal methods”. In: *Commun. ACM* 58.4 (2015), pp. 66–73. DOI: 10.1145/2699417. URL: https://doi.org/10.1145/2699417.

[22] Antti Pakonen et al. “User-friendly formal specification languages - conclusions drawn from industrial experience on model checking”. In: *21st IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2016, Berlin, Germany, September 6-9, 2016*. IEEE, 2016, pp. 1–8. DOI: 10.1109/ETFA.2016.7733717. URL: https://doi.org/10.1109/ETFA.2016.7733717.

[23] Cheng Pang et al. “A study on user-friendly formal specification languages for requirements formalization”. In: *14th IEEE International Conference on Industrial Informatics, INDIN 2016, Poitiers, France, July 19-21, 2016*. IEEE, 2016, pp. 676–682. DOI: 10.1109/INDIN.2016.7819246. URL: https://doi.org/10.1109/INDIN.2016.7819246.

[24] Stefan Resch and Michael Paulitsch. “Using TLA+ in the Development of a Safety-Critical Fault-Tolerant Middleware”. In: *2017 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Toulouse, France, October 23-26, 2017*. IEEE Computer Society, 2017, pp. 146–152. DOI: 10.1109/ISSREW.2017.43. URL: https://doi.org/10.1109/ISSREW.2017.43.

[25] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. “The TLA+ Toolbox”. In: *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE at FM 2019, Porto, Portugal, 7th October 2019*. Ed. by Rosemary Monahan, Virgile Prevosto, and José Proença. Vol. 310. EPTCS. 2019, pp. 50–62. DOI: 10.4204/EPTCS.310.6. URL: https://doi.org/10.4204/EPTCS.310.6.

[26] Rustam Zhumagambetov. “Teaching Formal Methods in Academia: A Systematic Literature Review”. In: *Formal Methods - Fun for Everybody - First International Workshop, FMFun 2019, Bergen, Norway, December 2-3, 2019, Revised Selected Papers*. Ed. by Antonio Cerone and Markus Roggenbach. Vol. 1301. Communications in Computer and Information Science. Springer, 2019, pp. 218–226. DOI: 10.1007/978-3-030-71374-4_12. URL: https://doi.org/10.1007/978-3-030-71374-4%5C_12.

[27] Ivan Beschastnikh et al. “Visualizing Distributed System Executions”. In: *ACM Trans. Softw. Eng. Methodol.* 29.2 (2020), 9:1–9:38. DOI: 10.1145/3375633. URL: https://doi.org/10.1145/3375633.

[28] Delano Oliveira et al. “Evaluating Code Readability and Legibility: An Examination of Human-centric Studies”. In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 348–359. DOI: 10.1109/ICSME46990.2020.00041. URL: https://doi.org/10.1109/ICSME46990.2020.00041.

[29] Judah Schvimer, A. Jesse Jiryu Davis, and Max Hirschhorn. “eXtreme Modelling in Practice”. In: *Proc. VLDB Endow.* 13.9 (2020), pp. 1346–1358. DOI: 10.14778/3397230.3397233. URL: http://www.vldb.org/pvldb/vol13/p1346-schvimer.pdf.

[30] James Noble et al. “More Programming Than Programming: Teaching Formal Methods in a Software Engineering Programme”. In: *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Vol. 13260. Lecture Notes in Computer Science. Springer, 2022, pp. 431–450. DOI: 10.1007/978-3-031-06773-0_23. URL: https://doi.org/10.1007/978-3-031-06773-0%5C_23.

[31] Mario Gleirscher, Jaco van de Pol, and Jim Woodcock. “A manifesto for applicable formal methods”. In: *Softw. Syst. Model.* 22.6 (2023), pp. 1737–1749. DOI: 10.1007/S10270-023-01124-2. URL: https://doi.org/10.1007/s10270-023-01124-2.

[32] A. Finn Hackett, Joshua Rowe, and Markus Alexander Kuppe. “Understanding Inconsistency in Azure Cosmos DB with TLA+”. In: *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP at ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1–12. DOI:

10.1109/ICSE-SEIP58684.2023.00006. URL: https://doi.org/10.1109/ICSE-SEIP58684.2023.00006.

[33] Arut Prakash Kaleeswaran et al. “A user study for evaluation of formal verification results and their explanation at Bosch”. In: *Empir. Softw. Eng.* 28.5 (2023), p. 125. DOI: 10.1007/S10664-023-10353-4. URL: https://doi.org/10.1007/s10664-023-10353-4.

[34] Patrick Rein et al. “Too Simple? Notions of Task Complexity used in Maintenance-based Studies of Programming Tools”. In: *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE. 2023, pp. 254–265.

[35] Daniel Stachnik et al. *Supplemental Material to “SpecTacles: Supporting Control Flow Comprehension of Software Developers in TLA+”*. July 2024. DOI: 10.5281/zenodo.12666180. URL: https://doi.org/10.5281/zenodo.12666180.