Visual Analysis of Hyperproperties for Understanding Model Checking Results

Tom Horak, Norine Coenen, Niklas Metzger, Christopher Hahn, Tamara Flemisch, Julián Méndez, Dennis Dimov, Bernd Finkbeiner, and Raimund Dachselt

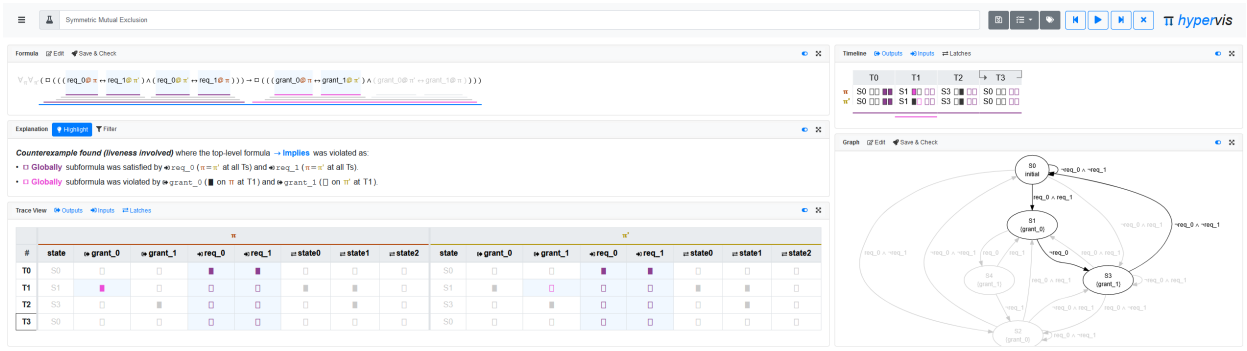

Fig. 1. HYPERVIS provides linked interactive views with highlighting mechanisms for analyzing counterexamples of hyperproperties.

Abstract— Model checkers provide algorithms for proving that a mathematical model of a system satisfies a given specification. In case of a violation, a counterexample that shows the erroneous behavior is returned. Understanding these counterexamples is challenging, especially for hyperproperty specifications, i.e., specifications that relate multiple executions of a system to each other. We aim to facilitate the visual analysis of such counterexamples through our HYPERVIS tool, which provides interactive visualizations of the given model, specification, and counterexample. Within an iterative and interdisciplinary design process, we developed visualization solutions that can effectively communicate the core aspects of the model checking result. Specifically, we introduce graphical representations of binary values for improving pattern recognition, color encoding for better indicating related aspects, visually enhanced textual descriptions, as well as extensive cross-view highlighting mechanisms. Further, through an underlying causal analysis of the counterexample, we are also able to identify values that contributed to the violation and use this knowledge for both improved encoding and highlighting. Finally, the analyst can modify both the specification of the hyperproperty and the system directly within HYPERVIS and initiate the model checking of the new version. In combination, these features notably support the analyst in understanding the error leading to the counterexample as well as iterating the provided system and specification. We ran multiple case studies with HYPERVIS and tested it with domain experts in qualitative feedback sessions. The participants' positive feedback confirms the considerable improvement over the manual, text-based status quo and the value of the tool for explaining hyperproperties.

Index Terms—Analyzing Counterexamples, Hyperproperties, Multiple Coordinate Views, Explainable Formal Methods.

1 INTRODUCTION

Model checking [23] is a highly efficient technique for the computer-aided verification of computer systems such as integrated circuits, network protocols, and software. Model checking has long made the transition from research into practice and is routinely used by companies like Intel, Microsoft, or Amazon. Intel, for example, replaced testing with verification for the core execution cluster in their design of the Intel Core i7 processor [46] and, recently, the initial boot code in data centers at Amazon Web Services (AWS) has been model checked to be memory safe [27]. The key advantage of model checking is that it is an *automatic* method: given a system description M and a logical specification φ of a desired behavioral property, the model checker automatically determines whether or not M satisfies φ. If the system design is erroneous, the model checker generates a counterexample in the form of a specific execution of M that violates φ. While finding the counterexample is completely automatic, model checking typically provides very little assistance in actually *understanding* the counterexample and its underlying design flaw. Model checkers typically output the counterexample in the form of a detailed listing that contains the complete state information for every step of a computation that leads to the violation. Understanding all this data is already difficult for small designs and, for more complex systems and specifications, quickly becomes a daunting task.

In this paper, we present a visualization system that aids the analyst in understanding the counterexamples found by the model checker. The visualization views communicate the core aspects of the model checking result to the analyst and support an iterative analysis process. We specifically focus on *hyperproperties* [51], a class of system specifications that is essential for the analysis of security-critical systems. Hyperproperties express the absence of undesired dependencies or flows of information, such as those exploited in the infamous Meltdown [56] and Spectre [48] attacks. A counterexample to a hyperproperty is a set of executions of the system that, together, are problematic. For example, if some software needs to keep certain data secret, then two executions that result from different values of the secret (but agree

- *T. Horak, T. Flemisch, J. Méndez, and D. Dimov are with the Interactive Media Lab at Technische Universität Dresden. Emails: horakt@acm.org, tamara.flemisch@tu-dresden.de, dennis.dimov@tu-dresden.de, julian_jesus.mendez_oconitrillo@mailbox.tu-dresden.de*
- *R. Dachselt is with the Interactive Media Lab, the Centre for Tactile Internet (CeTI), and the Cluster of Excellence Physics of Life (PoL) at Technische Universität Dresden. Email: dachselt@acm.org*
- *N. Coenen, N. Metzger, C. Hahn, and B. Finkbeiner are with the Reactive Systems Group at CISPA Helmholtz Center for Information Security. Emails: {norine.coenen, niklas.metzger, christopher.hahn, finkbeiner}@cispa.de*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

on the public inputs) should not show any difference on the public outputs. Consequently, the model checker searches for such a pair of executions that differ in their public output values. Our goal is to help the analyst understand the violation of the hyperproperty by visualizing the relationship between the individual system executions, as well as the relationship to the system description and the logical formula. To support this, we have implemented the interactive tool HYPERVIS (Fig. 1, imld.de/hypervis), that follows a multiple coordinated views approach [21, 72]. We provide five interconnected views. The hyperproperty specification is shown as a logical formula in the *formula view*, the system as a state machine in the *graph view*, the executions over time in a tabular-like *trace view* and in a more compact *timeline view*. Additionally, there is a textual explication in the *explanation view*.

The fundamental challenge is that the connections between the different views and the relevance of their individual components is not known in advance, but rather must be deduced specifically for the hyperproperty of interest. We address this challenge with an automated causal analysis of the counterexample, where we identify those elements of the different views that directly contribute to the violation of the specification. The textual explication in the *explanation view* is directly based on this analysis. In all other views, the relevant elements can be directly highlighted. By incorporating easy-to-parse value encodings and clear color mappings alongside interactive mechanisms such as linked highlighting and debugger-like functionalities, we support the analyst in recognizing the counterexample's characteristics and in relating its different components. Finally, after the cause of the violation is understood, the analyst can correct the system and the specification directly within the interface through integrated editing functionalities.

HYPERVIS is the result of an interdisciplinary effort and a highly iterative design process which included joint brainstormings and discussions between visualization and model checking experts. The results and insights from this joint effort are presented in this paper. Specifically, we contribute: (1) an in-depth analysis of challenges, (2) the design of visualization and interaction concepts enabling the visual analysis of the model checking results, (3) the realization of these concepts with HYPERVIS as a web-based tool alongside integrated editing facilities, and (4) insights from applying multiple case studies to our tool and conducting user feedback sessions with 6 participants. In summary, our work contributes to a class of visualization solutions that aims at visually explaining complex and abstract computing concepts.

2 WORKING WITH HYPERPROPERTIES

To support the analysis of counterexamples, we first need to understand the involved components, current workflows, and prevalent challenges. Therefore, we will first describe the formal objects utilized during the model checking process on a toy information-flow control problem, where a system needs to satisfy observational determinism (Sect. 2.1). Then, we will detail the current workflow using a slightly bigger example (Sect. 2.2), before outlining the resulting challenges for analyzing counterexamples (2.3).

2.1 Example: Verifying Observational Determinism

In general, the considered objects include the system model M, the counterexample executions π and π', and the hyperproperty specification φ. In this simplified example, our model M (Fig. 2) is prone to leak a secret s via the publicly observable outputs o_1 and o_2 to an attacker. The underlying security lattice considers the secret s to be a confidential input that should not be visible to any observer while the input i and the outputs o_1 and o_2 are publicly observable. The model M can be represented as a finite state machine, where the current state determines the system's output, and the transitions of the finite state machine are labelled with the inputs to the system. All inputs and outputs are binary values, thus, they are either present or absent. When executing such a system, the present inputs and outputs are observed over multiple time steps. The given system in Fig. 2 cycles in the first state, outputting nothing, until an input i is present. Depending on whether a secret s is also given, the system then either outputs o_2 or o_1 indefinitely. If an attacker now happens to observe two executions of the system where the outputs are different although the input i were the same on both executions, they can conclude about the secret s at this time step.

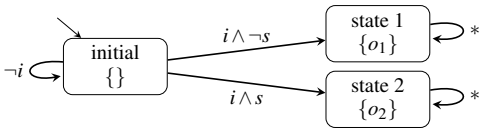

Fig. 2. A simple system leaking a secret s through the observable outputs o_1 and o_2.

The specification φ that we would like to verify for the system M is given as a HYPERLTL formula [24], a linear-time temporal logic for hyperproperties that can relate multiple executions. For the example above, we would like to require observational determinism, which is formalized in HYPERLTL as follows: $\forall\pi\,\forall\pi'\,\square(i_\pi \leftrightarrow i_{\pi'}) \rightarrow \square(o_\pi \leftrightarrow o_{\pi'})$. The formula quantifies universally ($\forall$) over two traces π and π'. The temporal modality $\square$ means "globally", i.e., the formula $\square\varphi$ requires the subformula φ to hold at every point in time. The given formula thus states that for all trace pairs π and π' it must hold that when the observable inputs are the same at every point in time, the respective observable outputs must also be equal. Given the model and formula, a model checker would now provide two specific executions where at a given time step the outputs differ while the inputs are equal.

2.2 Current Workflow

With the growing complexity of both the system and the specification, the model checking of hyperproperties quickly becomes complicated. We demonstrate the current workflow and the corresponding challenges when invoking a model checker for hyperproperties on a more involved example. To this end, we consider a system that arbitrates the access of two processes to a shared resource. Both processes can request access to their critical section (using *req_i*) where they can interact with the shared resource, and the arbiter grants the access (with *grant_i*) while ensuring mutual exclusion, i.e., only one of the processes can enter its critical section at any given time. The arbiter guarantees that every request will eventually be answered while not giving out spurious grants, i.e., every grant will have been requested before. The finite state machine for this system is sketched on the right in Fig. 3. We want to check whether the arbiter is symmetric, thus, if a pair of traces π and π' with symmetric requests at every step (i.e., $\square(req\_0_\pi \leftrightarrow req\_1_{\pi'})$ and vice versa) also gives the grants symmetrically. This hyperproperty checks if any of the processes has an unfair advantage and is favoured when granting access to the critical section. The corresponding HYPERLTL formula expressing symmetry is noted on the left in Fig. 3. The system grants processes asymmetrically: If $\pi = \pi'$ and both processes request initially, then always process 0 is granted first (Fig. 4b).

The symmetry specification and the model of the arbiter can be given to a HYPERLTL model checking tool, such as MCHYPER [31], which are typically command-line based. The provided system models are usually considered as hardware specifications that could be implemented, e.g., on a chipset. Consequently, the model checkers also consume low-level circuit representation like AIGER [9, 10], encoding the system as an And-Inverter Graph. This representation is hard to read for human developers, who often sketch the system by hand in a more visual way (Fig. 3) and realize the models using hardware description languages such as VERILOG [43], which are then compiled down to AIGER. Given such a system description and a hyperproperty, the model checker then tries to find a counterexample, i.e., a set of system

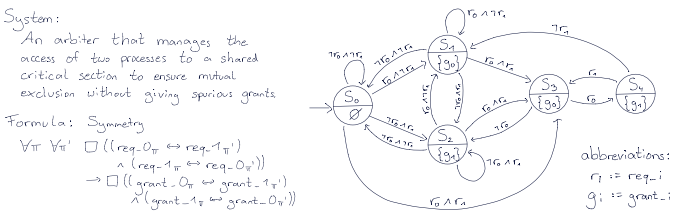

Fig. 3. Handwritten details of an arbiter system and a symmetry formula checking if a process has an unfair advantage over the other process when requesting access to a shared resource.

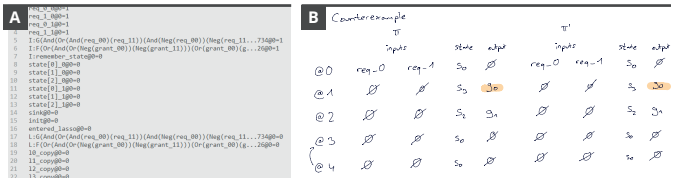

Fig. 4. (a) Instance of a counterexample produced by MCHYPER (excerpt), here for the system and formula described in Fig. 3. (b) Handwritten notation of the provided output in a table-like format. The marked outputs in the second row do not fulfil the requirements of the formula.

executions that together violate the HYPERLTL formula.

If a violation occurred, a counterexample is reported in a textual representation where each line represents variable's value on a given trace at a given time step (Fig. 4a). This representation is hard to grasp as even for smaller counterexamples, this output consists of a few hundred lines (140 lines for this arbiter example), rendering it almost impossible to quickly understand the violation. Consequently, system designers might write down the values in a table-like representation (Fig. 4b). Only then they can start to relate formula, system, and the counterexample executions with each other in order to identify and understand the violation of the specification.

2.3 Challenges

This whole process quickly becomes cumbersome and poses multiple challenges. First of all, hyperproperties can express arbitrarily complex relations across traces and time, making it hard to recognize the patterns in the executions that violate such a hyperproperty. Further, analysts need to identify which subformulas were relevant (i.e., violated) and which parts can be ignored. Any visual support for scanning the present executions and identifying the relevant elements will be highly beneficial. After identifying the violation, it is still necessary to reason about *why* the violation could occur, thus, why one execution reached a particular state. For this, the executions must be considered in the context of the provided system. Further, both formula and system can grow quickly in size. On the one hand, this makes it increasingly challenging to sketch the components adequately and recognize specific characteristics. On the other hand, the likelihood of faulty formula specifications or system definitions increases as well, leading to the need to identify these issues and to correct them. Thus, editing facilities for formula and system are of interest. With our tool, we aim to significantly improve these analysis workflows and help the system designers in their development process.

3 BACKGROUND & RELATED WORK

We first provide formal details of the model checking problem of hyperproperties. Secondly, we elaborate on the importance of visualization methods to better understand abstract models or processes by giving an overview of related work. Finally, we discuss existing work for editing formula and graph representations.

3.1 Model Checking of Hyperproperties

Model checking [23] answers the following question: Given a system description M and a specification φ, formally describing the desired property, does M satisfy φ. More specifically in the context of hyperproperties, we require that the set of executions of M satisfies the hyperproperty. For the interested reader, we will define these concepts formally in the following.

The system description M is typically provided as a finite Moore state machine, formally defined as a tuple (S, s_0, I, O, τ, l) with: S: a finite set of states; s_0: the initial state; I: the input alphabet; O: the output alphabet; $\tau : S \times I \to S$: a transition function; and $l : S \to O$: an output labeling. Figure 2, for example, depicts a finite Moore state machine with three states. The input alphabet contains variables i and s and the output alphabet contains the variables o_1 and o_2. Edges of the state machine (arrows) are labeled with the input and states (circles) are labeled with the system's output. An execution (trace) of a model M is an infinite sequence of sets of atomic propositions AP through the state machine, where $AP = I \cup O$. An example trace of the model in Figure 2 is $\{i,s\}(\{o_2\})^\omega$. In the first position of the trace (corresponding to the initial state and first input), there is no output but the input i and s. Defined by the transition function, we proceed from the initial state to state 2, where we reside indefinitely by outputting o_2 without receiving a further input. The notation $(x)^\omega$ denotes that x is repeated infinitely often. Formally, the set of all traces for a set of atomic propositions is thus $(2^{AP})^\omega$, i.e., the set of above mentioned infinite sequences over atomic propositions. The set of traces of a system model M, denoted by $Traces(M)$, is a subset of $(2^{AP})^\omega$.

Formally, a hyperproperty H is a set of sets of traces; meaning it defines all trace sets that comply to the hyperproperty. If the traces of a system model M are no element of the hyperproperty, i.e., if $Traces(M) \notin H$ then the system does not satisfy the hyperproperty. In this case, a counter example is provided by the model checker, i.e., a set of system traces that violates the hyperproperty.

The desired behavior of the system is provided in a formal specification language such as HyperLTL, a temporal logic for hyperproperties. In HyperLTL, variables are interpreted as atomic propositions which can be connected with either Boolean operators (e.g., *equivalence* $\leftrightarrow$, *implies* $\rightarrow$, *or* $\vee$) or temporal operators. The most prominent temporal operators are *globally* ($\square\varphi$, where φ must be true at all times) and *eventually* ($\diamondsuit\varphi$, meaning that φ will hold at some point in time); further operators include *until* ($\mathscr{U}$), *release* ($\mathscr{R}$), and *next* ($\bigcirc$). As an example, consider again the HYPERLTL formula from Sect. 2.1: $\forall\pi\,\forall\pi'\,\square(i_\pi \leftrightarrow i_{\pi'}) \rightarrow \square(o_\pi \leftrightarrow o_{\pi'})$. HyperLTL formulas start with a quantifier prefix introducing universally ($\forall$) or existentially ($\exists$) quantified trace variables (π and π') followed by a formula ψ in the body (here $\square(i_\pi \leftrightarrow i_{\pi'}) \rightarrow \square(o_\pi \leftrightarrow o_{\pi'})$). Within this formula, the variables are indexed with trace variables to indicate to which trace quantifier they refer to (e.g., i_π). For a formal definition of the semantics and more examples of hyperlogics, we refer the interested reader to [24, 25].

With HYPERVIS, we visualize hyperproperty counterexamples returned by a hyperproperty model checker [26, 30, 31]. We use MCHYPER [31], which builds on ABC [14]. MCHYPER takes as inputs a hardware circuit, specified in the AIGER format [9, 10], and a HYPERLTL formula. MCHYPER solves the model checking problem by computing the self-composition [5] of the system. If the system violates the HYPERLTL formula, MCHYPER returns a counterexample. This counterexample is a set of traces through the original system that together violate the HYPERLTL formula. Depending on the type of violation, this counterexample can then be used to debug the circuit or refine the specification iteratively.

3.2 Visualization and Explication of Formal Methods

In recent years, research started more intensively to investigate how complex and abstract algorithms and models can be visualized and interactively explored, and, thus, be made more transparent. Most prominently, this includes work within the area of *explainable artificial intelligence* (XAI) [40, 79, 82], but can also be extended to related fields such as formal methods [32]. For example, proof attempts have been visualized by SATVIS [34] and an improved version of the Z3 Axiom Profiler [74]. They visually represent attempts from the VAMPEXCIRE theorem prover and Z3 SMT solver, respectively, in order to support users and developers of the tools in understanding the results.

Textual Explications One instance of textual explications are automatically generated facts based on the underlying data [81]. Typically, machine learning algorithms extract facts which are then verbalized using natural language generation (NLG) [57, 65, 80]. These facts can then aid interpreting a visualization by verifying the viewer's thoughts and pointing at potentially overlooked aspects [81]. The generated facts can be shown as a single caption for a chart [78] or be provided as a collection of statements next to the visualization [28]. Applications in the areas of student-teacher communication [60], XAI [42, 77], and supporting safe handovers in cyber-physical systems [84] further indicate their practical benefits for interpreting visualizations.

Visualizing Counterexamples Visually representing counterexamples for trace properties, e.g., for LTL, is a known challenge for

which various approaches have already been proposed. Techniques, such as state diagrams [3, 36, 45, 58], sequence diagrams [16, 55, 58], and variable tables [45, 58, 68], convert the counterexample and the system model to more readable formats. The model view [16, 36, 55] takes a different route by mimicking the counterexample and providing a step-wise navigation. Further, visualization approaches for such counterexamples with single executions have been considered for various domains and applications [11, 12, 44, 69]. Additionally, the established model checker UPPAAL [55] visualizes timed automata for real-time systems, allowing for interacting with simulations of the system.

Approaches for supporting the analysis of counterexamples include minimizing [53] and explaining counterexamples [7], as well as investigating several system executions simultaneously [11, 39, 75]. Multiple works explore how individual counterexamples can be visualized and explained, e.g., for function block diagrams [44, 68], with the newest version of MODCHK [68] being highly related to HYPERVIS. MODCHK provides a causality analysis [7] which delivers an over-approximation of a set of causes. In contrast, HYPERVIS produces minimal explanations using a more efficient explanation algorithm. Further approaches to identifying the causes of a *trace property* violation [39] have been made, for instance, the EXPLAIN [38] tool, which has been incorporated into multiple model checkers [19, 20, 22].

Visualizing Parallel Executions Research on distributed systems has examined how to visualize multiple, parallel executions of a system. Two examples are ODDITY [63, 87] and SHIVIS [2, 8]. Oddity consists of an interactive visual debugger and is part of the DSLABS framework, which also includes a model checker for distributed systems. SHIVIS is a web tool that uses space-time diagrams to visualize the execution of a distributed system. Particularly these diagrams highlight the communication between components and the partial ordering between events that happen across components (the happens-before relationship). However, while related investigations have been conducted, the specific case of visualizing model checking results of hyperproperties was not considered.

3.3 Editing Formulas and Systems

The modeling of systems that fulfill certain specifications is an iterative process of checking, correcting, and refining both the specifications and the system models. Therefore, editing the specification, i.e., formula or system, are essential parts of the workflow. In the following, we present existing work providing techniques for efficiently editing them.

Formula Editing Established online tools like Wolfram Alpha [86] feature advanced text editors that facilitate writing mathematical notations. Specifically for formula editing, most interfaces provide a real-time preview of the formula, translated from the markup language used for writing the mathematical expressions. The most common markup languages for mathematical input are LaTeX [37, 76] as well as OpenMath and MathML [17, 50]. Via markup alternatives or special characters keyboards, WYSIWYG approaches can allow users without knowledge of the markup language to still write the desired mathematical expressions [54, 71]. Such visual interfaces can also support focusing on specific formula parts by collapsing selected subformulas [18, 49]. Finally, more experimental interfaces are starting to provide handwriting and speech recognition capabilities [29, 54, 85].

System Editing System models are usually edited in a hardware description language like VERILOG [43] within an integrated development environment. As an alternative to these textual representations, the systems can also be modeled through finite state machines [1, 61, 70]. These models can then be visually edited, e.g., by adding, relocating and removing nodes or edges from the node-link diagram [33, 73]. Lightweight versions of such editing are already provided within commercial tools for general diagram editing, such as Stateflow [4].

4 HYPERVIS: VISUALIZING MODEL CHECKING RESULTS

Based on the identified challenges of analyzing model checking results (Sect. 2.3), we iteratively developed HYPERVIS. In this section, we will first recap the main components of the considered counterexamples (Sect. 4.1) and outline our set design goals (Sect. 4.2). Then, as the main part, we will present our visualization design (Sect. 4.3), including its interaction concepts. This is followed by the description of the considered editing and debugging facilities (Sect. 4.4). Finally, we provide further insights into the design process as well as the actual implementation (Sect. 4.5). The tool is provided online (imld.de/hypervis).

4.1 Components of Counterexamples

Strictly speaking, a counterexample to a hyperproperty is only the set of executions that are returned by the model checking tool. However, for the remainder of this work, we depict a counterexample to comprise the formula and system provided by the analyst as well. Thus, it consists of three main components: the *system*, the *formula*, as well as the specific *executions*. In addition, we introduce *explanations* as a fourth component, indicating and explicating relevant bits of the violation.

The system describes the hardware circuit as a transition system with states providing the *outputs* and transitions implementing state changes based on *inputs*. Due to the system being a hardware circuit, states are internally represented by *latches*, i.e., sub-circuits that can preserve information. Together, all available variables, i.e, outputs, inputs, and latches, are the *atomic propositions*. The formula can be represented as a syntax tree over propositional and temporal *operators* where leaves are selected atomic propositions on a specified *trace* (or *execution*). Here, a formula typically describes relations on pairs of executions, i.e., two instances of the system. Each execution is representing values of atomic propositions for every *time step*. Notably, these executions can be infinite and contain a *lasso* (or loop), which marks subsequent time steps that are repeated infinitely.

Through a causal analysis of the counterexample, we are able to identify which atomic propositions in the formula contributed to its overall violation and are, therefore, *relevant* for the counterexample. For the explanations, we extract textual explications of the most top level *subformulas* with temporal operators. Depending on the actual top level operator, these subformulas can either be satisfied or violated (e.g., in case of an 'implies' $\rightarrow$ operator, the premise has to be satisfied while the conclusion is then violated). For an analyst, all mentioned components and elements are relevant for understanding the counterexample in general, reasoning about why it can occur, and identifying possible corrections to either the formula or system.

4.2 Design Goals

When starting the design process, we identified multiple design goals that a tool for visually analyzing hyperproperty counterexamples should fulfill. The goals DG1–DG5 describe desired visualization aspects, while DG6–DG7 outline more general tool characteristics.

DG1: Build Upon Familiar Presentations. As illustrated in Fig. 3 and Fig. 4, analysts often sketch the system or list the executions in a certain way. We aim to foster an intuitive understanding of the views by building upon these typical representations, but extending them with more effective encoding strategies.

DG2: Support Recognizing Trace Relations. In many cases, a specific combination of absent or present atomic propositions must be identified and compared across the executions. We aim to simplify such pattern recognition within the executions.

DG3: Relating Components. A major challenge for analysts is relating the different components to each other, e.g., mapping back atomic propositions in the execution to corresponding subformulas or to taken transitions in the system. Thus, the tool should support the analyst in mentally linking elements across views.

DG4: Provide Guidance for Identifying Violations. Counterexamples can quickly become overwhelming, with a multitude of variables or time steps being involved. Our goal is to support analysts in identifying the relevant elements that led to the violation and, thus, in understanding the model checking result.

DG5: Enable Editing of Formula and System. Due to their complexity, formulas and systems can easily contain small but hard to recognize bugs leading to a counterexample. For this, the tool should provide integrated functionalities for fixing such issues.

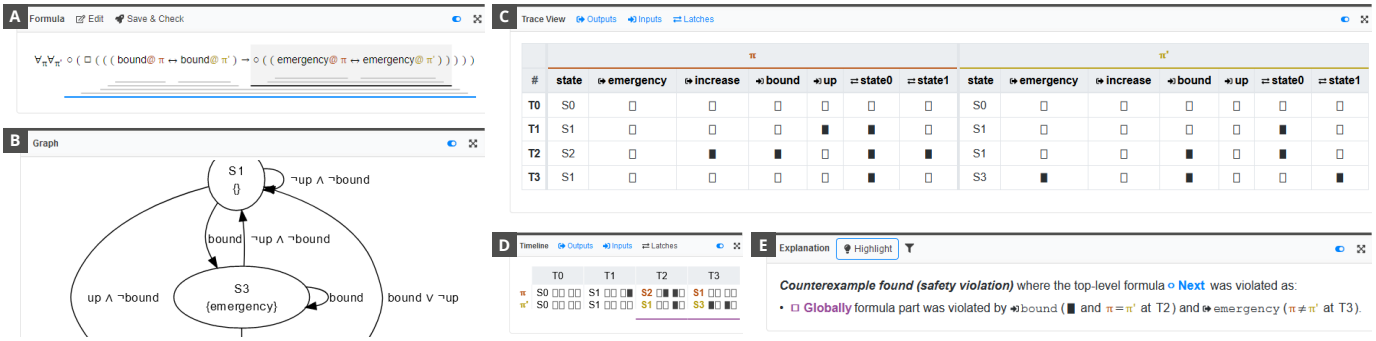

Fig. 5. Views of HyperVis: (a) formula view, here with hover of a subformula; (b) graph view showing the system as a Moore transition system, here zoomed in; (c) trace view providing for both executions π and π' the values of all atomic propositions across all time steps; (d) timeline view showing the executions in a compact format; and (e) explanation view with textual statements on the counterexample, here with one relevant subformula.

DG6: Provide a Holistic Interface. Model checking of hyperproperties is a multi-step process; from providing the input, to analyzing the counterexample, to iterating the specification or system. Thus, a tool should consolidate these steps within one interface.

DG7: Avoid Setup Efforts. The tools used for model checking are often command-line based and implemented with different dependencies. We aim at avoiding the setup effort for the analyst and providing a unifying ready-to-use tool.

In the following, we detail how we addressed these design goals within the visualization design of HyperVis' views.

4.3 Visualization Design

Guided by the described design goals, we developed HyperVis and its general interface including five visualization views. The focus in this section lies on how we visualize the counterexample components specifically as well as efficiently guide the analysis.

4.3.1 Visualizing a Counterexample: Provided Views

For HyperVis, we developed five different views; the formula view, graph view, trace view, timeline view, and explanation view. In the following, their design is detailed.

Formula View The HyperLTL formula provided by the user is transformed into a representation using the actual logical and temporal operator symbols (DG1). Internally, the formula is in a hierarchical structure; this structure is indicated with bars below the formula string. The bars allow emphasizing the different subformula levels, with the uppermost bars representing the atomic propositions and the lowermost bar (marked in blue) the entire formula. Hovering over the bars emphasizes the corresponding subformula (Fig. 5a), simplifying recognizing the formula structure and corresponding bracket. The stated atomic propositions always relate to one specific trace. To simplify distinguishing which proposition corresponds to which trace, we introduced fixed colors for the traces and added labels to the proposition, i.e., either $@\pi$ or $@\pi'$. These trace colors are re-used in all views.

Graph View The system is visualized as a Moore transition system, i.e., a graph with the states as nodes and transitions as edges (Fig. 5b). Following their convention, the set of present outputs on a given state is printed into the node label, e.g., $\{emergency\}$ in state S3. If an output is absent, its value is *false*. Further, we show symbolic transitions, i.e., edges can be labeled with formulas expressing specific input combinations, such as logical conjunction (e.g., $up \wedge \neg bound$). The graph can be freely zoomed and panned.

Trace View As previously described, analysts typically transform the textual output of the model checker into a table-like format, thus creating an overview of all atomic propositions and their values on the traces across time steps. Our trace view builds upon that (DG1) and prints the atomic propositions per trace as columns and the time steps as rows (Fig. 5c). The values themselves are binary, thus are either *true* when a variable was present or *false* when it was absent. We propose to replace the common notation of the values as 1 and 0 with a graphical representation: a filled rectangle ■ represents present variables and a hollow rectangle □ absent variables. This representation simplifies recognizing patterns of occurring values in and across traces (DG2).

Small icons before the proposition name indicate its type, i.e., either output, input, or latch. The propositions are sorted first by type and then alphabetically. Controls in the view head allow for hiding an proposition type. In addition to the atomic propositions, we also show a numbered state indicator (e.g., S3) in the first column of each trace. These state indicators are abstractions of the latches, which together encode the current state. The time steps are labeled as T0, T1, and so forth. Further, if a lasso (see Sect. 4.1) is present in the counterexample, it is indicated with gray borders at the respective time steps.

Timeline View So far, the view design was influenced by common ways to write down the counterexample. However, the timeline view is a new visualization that aims to provide a more compact representation of the executions (Fig. 5d). Similarly to the trace view, it shows the specific values of the atomic propositions, but with the time mapped horizontally. By omitting the atomic proposition labels, the rectangles indicating present or absent variables are placed next to each other. This allows for a further improved pattern recognition, either across traces or across time steps (DG2). For example, considering a set of four variables, it is easily possible to observe differences or similarities across instances: □■■□ and □□■□ differ only in the second variable. The label of the represented proposition can be accessed by hovering over the rectangle; their order is equal to the order in the trace view.

In an earlier iteration, the view was intended to emphasize diverging behaviors of the executions for one variable, e.g., showing when they were in different states or read different values for an atomic proposition. We opted to develop the view further into its more compact format while also showing the atomic propositions. To still indicate diverging executions, the state indicator (e.g., S0) is colored black if both executions are in the same state and colored according to the trace color when they diverge (e.g., S2 and S1). Finally, in the case of a present lasso, an indicator at the time steps is provided.

Explanation View The explanation view shows a verbal summary of the counterexample alongside statements on the most top-level subformulas relevant to the found violation (Fig. 5e). The basis of this is an automated causal analysis of the counterexample, which extracts a minimal set of subformulas that contributed to the overall violation at one or more time steps. With this information, we can relate the subformulas to specific values at specific time steps and derive a textual statement. The statements' structure is always the same: first, the temporal operator of the subformula is stated, followed by a list of involved atomic propositions. For each proposition, it is indicated at which time step it became relevant, how the values relate to each other across traces and whether the values were always *true* or *false*. This information is provided as inline or word-sized representations [6, 35], seamlessly integrating into the textual description. Further, each statement is assigned a unique color, allowing for indicating it in other views (DG3). For example, as visible in Fig. 5d, the timeline view shows bars at the bottom, hinting at which time steps a subformula was relevant.

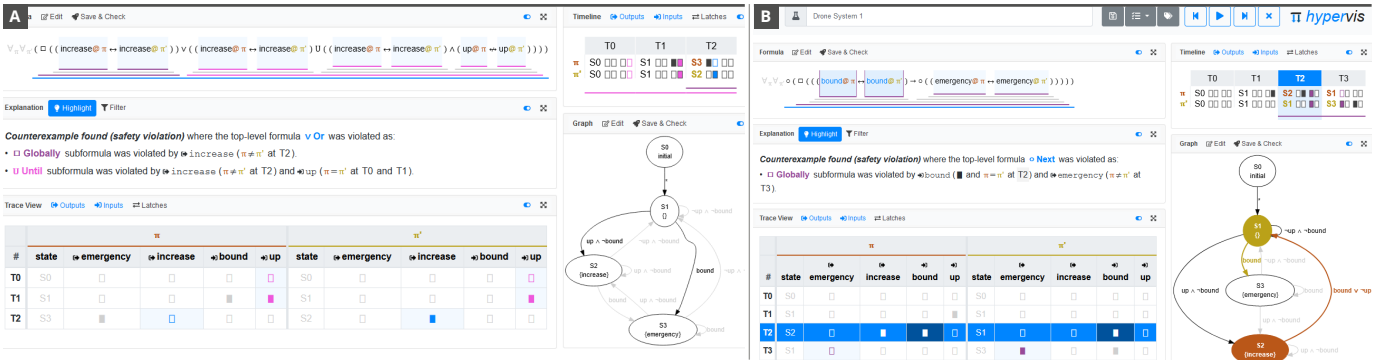

Fig. 6. The interactive analysis of counterexamples is at the core of HyperVis: (a) activated explanation highlighting points analysts to relevant elements; (b) stepping through time steps fosters the understanding of the executions' behavior (controlled by the buttons in the top right corner).

A complete statement is shown in Fig. 5e. For *bound* the value was *true* (■) and equivalent on both traces ($\pi{=}\pi'$) at T2, while *emergency* was unequal on both traces ($\pi{\neq}\pi'$) at T3. These statements can help analysts to quickly grasp the essential aspects of the violation and locate the time steps and atomic propositions of interest. Currently, the provided explanations for the atomic propositions can indicate found equivalences of traces across n time steps as well as consistent values across time steps (DG2). However, as HyperLTL formulas can describe arbitrary relations of atomic propositions across traces, not all relevant patterns are currently recognized and expressed. Similarly, we only provide statements for subformulas being present at the top two levels; thus, more nested formulas might not be verbalized adequately.

Interface Arrangement By default, the views are arranged in a simple 2-column grid, with formula, explanation, and trace view being placed in a wider column on the left, and timeline and graph view on the right (Fig. 1). However, as the space requirement of the views can heavily vary between counterexamples, the interface also supports arranging the views differently. For example, if a formula is becoming rather long, it is placed in full width on top. Similarly, if counterexamples involve many time steps, trace view and timeline view are devoted more space. In general, the goal is to provide all views within the initial viewport and avoid scrolling as much as possible. Finally, analysts can also manually collapse or maximize views.

4.3.2 Analyzing a Counterexample: Interactive Guidance

For analyzing the counterexample, we provide further interactive mechanisms fostering the comprehension of the counterexample's specifics. These mechanisms include an explicit highlighting of relevant elements, linked highlighting across views, as well as a debugger-like stepping through the counterexample. These dynamic functionalities of HyperVis are also shown and explained in the accompanying video.

Highlighting Relevant Elements As stated in the context of the explanation view, we are identifying the subformulas that contributed to the overall violation at specific time steps. This knowledge is not only used for the explanation statements but also to indicate the relevant elements across all views (DG4). To activate this indication, the explanation view features a 'Highlight' toggle button (Fig. 5e). Upon activation, as in Fig. 6a and Fig. 1, the non-relevant subformulas in the formula view are grayed out, as are the non-taken states and transitions of the executions in the graph view. Similarly, non-relevant values in the trace view and timeline view are shown less opaque while relevant values are emphasized. Further, a filter button next to the highlight button allows for removing non-relevant elements from the views.

We also relate the relevant elements to the provided statements in the explanation view (DG3). Specifically, we identify which atomic proposition is part of which statement, i.e., in which subformula it occurs. Further, we propose to use the statements' assigned color for highlighting: the rectangle representing binary values are colored accordingly, as are the bars indicating the atomic proposition.

Linked Highlighting In general, all views react to hovering over displayed elements, e.g., subformulas, states, or time steps. Hovering also results in a linked highlighting across views [15, 72], i.e., the corresponding elements in other views are also highlighted (DG3). Only in a few cases, elements are shown in exactly the same way in other views. For example, subformulas in the formula view may occur again the explanation view. As in most cases elements appear slightly differently, e.g, formula view and timeline view show atomic propositions differently, the correspondence is not immediately apparent and is then indicated through the linked highlighting (DG3).

Specifically, hovering over a trace indicator (i.e., π or π') in either view highlights the execution in the graph view, i.e., all taken transitions and states are colored in the trace color. Vice versa, hovering a state or transition highlights instances in the executions where this state and the inputs of the transition were present. Hovering over a time step highlights the corresponding row or column in trace and timeline view, the relevant subformulas at this time step, and the executions' current states and transitions taken next. The atomic propositions in formula and explanation view allow for highlighting the corresponding labels in the trace view and, if applicable, the specific values that were relevant at certain time steps in both trace and timeline view (DG4).

Stepping through a Counterexample It is important to understand the sequence of events that lead to the violation. Therefore, we enable stepping through the counterexample in a debugger-like fashion (Fig. 6b). Through control buttons provided in the interface header, the analyst can move forward and backward. For the current step, a stronger visual highlight is used (Fig. 6b), with the time step colored in blue and relevant subformulas further emphasized. For the graph view, we color the states and transitions in the respective trace colors; if they share the same state or transition, blue is used. The same effect can also be achieved by selecting a time step in the trace view or timeline view. Further, when stepping through, the highlight is permanent and can be used in combination with the highlighting of relevant elements as well as the linked highlighting triggered on hover.

4.4 Tool Functionalities & Editing Facilities

Following DG6, we provide one unified interface that allows for performing model checking, analyzing the counterexamples, and iterating specification and system within it. In the following, we describe the tool functionalities of HyperVis as well as its editing facilities.

Tool Functionalities We extended HyperVis with tool functionalities allowing for using it in a productive way for many model checking projects simultaneously. Among others, this includes functionalities for (re-)loading projects, re-running model checks, or managing different versions of them. The project manager provides access to all model checking projects, i.e., loaded systems and specifications checked, in a sidebar widget. The different projects can have multiple versions (here, marked with the latest modification timestamp), helping analysts to quickly jump to older iterations. For each version, multiple checks can be created, i.e., multiple hyperproperties that a system should fulfill. The versions of the projects can be manually created, but are also automatically introduced when editing a formula or system. In case of faulty edits that result in an error thrown by the model checker, a

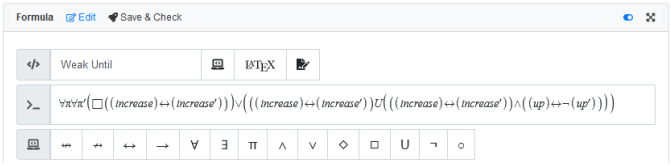

Fig. 7. Within the formula view, the formula editor can be toggled, providing a WYSIWYG inline LaTeX editor.

rollback to the previous version is offered. Each version of the project can be freely tagged and each checked hyperproperty can be named.

Editing of the Specification HyperVis provides a formula editor that uses LaTeX markup to edit the hyperproperty specifications (Fig. 7). Specifically, the analyst can directly type LaTeX markup, which is automatically rendered inline. By using LaTeX, copying and pasting formulas edited in common tools outside of HyperVis becomes directly possible. Similarly, formulas could also be sketched by hand via pen input. However, this is currently only supported by external tools like MyScript [64]. Our editor also features an extended keyboard, providing access to the common logical and temporal operators. As mentioned before, HyperVis supports the editing of multiple separate formulas for a single system, potentially allowing to split up complex hyperproperties or to test very different specifications.

Editing of the System For editing the system model, changes can be made either through visual editing or by changing the original text input. The visual editing could involve providing a special mode allowing for, e.g., drawing edges, rerouting existing ones, or creating nodes. At the same time, some users might still prefer to directly textually edit the originally provided Verilog definition. However, as providing such editing supporting is not straightforward, HyperVis currently only features a mock-up editing of the system. Specifically, the challenge comes with the representation of the system as a hardware circuit in the AIGER format [10]. These AIGER files are automatically generated from definitions implemented in Verilog and are hardly readable by humans. For displaying the graph view, a DOT [52] representation is generated from AIGER, however, the transformation comes with information loss and is therefore not invertible. To sidestep this, an intermediate format for automatons could be used, which can be transformed from or to AIGER without information loss and also more easily changed in programmatic way. For now, we allow the editing of the DOT notation, which updates the shown graph view to illustrate the intended functionality but cannot trigger an updated model check.

4.5 Design Process & Implementation

In order to develop HyperVis, we followed an iterative design process within an interdisciplinary team. This team consists of formal methods researchers on the one hand and HCI as well as visualization researchers on the other hand. While not end-users, the first group are domain experts for model checking of hyperproperties, knowing the challenges and main goals. In the following, we detail the design phases as well as the current implementation of HyperVis.

4.5.1 Design Phases

The first phase involved introducing the visualization researchers to the domain of model checking and hyperproperties in order to establish a common understanding of the current processes and present challenges. Afterward, we jointly developed a first click prototype illustrating a possible interface visualizing the found counterexamples. Then, the first implementations of the visualizations were realized in a web prototype alongside a parser consuming the output file of the model checking generated by MCHyper. Within this process, it became apparent that plainly representing the counterexample will be insufficient and that it will be essential to become able to extract the relevant bits of the violation and presenting it to the analyst. At this stage, a first version of the causal analysis algorithm was developed alongside early highlighting mechanisms. This enabled testing various case studies, and thus incorporating further improvements into the visualization design.

On the tool side, we started to develop approaches for editing the formula and system as well as the general structure of the tool interface, e.g., providing access to the project list, their versions, or loading new ones. With these tool aspects implemented, we ran a first feedback session with 3 participants and collected comments on the interface. This feedback allowed us to iterate, e.g., the menu structures, button icons and labels, or features of the inline formula editor. The result of the overall design process is the current version of HyperVis.

4.5.2 Implementation as Web Tool

HyperVis is implemented as a web-based tool, featuring a Node.js-based [67] backend and JavaScript-based frontend. In the frontend, the views are implemented with plain HTML or SVG. Except for the explanation view, the rendering of all views is controlled by D3.js [13]. To support the linked highlighting, custom events were introduced that are sent and consumed by the views. For the formula editing, we incorporate the MathQuill library [76]. For translating the formula and producing the polish notation that MCHyper requires, we use Spot. Graph editing is not fully functionally implemented yet. To illustrate the general possibility, we provide an embedded CodeMirror [41] editor to change the DOT representation of the system.

The backend is responsible for managing the model checking pipeline. Based on the analyst's inputs, it calls the MCHyper Python tool before handing over the found counterexample to our own Python script extracting the relevant subformulas. It computes a minimal set of variable and time step pairs, which cause the violation. In addition, this script also writes all required information into a JSON file. In parallel, a separate script is used to generate the DOT representation based on the AIGER file. As for larger systems this generation might not terminate in a reasonable time, for some counterexamples the graph is not available. After parsing these generated outputs in the Node.JS server, the data is provided to the frontend. For each project, the results are stored in a folder structure, allowing to quickly reload the counterexamples later on and implement the versioning concept. The communication between the frontend and backend is based on HTTP requests.

The HyperVis tool is hosted online but can also be locally run, either as a Docker container or by fully installing it and all its dependencies. In general, we envision the usage as an online tool as the primary usage style, which then also allows for avoiding setup efforts (DG7).

5 Validating HyperVis

In this section, we validate HyperVis by discussing multiple case studies and reporting on user feedback sessions. Both illustrate that HyperVis indeed advances the state-of-the-art significantly by helping to quickly identify the violations in counterexamples of hyperproperties.

5.1 Case Studies

Here, we detail two selected case studies: In the first one, we visualize the results of model checking information-flow properties on an open source implementation of the I2C bus protocol. In the second case study, we take a look at one of the core building blocks of such bus implementations: mutual exclusion protocols.

CS1: I2C Bus Protocol The I2C bus protocol coordinates the communication between multiple components in a master-slave hierarchy and is widely used in practice. As it has no security features, this has led to exploits, for example, in smart cards of German public health insurance companies [83]. The implementation used in this case study is taken from OpenCores [66]. Its AIGER circuit consists of 254 latches plus 86 input and output variables. Typically, this protocol consists of a master, one controller, and several slaves, where the master communicates to the slaves while the controller ensures properties like mutual exclusion. Suppose information has to be sent over the bus, the master addresses the slaves with a designated address bit. In this case study, we visualized the result of model checking the following information-flow policy: The information *which* slave the master is addressing should not be identifiable from the bus' output. This property is violated, but the counterexample is highly complex (e.g., it is not possible to generate a state graph). Still, the visualizations provided by HyperVis help to understand the violation.

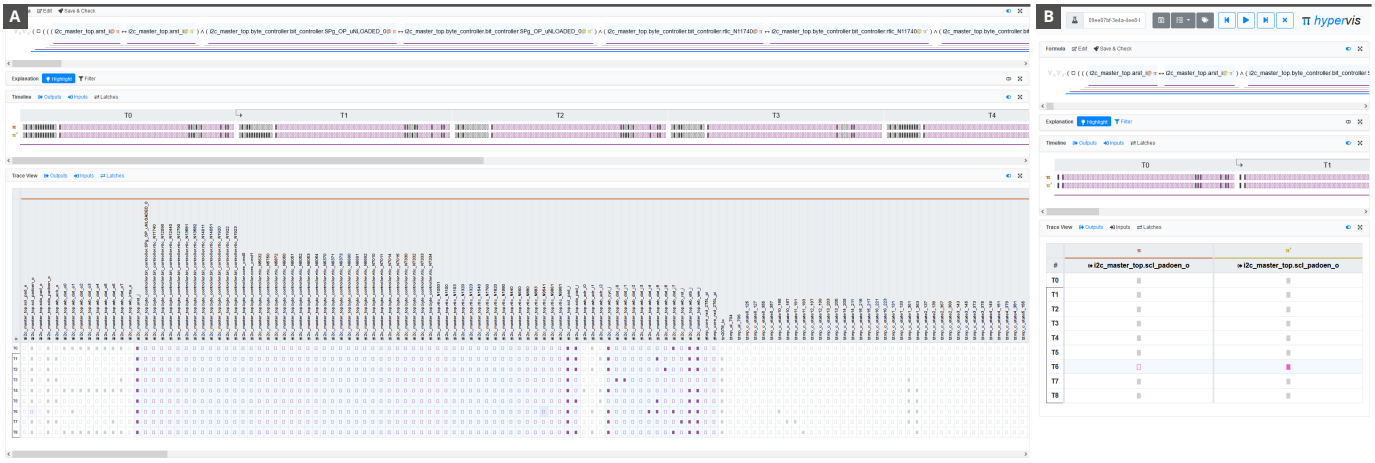

Fig. 8. (a) Excerpt of the interface for the I2C case study with activated explanation highlighting, where the trace view can only show a fraction of the 680 columns (see scroll bar); (b) with the explanation filtering activated as well as latches and inputs disabled, the trace view can be limited to the one relevant output variable. In both views, the explanation view has been collapsed; the full view is provided in the supplement.

The analyst benefits especially from the highlighting and filtering functionalities. The trace view of HyperVis is shown in Fig. 8. Fig. 8a shows parts of the interface with highlighting activated. While this already helps to see relevant elements, it is still nearly impossible to grasp the violation as large parts are outside of the browser's viewport. By enabling the filtering and hiding variable types currently not of interest, see Fig. 8b, HyperVis can yield the proposition and the time step responsible for the violation and support the user in limiting the information to a reasonable amount. Then, we can see that one output diverged at time step T6, resulting in a counterexample.

CS2: Symmetric Mutual Exclusion Arbiters form the basic building blocks in many protocols, such as the AMBA protocol or the above-mentioned I2C protocol. Ensuring that no process or slave has an unfair advantage is highly desirable, also referred to as *symmetry* in protocols. In this case study, we visualize the results of checking if an arbiter implementation satisfies symmetry (cf. the arbiter system from Sect. 2.2). Specifically, we check whether for two executions with symmetrically arriving requests the grants are also given symmetrically.

This case study features the interplay between the views implemented in HyperVis. The explanation view directly tells the analyst why the overall formula is violated. When the highlighting button is pressed, HyperVis pinpoints the atomic propositions, time steps, and subformulas that caused the violation in the trace view, the timeline view, and the formula view (Fig. 1). In the formula view, for example, the subformula $grant\_0@\pi \leftrightarrow grant\_1@\pi'$ is highlighted in the conclusion because only this subformula is needed to understand why the symmetry specification is violated: In the counterexample, $grant\_0@\pi$ holds at time step 1 while $grant\_1@\pi'$ does not hold at that time step. Highlighting relevant subformulas decreases the number of subformulas that the analyst needs to consider when trying to understand the counterexample. This illustrates that HyperVis fulfills DG4, providing guidance for understanding the formula violation.

After the violation is identified, the bug in the system needs to be found. Since DG3 is supported through the linked highlighting of elements and the highlight button, the graph view is restricted to the relevant states for the counterexample executions. This feature again allows the analyst to focus their attention on the most relevant aspects. By using HyperVis to explicitly step through the time steps, one observes that both executions represent the same system trace, thereby violating the symmetry in the grants. The solution to achieve symmetry is: Adding a new input to the system that allows giving grants symmetrically when both processes send requests simultaneously [59].

5.2 User Feedback Sessions

We conducted feedback sessions with domain users to better assess the merit of our tool for them. In the following, we first describe the study design before reporting on the received feedback.

5.2.1 Study Design

Participants We recruited six participants (age M=27.5 yrs, SD=3.33 yrs; 1 female, 5 male) that have significant knowledge on model checking and hyperproperties. On average, participants rated their theoretical expertise on model checking with 4.5 out 5 and on hyperproperties with 4.0.

Apparatus The sessions were conducted remotely through a video call with screen sharing. We hosted the latest version of the tool online and provided participants with the link. Two investigators moderated the videotaped sessions, and a third one was taking notes. Participants were asked to follow a think-aloud protocol, i.e., verbally phrasing their thoughts and actions while interacting with the tool.

Procedure After a short welcome and general introduction, participants were asked to provide consent for the video recording. Then, we outlined the procedure and think-aloud protocol before starting the demonstration of HyperVis via screen sharing, introducing all views and their functionalities. Afterward, participants were asked to open the tool, start screen sharing from their end, and analyze three provided examples (detailed below). For each example, we provided a short introduction on the specification and system and then asked them to reason about the counterexample. Further, for the first example, participants had to propose a fix for a corrupt system, while in the second, they had to edit the formula to a working version. For the last example, they had to identify the "needle in the haystack". After working on each example, we asked them to reflect on the interface and which views they found helpful in the specific context. Lastly, we concluded the session with an open discussion and provided them with a link to our questionnaire. Sessions lasted one hour on average.

Provided Examples For the demonstration of HyperVis, we used an arbiter example similar to the one described in CS2. Further, we prepared three examples for the hands-on part: The first two consider a straightforward drone system that is supposed to increase the drone's height when it reads an *up* input, and to go into an emergency state when a *bound* input is read. In the first version, the specification stated that equal *bound* inputs must result in equal *emergency* outputs in the next time step; however, the specification was violated due to an incorrect transition in the system. Participants had to identify this issue and verbally provide a fix. The counterexample is visible in Fig. 5 and Fig. 6b. In the second version, the fixed system was used, but now with a different but incorrect formula. Participants had to pinpoint this issue and, this time, edit the formula in HyperVis. The counterexample is shown in Fig. 6a. Lastly, the third example was a larger counterexample involving 29 time steps and 50 atomic propositions, where a mutual exclusion specification was violated. Due to the system's size, the graph view was not available. We asked participants to describe the violation in their own words and did not inquire any fixes.

5.2.2 Results

Overall, all participants (P1–P6) were able to work with HYPERVIS without larger issues and considered the tool to be useful for experts and novices. Our two main insights are: 1) Our proposed interface allows to quickly identify the violations in counterexamples and provides valuable guidance for understanding the underlying issue. 2) Personal preferences and the different analysis workflows influence how the different views were used by the participants.

All participants were able to correctly identify the violations and issues in the provided examples within the given time. We could observe that the *trace view* was used as a central component within the analysis process, providing detailed information on the executions, while the *linked highlighting* allowed for seeing the corresponding information in the other views. For the *formula view*, all participants (P1–P6) explicitly stated that they are intrigued by the hierarchy indicators below the formula. Similarly, the *graph view* proved to be of great value, particularly when the traces were highlighted while stepping through the counterexample. The comments and ratings also showed that the used *colors for traces and statements* were appreciated for relating the different components. Finally, the *explanation highlighting* was considered "extremely important" (P3) for understanding the counterexample and identifying the relevant pieces for the violation (all Ps).

At the same time, not all views were used to the same extent. For example, while most participants (P2–P6) found the *textual explanations* "extremely good" (P6) and used it as starting point for understanding the violation, participant P1 preferred working with the other views. Similarly, while participants P2+P5 only briefly used the *stepping through mechanisms*, the others found it very helpful and used it extensively. The *timeline view* was intensively utilized by participants P2 and P4–P6, particularly for comparing traces and recognizing specific patterns. At the same time, P1 used the trace view more extensively, while P3 used the timeline view only for the larger example.

While working with HYPERVIS, participants also provided multiple suggestions for various improvements. One commonly stated shortcoming was the missing graph view for more extensive examples. Further, P5+6 would prefer some indication of the explanation statements in the graph view as well. As participant P4 intensively used the stepping through functionalities, he proposed to improve the coloring of nodes and edges in the graph view when both executions are overlapping. Participants P4+6 suggested activating the explanation highlight on default. Some extended filter options were also proposed, e.g., P5 suggested the filtering of single atomic propositions, while P2 proposed to allow for hiding time steps. For the formula editing, multiple possible improvements were stated, e.g., better highlighting of corresponding brackets (P5+P6), a semantic check (P4+P5), or separating the LaTeX input and rendered formula (P2). Still, the formula editor was appreciated, with P4 stating that it is "something that we needed for a long time".

6 DISCUSSION

The positive feedback that we received emphasizes that there is a clear need for visualization and analysis interfaces within the formal methods domain. We found that the most important aspect when working on visualization solutions within this space is to have access to the specific knowledge that is involved in the rather abstract and formalized concepts. From a visualization perspective, the incorporated encoding strategies or interactive mechanisms are mostly already known. However, when applied and combined in the right way, they become extremely helpful. Importantly, as it was also commented in our study, such a solution is not only an improvement for domain experts, but can also support novices in understanding the underlying principles.

Consequently, our work is in line with other efforts of providing explications and intuition for abstract or black-box-like processes [32, 82, 84]. However, in this area, work around explainable AI [40] has received most attention in recent years, while formal methods themselves are only rarely considered. This is particularly interesting for two reasons: (1) formal methods, and especially model checking, are largely built around mathematical and logical representations that are consumed in command-line tools, while visual representations remain underestimated. Therefore, there is a big potential for making the concepts more accessible by using visualizations. Further, the mathematical nature of it requires a rigorous treatment of the visualized elements, which poses special challenges to the visualization design. (2) Formal methods also play an important role for AI in general and when trying to provide explanations to computations of an AI agent. For example, Marabou [47] is a recently introduced framework for verifying and providing counterexamples of properties of deep neural networks (e.g., robustness, which is in general a hyperproperty). However, as it is command-line based and does not provide an explanation on the counterexample, users have to cope with the same problems described in this work. The here presented visualization approaches might be directly applicable to many tools in the area of formal methods.

In the light of these considerations, HYPERVIS can be seen as a first foundation for explaining hyperproperties and counterexamples. As the immediate next steps, the suggested improvements from the user feedback sessions can be incorporated. For the editing facilities, this includes the general possibility for modifying the system plus a visual editing mode. This could also allow for providing a stand-alone editing mode with improved live previews of formula and system. For the analysis of counterexamples themselves, an interesting addition would be support for adding annotations or storing derived insights [62]. In this context, it can also be considered to automatically track the analysis history or provenance [88] and allow analysts to review it.

Currently, HYPERVIS is focused on visualizing a specific counterexample to a hyperproperty. However, on the one hand, the need for visually representing hyperproperties can also occur independently from violated specifications, i.e., for correctly implemented systems and specified properties. While it is always possible to generate a so-called witness for a proved hyperproperty by negating the specification, the found witness is one of many possible ones and might not adequately represent the underlying hyperproperty. On the other hand, the challenging but promising interactive synthesis problem potentially benefits from the presented visualizations. Synthesis constructs per definition a correct implementation directly from the provided specification, making the model checking process superfluous. Here, it would be beneficial to visualize the iterative synthesis process (i.e., how the system was derived) as well as the proposed implementation itself. The visualization and interaction designs presented here can guide the development of such novel hyperproperty visualization tools.

7 CONCLUSION

In this paper, we presented concepts for visually analyzing counterexamples to hyperproperties as well as for editing the provided formula and system. As demonstrated through case studies and attested by user feedback, our HYPERVIS tool notably improves the analysis and understanding of the counterexamples. At the core of this is the targeted usage of encoding strategies and interactive mechanisms that pointedly represent the different aspects and help to guide the analyst to the relevant information in the example. In particular, the right combination of allegedly simple measures, such as color encoding, linked highlighting, and relevance indication, can allow experts to quickly recognize the violation cause and also novices to understand the complex relations in the first place. Notably, the key to such solutions is the understanding of the domain, which in this case enabled us to embed the causal analysis of the counterexample and to automatically derive textual explanations and corresponding highlights. The provided editing facilities support fixing the identified issues, turning HYPERVIS into a valuable tool for analyzing hyperproperties. With this, we contribute a foundation for explaining and visualizing hyperproperties in general and hope to inspire further visualization solutions for more formal methods concepts.

ACKNOWLEDGMENTS

We thank Weizhou Luo for his valuable support during the overall project duration. This work was funded by DFG grant 389792660 as part of TRR 248 – CPEC, by the DFG as part of the Germany's Excellence Strategy EXC 2050/1 - Project ID 390696704 - Cluster of Excellence "*Centre for Tactile Internet*" (CeTI) of TU Dresden, by the European Research Council (ERC) Grant OSARES (No. 683300), and by the German Israeli Foundation (GIF) Grant No. I-1513-407./2019.

REFERENCES

[1] A. T. Abdel-Hamid, M. Zaki, and S. Tahar. A tool converting finite state machine to vhdl. In *Canadian Conference on Electrical and Computer Engineering 2004*, pp. 1907–1910, 2004. doi: 10.1109/CCECE.2004.1347584

[2] J. Abrahamson, I. Beschastnikh, Y. Brun, and M. D. Ernst. Shedding light on distributed system executions. In *Companion Proc. International Conference on Software Engineering*, pp. 598–599. ACM, New York, NY, USA, 2014. doi: 10.1145/2591062.2591134

[3] H. Aljazzar and S. Leue. Debugging of dependability models using interactive visualization of counterexamples. In *International Conference on Quantitative Evaluation of Systems*, pp. 189–198. IEEE, Piscataway, NJ, USA, 2008. doi: 10.1109/qest.2008.40

[4] A. Angermann, M. Beuschel, M. Rau, and U. Wohlfarth. *MATLAB – Simulink – Stateflow*. De Gruyter Oldenbourg, 2020. doi: doi:10.1515/9783110636420

[5] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, Oct. 2011. doi: 10.1017/s0960129511000193

[6] F. Beck and D. Weiskopf. Word-sized graphics for scientific texts. *IEEE Trans. Visualization and Computer Graphics*, 23(6):1576–1587, June 2017. doi: 10.1109/tvcg.2017.2674958

[7] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. In *Computer Aided Verification*, pp. 94–108. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-642-02658-4_11

[8] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst. Visualizing distributed system executions. *ACM Transactions on Software Engineering and Methodology*, 29(2):9:1–9:38, Apr. 2020. doi: 10.1145/3375633

[9] A. Biere. The AIGER And-Inverter Graph (AIG) format version 20071012. Technical Report Report 07/1, Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria, 2007.

[10] A. Biere, K. Heljanko, and S. Wieringa. Aiger 1.9 and beyond. Technical report, Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria, 2011.

[11] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels. Paths to property violation: A structural approach for analyzing counter-examples. In *IEEE International Symposium on High Assurance Systems Engineering*, pp. 74–83. IEEE, Piscataway, NJ, USA, 2010. doi: 10.1109/hase.2010.15

[12] M. L. Bolton and E. J. Bass. Using task analytic models to visualize model checker counterexamples. In *Intl. Conference on Systems, Man and Cybernetics*, pp. 2069–2074. IEEE, 2010. doi: 10.1109/icsmc.2010.5641711

[13] M. Bostock, V. Ogievetsky, and J. Heer. D^3: Data-driven documents. *IEEE Trans. Visualization and Computer Graphics*, 17(12):2301–2309, Dec. 2011. doi: 10.1109/TVCG.2011.185

[14] R. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *Computer Aided Verification*, pp. 24–40. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-14295-6_5

[15] A. Buja, J. McDonald, J. Michalak, and W. Stuetzle. Interactive data visualization using focusing and linking. In *Proc. Visualization '91*, pp. 156–163. IEEE, Piscataway, NJ, USA, 1991. doi: 10.1109/visual.1991.175794

[16] A. Campetelli, F. Hölzl, and P. Neubeck. User-friendly model checking integration in model-based development. In *International Conference on Computer Applications in Industry and Engineering*, 2011.

[17] D. Carlisle, P. D. F. Ion, and R. R. Miner. *Mathematical Markup Language (MathML) Version 3.0 2nd Edition*. W3C, 2014.

[18] D. Cervone, P. Krautzberger, and V. Sorge. Towards meaningful visual abstraction of mathematical notation. *Proc. CICM*, 2015.

[19] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004. doi: 10.1109/tse.2004.22

[20] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *Proc. Intl. Symposium on Foundations of Software Engineering*, pp. 73—82. ACM, New York, NY, USA, 2004. doi: 10.1145/1029894.1029908

[21] X. Chen, W. Zeng, Y. Lin, H. M. Al-maneea, J. Roberts, and R. Chang. Composition and configuration patterns in multiple-view visualizations. *IEEE Trans. Visualization and Computer Graphics*, 27(2):1514–1524, Feb. 2021. doi: 10.1109/tvcg.2020.3030338

[22] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-24730-2_15

[23] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 2000.

[24] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In *Lecture Notes in Computer Science*, pp. 265–284. Springer Berlin Heidelberg, 2014. doi: 10.1007/978-3-642-54792-8_15

[25] N. Coenen, B. Finkbeiner, C. Hahn, and J. Hofmann. The hierarchy of hyperlogics. In *Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, June 2019. doi: 10.1109/lics.2019.8785713

[26] N. Coenen, B. Finkbeiner, C. Sánchez, and L. Tentrup. Verifying hyperliveness. In *Intl. Conference on Computer Aided Verification*, pp. 121–139. Springer, 2019. doi: 10.1007/978-3-030-25540-4_7

[27] B. Cook, K. Khazem, D. Kroening, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Model checking boot code from AWS data centers. In *International Conference on Computer Aided Verification*, pp. 467–486. Springer, 2018. doi: 10.1007/978-3-319-96142-2_28

[28] Z. Cui, S. K. Badam, M. A. Yalçin, and N. Elmqvist. DataSite: Proactive visual data exploration with computation of insight-based recommendations. *Information Visualization*, 18(2):251–267, 2019. doi: 10.1177/1473871618806555

[29] H. Dai Nguyen, A. D. Le, and M. Nakagawa. Deep neural networks for recognizing online handwritten mathematical symbols. In *IAPR Asian Conference on Pattern Recognition*, pp. 121–125, 2015. doi: 10.1109/ACPR.2015.7486478

[30] B. Finkbeiner, C. Hahn, and H. Torfah. Model checking quantitative hyperproperties. In *Intl. Conference on Computer Aided Verification*, pp. 144–163. Springer, 2018. doi: 10.1007/978-3-319-96145-3_8

[31] B. Finkbeiner, M. N. Rabe, and C. Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Computer Aided Verification*, Lecture Notes in Computer Science, pp. 30–48. Springer International Publishing, 2015. doi: 10.1007/978-3-319-21690-4_3

[32] T. Flemisch, R. Langner, C. Alrabbaa, and R. Dachselt. Towards designing a tool for understanding proofs in ontologies through combined node-link diagrams. In *International Workshop on Visualization and Interaction for Ontologies and Linked Data*, Nov. 2020.

[33] M. Frisch. *Visualization and Interaction Techniques for Node-Link Diagram Editing and Exploration*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, München, 6 2012.

[34] B. Gleiss, L. Kovács, and L. Schnedlitz. Interactive visualization of saturation attempts in vampire. In *Lecture Notes in Computer Science*, pp. 504–513. Springer International Publishing, 2019. doi: 10.1007/978-3-030-34968-4_28

[35] P. Goffin, J. Boy, W. Willett, and P. Isenberg. An exploratory study of word-scale graphics in data-rich text documents. *IEEE Trans. Visualization and Computer Graphics*, 23(10):2275–2287, Oct. 2017. doi: 10.1109/tvcg.2016.2618797

[36] H. Goldsby, B. H. C. Cheng, S. Konrad, and S. Kamdoum. A visualization framework for the modeling and formal analysis of high assurance systems. In *Model Driven Engineering Languages and Systems*, pp. 707–721. Springer Berlin Heidelberg, 2006. doi: 10.1007/11880240_49

[37] G. Grätzer. *Math into LaTeX*. Birkhäuser, 3rd ed., 2000.

[38] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In *Computer Aided Verification*, pp. 453–456. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-27813-9_35

[39] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Model Checking Software*, pp. 121–136. Springer Berlin Heidelberg, 2003. doi: 10.1007/3-540-44829-2_8

[40] D. Gunning. Explainable artificial intelligence (XAI). Technical report, Defense Advanced Research Projects Agency (DARPA), 2016.

[41] M. Haverbeke. CodeMirror, 2011. `https://codemirror.net/`.

[42] F. Hohman, A. Srinivasan, and S. M. Drucker. TeleGam: Combining visualization and verbalization for interpretable machine learning. In *IEEE Visualization Conference*, pp. 151–155. IEEE, 2019. doi: 10.1109/VISUAL.2019.8933695

[43] IEEE Computer Society. *IEEE Standard for Verilog Hardware Description Language*, 2006. doi: 10.1109/IEEESTD.2006.99495

[44] E. Jee, S. Jeon, S. Cha, K. Koh, J. Yoo, G. Park, and P. Seong. Fbdverifier: Interactive and visual analysis of counter-example in formal verification of function block diagram. *Journal of Research and Practice in Information Technology*, 42(3):171—188, 2010.

[45] S. Jeong, J. Yoo, and S. Cha. VIS analyzer: A visual assistant for VIS verification and analysis. In *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, Piscataway, NJ, USA, 2010. doi: 10.1109/isorc.2010.41

[46] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. A. Frolov, E. Reeber, and A. Naik. Replacing testing with formal verification in intel coretm i7 processor execution engine validation. In *Intl. Conference on Computer Aided Verification*, pp. 414–429. Springer, 2009. doi: 10.1007/978-3-642-02658-4_32

[47] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett. The marabou framework for verification and analysis of deep neural networks. In *Intl. Conference on Computer Aided Verification*, pp. 443–452. Springer, 2019. doi: 10.1007/978-3-030-25540-4_26

[48] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*, 2019.

[49] M. Kohlhase, C. Lange, C. Müller, N. Müller, and F. Rabe. Adaptation of notations in living mathematical documents, 2008.

[50] M. Kohlhase and F. Rabe. Semantics of openmath and mathml 3. *Mathematics in Computer Science*, 6(3):235–260, 2012.

[51] M. Koleini, M. R. Clarkson, and K. K. Micinski. A temporal logic of security. *CoRR*, abs/1306.5678, 2013.

[52] E. Koutsofios and S. C. North. Drawing graphs with dot. Technical report, AT&T Bell Laboratories, 1992.

[53] J. Lahtinen, T. Launiainen, K. Heljanko, and J. Ropponen. *Model Checking Methodology for Large Systems, Faults and Asynchronous Behaviour: SARANA 2011 Work Report*. VTT Technical Research Centre of Finland, Finland, 2012.

[54] J. Lao-Tebar, F. Alvaro, and D. Marques. Proposal for coexistence of mathematical handwritten and keyboard input in a wysiwyg expression editor. In *FM4M/MathUI/ThEdu/DP/WIP@CIKM*, 2016.

[55] K. G. Larsen, F. Lorber, and B. Nielsen. 20 years of UPPAAL enabled industrial model-based validation and beyond. In *Lecture Notes in Computer Science*, pp. 212–229. Springer International Publishing, 2018. doi: 10.1007/978-3-030-03427-6_18

[56] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[57] C. Liu, L. Xie, Y. Han, D. Wei, and X. Yuan. AutoCaption: An approach to generate natural language description from visualization automatically. In *Proc. IEEE Pacific Symposium on Visualization*, pp. 191–195. IEEE, 2020. doi: 10.1109/PacificVis48177.2020.1043

[58] K. Loer and M. D. Harrison. An integrated framework for the analysis of dependable interactive systems (IFADIS): Its tool support and evaluation. *Automated Software Engineering*, 13(4):469–496, May 2006. doi: 10.1007/s10515-006-7999-y

[59] Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

[60] R. Martinez-Maldonado, V. Echeverria, G. Fernandez Nieto, and S. Buckingham Shum. From data to insights: A layered storytelling approach for multimodal learning analytics. In *Proc. CHI Conference on Human Factors in Computing Systems*, pp. 1—15. ACM, New York, NY, USA, 2020. doi: 10.1145/3313831.3376148

[61] S. H. Masoumi, S. A. R. Al-Haddad, and F. Z. Rokhani. New tool for converting high-level representations of finite state machines to verilog hdl. In *IEEE Student Conference on Research and Development*, pp. 1–6, 2017. doi: 10.1109/SCORED.2017.8305431

[62] A. Mathisen, T. Horak, C. N. Klokmose, K. Grønbæk, and N. Elmqvist. InsideInsights: Integrating Data-Driven Reporting in Collaborative Visual Analytics. *Computer Graphics Forum*, 38(3), June 2019. doi: 10.1111/cgf.13717

[63] E. Michael, D. Woos, T. Anderson, M. D. Ernst, and Z. Tatlock. Teaching rigorous distributed systems with efficient model checking. In *Proc. EuroSys Conference*, pp. 32:1–32:15. ACM, New York, NY, USA, 2019. doi: 10.1145/3302424.3303947

[64] MyScript. Handwriting recognition - MyScript, 2017. `https://www.myscript.com/handwriting-recognition`.

[65] J. Obeid and E. Hoque. Chart-to-Text: Generating natural language descriptions for charts by adapting the transformer model. In *Proc. International Conference on Natural Language Generation*, pp. 138–147. Association for Computational Linguistics, Dublin, Ireland, 2020.

[66] OpenCores.org. OpenCores, 1999. `https://opencores.org/`.

[67] OpenJS Foundation. Node.js, 2009. `http://nodejs.org/`.

[68] A. Pakonen, I. Buzhinsky, and V. Vyatkin. Counterexample visualization and explanation for function block diagrams. In *IEEE International Conference on Industrial Informatics*, pp. 747–753. IEEE, 2018. doi: 10.1109/indin.2018.8472025

[69] S. Patil, V. Vyatkin, and C. Pang. Counterexample-guided simulation framework for formal verification of flexible automation systems. In *IEEE International Conference on Industrial Informatics*. IEEE, Piscataway, NJ, USA, 2015. doi: 10.1109/indin.2015.7281905

[70] V. A. Pedroni. *Finite State Machines in Hardware: Theory and Design (with VHDL and SystemVerilog)*. The MIT Press, 2013.

[71] M. Pollanen, J. Hooper, B. Cater, and S. Kang. Towards a universal interface for real-time mathematical communication. In *CICM Workshops*, 2014.

[72] J. C. Roberts. State of the art: Coordinated & multiple views in exploratory visualization. In *Proc. IEEE Conference on Coordinated and Multiple Views in Exploratory Visualization*, pp. 61–71. IEEE, Piscataway, NJ, USA, July 2007. doi: 10.1109/cmv.2007.20

[73] H. Romat, C. Appert, and E. Pietriga. Expressive authoring of node-link diagrams with graphies. *IEEE Trans. Visualization and Computer Graphics*, 27(4):2329–2340, Apr. 2021. doi: 10.1109/tvcg.2019.2950932

[74] F. Rothenberger. Integration and analysis of alternative smt solvers for software verification. Master's thesis, ETH Zurich, 2016. doi: 10.3929/ETHZ-A-010608394

[75] V. Schuppan and A. Biere. Shortest counterexamples for symbolic model checking of LTL with past. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 493–509. Springer Berlin Heidelberg, 2005. doi: 10.1007/978-3-540-31980-1_32

[76] H. Seoul-Oh, J. Adkisson, and M. Stufflebeam. Mathquill, 2012. `http://mathquill.com/`.

[77] R. Sevastjanova, F. Beck, B. Ell, C. Turkay, R. Henkin, M. Butt, D. A. Keim, and M. El-Assady. Going beyond visualization: Verbalization as complementary medium to explain machine learning models. In *Workshop on Visualization for AI Explainability at IEEE VIS*, 2018.

[78] D. Shi, X. Xu, F. Sun, Y. Shi, and N. Cao. Calliope: Automatic visual data story generation from a spreadsheet. *IEEE Trans. Visualization and Computer Graphics*, 27(2):453–463, 2021. doi: 10.1109/TVCG.2020.3030403

[79] T. Spinner, U. Schlegel, H. Schäfer, and M. El-Assady. explAIner: A visual analytics framework for interactive and explainable machine learning. *IEEE Trans. Visualization and Computer Graphics*, 26(1):1064–1074, 2020. doi: 10.1109/TVCG.2019.2934629

[80] A. Spreafico and G. Carenini. Neural data-driven captioning of time-series line charts. In *Proc. International Conference on Advanced Visual Interfaces*. ACM, New York, NY, USA, 2020. doi: 10.1145/3399715.3399829

[81] A. Srinivasan, S. M. Drucker, A. Endert, and J. Stasko. Augmenting visualizations with interactive data facts to facilitate interpretation and communication. *IEEE Trans. Visualization and Computer Graphics*, 25(1):672–681, Jan. 2019. doi: 10.1109/tvcg.2018.2865145

[82] H. Strobelt, S. Gehrmann, M. Behrisch, A. Perer, H. Pfister, and A. M. Rush. Seq2seq-Vis: A visual debugging tool for sequence-to-sequence models. *IEEE Trans. Visualization and Computer Graphics*, 25(1):353–363, 2019. doi: 10.1109/TVCG.2018.2865044

[83] W. Thielke. Code geknackt. `https://www.focus.de/finanzen/news/krankenkassen-code-geknackt_aid_148829.html`, 2013.

[84] F. Wiehr, A. Hirsch, F. Daiber, A. Kruger, A. Kovtunova, S. Borgwardt, E. Chang, V. Demberg, M. Steinmetz, and H. Jorg. Safe handover in mixed-initiative control for cyber-physical systems, 2020.

[85] A. Wigmore, G. Hunter, E. Pflügel, J. Denholm-Price, and V. Binelli. Using automatic speech recognition to dictate mathematical expressions: The development of the "talkmaths" application at kingston university. *Journal of Computers in Mathematics and Science Teaching*, 28(2):177–189, Apr. 2009.

[86] S. Wolfram. Symbolic mathematical computation. *Communications of the ACM*, 28(4):390—394, Apr. 1985. doi: 10.1145/3341.3347

[87] D. Woos. *A Step-through Debugger for Distributed Systems*. PhD thesis, University of Washington, 2019.

[88] K. Xu, A. Ottley, C. Walchshofer, M. Streit, R. Chang, and J. Wenskovitch. Survey on the analysis of user interactions and visualization provenance. *Computer Graphics Forum*, 39(3):757–783, June 2020. doi: 10.1111/cgf.14035