THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Quickstrom: property-based acceptance testing with LTL specifications

Citation for published version:
O'Connor, L & Wickström, O 2022, Quickstrom: property-based acceptance testing with LTL specifications. in R Jhala & I Dillig (eds), *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* ACM Association for Computing Machinery, New York, NY, USA, pp. 1025-1038, 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, California, United States, 15/06/22. https://doi.org/10.48550/arXiv.2203.11532, https://doi.org/10.1145/3519939.3523728

Digital Object Identifier (DOI):
10.48550/arXiv.2203.11532
10.1145/3519939.3523728

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

OPEN ACCESS

Download date: 22. Jul. 2022

Quickstrom: Property-based Acceptance Testing with LTL Specifications

Liam O'Connor
University of Edinburgh
Edinburgh, Scotland
l.oconnor@ed.ac.uk

Oskar Wickström
Monoid Consulting
Malmö, Sweden
oskar@wickstrom.tech

arXiv:2203.11532v1 [cs.PL] 22 Mar 2022

Abstract

We present Quickstrom, a property-based testing system for acceptance testing of interactive applications. Using Quickstrom, programmers can specify the behaviour of web applications as properties in our testing-oriented dialect of Linear Temporal Logic (LTL) called QuickLTL, and then automatically test their application against the given specification with hundreds of automatically generated interactions. QuickLTL extends existing finite variants of LTL for the testing use-case, determining likely outcomes from partial traces whose minimum length is itself determined by the LTL formula. This temporal logic is embedded in our specification language, Specstrom, which is designed to be approachable to web programmers, expressive for writing specifications, and easy to analyse. Because Quickstrom tests only user-facing behaviour, it is agnostic to the implementation language of the system under test. We therefore formally specify and test many implementations of the popular TodoMVC benchmark, used for evaluation and comparison across various web frontend frameworks and languages. Our tests uncovered bugs in almost half of the available implementations.

Keywords: property-based testing, linear temporal logic, web frontend programming, semantics

1 Introduction

Property-based testing, such as that of QuickCheck [16], is a popular bug testing methodology whereby software is specified in the form of logical properties, and automatically tested against randomly-generated inputs to find possible counterexamples to those specifications. Property-based testing specifications are more high-level than unit tests, and facilitate greater maintainability with less effort. Unlike unit testing, property-based testing allows the programmer to specify the behaviour of a module without also specifying the expected behaviour of the module's user, i.e. the expected inputs to a function.

With the increasing use of web browser technology for user interfaces of applications, automatic testing of these interfaces using browser testing technology such as Selenium WebDriver [2], has become more necessary. To write a test in Selenium, the programmer must first script a specific interaction with their application's user interface, and then test that the interaction produces the expected result. For example, to test the property:

When I click Cancel*, I should return to the main menu.*

The programmer would write a script that first simulates a click to the Cancel button and then inspects the state of the user interface to confirm that we have indeed returned to the main menu. Other properties, however, are not so simple, such as this invariant:

I should not reach the finances page without logging in.

Or this *temporal* property:

The menu should never be disabled forever.

These properties cannot be easily translated into a Selenium script, because Selenium tests, like unit tests, require the programmer to specify not just the intended behaviour of the application but also the expected behaviour of the application's user. This is where Quickstrom comes in.

Quickstrom [40] is an in-development open-source tool which uses property-based testing techniques to enable automatic behavioural acceptance testing of web user interfaces from high-level specifications. Using a simple specification language, engineers inform Quickstrom of their desired properties, as well as how to interact with their user interface. Then, Quickstrom generates and tests hundreds of possible interactions, just as property-based testing libraries generate inputs, checking that the given properties are not violated.

In conventional property-based testing frameworks, the properties that make up specifications usually take the form of equations relating inputs to expected outputs of functions under test. Quickstrom, however, is not designed for testing functions, but for testing *whole applications*. These applications cannot be viewed as functions—instead they are *reactive* systems: they continuously respond to signals such as user actions and environmental events.

One of the most common logics used to specify reactive systems is Linear Temporal Logic (LTL) [32], a logic equipped with temporal modalities to describe *behaviours*: completed, infinite traces of a system's execution. Our tests, however, only produce finite traces: as only a finite number of actions can be taken, only a finite prefix of a desirable behaviour can be observed. Our dialect of LTL, called QuickLTL, is extended to accommodate this testing use-case. It is a *multi-valued* version of LTL defined for finite, partial traces whose minimum length is determined by the given formula. The logic is multi-valued to enable Quickstrom to give *presumptive* answers for when the formula cannot be definitively proven nor refuted by the steps taken so far. The syntax and semantics of QuickLTL are given in Section 2.

QuickLTL is embedded in our bespoke specification language Specstrom. This language is designed to be familiar to web programmers, expressive for writing specifications, and simple to analyse. In addition to writing QuickLTL formulae, engineers also use Specstrom to tell Quickstrom which actions to take and which events to expect when running tests. Details of the design of Specstrom and examples of its use are given in Section 3.

Our framework is designed for *acceptance testing*, that is, it only tests the user-observable behaviour of the application as a whole. Therefore, Quickstrom specifications are independent of the language used to implement the application under test. TodoMVC is a widely-implemented benchmark and sample application for a variety of web application frameworks and languages. We have converted the (informal) English specification of TodoMVC to a formal Specstrom specification, and used Quickstrom to test its various implementations, uncovering bugs and problems in more than a third of the available implementations. Our specification and our test results are discussed in Section 4.

Contributions

- The design and implementation of the Quickstrom tool itself, including its Specstrom interpreter and its test executor based on Selenium WebDriver,
- The QuickLTL temporal logic, a multi-valued dialect of Linear Temporal Logic for partial traces which incorporates minimum constraints on the length of the trace. We specify its semantics by formula progression and provide examples of its use.
- The design and implementation of the specification language Specstrom, which includes a variety of features, such as control over evaluation, which make it easy to specify systems. We specify an egg timer as a worked example.
- A formal specification of the TodoMVC benchmark in Specstrom/QuickLTL, and our evaluation of various implementations of this benchmark against our formal specification, in which we find faults in over one third of available implementations.

2 LTL and QuickLTL

Formulae:

$$\begin{array}{lcllll}
\varphi,\psi & ::= & p \mid \neg\varphi \mid \top \mid \bot & & & \\
& \mid & \varphi \wedge \psi & \mid & \varphi \vee \psi & \\
& \mid & \bigcirc \varphi & & & \textit{(next)}\\
& \mid & \Box\varphi & \mid & \Diamond \varphi & \textit{(henceforth/eventually)}\\
& \mid & \varphi\,\mathcal{U}\,\psi & \mid & \varphi\,\mathcal{R}\,\psi & \textit{(until/release)}\\
p & \in & \Sigma \to \{\top,\bot\} & & & \text{predicates}\\
\sigma & \in & \Sigma & & & \text{states}\\
\rho & \in & \Sigma^{\omega} & & & \text{behaviours}
\end{array}$$

Figure 1. Syntax of LTL

For $\rho = \sigma_0\sigma_1\sigma_2\cdots$:

$$\begin{array}{lcl}
\rho \models p & \Leftrightarrow & p(\sigma_0)\\
\rho \models \varphi \wedge \psi & \Leftrightarrow & \rho \models \varphi \text{ and } \rho \models \psi\\
\rho \models \neg\varphi & \Leftrightarrow & \rho \not\models \varphi\\
\rho \models \bigcirc\varphi & \Leftrightarrow & \sigma_1\sigma_2\ldots \models \varphi\\
\rho \models \Diamond\varphi & \Leftrightarrow & \text{There exists an } i \text{ such that } \sigma_i\ldots \models \varphi\\
\rho \models \Box\varphi & \Leftrightarrow & \text{For all } i \geq 0,\ \ \sigma_i\ldots \models \varphi\\
\rho \models \varphi\,\mathcal{U}\,\psi & \Leftrightarrow & \text{There exists an } i \text{ such that } \sigma_i\ldots \models \psi\\
& & \text{and for all } j < i,\ \sigma_j\ldots \models \varphi\\
\rho \models \varphi\,\mathcal{R}\,\psi & \Leftrightarrow & \text{For all } i \geq 0,\ \sigma_i\ldots \models \psi \text{ or}\\
& & \text{there exists } j < i \text{ such that } \sigma_j\ldots \models \varphi
\end{array}$$

Figure 2. Semantics of LTL

Linear Temporal Logic [32] is a modal logic that describes *behaviours*: infinite, linear sequences of states ordered by time. The syntax of LTL is given in Figure 1 and its semantics in Figure 2. When our behaviours are the completed traces or executions of our application, we can use LTL to write its specification. For instance, we can express invariants using the modality $\Box$ (read "henceforth" or "always"), as in this invariant which states that users should not be able to access the "Finances" page without being logged in:

$$\Box(\mathsf{LoggedIn} \vee \mathsf{page} \neq \texttt{"Finances"})$$

Invariants are an example of *safety properties*, which say that "bad" states will not be reached. It is straightforward to find counterexamples to safety properties by testing, as safety properties are exactly those that can be refuted in a finite number of steps [5], but many specifications also include *liveness properties*, which say that a "good" state will (eventually) be reached. We can express liveness properties by using the modality $\Diamond$ (read "eventually"), dual to $\Box$. For example, this property states that a menu will eventually be enabled:

$$\Diamond\,\mathsf{menuEnabled}$$

Counterexamples to liveness properties are more difficult to find via testing, as they take the form of infinite traces where the desired "good" state is never reached—no finite amount of testing will ever produce a complete counterexample. Conversely, if, rather than search for counterexamples, we instead search for a positive witness that the property holds, liveness properties become easy and safety properties become hard.

We can combine $\Diamond$ with $\Box$ to state that the menu will be enabled infinitely often; or, equivalently, that the menu will

$$\begin{aligned}
\neg \Diamond \varphi &= \Box \neg \varphi && (1)\\
\neg \Box \varphi &= \Diamond \neg \varphi && (2)\\
\neg \bigcirc \varphi &= \bigcirc \neg \varphi && (3)\\
\neg(\varphi\, \mathcal{U}\, \psi) &= \neg \varphi\, \mathcal{R}\, \neg \psi && (4)\\
\neg(\varphi\, \mathcal{R}\, \psi) &= \neg \varphi\, \mathcal{U}\, \neg \psi && (5)\\[6pt]
\Diamond \varphi &= \top\, \mathcal{U}\, \varphi && (6)\\
\Box \varphi &= \bot\, \mathcal{R}\, \varphi && (7)\\[6pt]
\Box \varphi &= \varphi \wedge \bigcirc \Box \varphi && (8)\\
\Diamond \varphi &= \varphi \vee \bigcirc \Diamond \varphi && (9)\\
\varphi\, \mathcal{U}\, \psi &= \psi \vee (\varphi \wedge \bigcirc (\varphi\, \mathcal{U}\, \psi)) && (10)\\
\varphi\, \mathcal{R}\, \psi &= \psi \wedge (\varphi \vee \bigcirc (\varphi\, \mathcal{R}\, \psi)) && (11)
\end{aligned}$$

Figure 3. Important LTL identities

never be disabled forever:

$$\Box \Diamond \mathsf{menuEnabled} \qquad \neg \Diamond \Box\, \mathsf{menuDisabled}$$

Both of the temporal operators $\Diamond$ and $\Box$ are special-cases of the more general temporal operators $\mathcal{U}$ (read "until") and its dual $\mathcal{R}$ (read "release") respectively, as can be seen in identities 6–7 of Figure 3. Using these operators we can express more sophisticated requirements on the ordering of events, such as these (equivalent) properties that state that we cannot access a secret page without logging in first:

$$\mathsf{LogIn}\, \mathcal{R}\, \neg \mathsf{SecretPage} \qquad \neg(\neg \mathsf{LogIn}\, \mathcal{U}\, \mathsf{SecretPage})$$

All of these operators can be thought of as fixed points of expansion identities involving the $\bigcirc$ (read "next") operator, such as identities 8–11 of Figure 3. We can also use $\bigcirc$ in our specifications, such as this example that describes a flashing screen, alternating between dark and light:

$$\Box(\mathsf{dark} \wedge \bigcirc \mathsf{light} \vee \mathsf{light} \wedge \bigcirc \mathsf{dark})$$

2.1 LTL with Finite Testing

As can be seen from these examples, LTL makes it easy to specify our application, but actually checking that our application meets our specification remains a challenge. As Quickstrom does not have any view of the application's structure beyond the current trace, we cannot construct a model of the system and apply the usual LTL model-checking techniques [39]. Instead, we randomly explore the state space of the system by performing randomly-chosen interface actions from a list given in the specification. This gives us finite, partial traces of the system's execution. LTL, however, is defined on behaviours—infinite, completed traces. As no finite amount of testing will give an infinitely long trace, we must instead turn to variants of LTL for finite traces.

The most glaring problem when moving LTL to finite traces is the $\bigcirc$ operator: what does $\bigcirc \varphi$ mean if there is no next state? Pnueli[1] answers by splitting the $\bigcirc$ operator into two dual "next" operators: The "weak next" $\overline{\bigcirc}$, which defaults to $\top$ when there is no next state; and the "strong next" $\underline{\bigcirc}$, which defaults to $\bot$. The $\Box$ and $\Diamond$ (resp. $\mathcal{R}$ and $\mathcal{U}$) expansion identities then use $\overline{\bigcirc}$ and $\underline{\bigcirc}$ respectively, so $\Box \varphi$ holds when a violation of φ does not occur in the trace, and $\Diamond \varphi$ holds when a state satisfying φ occurs at some point in the trace.

Pnueli's finite LTL is still defined for *completed* traces, however. It assumes that the application terminates when the trace ends, and no further states could follow. By contrast, Quickstrom traces are partial: they can be extended with more states simply by Quickstrom further interacting with the application. This means that if we were to use Pnueli's finite LTL, a liveness property for example about a timer application such as

$$\Diamond(\mathsf{timeRemaining} = 0)$$

could be marked as false simply because we didn't wait long enough for the remaining time to reach zero.

Bauer et al. [13] describe a tri-valued LTL for partial traces called LTL_3 which distinguishes between those formulae that are evidently true or false only from the trace provided, and those formulae which are *indeterminate*, i.e. require further states to evaluate definitively. Bauer et al. [12] later refine LTL_3 into RV-LTL, an LTL designed for runtime verification. This logic has four values: formulae may be *definitively* false, such as when a safety property is shown to be violated; *presumptively false*, such as when a liveness property fails to be fulfilled in the trace; *presumptively true*, such as when no counterexample to a safety property is found in the trace; or *definitively true*, such as when a liveness property is shown to be satisfied. The definitive cases correspond to the same in LTL_3. In the indeterminate cases, the presumptive results correspond to the answers given by Pnueli's finite LTL.

While RV-LTL is suitable for run-time monitoring or verification, it is still insufficient for testing. Consider our example from earlier that the menu will not be forever disabled:

$$\Box \Diamond \mathsf{menuEnabled}$$

As this formula nests $\Box$ and $\Diamond$ operators, it is definitive in neither positive nor negative cases and will only give presumptive answers. But the presumptive answer given in RV-LTL depends only on the value of menuEnabled in the last state of the trace. For a trace where menuEnabled continuously alternates off and on, the correct presumptive answer would be true, but this formula would be considered presumptively false if we happen to end testing in a state where menuEnabled is false. This would lead to many spurious counterexamples that, like the liveness property earlier, are merely due to ending our partial trace at the wrong time.

[1] This technique is found in many early papers on LTL with Pnueli as a coauthor such as Lichtenstein et al. [28], but Manna and Pnueli [33], which is usually cited for this technique, does not mention finite traces at all.

Formulae:

$$\begin{array}{lcllll}
\varphi, \psi & ::= & p \mid \neg\varphi \mid \top \mid \bot & & & \\
& \mid & \varphi \wedge \psi & \mid & \varphi \vee \psi & \\
& \mid & \odot\,\varphi & & & \textit{(required next)} \\
& \mid & \overline{\bigcirc}\,\varphi & \mid & \underline{\bigcirc}\,\varphi & \textit{(weak/strong next)} \\
& \mid & \Box_n\,\varphi & \mid & \Diamond_n\,\varphi & \textit{(henceforth/eventually)} \\
& \mid & \varphi\,\mathcal{U}_n\,\psi & \mid & \varphi\,\mathcal{R}_n\,\psi & \textit{(until/release)}
\end{array}$$

Guarded form:

$$\begin{array}{lcl}
F, G & ::= & F \wedge G \mid F \vee G \\
& \mid & \odot\varphi \mid \overline{\bigcirc}\varphi \mid \underline{\bigcirc}\varphi
\end{array}$$

Figure 4. Syntax of QuickLTL

$$\begin{array}{lcl}
\Box_0\,\varphi & = & \varphi \wedge \overline{\bigcirc}\,\Box_0\,\varphi \\
\Box_{n+1}\,\varphi & = & \varphi \wedge \odot\,\Box_n\,\varphi \\
\Diamond_0\,\varphi & = & \varphi \vee \underline{\bigcirc}\,\Diamond_0\,\varphi \\
\Diamond_{n+1}\,\varphi & = & \varphi \vee \odot\,\Diamond_n\,\varphi
\end{array}$$

Figure 5. QuickLTL expansions for basic temporal operators.

Exactly when testing should stop and traces should end to give correct presumptive answers depends on the specific formula being tested. Therefore, our QuickLTL dialect of LTL extends RV-LTL with additional information, allowing users to specify the required length of traces as part of the formula itself.

2.2 QuickLTL

As can be seen in Figure 4, we annotate temporal operators with numbers that specify the minimum length of the trace required to give accurate presumptive answers for that operator. For instance, to check $\Box_n\,\varphi$, Quickstrom must check at least n states for φ before concluding that the formula is presumptively true; and for $\Diamond_m\,\psi$ it must check at least m states for ψ before giving up and concluding that the formula is presumptively false. Adding annotations to our previous example, we get:

$$\Box_{100}\,\Diamond_5\,\mathsf{menuEnabled}$$

These annotations instruct Quickstrom to check (at least) 100 states for the property $\Diamond_5\,\mathsf{menuEnabled}$, which itself requires Quickstrom to check at least 5 states for $\mathsf{menuEnabled}$. These annotations eliminate the spurious counterexamples mentioned in the previous section, so long as the menu is re-enabled within 5 states of being disabled. The semantics of these annotations is best explained by their expansions into the "next" operators, given in Figure 5. In addition to the "weak next" $\overline{\bigcirc}$ and "strong next" $\underline{\bigcirc}$ of RV-LTL, we also introduce the self-dual "required next" $\odot$, which, rather than default to a value in the absence of a next state, simply requires Quickstrom to perform more actions to *produce* a next state if one does not exist. As can be seen in Figure 5, the numeric annotation n on a temporal operator expands into n uses of the $\odot$ operator, thus requiring Quickstrom to generate and check at least n states to evaluate the formula for that operator.

$$\boxed{\varphi \overset{\sigma}{\mapsto} \psi}$$

$$\frac{}{\top \overset{\sigma}{\mapsto} \top} \qquad \frac{}{\bot \overset{\sigma}{\mapsto} \bot}$$

$$\frac{}{p \overset{\sigma}{\mapsto} p(\sigma)} \qquad \frac{\varphi \overset{\sigma}{\mapsto} \varphi'}{\neg\varphi \overset{\sigma}{\mapsto} \neg\varphi'}$$

$$\frac{\varphi \overset{\sigma}{\mapsto} \varphi' \qquad \psi \overset{\sigma}{\mapsto} \psi'}{\varphi \wedge \psi \overset{\sigma}{\mapsto} \varphi' \wedge \psi'} \qquad \frac{\varphi \overset{\sigma}{\mapsto} \varphi' \qquad \psi \overset{\sigma}{\mapsto} \psi'}{\varphi \vee \psi \overset{\sigma}{\mapsto} \varphi' \vee \psi'}$$

$$\frac{}{\odot\varphi \overset{\sigma}{\mapsto} \odot\varphi} \qquad \frac{}{\underline{\bigcirc}\varphi \overset{\sigma}{\mapsto} \underline{\bigcirc}\varphi} \qquad \frac{}{\overline{\bigcirc}\varphi \overset{\sigma}{\mapsto} \overline{\bigcirc}\varphi}$$

$$\frac{\varphi \overset{\sigma}{\mapsto} \varphi'}{\Box_{n+1}\,\varphi \overset{\sigma}{\mapsto} \varphi' \wedge \odot\,\Box_n\,\varphi} \qquad \frac{\varphi \overset{\sigma}{\mapsto} \varphi'}{\Box_0\,\varphi \overset{\sigma}{\mapsto} \varphi' \wedge \overline{\bigcirc}\,\Box_0\,\varphi}$$

$$\frac{\varphi \overset{\sigma}{\mapsto} \varphi'}{\Diamond_{n+1}\,\varphi \overset{\sigma}{\mapsto} \varphi' \vee \odot\,\Diamond_n\,\varphi} \qquad \frac{\varphi \overset{\sigma}{\mapsto} \varphi'}{\Diamond_0\,\varphi \overset{\sigma}{\mapsto} \varphi' \vee \underline{\bigcirc}\,\Diamond_0\,\varphi}$$

$$\frac{\varphi \overset{\sigma}{\mapsto} \varphi' \qquad \psi \overset{\sigma}{\mapsto} \psi'}{\varphi\,\mathcal{U}_{n+1}\,\psi \overset{\sigma}{\mapsto} \psi' \vee (\varphi' \wedge \odot(\varphi\,\mathcal{U}_n\,\psi))}$$

$$\frac{\varphi \overset{\sigma}{\mapsto} \varphi' \qquad \psi \overset{\sigma}{\mapsto} \psi'}{\varphi\,\mathcal{U}_0\,\psi \overset{\sigma}{\mapsto} \psi' \vee (\varphi' \wedge \underline{\bigcirc}(\varphi\,\mathcal{U}_0\,\psi))}$$

$$\frac{\varphi \overset{\sigma}{\mapsto} \varphi' \qquad \psi \overset{\sigma}{\mapsto} \psi'}{\varphi\,\mathcal{R}_{n+1}\,\psi \overset{\sigma}{\mapsto} \psi' \wedge (\varphi' \vee \odot(\varphi\,\mathcal{R}_n\,\psi))}$$

$$\frac{\varphi \overset{\sigma}{\mapsto} \varphi' \qquad \psi \overset{\sigma}{\mapsto} \psi'}{\varphi\,\mathcal{R}_0\,\psi \overset{\sigma}{\mapsto} \psi' \wedge (\varphi' \vee \overline{\bigcirc}(\varphi\,\mathcal{R}_0\,\psi))}$$

Figure 6. Unrolling a formula, evaluating it against one state

2.3 Evaluation by Formula Progression

We evaluate QuickLTL formulae in a step-by-step manner, unrolling and partially evaluating the formula for each state of the trace in succession, similar to an operational semantics but for LTL formulae. Evaluation of a formula φ proceeds in three phases, repeated in a loop:

1. Given the state σ, unroll the formula φ one step and partially evaluate it against σ, according to the rules given in Figure 6. This relation $\varphi \overset{\sigma}{\mapsto} \varphi'$ evaluates all atomic propositions about the state σ, leaving a formula φ' where all nontrivial propositions are surrounded by a "next" operator. Note that the rules for temporal operators are expanding formulae exactly as in the expansion identities of Figure 5.

$$\boxed{G \Mapsto \varphi}$$

$$\frac{G \Mapsto \varphi \qquad F \Mapsto \psi}{G \land F \Mapsto \varphi \land \psi} \qquad \frac{G \Mapsto \varphi \qquad F \Mapsto \psi}{G \lor F \Mapsto \varphi \lor \psi}$$

$$\frac{}{\odot\varphi \Mapsto \varphi} \qquad \frac{}{\underline{\bigcirc}\varphi \Mapsto \varphi} \qquad \frac{}{\overline{\bigcirc}\varphi \Mapsto \varphi}$$

Figure 7. Stepping a formula forward

2. Simplify the resultant formula φ' using simple logical identities and the negation identities 1–5 from Figure 3. This will either result in a definitive answer like $\top$ or $\bot$, in which case Quickstrom will cease checking; or it will result in a formula F in *guarded form*, syntax of which is given in Figure 4. A formula is in guarded form if it consists solely of conjunctions and disjunctions of formulae guarded by "next" operators. If none of these "next" operators are the "required next" $\odot$, then a presumptive answer can be given by treating all $\overline{\bigcirc}$-guarded terms as $\top$ and all $\underline{\bigcirc}$-guarded terms as $\bot$, then simplifying the formula.
3. If the guarded-form formula F contains $\odot$-guarded terms, then Quickstrom must perform more actions to generate a new state σ. We then step the formula forward according to the rules in Figure 7. This relation $F \Mapsto \varphi$ progresses the formula to the next state, giving a new formula φ that can be used in the next iteration of the loop with the new state σ.

This procedure is very similar to the *formula progression* technique proposed by Kabanza et al. [8, 26] for standard LTL, however we split the progression relation into the two relations in Figures 6 and 7 to allow us to distinguish between formulae that already have definitive answers and those that have only presumptive answers due to the presence of remaining "next" operators.

Roşu and Havelund [37] warn that this technique can result in exponential blow-up in the size of the formula relative to the number of steps taken, however we have found in our case studies that this is avoided in all practical cases by our simplification of the formula at each step. Nested temporal operators can cause the formula size to grow at each step but, as our traces are rarely longer than a few hundred states, this is not prohibitively expensive. Therefore, this technique remains effective for our practical scenarios.

3 Specstrom

QuickLTL formulae are only a small component of a Quickstrom specification. Specification writers must also describe which *actions* can occur, either due to Quickstrom interacting directly with the interface or due to asynchronous environmental events, as well as the *state queries* that make up the atomic propositions in a QuickLTL formula.

Our specification language, Specstrom, is a simple language with syntax that superficially resembles JavaScript and has smooth interoperability with JavaScript data such as objects and arrays, but with a significantly more restricted semantics: recursion is not allowed, and all expressions terminate. Specstrom also includes a number of built-in primitives for constructing formulae, actions and state queries.

Although Specstrom supports higher order functions, it still guarantees termination through use of a very simple type system. Because most web programmers are not accustomed to strict type-checking, this type system is designed to be mostly invisible to the programmer: it distinguishes only between *functions* and *non-functions*, and all types are inferred. To avoid circumvention of this type system, functions may not be placed inside data types such as arrays or objects. The termination guarantee that we obtain from this type system enables us to analyse Specstrom code more easily, as in Section 3.3.

3.1 Evaluation Control

Specstrom also gives the user fine-grained control over evaluation, allowing programmers to define their own temporal operators or connectives. As an illustrative example, consider the following temporal predicate $\mathsf{evovae}(x)$, which states that x shall forever have the same value it had initially:

$$\mathsf{evovae}(x) = \textbf{let}\ v = x; \Box(x == v);$$

In a language which evaluates in applicative order (i.e. "strict" evaluation), this would trivially be true, because the parameter x would be fully evaluated to a value before evovae was even invoked, and thus $x == v$ would be true independently of the state in which it is executed. On the other hand, in a language where bindings are only evaluated when they are used, this would *also* be trivially true, because binding v to x would not evaluate x until inside the $\Box$ operator, making $\mathsf{evovae}(x)$ equivalent to the trivial $\Box(x == x)$. For this reason, Specstrom allows the user to specify which bindings are to be left unevaluated explicitly, with a ~ prefix before the binding. This allows us to define evovae with the intended semantics, where only x is left unevaluated and v is evaluated eagerly:

$$\mathsf{evovae}({\sim}x) = \textbf{let}\ v = x; \Box(x == v);$$

3.2 Specifying an Egg Timer

Figure 8 gives an illustrative example of a complete Specstrom specification for a three-minute egg timer application, with syntax slightly adjusted for brevity. The application consists of a start/stop toggle button and a label containing the remaining time in seconds.

State projections. The first two lines introduce atomic propositions, *stopped* and *started*, which indicate the status of the timer. Strings surrounded in \`backticks\` are *CSS selectors* which extract part of the application's UI state. In this

```
let ~stopped = `#toggle`.text == "start";
let ~started = `#toggle`.text == "stop";
let ~time = parseInt(`#remaining`.text);

action start! = click!(`#toggle`) when stopped;
action stop! = click!(`#toggle`) when started;
action wait! = noop! timeout 1000 when started;
action tick? = changed?(`#remaining`);

let ~ticking {
  let old = time;
  started
    ∧ ◯̄ (tick? in happened
        ∧ time == old − 1
        ∧ if time == 0 {stopped} else {started})
}
let ~waiting =
  started ∧ ◯̄(wait! in happened ∧ started);
let ~starting =
  stopped ∧ ◯̄ (start! in happened
                ∧ if time == 0 {stopped} else {started});
let ~stopping =
  started ∧ ◯̄(stop! in happened ∧ stopped);

let ~safety =
  loaded? in happened ∧ time == 180
    ∧ □_400(starting ∨ stopping ∨ waiting ∨ ticking);
let ~liveness =
  □_400(start! in happened ⇒ ◇_360 stopped);
let ~timeUp =
  □_400(start! in happened ⇒ ◇_360(time == 0));

check safety liveness;
check timeUp with start! wait! tick?;
```

Figure 8. An Example of a Specstrom specification

case, our propositions are determined by the text label on the toggle button. Note that these definitions are expected to change over time and are thus bound with the ~ operator to prevent them from being evaluated at definition-time. We similarly define a state-dependent quantity *time* which is determined from the label containing the number of seconds remaining.

Actions. The next four lines define *actions* that may occur, either due to *user actions*, which are initiated by Quickstrom, or due to *events*, which are asynchronously initiated by the application. In our specifications and libraries, we adopt the convention that user actions are suffixed with an exclamation mark (!) and events with a question mark (?). Thus, we define three user actions and one event. The event is tick?, which fires when the application updates the remaining time label each second. The user actions are start!, stop! and wait!, which all indicate actions Quickstrom may take when interacting with the application. They are defined in terms of built-in primitive actions such as noop! and click!(). We associate *guards* to our actions using the **when** operator, allowing us to differentiate between the start! and stop! actions: both of these actions simply involve clicking the toggle button, but in different contexts. In general, these guards take the form of atomic propositions, i.e. non-temporal boolean formulae. The action will only fire if the guard condition is met.

Timeouts. The definition for the action wait! associates a *timeout* to the built-in action noop! with the **timeout** keyword. This indicates to Quickstrom that, after performing this action, it should not attempt to perform another action for at least one second, or until an event occurs. These timeouts are designed to accommodate the very common use case where a user action causes an application to respond asynchronously. In this case, the action noop! does nothing, so the action wait! will cause Quickstrom to wait until a tick? occurs, or until one second elapses. This action is needed because otherwise Quickstrom will simply stop the timer as soon as it starts, as it has no other actions available to perform once the timer has started.

Safety properties. The next five definitions can be understood by looking at the property *safety*. This property states that the built-in event loaded? must happen first, and that in the initial state, the time remaining should be three minutes. It then states that one of *ticking*, *waiting*, *starting*, and *stopping* must be true forevermore. Each of these properties describes one allowable state transition. For example, *stopping* describes a transition from a state where the timer was *started* to a state where the timer has *stopped* due to the stop! action. The variable "happened" is a special state-dependent variable that contains all events or actions that occurred immediately prior to the current state. The *ticking* transition uses a **let**-binding to freeze the value of *time* before the tick? event occurred. This allows us to then specify that the value of *time* after the event must be decremented.

Liveness properties. While the safety property defined above thoroughly describes what transitions are allowable, it does not say anything about what transitions will be taken. For this, we need liveness properties. The simplest liveness property of our egg timer is that the timer will eventually stop—either by running out of time or by the user pressing the stop button. This is easily expressed in the property *liveness*. For an egg timer, however, it is reasonable to want a stronger property: eventually, time will run out. Unfortunately this property, which we call *timeUp* in Figure 8, is not necessarily true for all implementations. For example, if the timer is implemented with a one-second granularity, the user might repeatedly press the start and stop button faster than the granularity of the timer, and prevent it from

Checker

Start ⟨*dependencies*⟩
Request a new session be started
(also specifies which selectors are relevant)

Act ⟨*action*⟩ ⟨*version*⟩ ⟨*timeout*⟩
Request the given action be performed
(rejected if version < trace length)

Wait ⟨*time*⟩ ⟨*version*⟩
Request to signal a Timeout after the given time
if no event occurs first.

Executor

Event ⟨*event*⟩ ⟨*state*⟩
Notify the checker about an event that occurred
along with the updated state

Acted ⟨*state*⟩
Notify the checker that an action was performed
along with the updated state

Timeout ⟨*state*⟩
Notify the checker that a timeout has elapsed
along with the (possibly) updated state

Figure 9. The protocol between the checker and executor.

ever making progress. An optional parameter to the **check** command, seen at the bottom of Figure 8, allows us to specify which actions may fire when testing a given property. Therefore, we can still check this property by excluding the stop! action from the set of allowable actions. Then, the only way the timer will stop is if it runs out of time.

Currently, the Quickstrom checker makes a completely random selection from the set of allowable actions for the current state. Refining this action selection to be more *targeted*, methodically exploring previously unreached parts of the state space, is left as future work (see Section 5.1).

3.3 Static Analysis

Quickstrom is built on top of Selenium WebDriver [2], a programmatic testing tool which can simulate user interaction with a web application using a headless browser instance. When given a Specstrom specification, Quickstrom must determine what parts of the browser state are relevant for the properties at hand *before* checking, to properly instrument the running application with listeners for changes to relevant components of the user interface, and to get a consistent view of a state by retrieving all relevant information in bulk. We determine this information automatically by statically analysing Specstrom code. Because Specstrom guarantees termination and does not support recursion, this analysis is a very simple abstract interpretation for dependency analysis. In addition to direct dependencies, such as the expression `` `#toggle` ``.text which depends obviously on the UI element `` `#toggle` ``, we must also track indirect dependencies, such as in the expression **if** `` `#toggle` ``.enabled {0} **else** {1}, which also depends on `` `#toggle` ``. Running this analysis on the property under test yields a set of state elements which are instrumented and recorded by Quickstrom as it runs actions.

3.4 Checker and Executor

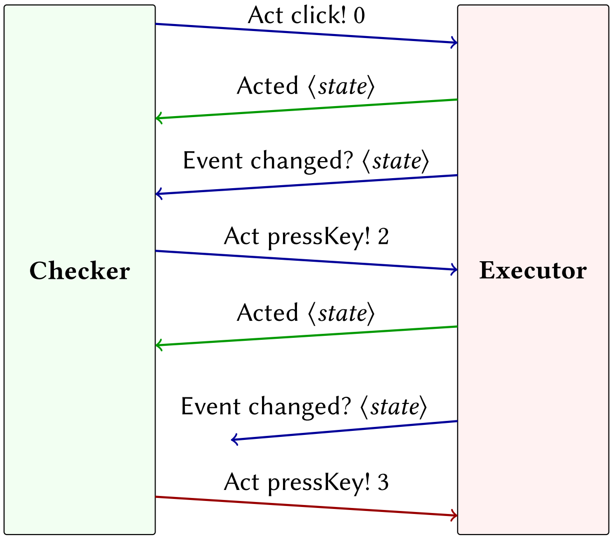

Figure 10. Example of communication between checker and executor.

Quickstrom is divided into two main components: the *checker*, which is the Specstrom interpreter that evaluates the formula and selects actions to perform; and the *executor*, which interacts with Selenium WebDriver to actually interact with the application under test using a headless browser instance.

Figure 9 describes the protocol for communication between the checker and executor. Each column describes the messages sent by the checker and executor respectively. When the checker intends to test a property, it signals the executor to load the application (Start) and tells it which parts of the application's state are relevant to the property. As mentioned, this information is determined by simple static analysis of the Specstrom code. The executor uses this information to add event listeners to the application under test, and to determine what parts of the state to include in future messages. The executor then waits for events to occur in the application or action requests (Act) to come from the checker. In either case it reports the updated application state to the checker, after performing the requested action if necessary (Acted and Event). The checker will wait until the initial event of the property is observed (usually, this is when the page is loaded) before beginning to request actions.

For user actions that include a timeout, such as our wait! action from the egg timer example, the Act message may optionally include a **timeout** parameter. If the specified time elapses without an event occurring, the executor will send a Timeout message along with the state (which might have changed since the last action occurred). The checker can also request such a timeout separately from an action using the Wait message, which is used when a **timeout** is associated with an event: if the event occurs, the checker requests a timeout from the executor.

Because the application under test is running in a separate process and cannot be paused, it is possible that asynchronous events could change the application's state while the checker is deciding what action to perform. Thus, the checker might make a decision based on out-of-date information. We solve this problem by including the *length of the trace so far* in every message after checking begins. Figure 10 illustrates an interaction between the checker and executor after initiating a run for a particular property. Time flows from the top to the bottom. Initially, the checker tells the executor to click a button, which the executor dutifully does, returning the updated state along with its acknowledgment that the action was performed. Then, part of the application's state is asynchronously changed, which the executor reports to the checker along with an updated state. The checker acknowledges receipt of all these updated states by including the current trace length (2) in its next action request, to press a key. The executor then performs this action and sends the new state to the checker. Then, the application state is again asynchronously changed, but before the checker is notified of this, it requests that the executor perform another action to press a key. Because this request has an out of date trace length (3, not 4), the executor knows to ignore this request.

Nothing about the *checker* is specific to Selenium WebDriver: paired with a different executor, the same checker could be used to test any reactive system. While the only production-ready executor is the WebDriver-based one, to simplify testing of our Specstrom interpreter we have also implemented another executor, which interprets models written in Milner's Calculus of Communicating Systems [34]. Developing other executors is promising future work.

4 Evaluation

The TodoMVC benchmark is a suite of various implementations of the same to-do list application, which should all look the same and behave according to the same (plain English) specification [4]. The various implementations are provided by independent developers, usually the developers of the frameworks themselves. The purpose of the benchmark is to provide a non-trivial application that can be used to compare frameworks for performance, functionality and ease of use.

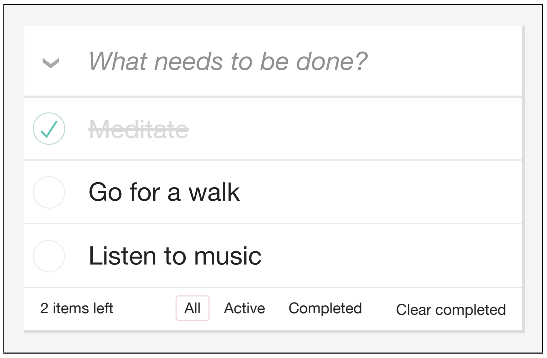

Figure 11. A TodoMVC implementation in action.

Figure 11 contains a screenshot of one of the TodoMVC implementations in action. As can be seen, items can be added by typing into the text box at the top of the list, and items may be marked "completed" by clicking the checkbox to their left. The arrow icon to the left of the text entry box allows all items to be toggled simultaneously. Items may be filtered by their status using the buttons below the list, and items may be edited by double clicking on them. A delete button appears to the right of an item when the user hovers over it. The to-do list is *persistent*, stored in local storage, so page reloads should not affect the content of the to-do list.

4.1 A Formal TodoMVC Specification

The TodoMVC specification is quite precise, but it is written in technical English, not a formal specification language. Therefore, we have translated the TodoMVC specification into a formal specification, consisting of 300 lines of Specstrom. As has been observed with other natural language specifications [11], our formalisation efforts show the official specification is rife with ambiguity and under-specification.

For instance, the official TodoMVC specification defines what items should be shown when the user changes the current filter, but it does not say what happens to the *rest* of the user interface. Our formal specification makes the reasonable assumption that no other part of the interface (such as pending input) should be modified when switching between filters, even though the official specification does not explicitly rule out such behaviour.

The official specification also says nothing about which filter should be active after all to-do items have been removed, and our specification does the same, i.e. it leaves it undefined. Interestingly, there seems to be a commonly understood de-facto specification that the filter should be unchanged—developers have submitted bug-fix pull requests to implementations that behave differently—but we have not formalised this as it is not officially specified by TodoMVC.

Figure 12 gives a high-level sketch illustrating the main safety property for TodoMVC in our formal specification.

```
let ~safety = initial
  ∧ □ ( focusNewTodo
        ∨ enterNewTodoText
        ∨ addNew
        ∨ changeFilter
        ∨ setSameFilter
        ∨ toggleAll
        ∨ checkOne
        ∨ uncheckOne
        ∨ delete
        ∨ enterEditMode
        ∨ inEditMode )
  ∧ □ ··· ⟨invariants⟩
let ~enterEditMode = startEditing ∧ ◯̅ editMachine
let ~editMachine {
  let item = itemInEditMode;
  exitEditMode(item) 𝓡
    (enterEditText ∨ exitEditMode(item))
}
let exitEditMode(initialItem) =
      commitEdit(initialItem)
    ∨ abortEdit(initialItem)
```

Figure 12. Sketch of our TodoMVC specification

When numeric subscripts on temporal operators are omitted, they use a user-specified default value. Using a higher value increases test accuracy but also test running time. See Section 4.3 for a detailed analysis of this trade-off. Like our timer specification, we specify the application similarly to a state machine. The property consists of three conjuncts: one specifying the initial state, one specifying the allowable transitions corresponding to user actions, and one containing a list of invariants. These invariants mostly just state the requirement that the various elements that make up the user interface are actually present.

This kind of state machine specification is a very common pattern when writing Quickstrom specifications, and our TodoMVC example also demonstrates another pattern: We can use the temporal operator $\mathcal{R}$ to *nest* these state machine specifications. Notice that the transition conjunct of the main safety property is easily satisfied if we are editing an item (i.e. the *inEditMode* disjunct is true). This is because we specify editing an item as a separate state machine specification in *editMachine*, which is invoked by the *enterEditMode* transition. From the perspective of the main, high-level state machine in *safety*, editing an item is a single abstract state with a single internal transition, described by the formula *inEditMode*. But in *editMachine*, we refine this into three transitions: editing the text of an item, committing changes, and aborting changes. If either of those last two transitions are taken, we leave the nested state machine: we are no longer in edit mode and need no longer abide by the nested state machine specification. We use the release operator $\mathcal{R}$ to indicate this (Note that the top-level *safety* state machine uses $\square$, which is equivalent to $\mathcal{R}$ with an exit condition of $\bot$). In addition, we also "remember" the original value of an item that is being edited, using the **let** binding in *editMachine*, so that we can specify that the text returns to its original value if an edit is aborted.

Passed — 23 (*9 beta*, 14 mature)
angularjs_require, *aurelia*, *backbone_require*, backbone, binding-scala, closure, emberjs, *enyo_backbone*, *exoskeleton*, js_of_ocaml, *jsblocks*, knockback, knockoutjs, kotlin-react, *react-alt*, *react-backbone*, react, *riotjs*, scalajs-react, typescript-angular, typescript-backbone, typescript-react, vue
Failed — 20 (*8 beta*, 12 mature)
angular-dart[14], *angular2_es2015*[1], *angular2*[5], angularjs[7], backbone_marionette[11], *canjs_require*[13], canjs[13], *dijon*[2], dojo[9], *duel*[4], elm[4], jquery[10], *knockoutjs_require*[2], *lavaca_require*[4], mithril[7], polymer[6], *ractive*[12], reagent[4], vanilla-es6[8,3], vanillajs[8]

Table 1. Summary of Results

We have not yet formalized the persistence aspect of the official specification. We expect that this could be modelled by inserting page reloads as another possible action, and may expose further problems in the implementations' handling of local storage, but this is left as future work.

4.2 Results

From the many standard TodoMVC 1.3 implementations listed on the TodoMVC website [4], we selected 43 implementations for our evaluation. We selected only those implementations that are stored on the TodoMVC repository (commit version `41ba86d` from February 2020) to ensure reproducibility. We also excluded any implementations that were not standard, single-page TodoMVC applications (e.g. streaming variants such as those based on Firebase), those that didn't successfully start (i.e. `cujo`), those whose markup didn't match the specification (i.e. `gwt`), and those for whom compiled, testable artifacts were not available (i.e. `react-hooks`, `emberjs-require`). Some of these implementations are labelled as *beta*, i.e. still under evaluation from the TodoMVC team. As can be seen in Table 1, which gives a high level overview of our results on each of these implementations, we found bugs or faults in 20 of those implementations—almost half. Surprisingly, this fault rate was not significantly higher for the implementations marked as beta, although bugs due to missing features are more common.

Table 2 describes in detail the specific faults that Quickstrom exposed. The problem found in `angular-dart`, number 14, does not actually impede the overall operation of the

	Description	Count
1	Items have no checkboxes	1
2	There are no filter controls	2
3	A `<strong>` element is missing	1
4	Blank items can be added	1
5	Edit input is not focused after double-click	1
6	Incorrectly pluralizes the to-do count text	1
7	Any pending input is cleared on filter change or removal of last item	4
8	A new item is created from pending input after non-create actions	2
9	"Toggle all" does not untoggle all items when certain filters are enabled	1
10	The "Toggle all" button disappears when the current filter contains no items.	1
11	Commiting an empty to-do item in edit mode does not fully delete it—it can later be restored with "Toggle all"	1
12	Editing an item hides other items	1
13	Adding an item changes the filter to "All"	2
14	Adding an item first shows an empty state	1

Table 2. Problems found in TodoMVC implementations

application. However, because it temporarily empties the list before re-populating it when adding a new item, this is a bug according to our formal specification. Because this is not explicitly forbidden by the official English specification, however, we consider it a "dubious" case, and it could be considered "correct" by a more generous interpretation of the specification. Of the remaining problems, three (Problems 1–3) are just unimplemented functionality or missing UI elements that are required by the specification. These problems appear only in beta versions for the most part, and would likely be found in a cursory review. The others all appear to be programming mistakes. Problems 4–6 are simple bugs that are easily found manually, but the remaining problems (Problems 7–13) require nontrivial steps to uncover. In particular, problems often manifest when the user does something unexpected after entering (but not committing) some input text, as in Problem 7 (the most common fault at four implementations) and Problem 8 (which also appeared in multiple implementations). In addition, the interaction between filters and the "Toggle all" button is another common source of bugs, as in Problems 9–11.

Problem 11 is particularly involved to uncover, and could easily slip past cursory review. In order to reproduce this bug, the user must create a to-do item, then immediately double click it to start editing, erase all text and press Enter (the item now appears deleted, but filters are still visible). Then, the user must click the "toggle all" button, at which point the supposedly deleted item re-appears.

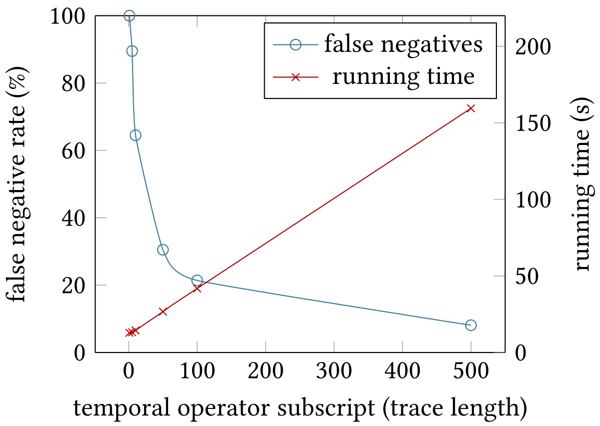

Figure 13. False negative rate and average running-time

The bugs found in the various TodoMVC implementations run the gamut from trivial to complex. Notably, we found roughly as many faults in mature implementations as we did in beta ones. We even found problems in all three of the "Pure JavaScript" examples (`vanillajs`, `vanilla-es6`, and `jquery`) considered as reference implementations by the TodoMVC specification. These case studies demonstrate Quickstrom's effectiveness as a bug-finding tool, even for mature software with extensive manual testing.

4.3 Running Time and False Negative Rate

Figure 13 summarises the relationship between the subscripts on temporal operators, test accuracy, and running time for our TodoMVC specification. Specifically we measure the false negative rate (percentage of tests on faulty implementations that unexpectedly pass) for failing implementations and compare it to the average running time for testing passing implementations. This is because the TodoMVC specification consists only of safety properties: when checking a safety property, passing cases will always take more time than failing ones, as the testing tool will always exit as soon as a counterexample is found. Similarly, the only way that testing of safety properties could give inaccurate results is in the form of a false negative, because Quickstrom will only report a test failure if a concrete counterexample is found. Conversely, when testing liveness properties, the situation would be reversed: failing cases would take the most time, and inaccurate results would be false positives. For each subscript, each implementation was tested 10 times on a 2020 Apple MacBook Pro M1 with 16GB of RAM, although testing time is dominated by waiting for events, so performance of hardware does not greatly affect running time.

As can be seen, testing takes linearly more time but becomes logarithmically more accurate as the temporal subscript increases. All of the faults found in TodoMVC can be exposed with a subscript of merely 50, but the more involved faults such as Problem 11 are only found infrequently, resulting in flaky tests. The vast majority of faults are reliably found with a subscript of 100—the default value in Quickstrom—and testing takes less than a minute (approx. 42 seconds for passing cases). After that, higher subscripts are still more likely to uncover faults, but there are diminishing returns in terms of faults found for time taken.

5 Related and Future Work

5.1 Automated Browser Testing

While tools such as Selenium WebDriver are now well established and enjoy widespread industry use, higher-level automation of such acceptance testing is an area that has not yet been thoroughly explored. Like us, Bainczyk et al. [9] apply their testing tool ALEX to the TodoMVC benchmark, however ALEX is based on learning-based testing without models or specifications, and is therefore limited to finding inconsistencies between TodoMVC implementations. By contrast, Quickstrom generates tests based on user-provided logical specifications, and can therefore find more bugs, albeit with greater effort required for writing specifications. We believe model inference techniques [6] such as those in ALEX and other model checking techniques such as counterexample-guided abstraction-refinement [18] are highly compatible with Quickstrom, and could potentially be used to make Quickstrom more intelligently select actions and search for counterexamples—a kind of *targeted* property-based testing for LTL specifications [29, 30].

Panchekha et al. [35] present a tool to automatically verify constraints on page *layout* and appearance based on high-level specifications. While it is possible to verify some layout constraints with Quickstrom, that is not its primary purpose. Quickstrom does not presently feature any specific functionality to verify that pages are laid out correctly, focusing instead on behavioural specifications. Thus, this tool is complementary with Quickstrom.

5.2 LTL for User Interfaces

We are not the first to realise the suitability of LTL for describing user interfaces. Jeffrey [24] and Jeltsch [25] simultaneously observed that, just as logical formulae correspond to types of programs, LTL formulae can correspond to types for functional-reactive programs (FRP), including user interfaces and interactive applications. Perez and Nilsson [36] used LTL formulae for testing and debugging of FRP programs.

The *Model-View-Update* (MVU) architecture, pioneered by the Elm programming language [3], itself descended from FRP, is a simple design pattern for user interfaces that has now become widespread, with variants for most programming languages and UI frameworks. At its core, it describes interactive applications with a type for the *model* or application state M, a type for the *view* V, a type for *actions* A, and a pair of functions $\mathit{display} : M \rightarrow V$ and $\mathit{update} : M \times A \rightarrow M$. This model is highly compatible with the view of states and actions used in Quickstrom. As the Quickstrom checker is not WebDriver-specific, we could repurpose it with custom executors to produce language- or framework-specific testing tools, allowing Specstrom and QuickLTL specifications to be applied to these applications more directly.

5.3 Other Executors and Debuggers

As neither Specstrom nor QuickLTL are specific to web applications, it is worth investigating other domains to see if they would be a suitable fit. Other GUI frameworks such as GTK and Qt both have acceptance testing frameworks similar to Selenium WebDriver, and are obvious candidates, but there may be more interesting use cases further afield: for example, our Specstrom checker could also be attached to an emulator or a debugger for an embedded system, where actions and events take the form of IO signals and the accessible state is the memory on the system. Model checkers based on LTL such as Spin are already used in the embedded systems area, so programmers in this area may already be amenable to LTL specifications.

5.4 State Machine Specifications

As previously mentioned, our specifications for both our egg timer example and our TodoMVC case study strongly resemble a specification of a state machine. Model-oriented property-based testing using state machine models was originally developed for the implementation of QuickCheck for Erlang, which was used to find linearisable instances of race conditions [17]. The same version of QuickCheck was later used to test AUTOSAR implementations [7, 23]. This state machine idea has now been implemented for several other property-based testing systems, including the original Haskell QuickCheck. Unlike Quickstrom, these frameworks require that the model capture the essential complexity of the application under test: it needs to be functionally complete to be a useful oracle. For a system that is conceptually simple, such as a key-value database engine, this is not a problem, but for systems that are burdened with inherent complexity, such as a business application with many intricate rules, a useful model tends to grow as complex as the system itself. Quickstrom specifications can be more abstract: the engineer does not have to implement a complete functional model of their system, and is free to leave out details and specify only the most important aspects of their application. For example, the timer specification given in Figure 8 intentionally applies both to timers that reset when stopped and to timers that pause when stopped.

5.5 QuickLTL as a Temporal Logic

While QuickLTL is by definition a superset of other partial trace variants of LTL such as RV-LTL [12], we have not yet formally explored the relationship between QuickLTL and conventional infinite-trace LTL dialects. Recall that actions in Quickstrom are divided into *user actions*, under the control of the user, and *events*, under the control of the application. It is not reasonable to assume progress for all actions, as conventional LTL dialects do, as this would impose a requirement on applications that events must eventually occur if no user actions can be taken. The *reactive LTL* of van Glabbeek [38] is designed specifically to address this problem, and would serve as a good starting point for this investigation.

5.6 Fault Injection

Majumdar and Niksic [31] provide a theoretical justification for the surprising effectiveness of randomly testing distributed systems with *fault injection*—intentional simulation of network faults for testing purposes. This kind of fault injection is often provided by tools such as Jepsen [1] and Elle [27]. While these tools are for systems like distributed databases, the same fault injection technique may also be useful in Quickstrom: Modern web applications often try to handle network interruptions gracefully, defaulting to local storage or warning the user that the connection was lost. Simulating network faults would enable Quickstrom specifications to test such scenarios.

5.7 Security and Confidentiality

A drawback of our approach is that we can only write properties that are expressible in LTL, i.e. properties of a single trace. While we can specify some properties that relate to security, such as our property requiring the user to log in to see the "Finances" page, we cannot express security properties, such as information-flow security, in standard LTL as they are *hyperproperties*—properties relating multiple traces [20]. While temporal logics exist to express hyperproperties [10, 19, 21], random testing for hyperproperties may not be as fruitful as it is for QuickLTL properties. As hyperproperties relate multiple traces, a counterexample to a security property expressed as an n-hyperproperty would take the form of a n-tuple of traces, rather than a single trace. This makes counterexamples to security properties significantly harder to find. While Hrițcu et al. [22] report effectively testing security properties using randomised property-based testing, these tests were on abstracted models of security definitions rather than on realistic systems. We expect that, when applied to real-world web applications with large amounts of state, counterexamples to security properties will not be easily found by randomised exploration of the state space, and would likely benefit from the more *targeted* approaches previously mentioned.

5.8 Property-based Testing and Formal Methods

The specifications used for property-based testing resemble those used for formal verification of software. In particular, QuickCheck test suites have served as sources of specifications for deductive verification of Haskell code [14]. Our specifications too resemble temporal logic specifications that one might find for formal tools such as TLA+, Spin or, most recently, Alloy 6. In their work on information flow, Hrițcu et al. [22] observe that property-based testing is still valuable even in the context of formal verification, as it can eliminate the wasted effort of trying to prove a faulty or ill-specified system correct. Chen et al. [15] posit that property-based testing could be used as an incremental path towards more widespread adoption of formal verification among software engineers. Quickstrom very much fits into this theme, as specifications in Specstrom could, with little modification, be transliterated for use in more formal, exhaustive tools.

6 Conclusion

We have presented Quickstrom, a property-based browser testing framework for acceptance testing of web applications. Quickstrom users write formal specifications of their application's behaviour in our specification language Specstrom, based on our new dialect of Linear Temporal Logic for testing, QuickLTL. With this specification, Quickstrom will test the application with hundreds of possible interactions, all generated automatically from the specification.

Our case studies demonstrate that Quickstrom is an effective tool for finding non-trivial bugs in realistic web applications. Writing a Specstrom/QuickLTL specification enables programmers to find bugs more quickly and easily than by writing a comprehensive test suite with a browser testing framework. But, more than that, we hope that Quickstrom will make formal specification and modelling, immensely powerful tools for improving software reliability, more accessible to mainstream web application programmers.

Acknowledgments

Many thanks to Rob van Glabbeek for his early feedback on the design of our logic.

References

[1] 2021. Jepsen. https://jepsen.io/ accessed January 2021.

[2] 2021. Selenium WebDriver. https://www.selenium.dev/ accessed January 2021.

[3] 2021. The Elm Programming Language. https://www.elm-lang.org/ accessed January 2021.

[4] 2021. The TodoMVC Benchmark. https://www.todomvc.com/ accessed October 2021.

[5] Bowen Alpern and Fred B. Schneider. 1985. Defining liveness. *Inform. Process. Lett.* 21, 4 (1985). https://doi.org/10.1016/0020-0190(85)90056-0

[6] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

[7] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *International Conference on Software Testing, Verification and Validation Workshops*. 1–4. https://doi.org/10.1109/ICSTW.2015.7107466

[8] Fahiem Bacchus and Froduald Kabanza. 1996. *Using Temporal Logic to Control Search in a Forward Chaining Planner.* IOS Press, 141–153.

[9] Alexander Bainczyk, Alexander Schieweck, Bernhard Steffen, and Falk Howar. 2017. *Model-Based Testing Without Models: The TodoMVC Case Study.* Springer International Publishing, Cham, 125–144. https://doi.org/10.1007/978-3-319-68270-9_7

[10] Musard Balliu, Mads Dam, and Gurvan Le Guernic. 2011. Epistemic Temporal Logic for Information Flow Security. In *Programming Languages and Analysis for Security*. Association for Computing Machinery, San Jose, California, Article 6, 12 pages. https://doi.org/10.1145/2166956.2166962

[11] Ryan Barry, Rob van Glabbeek, and Peter Höfner. 2020. Formalising the Optimised Link State Routing Protocol. *Electronic Proceedings in Theoretical Computer Science* 316 (Apr 2020), 40–71. https://doi.org/10.4204/eptcs.316.3

[12] A. Bauer, M. Leucker, and C. Schallhart. 2010. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation* 20, 3 (2010), 651–674. https://doi.org/10.1093/logcom/exn075

[13] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM Transactions in Software Engineering Methodology* 20, 4, Article 14 (Sept. 2011), 64 pages. https://doi.org/10.1145/2000799.2000800

[14] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, Set, Verify! Applying Hs-to-Coq to Real-World Haskell Code (Experience Report). *Proceedings of the ACM in Programming Languages* 2, ICFP, Article 89 (jul 2018), 16 pages. https://doi.org/10.1145/3236784

[15] Zilin Chen, Liam O'Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. 2017. The Cogent Case for Property-Based Testing. In *Workshop on Programming Languages and Operating Systems* (Shanghai, China). ACM, 7 pages. https://doi.org/10.1145/3144555.3144556

[16] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming*. ACM, 268–279. https://doi.org/10.1145/351240.351266

[17] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. 2009. Finding Race Conditions in Erlang with QuickCheck and PULSE. In *International Conference on Functional Programming* (Edinburgh, Scotland). ACM, 149–160. https://doi.org/10.1145/1596550.1596574

[18] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *J. ACM* 50, 5 (Sept. 2003), 752–794. https://doi.org/10.1145/876638.876643

[19] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *Principles of Security and Trust*, Martín Abadi and Steve Kremer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–284.

[20] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (Sept. 2010), 1157–1210.

[21] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. 2012. Model Checking Information Flow in Reactive Systems. In *Verification, Model Checking, and Abstract Interpretation*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–185.

[22] Cătălin Hrițcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *International Conference on Functional Programming* (Boston, Massachusetts, USA). ACM, 455–468. https://doi.org/10.1145/2500365.2500574

[23] John Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *Successes That Can Change the World - Essays Dedicated to Philip Wadler on his 60th Birthday*. https://doi.org/10.1007/978-3-319-30936-1_9

[24] Alan Jeffrey. 2012. LTL Types FRP: Linear-Time Temporal Logic Propositions as Types, Proofs as Functional Reactive Programs. In *Programming Languages Meets Program Verification* (Philadelphia, Pennsylvania, USA). Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2103776.2103783

[25] Wolfgang Jeltsch. 2013. Temporal Logic with "Until", Functional Reactive Programming with Processes, and Concrete Process Categories. In *Programming Languages Meets Program Verification* (Rome, Italy). Association for Computing Machinery, New York, NY, USA, 69–78. https://doi.org/10.1145/2428116.2428128

[26] Froduald Kabanza and Sylvie Thiébaux. 2005. Search Control in Planning for Temporally Extended Goals. In *International Conference on Automated Planning and Scheduling*. AAAI, Monterey, California, USA, 130–139.

[27] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. arXiv:2003.10554 [cs.DB]

[28] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. 1985. The glory of the past. In *Logics of Programs*, Rohit Parikh (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 196–218.

[29] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-Based Testing. In *International Symposium on Software Testing and Analysis*. ACM, Santa Barbara, CA, USA, 46–56. https://doi.org/10.1145/3092703.3092711

[30] Andreas Löscher and Konstantinos Sagonas. 2018. Automating Targeted Property-Based Testing. In *International Conference on Software Testing, Verification and Validation*. 70–80. https://doi.org/10.1109/ICST.2018.00017

[31] Rupak Majumdar and Filip Niksic. 2017. Why is Random Testing Effective for Partition Tolerance Bugs? *Proceedings of the ACM in Programming Languages* 2, POPL, Article 46 (dec 2017), 24 pages. https://doi.org/10.1145/3158134

[32] Zohar Manna and Amir Pnueli. 1992. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Berlin, Heidelberg.

[33] Zohar Manna and Amir Pnueli. 1995. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, Berlin, Heidelberg.

[34] Robin Milner. 1982. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg.

[35] Pavel Panchekha, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. 2019. Modular Verification of Web Page Layout. *Proceedings of the ACM in Programming Languages* 3, OOPSLA, Article 151 (2019), 26 pages. https://doi.org/10.1145/3360577

[36] Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proceedings of the ACM in Programming Languages* 1, ICFP, Article 2 (aug 2017), 27 pages. https://doi.org/10.1145/3110246

[37] Grigore Roşu and Klaus Havelund. 2005. Rewriting-Based Techniques for Runtime Verification. *Automated Software Engineering* 12, 2 (April 2005), 151–197. https://doi.org/10.1007/s10515-005-6205-y

[38] Rob van Glabbeek. 2020. Reactive Temporal Logic. In *Expressiveness in Concurrency, and Structural Operational Semantics (EXPRESS/SOS 2020) (Electronic Proceedings in Theoretical Computer Science 322, Vol. 322)*. Open Publishing Association, Online, 51–68. https://doi.org/10.4204/EPTCS.322.6

[39] Moshe Y. Vardi and Pierre Wolper. 1986. Automata-theoretic techniques for modal logics of programs. *J. Comput. System Sci.* 32, 2 (1986), 183–221.

[40] Oskar Wickström. 2020. Quickstrom. https://quickstrom.io/ accessed January 2021.