Automated Analyses of IOT Event Monitoring Systems

Andrew Apicelli[1], Sam Bayless[1], Ankush Das[1], Andrew Gacek[1], Dhiva Jaganathan[1], Saswat Padhi[2], Vaibhav Sharma[3](✉), Michael W. Whalen[1], and Raveesh Yadav[1]

[1] Amazon Web Services, Inc., Seattle, USA
{apicea,sabayles,daankus,gacek,dhivasj,mww,raveeshy}@amazon.com
[2] Google LLC, Mountain View, USA
spadhi@google.com
[3] Amazon.com Services LLC, Seattle, USA
svaib@amazon.com

Abstract. AWS IoT Events is an AWS service that makes it easy to respond to events from IoT sensors and applications. *Detector models* in AWS IoT Events enable customers to monitor their equipment or device fleets for failures or changes in operation and trigger actions when such events occur. If these models are incorrect, they may become out-of-sync with the actual state of the equipment causing customers to be unable to respond to events occurring on it.

Working backwards from common mistakes made when creating detector models, we have created a set of automated analyzers that allow customers to prove their models are free from six common mistakes. Our analyzers have been running in the AWS IoT Events production service since December 2021. Our analyzers check six correctness properties in the production service in real time. 93% of customers of AWS IoT Events have run our analyzers without needing to have any knowledge of them. Our analyzers have reported property violations in 22% of submitted detector models in the production service.

1 Introduction

AWS IoT Events is a managed service for managing fleets of IoT devices. Customers use AWS IoT Events in diverse use cases such as monitoring self-driving wheelchairs, monitoring a device's network connectivity, humidity, temperature, pressure, oil level, and oil temperature sensing. Customers use AWS IoT Events by creating a *detector model* that detects events occurring on IoT devices and notifies an external service so that a corrective action can be taken. An example is an industrial boiler which constantly reports its temperature to a detector. The detector tracks the boiler's average temperature over the past 90 min and notifies a human operator when it is running too hot.

S. Padhi–Work done while at Amazon Web Services, Inc.

Each detector model is defined as a finite state machine with dynamically typed variables and timers, where timers allow detectors to keep track of state over time. A model processes inputs from IoT devices to update internal state and to notify other AWS services when events are detected. Customers can use a single detector model to instantaneously detect events in thousands of devices. Ensuring well-formedness of a detector model is crucial as ill-formed detector models can miss events in *every* monitored device.

Starting from a survey that identified sources of well-formedness problems in customer models, we identified most common mistakes made by customers and detect them using type- and model-checking. To use a model-checker for checking well-formedness of a detector model, we formalize the execution semantics of a detector model and translate this semantics into the source-language notation of the JKind model checker [1]. Model checking [2–9] verifies desirable properties over the behavior of a system by performing the equivalent of an exhaustive enumeration of all the states reachable from its initial state. Most model checking tools use *symbolic encodings* and some form of *induction* [6] to prove properties of very large finite or even infinite state spaces.

We have implemented type-checking and model-checking as an analysis feature in the production AWS IoT Events service. Our analyzers have reported well-formedness property violations in 22% of submitted detector models. 93% of customers of AWS IoT Events have checked their detector models using our analyzers. Our analyzers report property violations to customers with an average latency of 5.6 s (see Sect. 4).

Our contributions are as follows:

1. We formalize the semantics of AWS IoT Events detector models.
2. We identify six well-formedness properties whose violations detect common customer mistakes.
3. We create fast, push-button analyzers that report property violations to customers.

2 Overview

Consider a user of AWS IoT Events who wants to monitor the temperature of an industrial boiler. If the industrial boiler overheats, it can cause fires and endanger human lives. To detect an early warning of an overheating event, they want to automatically identify two different alarming events on the boiler's temperature. They want their first alarm to be triggered if the boiler's reported temperature is outside the normal range for more than 1 min. They want their second alarm to be triggered if the temperature is outside the normal range for another 5 min after the first alarm.

A user might try to implement these requirements by creating the (flawed) detector model shown in Fig. 1. This detector receives temperature data from the boiler and responds by sending a text message to the user. The detector model contains four states:

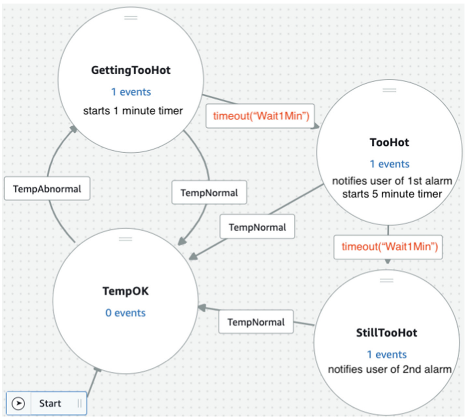
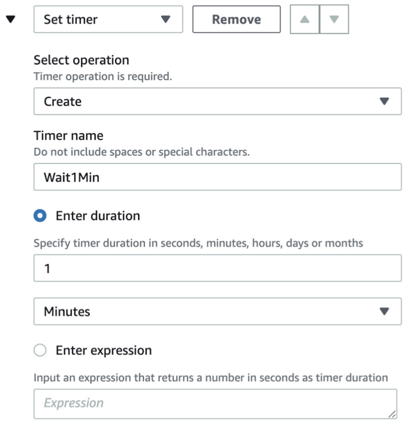


Fig. 1. AWS IoT Events detector model with two alarms (buggy version)



Select operation
Timer operation is required.
Create

Timer name
Do not include spaces or special characters.
Wait1Min

[✓ Enter duration]
Specify timer duration in seconds, minutes, hours, days or months
1
Minutes

Enter expression
Input an expression that returns a number in seconds as timer duration
Expression

Fig. 2. An action in the detector model from Fig. 1

- TempOK: starting state of the detector model. The detector stays in this state as long as the boiler's temperature lies in a normal range. The detector transitions from TempOK to GettingTooHot on detecting a temperature outside normal range, indicated by TempAbnormal.
- GettingTooHot: detector starts a 1 min timer and transitions back to TempOK if the boiler cools down. When the timer expires, it transitions to TooHot.
- TooHot: detector first notifies the user of the 1st alarm. It then starts a 5 min timer and transitions back to TempOK if the boiler cools down. When the 5 min timer expires, it transitions to StillTooHot.
- StillTooHot: detector notifies user of the 2nd alarm.

Understanding the Bug: Every state in the detector model consists of *actions*. An action changes the internal state of a detector or triggers an external service. For example, the GettingTooHot state consists of an action that starts a timer. The user can edit these actions with an interface shown in Fig. 2. This action starts a one minute timer named Wait1Min. Note that timers are accessible from every state in the detector model. Even though the Wait1Min timer is created in the GettingTooHot state of Fig. 1, it can be checked for expiration in all the four states of Fig. 1.

The detector model in Fig. 1 has a fatal flaw based on a typo. The user has written timeout("Wait1Min") instead of timeout("Wait5Min") when transitioning out of TooHot. This is allowed as timers are globally referenceable. However, it is a bug because each global timer has a unique name and the Wait1Min timer has already been used and expired. This makes StillTooHot unreachable, meaning the 2nd alarm won't ever fire, since a timer can expire at most once.

Related Work. Languages such as IOTA [10], SIFT [11], and the system from Garcia et al. [12] use *trigger-condition-action rules* [13] to control the behavior of internet of things applications. These languages have the benefit of being largely declarative, allowing users to specify desired actions under different environmental stimuli. Similar to our approach, SIFT [11] automatically removes common user mistakes as well as compiles specifications into controller implementations without user interaction, and IOTA [10] is a reasoning calculus that allows custom specifications to be written both about why something *should* or *should not* occur. AWS IoT Events is designed explicitly for monitoring, rather than control, and our approach is imperative, rather than declarative: detector models do not have the same inconsistencies as rule sets, as they are disambiguated using explicit priorities on transitions. On the other hand, customers may still construct machines that do not match their intentions, motivating the analyses described in this paper.

3 Technique

In this section, we present a formal execution semantics of an AWS IoT Events detector model and describe specifications for the correctness properties.

Formalization of Detector Models. Defining the alphabet and the transition relation for the state machine is perhaps the most interesting aspect of our formalization. Since detector models may contain global timers, *timed automata* [14] might seem like an apt candidate abstraction. However, AWS IoT Events users are not allowed to change the clock frequency of timers, nor specify arbitrary *clock constraints*. These observations allow us to formalize the detector models as a regular state machine, with timeout durations as additional state variables.

Formally, we represent the state machine for a detector model $\mathbf{M}$ as a tuple $\langle \mathbf{S}, \mathbf{S}_0, \mathbf{I}, \mathbf{G}, \mathbf{T}, \mathcal{E}_E, \mathcal{E}_X, \mathcal{E}_I \rangle$, where:

- $\mathbf{S}$: finite set of states in the FSM,
- $\mathbf{S}_0 \subseteq \mathbf{S}$: set of *initial* state(s),
- $\mathbf{I}$: set of input variables assigned by the environment
- $\mathbf{G}$: set of global variables assigned by the state machine
- $\mathbf{T}$: set of timer variables that are reset by the model and updated as time evolves in the environment
- $\mathcal{E}_E : \mathbf{S} \rightarrow \kappa\ \texttt{list}$: mapping from states to a (possibly empty) list of entry events to be performed when entering a state. κ describes an *event*, further explained in the description of the grammar.
- $\mathcal{E}_X : \mathbf{S} \rightarrow \kappa\ \texttt{list}$ is a mapping from states to a list of exit events to be performed when exiting a state.
- $\mathcal{E}_I : \mathbf{S} \rightarrow (\kappa\ \texttt{list} \times \mu\ \texttt{list})$: mapping from states to a list of input events, including transitions to other states.

It is assumed that the sets $\mathbf{I}$, $\mathbf{G}$, and $\mathbf{T}$ are pairwise disjoint, and we define the set $\mathbf{V} \triangleq \mathbf{I} \cup \mathbf{G}$ to represent input and global variables in the model.

We denote by $\mathbb{V}$ the set of values for global ($\mathbf{G}$) and input ($\mathbf{I}$) variables; $\mathbb{V}$ ranges over the values of primitive types: integers, decimals (rationals), booleans,

$$\begin{aligned}
\tau &::= \texttt{int} \mid \texttt{dec} \mid \texttt{str} \mid \texttt{bool} \\
\epsilon &::= e_0 \ bop \ e_1 \mid uop \ e_0 \mid l \mid v \mid \texttt{timeout}(t) \mid \texttt{isundefined}(v) \mid \ldots \\
\alpha &::= \texttt{setTimer}(t, e) \mid \texttt{resetTimer}(t) \\
&\quad \mid \texttt{clearTimer}(t) \mid \texttt{setGlobal}(g, e) \\
\kappa &::= \texttt{event}(e, a*) \\
\mu &::= \texttt{transition}(e, a*, s) \\
\iota &::= \texttt{message}(i, v) \mid \texttt{timeout}(t)
\end{aligned}$$

Fig. 3. Types, expressions, actions, and events in IoT Events Detector Models

and strings. Integers and rationals are assumed to be unbounded, and rationals are arbitrarily precise. We use $\mathbb{N}$ as the domain for time and timeout values. Sets $\mathbb{V}^{\perp}$ and $\mathbb{N}^{\perp}$ are extended with the value $\perp$ to represent an *uninitialized* variable.

The grammar for types (τ), expressions (ϵ), actions (α), events (κ), transitions (μ) and input triggers (ι) is shown in Fig. 3. In the grammar, metavariable e stands for an expression, l stands for a literal value in $\mathbb{V}$, v stands for any variable in $\mathbf{V}$, t is a timer variable in $\mathbf{T}$, a is an action, and i is an input in $\mathbf{I}$. The unary and binary operators include standard arithmetic, Boolean, and relational operators. The `timeout` expression is true at the instant timer t expires, and the `isundefined` expression returns true if the variable or timer in question has not been assigned. Actions (α) describe changes to the system state: `setTimer` starts a timer and sets the periodicity of the timer, while the `resetTimer` and `clearTimer` reset and clear a timer (without changing the periodicity of the timer). The `setGlobal` action assigns a global variable. Events (κ) describe conditions under which a sequence of actions occur.

We define configurations $\mathbf{C}$ for the state machine as:

$$\mathbf{C} \triangleq \mathbf{S} \times (\mathbf{I} \rightarrow \mathbb{V}^{\perp}) \times (\mathbf{T} \rightarrow (\mathbb{N}^{\perp} \times \mathbb{N}^{\perp})) \times (\mathbf{G} \rightarrow \mathbb{V}^{\perp})$$

Each configuration $C = \langle s, i, t, g \rangle$ tracks the following:

- a state $s \in \mathbf{S}$ in the detector model,
- the input valuation $i \in (\mathbf{I} \rightarrow \mathbb{V}^{\perp})$ containing the values of inputs,
- the timer valuation $t \in (\mathbf{T} \rightarrow (\mathbb{N}^{\perp} \times \mathbb{N}^{\perp}))$ for user-defined timers. Each timer has both a periodicity and (if active) a time remaining, and
- the global valuation $g \in (\mathbf{G} \rightarrow \mathbb{V}^{\perp})$ for global variables in the detector model.

Example 1. Consider a corrected version of our example detector model from Fig. 1 which has two timers, `Wait1Min` and `Wait5Min`, and no global variables. Some examples of configurations for this model are:

- $\langle \texttt{TempOK}, \{\texttt{temp} : \perp\}, \{\texttt{Wait1Min} : (\perp, \perp), \texttt{Wait5Min} : (\perp, \perp)\}, \{\}\rangle$ is the initial configuration. The model contains input `temp`, timers `Wait1Min` and `Wait5Min`, and no global variables. As no variables or timers have been assigned, all variables have value undefined ($\perp$).

$$\boxed{\begin{array}{ll} C \vdash_\epsilon e \to v & \\ C \vdash_\alpha a \to C' & C \vdash_{\alpha*} al \to C' \\ C \vdash_\kappa k \to C' & C \vdash_{\kappa*} kl \to C' \\ C \vdash_{\mu*} ml \to C' & C \vdash_{\mathcal{E}_I} \mathcal{E}_I \to C' \\ \vdash_\iota C \overset{i}{\mapsto} C' & \end{array}} \qquad \frac{\langle s,i,t,g\rangle \vdash_\epsilon e \to v}{\begin{array}{l}\langle s,i,t,g\rangle \vdash_\alpha \texttt{setTimer}(tr,e) \to \\ \langle s,i,t[tr \leftarrow (v,v)],g\rangle\end{array}}$$

$$\frac{t(tr) = (p,v)}{\begin{array}{l}\langle s,i,t,g\rangle \vdash_\alpha \texttt{resetTimer}(tr) \to \\ \langle s,i,t[tr \leftarrow (p,p)],g\rangle\end{array}} \qquad \frac{t(tr) = (p,v)}{\begin{array}{l}\langle s,i,t,g\rangle \vdash_\alpha \texttt{clearTimer}(tr) \to \\ \langle s,i,t[tr \leftarrow (p,\bot)],g\rangle\end{array}}$$

$$\frac{\langle s,i,t,g\rangle \vdash_\epsilon e \to v}{\begin{array}{l}\langle s,i,t,g\rangle \vdash_\alpha \texttt{setGlobal}(gv,e) \\ \to \langle s,i,t,g[gv \leftarrow v]\rangle\end{array}} \qquad \frac{C \vdash_\epsilon e \to \texttt{false}}{C \vdash_\kappa \texttt{event}(e,al) \to C} \qquad \frac{\begin{array}{c}C \vdash_\epsilon e \to \texttt{true} \\ C \vdash_{\alpha*} al \to C'\end{array}}{C \vdash_\kappa \texttt{event}(e,al) \to C'}$$

$$\frac{}{C \vdash_{\mu*} \texttt{nil} \to C} \qquad \frac{\begin{array}{c}C \vdash_\epsilon e \to \texttt{false} \\ C \vdash_{\mu*} tl \to C'\end{array}}{C \vdash_{\mu*} \texttt{transition}(e,al,s') :: tl \to C'}$$

$$\frac{\begin{array}{c}C \vdash_\epsilon e \to \texttt{true} \qquad C \vdash_{\alpha*} al \to C' \\ C' \vdash_{\kappa*} \mathcal{E}_X(C.s) \to C'' \\ C''[s \leftarrow s'], ti \vdash_{\kappa*} \mathcal{E}_E(s') \to C'''\end{array}}{C \vdash_{\mu*} \texttt{transition}(e,al,s') :: tl \to C'''} \qquad \frac{\begin{array}{c}C \vdash_{\kappa*} kl \to C' \\ C' \vdash_{\mu*} ml \to C''\end{array}}{C \vdash_{\mathcal{E}_I} (kl,ml) \to C''}$$

$$\frac{\begin{array}{c}\texttt{matchesEarliest}(C.t,ti) \wedge \texttt{subtractTimers}(C,ti) \to C' \\ C' \vdash_{\mathcal{E}_I} \mathcal{E}_I(C'.s) \to C'' \wedge \texttt{clearTimers}(C'') \to C'''\end{array}}{\vdash_\iota C \xmapsto{timeout(ti)} C'''} \qquad \frac{\langle s,i[iv \leftarrow v],t,g\rangle \vdash_{\mathcal{E}_I} \mathcal{E}_I(C.s) \to C'}{\vdash_\iota \langle s,i,t,g\rangle \xmapsto{message(iv,v)} C'}$$

Fig. 4. Rules describing behavior of the system

– $\langle \texttt{TooHot}, \{\texttt{temp} : 300\}, \{\texttt{Wait1Min} : (60,\bot),\ \texttt{Wait5Min} : (300,260)\}, \{\}\rangle$ is the configuration at global time t if the temperature is still beyond the normal range and we transition to the `TooHot` detector model state. Note the `Wait1Min` timer is no longer set whereas the `Wait5Min` timer has a periodicity of 300 and is set to expire at $t + 260$.

To define the execution semantics, we create a structural operational semantics for each of the grammar rules and for the interaction with the external environment, as shown in Fig. 4. We distinguish semantic rules by decorating the turnstiles with the grammar type that they operate over ($\epsilon, \alpha, \kappa, \mu, \mathcal{E}_I$, and ι). The variables e, a, k, m, i stand in for elements of the appropriate syntactic class defined by the turnstile. For lists of elements, we decorate the syntactic class with * (e.g. $\vdash_{\alpha*}$), and the variables with 'l' (e.g. al). We use the following notation conventions: Given $C = \langle s,i,t,g\rangle$, we say $C.s = s$, and similarly with the other components of C. We also say $C[s \leftarrow s']$ is equivalent to $\langle s',i,t,g\rangle$, and similarly with the other components of C.

Expressions ($\vdash_\epsilon$) evaluate to values, given a configuration. We do not present expression rules (they are simple), but illustrate the other rule types in Fig. 4. For actions ($\vdash_\alpha$), the *setTimer* rule establishes the periodicity of a timer and also starts it. The *resetTimer* and *clearTimer* rules restart an existing timer given a periodicity p or clear it, respectively, and the *setGlobal* rule updates the value of a global variable. *Events* (κ) are used by entry and exit events for states. The list rules for actions ($\alpha*$) and events ($\kappa*$) are not presented but are straightforward: they apply the relevant rule to the head of the list and pass the updated configuration to the remainder of the list, or return the configuration unchanged for `nil`. *Transition event lists* ($\mu*$) cause the system to change state, executing (only) the first transition from the list whose guard e evaluates to true. Finally, the top-level rule $\vdash_\iota$ describes how the system evolves according to external stimuli.

A *run* of the machine is any valid sequence of configurations produced by repeated applications of the $\vdash_\iota$ rule. Timeout inputs increment the time to the earliest active timeout as described by the *matchesEarliest* predicate:

$$\texttt{matchesEarliest}(t, x) \equiv \exists t_i, p_i.(p_i, x) = t(t_i) \wedge \\ \forall t_j, p_j, y.((p_j, y) = t(t_j) \implies y = \bot \vee y \geq x)$$

The `subtractTimers` function subtracts t_i from each timer in C, and the `clearTimers` function, for any timers whose time remaining is equal to zero, calls the `clearTimer` action[1].

3.1 Well-formedness Properties

To find common issues with detector models, we surveyed (i) detector models across customer tickets submitted to AWS IoT Events, (ii) questions posted on internal forums like the AWS re:Post forum [15], and (iii) feedback submitted via the web-based console for AWS IoT Events. Based on this survey, we determined that the following correctness properties should hold over all detector models. For more details about this survey, please refer to Appendix A.

The Model does not Contain Type Errors: The AWS IoT Events expression language is *untyped*, and thus, may contain ill-typed expressions, e.g., performing arithmetic operations on Booleans. A large class of such bugs can be readily detected and prevented using a *type inference* algorithm. The algorithm follows the standard Hindley-Milner type unification approach [16–18] and generates (and solves) a set of type constraints or reports an error if no valid typing is possible. We use this type inference algorithm to detect type errors in the detector model. Every type error is reported as a warning to the customer. When our type inference successfully infers types for expressions, we use them to construct a well-typed abstract state machine using the formalization reported in Sect. 3.

For the remaining well-formedness properties we use model checking. We introduce one or more *indicator variables* in our global abstract state to track certain kinds of updates in the state machine, and then we assert temporal properties on these indicator variables. Because we use a model checker that

[1] In the interests of space, we do not cover the *batch* execution mode, where all variables used in expressions maintain their "pre-state" value until the step is completed; it is a straightforward extension.

checks only *safety properties*, in many cases we invert the property of interest and check that its negation is falsifiable, using the same mechanism often used for test-case generation [19].

Every Detector Model State is Reachable and Every Detector Model Transition and Event can be Executed: For each state $s \in \mathbf{S}$, we add a new Boolean *reachability indicator* variable v^s_{reached} to our abstract state that is initially `false` and assigned `true` when the state is entered (similarly for transitions and events). To encode the property in a safety property checker, we encode the following *unreachability* property expressed in LTL and check it is falsifiable. If it is provable, the tool warns the user.

$$\text{Unreachable}(s) \triangleq \square\ (\neg\ v^s_{\text{reached}})$$

Every Variable is Set Before Use: In order to test that variables are properly initialized, first we identify the places where variables are assigned and used. In detector models, there are three places where variables are used: in the evaluation of conditions for events and transitions, and in the `setGlobal` action (which occurs because of an event or transition). We want to demonstrate that the variables used within these contexts are never equal to $\perp$ during evaluation. In this case, we can reuse the reachability variables that we have created for events and transitions to encode that variables should always have defined values when they are used.

We first define some functions to extract the set of variables used in expressions and action lists. The function $Vars(e) : \epsilon \rightarrow v$ `set` simply extracts the variables in the expression. For action lists, it is slightly more complex, because variables are both defined and used:

$$\begin{aligned}
Vars(\texttt{nil}) &= \{\} \\
Vars(\texttt{setTimer}(t, e) :: tl) &= Vars(e) \cup Vars(tl) \\
Vars(\texttt{resetTimer}(t) :: tl) &= Vars(tl) \\
Vars(\texttt{clearTimer}(t) :: tl) &= Vars(tl) \\
Vars(\texttt{setGlobal}(g, e) :: tl) &= Vars(e) \cup (Vars(tl) - \{g\}) \\
Vars(\texttt{event}(e, al)) &= Vars(e) \cup Vars(al) \\
Vars(\texttt{transition}(e, al, s')) &= Vars(e) \cup Vars(al)
\end{aligned}$$

Every event or transition can be executed at most once during a computation step, so we can use the execution indicator variables to determine when a variable might be used.

$$\begin{aligned}
&\forall a_i, v_j \in Vars(a_i)\ . \\
&\quad \text{SetBeforeUse}(a_i, v_j) \triangleq \square\,(v^{a_i}_{\text{exec}} \implies v_j \neq \perp)
\end{aligned}$$

Input Read Only on Message Trigger: This property is covered in the previous property, with one small change. To enforce it, we modify the translation of the semantics slightly so that at the beginning of each step, prior to processing the input message, all input variables are assigned $\perp$.

Message Triggered Between Consecutive Timeouts: We conservatively approximate a liveness property (no infinite path consisting of only timeout events) with a safety property: the same timer should not timeout twice without an input message occurring in between the timeouts. This formulation may flag models that do not have infinite paths with no input events, but our customers consider it a reasonable indicator.

We begin by defining an indicator variable for each timer t_i (of type integer rather than Boolean): v^i_{timeout} and initialize it to zero. We modify the translation of `updateTimers` to increment this variable when its timer variable equals zero, and modify the translation of the *message* rule to reset all v^i_{timeout} variables to zero. The property of interest is then:

$$\text{NoConsecutiveTimeouts}(t_i) \triangleq \Box \ \left(v^i_{\text{timeout}} < 2\right)$$

4 Experiments

In this section, we evaluate the performance of model-checking safety properties on detector models, with a focus on model checking latency. Low analysis latency is crucial because our tool warns customers of property violations while they are editing their detector model. Our type inference implementation runs with an average latency of 10 milliseconds on all the detector models in our experiments. Since type inference is much faster than model checking and can be successfully run on all detector models, we do not evaluate it in this section.

AWS IoT Events has a commercial feature [20] which uses the type checking and model checking described in Sect. 3. The feature's implementation first infers types using the type inference algorithm. Next, it translates the detector model into the Lustre language [21]. The translation of IoT Events into Lustre is straightforward and directly follows from the semantics presented in Sect. 3. The safety properties described in Sect. 3.1 are attached to the model, along with location information. Then the feature analyzes the model using the JKind [1] tool suite, an open-source industrial model-checker. If JKind invalidates a safety property, the feature decodes the location from the safety property and includes it in the warning.

To evaluate this implementation, we randomly selected 210 detector models previously analyzed by the commercial feature. We checked the five properties described in Sect. 3.1 in parallel on a c4.8xlarge EC2 instance running Amazon Linux 2 x86_64 using JKind version 4.4.1, with a timeout of 60 s.

Of the safety properties that we were able to translate to Lustre, JKind resolved 96% within our timeout of 60 s, with 80% completing in less than 10 s.

Table 1 shows that checking the *no-unreachable-action* safety property requires the most time to complete. The detector models analyzed in the evaluation include models for monitoring self-driving wheel chairs, monitoring device connectivity, humidity, temperature, pressure, oil level, oil temperature, doors, motion, refrigerator temperature, dough fermentation, and vehicle speed-sensing. They consisted of between 1–7 states and from 0–14 state changes. The *no-unreachable-action* safety property is checked on every action, generating an

Table 1. Performance of our model-checking tool against 210 detector models

safety property	avg. latency (milliseconds)	# completed	# translation failed	# timeout
no-unreachable-state	3544	176	28	6
no-unreachable-action	5586	171	28	11
var-always-set-before-use	2968	179	28	3
no-infinite-timer-expiration	2875	174	28	8
no-input-read-with-timer-expiration	5477	177	30	3

average of 17 safety properties per detector model, the most of any kind of safety property. This large number of properties to be checked on every detector model caused checking the *no-unreachable-action* safety property to have the highest average latency (5.6 s per analysis).

Table 1 shows that about 13% of the properties could not be translated to Lustre. In 2% of the detector models, translation failures arose due to type errors or incorrect use of the AWS IoT Events expression language in the detector model. The remaining translation failures occurred due to either: (1) use of operations not supported by Lustre, (2) no types being inferred for inputs or variables in the detector model, or (3) use of non-linear arithmetic, which is unsupported in JKind. Bitwise functions, strings, and array data types are supported in the AWS IoT Events expression language but not in Lustre. This language gap prevented us from translating 19 of the 210 detector models. Failing to infer a type for a variable in the detector model prevented translation of 6 of the 210 detector models. JKind's lack of support for non-linear arithmetic prevented model-checking 2 of the 210 detector models. We are actively working to support more functions, string and array data types, type annotations, and non-linear arithmetic in our model-checking of detector models.

5 Conclusion

Our analyzers have been running in the AWS IoT Events production service since December 2021. Since then, 93% of AWS IoT Events customers have used our implementation to check their detector models for well-formedness, without needing to have any knowledge of the underlying type checking and model checking. Our analyzers successfully complete for 85% of real-world detector models and we are actively working on improving this support as explained in Sect. 4. Overall, our implementation has reported well-formedness property violations in 22% of submitted detector models in the production service, with an average latency of 5.6 s. We find giving customers push-button access to fast verification without requiring any knowledge of the underlying techniques enables adoption of automated reasoning-based tools.

A Common Issues with Detector Models

Table 2. Issues seen in detector models from customer questions

#	Issue	# of instances
1	incorrectly scaling detector model	1
2	unreachable action	2
3	infinite loop	3
4	variable-used-before-set	3
5	input read on timer expiration	3
6	insufficient logging permissions	3
7	incorrectly typed expression	5
8	incorrect cross-service setup	8
9	missing simplifications	8
		36

As mentioned in Sect. 3.1, we surveyed customer detector models for generic correctness problems. We present the root causes of the problems from this study in Table 2. Incorrect scaling (#1) occurs when the customer does not set up their detector model to be instantiated correctly for every IoT device in their fleet. Infinite loop (#3) occurs when the detector model has an infinite execution path involving only timeout events and no external input messages. IoT models should be eventually quiescent if no external inputs occur.

Variable-used-before-set (#4) occurs when a variable in the detector's state is read from before being set to an initial value. AWS IoT Events does not require variables in detector models to be initialized.

A step through a detector can be triggered due to both a timer expiration or a new value being sent to the detector by the outside world. Input read on timer expiration (#5) occurs when a step, triggered by timer expiration, causes the detector to read from its input(s). This is a problem because customers often do not realize that such a read will return the last value sent to the detector by the outside world. Insufficient logging permissions (#6) occurs when a detector is not given sufficient permissions to produce logging output. Incorrect cross-service setup (#8) occurs when customers do not correctly set up data flow across services in AWS IoT. While unnecessarily complex detector models (#9) is not a correctness problem, it poses a significant difficulty to customers in maintaining their detector models, and so, we include it in Table 2.

Of these 9 root causes, we identified that type checking and model checking detected 5 root causes highlighted in green in Table 2. These 5 root causes were responsible for 44% of issues in our survey. Based on Table 2, we determined that the following correctness properties should hold over all detector models:

1. *Detector models must be well-typed*
2. *Every detector model state must be reachable*

3. *Every detector model action must be executable*
4. *Every variable must be set before being used*
5. *Input reads shall not happen on timer expiration*
6. *Detector model must not have infinite timer expirations*

We explain these properties further s in Sect. 3.1.

References

1. Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JKind model checker. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 20–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_3
2. Cho, C.Y., D'Silva, V., Song, D.: Blitz: compositional bounded model checking for real-world programs. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 136–146 (2013)
3. Baranová, Z., et al.: Model checking of c and C++ with DIVINE 4. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_14
4. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC/SIGSOFT FSE -1999. LNCS, vol. 1687, pp. 146–162. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48166-4_10
5. Karamanolis, C., Giannakopoulou, D., Magee, J., Wheater, S.M.: Model checking of workflow schemas. In: Proceedings Fourth International Enterprise Distributed Objects Computing Conference, EDOC 2000, pp. 170–179 (2000)
6. Clarke, E.M., Henzinger, T.A., Veith, H.: Introduction to model checking. In: Handbook of Model Checking, pp. 1–26. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_1
7. Clarke Jr, E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model checking. cyber physical systems series (2018)
8. Bradley, A.R.: Incremental, inductive model checking. In: 2013 20th International Symposium on Temporal Representation and Reasoning, pp. 5–6 (2013)
9. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: from refutation to verification. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_2
10. Newcomb, J.L., Chandra, S., Jeannin, J.-B., Schlesinger, C., Sridharan, M.: IOTA: a calculus for internet of things automation. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, ser. Onward! 2017. New York, NY, USA, pp. 119–133. Association for Computing Machinery (2017). https://doi.org/10.1145/3133850.3133860
11. Liang, C.-J. M., et al.: SIFT: building an internet of safe things. In: Proceedings of the 14th International Conference on Information Processing in Sensor Networks, ser. IPSN 2015. New York, NY, USA, pp. 298–309. Association for Computing Machinery (2015). https://doi.org/10.1145/2737095.2737115
12. García-Herranz del Olmo, M., Haya, P.A., Alamán, X.: Towards a ubiquitous end-user programming system for smart spaces. J. Univers. Comput. Sci. **16**(12), 1633–1649 (2010)

13. Ur, B., McManus, E., Pak Yong Ho, M., Littman, M.L.: Practical trigger-action programming in the smart home. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI 2014. New York, NY, USA, pp. 803–812. Association for Computing Machinery (2014). https://doi.org/10.1145/2556288.2557420
14. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
15. Amazon Web Services, Inc., Find answers to AWS questions about AWS IoT Events, AWS (2021). https://repost.aws/tags/TANsxSwnClQ_Wfh-uklXi7hQ
16. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL 1982, New York, NY, USA, pp. 207–212. Association for Computing Machinery (1982). https://doi.org/10.1145/582153.582176
17. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978). https://www.sciencedirect.com/science/article/pii/0022000078900144
18. Hindley, R.: The principal type-scheme of an object in combinatory logic. Trans. Am. Math. Soc. **146**, 29–60 (1969). http://www.jstor.org/stable/1995158
19. Gay, G., Staats, M., Whalen, M., Heimdahl, M.P.E.: The risks of coverage-directed test case generation. IEEE Trans. Softw. Eng. **41**(8), 803–819 (2015)
20. AWS IoT Events, Troubleshooting a detector model by running analyses (2021). https://docs.aws.amazon.com/iotevents/latest/developerguide/iotevents-analyze-api.html
21. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proc. IEEE **79**(9), 1305–1320 (1991)