Samsung SmartThings Application Data Consistency Analysis Tool Documentation

Contributors: Atlas Kaan Yilmaz, Kelly Shaw

# What the Tool is Looking For

This tool is looking at groovy applications that run on the Samsung SmartThings platform for possible errors in data consistency with usage of state variables and undesired outcomes that impact the user like false actuation of devices and notification mishandling.

# SmartThings Concepts In Play[[1]](#footnote-0)

## General Overview

The above image is a breakdown of the SmartThings platform. We will be providing a short explanation of the layers but the ones we will be most interested in are the relations between the devices, subscription processing, SmartThings Application (SmartApp) execution, and server-side storage of data.

Devices are connected to the SmartThings network through a hub. The data sent from the hub is first handled by the device type handlers in the cloud. These device handlers are different from the handlers we mention that are present in the SmartApps we look at.

Device handlers call the associated event in the subscription processing and push the event onto the event stream to be processed by the SmartApp. In the SmartApps, the applications have methods that are subscribed to specific events from certain devices that the user selects and assigns in their setup of the application. When the said event enters the event stream and is processed, the application calls the method associated with that event in the SmartApp.

The SmartApp is running in the cloud only when the event is being handled, so any data that should persist across handlings must be stored in an external storage. This external storage is the state variable, which is stored in an external database. State information is read from the external storage every time before a handler that uses state is executed.(See more specifics are below.)

After the execution of the SmartApp event handler is done, any modifications to either state variables or device state changes are propagated through the platform.

The SmartThings platform implements eventually consistent guarantees for updating data. Consistency guarantees how data is updated when there are multiple updates and when multiple copies of the data exist in the system. In eventually consistent systems, there’s no guarantee for the ordering of the requests. The requests can be executed out of order. A request may also return an out of date value before the system reaches a stable state where all the copies of the data across the system have the same value.

The SmartThings platform also implements asynchronous execution and method handling. What this means is that once an event to be handled is fired, it does not stop everything else from running. The platform implements a “fire and forget” approach to handling commands. Meaning, when you execute a command, it is expected to be handled, but there’s a follow-up to check its completion. This approach does not guarantee the execution of your commands in the order you’ve sent them. They can arrive for execution out of order. For example you might turn a switch on but another app might sneak in and turn it off.

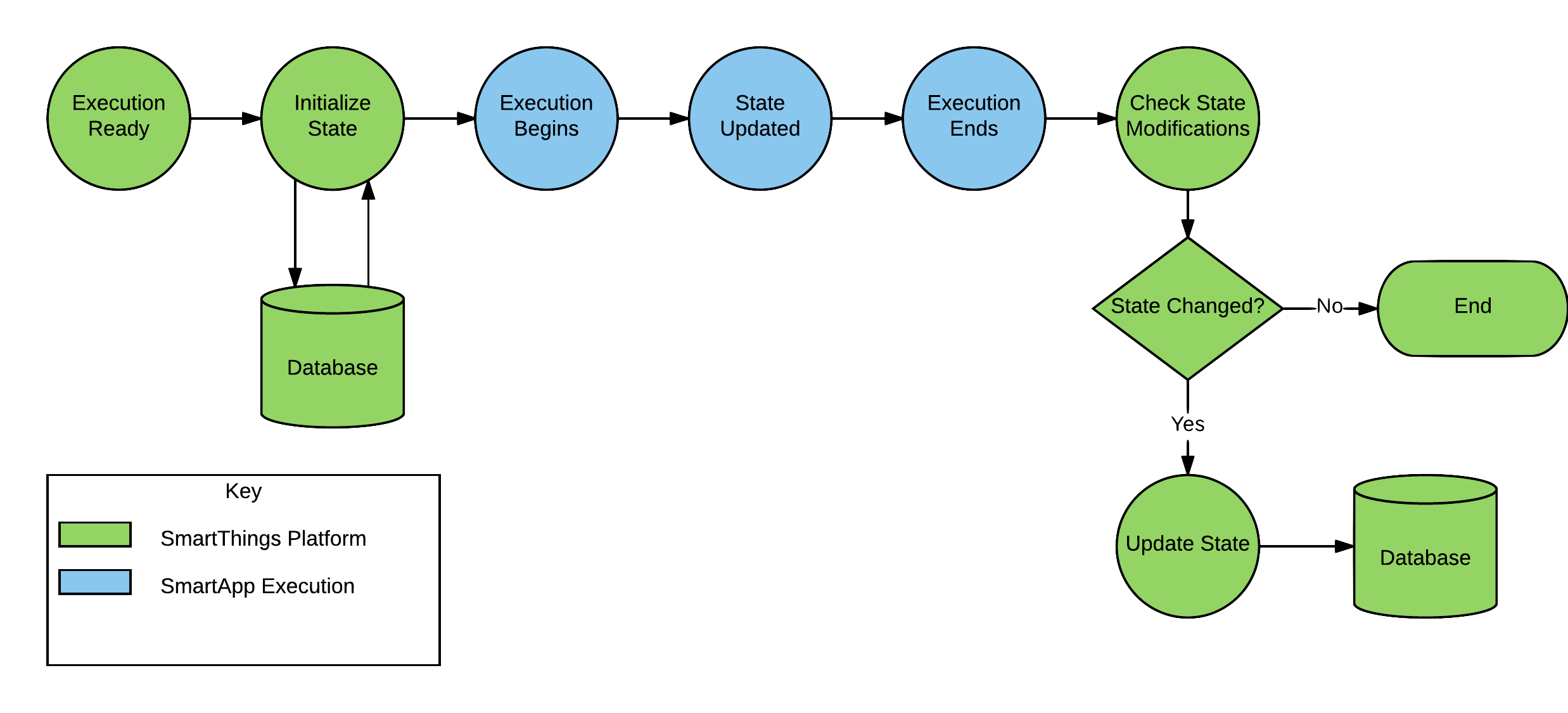
For more information, visit <https://docs.smartthings.com/en/latest/architecture/index.html>

## Challenges

No guaranteed order of how event handlers get executed. No serialisation and can also execute concurrently. They’re accessing and modifying shared state and devices.

## State

State is data associated with a SmartThings application that is stored in persistent storage that an application can read and modify. The State is fetched from the database at the beginning of a method's execution, a copy(copy A) is stored in the SmartThings Platform and a local copy(copy B) is sent to the handler. Then, modifications to the state fields by that handler are made to copy B. At the end of execution of a method handler, the final state values of the copy B are checked against copy A that was stored in the SmartThings cloud platform. If the values are different, the value of the modified state field is written to the database.



[[2]](#footnote-1)

## AtomicState

Similar to the state, atomic state is used to store data associated with a SmartThings application in persistent storage. Also similar to state, a local copy of that data is fetched at the beginning of execution of a method handler from the database. What’s different is that AtomicState updates the database right after a modification is made to the atomic state field, not waiting for the entire method handler to finish executing before sending the modification to the persistent storage.

## Scheduling

A method can be scheduled to execute at a later time using a variety of functions provided by the SmartThings API; these can be found [here](https://docs.smartthings.com/en/latest/smartapp-developers-guide/scheduling.html?highlight=runin). By default, a scheduling function overwrites the standing scheduled calls, but by setting the overwrite argument to false in the function call this can be turned off and instead the calls would be queued. There exists a maximum for how many calls can be queued this way.

## Event

Events are information packages sent by devices to the SmartApp and to the handler subscribed to the device.

## Input

Inputs are gathered in the preferences section of the SmartApp from the user using the SmartThings WebUI or mobile application. They can be values, button states, device bindings, or modes.

# A High-level Breakdown of How the Tool Works

The tool reads the abstract syntax tree (AST) of the input SmartThings Groovy application and by traversing the nodes of the AST it gathers the information relevant to the analysis. This information consists of inputs and event subscriptions as specified in the preferences, setup portions of the SmartThings application, conditional branching information, any state field accesses, method calls within the application, and method calls that modify device states; all of which are stored in the handler and method data structures within the tool. After the information gathering is done, the tool cross-analyses the event handlers in an input application against themselves and each other to detect when there are risks when the two of them are executing concurrently. Once the analysis is complete, the tool outputs all pairs of handlers that might potentially have issues when the pair executes concurrently.

# A Short Explanation of Data Inconsistency

The part of data consistency we’re concerned with in this analysis is race conditions that can happen when multiple handler methods are executing concurrently.

When two or more methods that modify a common data have started executing simultaneously and the modifications to the data are not propagated until the end of the method handlers' execution, then the method that finishes last will be the update that persists in the database. Meaning the methods will be in a race to finish their execution and the one that finishes first will result in a lost update as the last to finish will most likely overwrite the others' updates.

Here’s an example for a state write against state write concurrency example that explains possible consistency issues:

Let’s say we have Method B that sets a variable state.var to the value “B” and Method A that sets var to the value “A”. Due to the asynchronous model of the SmartThings platform, the methods called in order can start their execution concurrently or out of order. Let’s say, Method B finishes before Method A and sends a request to set the database copy of the state.var to “B”. After Method B completes, Method A finishes and sends a similar request to set state.var to “A”. This can happen in such a way that Method B is called **after** Method A from the device, but Method B finishes execution and propagates the updates to the state variable before Method A. In which case the desired final state would have been the modifications done by Method B, var = B, but that update gets lost when Method A finishes its execution and overwrites the desired data to var = A.

Here’s an example for a state write against state read concurrency example that explains possible consistency issues:

Let’s say there’s a handler A that reads the state variable state.var, and if the value of the state.var is on it switches a lighting device on and switches the device off when the value is off. And now let’s get a handler B that sets the value of state.var variable to off. Imagine a hypothetical case where the initial value of state.var is on. The event that triggers handler A is fired followed by the event for handler B. Due to the asynchronous aspects of the platform B starts execution before A. It creates a local copy of the state, modifies the value of state.var in the local copy, and updates the server copy before A gets the state from the database and reads it. What happens here is, handler A was called when state.var was set to ON and the handler’s desired outcome would be to turn the lights on. However, handler B sneaks in and updates the value of the state.var, causing handler A to execute with state.var set to OFF and turn the lights off, the opposite of the desired outcome when looked at the sequential firing order of the events.

# A List of Possible Outputs and What They Mean

## State

*A variable value is set to a state field:* When a variable value is written to the state field, the value of the local variable can be different from one instance of the handler to the next. So when multiple calls are fired to the same handler, it can produce different outcomes for the state variable dependent on the calculation of the variable within the handler. Constant values on the other hand do not change from call to call and are deemed safe as they produce the same outcome for the state field.

*Multiple unscheduled writes to the same field:* It is deemed unsafe when multiple modifications with different values are done to the state field from the handlers, since multiple modifications with concurrent calls to the handlers might result in race conditions for the state field. This error message will not occur when the state updates happen in a method call that is scheduled with overwriting turned on, since with scheduling with overwrite there will only be a single execution of the method that writes states.

*Modification is inside a conditional block dependent on time information:* Method handlers can have pieces of code that are executed conditionally via the use of control flow branching. If a modification is inside one of these blocks whose conditional check depends on a time related information, e.g. the current time is inside a certain time frame, then the handler can have different outcomes depending on their timestamps and create race conditions for the modification of the state field.

## User Impact

### Device

*Variable value is passed to the device modification call:* A variable value can change from one instance of a handler to another. In passing a variable value as a parameter to a device modification, there can be different final states of the device dependent on the internal calculation of the variable value and result in different outcomes to the handler and possible race conditions.

*Multiple unscheduled modification calls are made to the same device:* It is deemed unsafe when multiple different modifications are done to the device from the handlers, since multiple modifications with concurrent calls to the handlers might result in race conditions for the device state. This error message will not occur when the modification is the same or when the device modification happens in a method call that is scheduled with overwriting turned on, since with scheduling with overwrite there will only be a single execution of the method that modifies the device state.

*Modification is inside a conditional block dependent on time/state/event information:* The time, state, and event information can change from one instance of the handler to another, even when they might be executing concurrently. Hence a handler that results in different outcomes dependent on time/state/event information might incur a race condition when called against itself or other handlers and is unsafe for device state modification consistency.

### Notification

*Multiple notifications sent without scheduling:* If a notification or message sending function is called without a scheduling with overwrite, when a handler is executed concurrently by itself, the handler will send duplicate notifications to the user. If it was used in a scheduling method, thanks to the overwrite capability of the scheduling functions, only a single notification would be sent. This is considered a minor user impact, since it does not actually pose any security and device related risks.

### Scheduling

*Scheduling and unscheduling conflict:* This error message is given when in a pair of handlers under analysis, one of them unschedules an execution the other handler schedules. This can possibly unschedule and erase many modifications to both state and device depending on the outcome of the race condition, or schedule a series of modifications that would otherwise have been unscheduled.

1. Image: <https://docs.smartthings.com/en/latest/architecture/index.html> [↑](#footnote-ref-0)
2. <https://docs.smartthings.com/en/latest/smartapp-developers-guide/state.html> [↑](#footnote-ref-1)