Improving SVSHI's Verification Method

Ladina Roffler
Project partner: Aymeri Servanin
Supervisor: Samuel Chassot, Prof. George Candea
EPFL DSLAB

June 10, 2022

Abstract

In order to check recurring, time dependant properties such as "a boiler is above 60 °C for one hour every day", the verification strategy needed to be changed from "given a valid state, an app cannot put SVSHI in a invalid state" to "given an invalid state, apps will make the state valid in the future". To do so, SVSHI needed to be made time aware. Several methods to change the verification where tried, a symbolically constructed list, adding functions to stop the app execution until a certain condition and constructing a condition manually to pass to a z3 solver. The method using the z3 solver was selected as the new verification method, since it was the only feasible approach. The changes to the verification required some changes to the runtime of SVSHI. Apps are now combined to one single app that is run when SVSHI is run.

1 Introduction

The high-level goal of this project was to improve SVSHI's verification capabilities.

1.1 Background on SVSHI

SVSHI stands for "Secure and Verified Smart Home Infrastructure". It was developed by Samuel Chassot and Andrea Veneziano and aims to provide an infrastructure for developing smart home applications that is easy to use, reliable and secure. [CV22] The developer implements the application in python and the verification of the app's behaviour is done automatically at install time.

Every time a developer wishes to install a new application to the SVSHI engine, it is first verified by crosshair. Crosshair is a tool to analyse a python program using symbolic execution. [Cro] It provides a function called `check`, which ensures that given some preconditions, after running a function the function's postconditions are met.

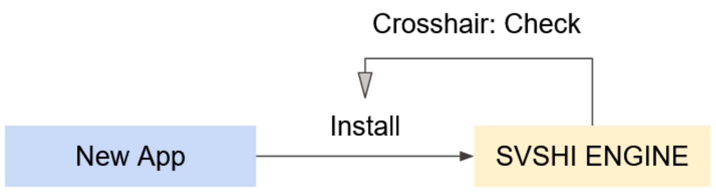

Figure 1: When a developer wants to install an app, the app is first verified using crosshair.

Every SVSHI app has two functions, the `invariant` function and the `iteration` function. The `invariant` function is used to impose conditions on apps that need to be true at all times. The `iteration` function is used for implementing an app's behaviour. At verification time, crosshair's check function is used on the `iteration` function. As a precondition, it is assumed that before running `iteration`, the `invariants` were true. Crosshair then checks that after running the `iteration` function, the `invariants` are still true.

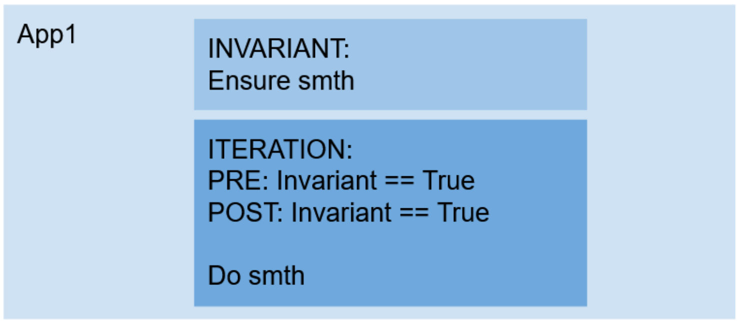

Figure 2: The structure of an app's code.

This means that previously, SVSHI's verification ensured that given a valid state of the engine, an app could not put SVSHI into an invalid state. This works for many cases but not all, as will be explained in the next chapter.

1.2 The Boiler Example

Legionella is bacteria that can be found in water. When ingested it can cause symptoms that are similar to pneumonia. To prevent this illness, the Swiss government requires that boilers should set the water temperature to be above 60 °C for at least one hour every day in order to kill the bacteria. [BAG]

Let us assume we write an application that fulfills this requirement. At 22:00 the developer sets the boiler to 60 °C until 23:45. He then sets it back to 40 °C, where it will remain until 22:00 on the next day.

Although one can quickly see that this application should pass verification, since the application only results in a valid state after 23:00 and is in an invalid state until that point, SVSHI's previous verification method would fail the application and not install it.

1.3 Project Aim

The aim of this project was to allow for SVSHI to handle situations such as the previously mentioned boiler example. In order to do this, the verification strategy needed to change. The strategy needed to be moved from "given a valid state, an app cannot put SVSHI in an invalid state" to "given an invalid state, ensure that in the future, the apps will set SVSHI to a valid state".

To achieve this goal, the SVSHI engine needed to be made time aware, which is detailed in section 2. The tried verification strategies, including the ultimately chosen one are presented in section 3. Section 4 details the changes that were required to SVSHI's runtime and finally the report is concluded in section 5.

2 Time Integration

In order to be able to check time dependent conditions, the SVSHI engine required a notion of time.

SVSHI already keeps state for every application's variables using a class called `AppState`. It also keeps state of sensor values in a class called `PhysicalState`.

A class was added called `InternalState`. It was originally created to keep track of the engine's time but can be expanded to include anything that the engine needs to be aware of. Unlike `AppState` and `PhysicalState`, a developer does not have access to the variables of `InternalState`.

A second class was added to SVSHI called the `SVSHI API`. It provides already implemented functions that a developer can use as a black box. It includes functions such as `get_hour` or `get_day_of_the_week`, thus providing a developer indirect, controlled access to the `InternalState`.

3 Verification Approaches

Three possible solutions were implemented, only the last of which was a feasible approach.

3.1 Symbolic List

One approach was to create a list containing tuples of time and the corresponding state. The list would never actually be created physically, instead its elements were heavily constrained with preconditions. The list would need to meet certain conditions, like having two consecutive items contain states where the boiler is above 60 °C.

When asking crosshair to check whether the list meets the required conditions, the list would then be built symbolically in such a way that it meets all the given constraints. If the list meets the conditions, the app being verified can be installed.

This approach worked for lists containing only very few items. The time required to verify the conditions grew exponentially as more items where added, and reached a timeout for lists with more than 10 elements. The same behaviour was observed when using a dictionary instead of a list.

The underlying issue is that crosshair is ill-equipped to deal with containers of variable length. Such cases lead to path explosion very quickly. Given the fact that we not only required using lists of variable size, but also of much greater size than just 10 elements, this approach was not feasible for our purposes.

3.2 Wait-Based Verification

Another approach was to add two functions to the `SVSHI API`, a function called `wait` and a function called `wait_sensor`.

The `wait_sensor` function suspends the app's execution until a sensor meets a specified condition. This is needed to handle the fact that physical systems require time to change state, so a boiler would not reach 60 °C immediately after being set to that temperature. At verification time, this behaviour was simulated by having the boiler sensor's temperature read-out increase linearly for a given time delta before reaching its steady state at the set temperature. The `wait` function suspends the app's execution until a specified point in time.

At verification time, when one of the two API functions is reached, the `InternalState`'s time is manipulated. This allows us to jump to the future and analyze the state at the future point in time. At runtime, the functions would have been implemented using python asyncio calls.

Although this approach worked, unfortunately it would have defeated SVSHI's event based nature, which is why it was discarded.

3.3 Current Implementation

The condition that should be verified is the following:

$$\forall S(\exists t : f(t,S).boiler > 60 \land \forall t < t' < t + 1h(\forall S'(If S'.boiler > 60 then f(t',S').boiler > 60))) \quad (1)$$

In words this means that for all states S, there exists a point in time t where the boiler is above 60°C and for all points in time t' that are between t and t plus one hour, the state remains at 60 °C.

It is not possible to verify this condition with crosshair, since crosshair does not support the notion of a "there exists". However, internally crosshair uses a z3 solver which does provide a "there exists".

The idea of this approach is to manually create the condition expressed by equation 1 and then to pass it to a z3 solver to verify whether the condition is satisfied or not.

To create the condition manually, a f(t, S) needed to be obtained. This is done by using a crosshair function called `cover`, which extracts all paths from a function. If all the apps are combined into one function, `cover` can be used to extract all the paths from the SVSHI engine. These paths are then concatenated to a z3 readable condition.

To build the time dependant components of equation 1 a new function called `check_time_property` was added to the `SVSHI API`. `check_time_property` takes three arguments, a frequency, a duration and a condition. For the boiler example, the arguments would be one day, one hour and boiler > 60°C.

During verification, when a `check_time_property` function call is encountered in the `invariants`, it is extracted and converted to a z3 readable format. Together with the extracted and concatenated paths, the condition in equation 1 can be created.

The functions to create the condition in equation 1 and to use the condition in a z3 solver were combined into a python module. At verification time, this new module is now called instead of calling crosshair's `check` function.

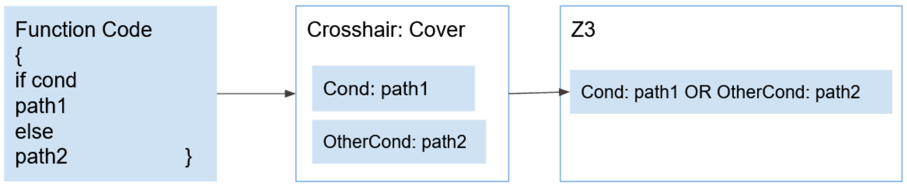

Figure 3: The workflow of the new verification method. Paths are extracted then concatenated to a z3 readable condition. Z3 checks whether the condition is satisfied.

4 Runtime Changes

The changes made to SVSHI's verification required some changes to SVSHI's runtime.

4.1 System Behaviour Function

In order to extract all paths from the SVSHI Engine at verification time, all of the app's `iteration` functions needed to be combined into one single function. This function is called `system_behaviour`.

Previously, every app was run one by one, either when something in the `PhysicalState` changed that affects the app, or periodically according to the app's timer. The switch from singular `iteration` functions to the combined `system_behaviour` function means that individual apps are no longer needed. There is only one app, called JointApp, which calls the `system_behaviour` function every time something in the `PhysicalState` changes. If apps are present that have a timer, the JointApp is run periodically, using the minimal timer of all the apps.

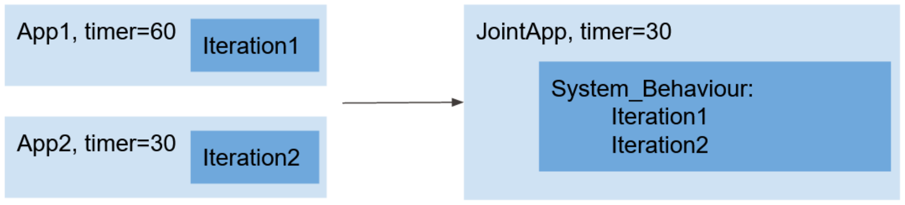

Figure 4: Apps are no longer run separately. Only one app is run which calls the joint iteration functions. The timer is set to be the minimum timer of all the apps.

4.2 Check_time_property calls

The `check_time_property` calls used for verification need to be handled at runtime as well. To do so, each function needs to store state.

This was solved by adding a new class `CheckState`. `CheckState` includes four variables, that are needed to keep state for one `check_time_property` call:

- An integer to store the time the frequency last passed.
- An integer to store the time the condition was last valid.
- A boolean to mark whether the condition is currently valid.

- A boolean to mark whether the condition was valid for the given duration in the given frequency.

Each `check_time_property` call is assigned a `CheckState` object, which is stored in the `InternalState`. Every time the call is made, the object is updated. `check_time_property` returns `True` so long as the frequency has not passed yet. Once the frequency has passed, it either returns `True` if the condition was valid for the given duration, otherwise it returns `False`.

5 Conclusion

In order to check recurring, time dependant properties such as "a boiler is above 60 °C for one hour every day", the verification strategy needed to be changed. A shift needed to be made away from the method "given a valid state, an app cannot put SVSHI in a invalid state" towards the new method "given an invalid state, apps will make the state valid in the future".

To achieve this shift, the verification is no longer done by crosshair's `check` function. Instead, the required condition is created manually and then passed to a z3 solver which checks that the condition is satisfied.

Part of the condition is constructed by extracting SVSHI's paths with crosshair's `cover` function. The other part is constructed by extracting `check_time_property` functions, which is a new function added to the `SVSHI API`, from the `invariants`.

Several new classes were added to SVSHI. The `CheckState` class keeps state for one `check_time_property` call at runtime. The `CheckState` objects are stored in a new class `InternalState`, which is also used to give SVSHI a notion of time. Finally, the new class `SVSHI API` provides already implemented functions that the developer can use as black boxes in their code.

Since extracting all of SVSHI's paths required joining all of the app's `iteration` functions into one joint function, SVSHI now only runs one app, a JointApp, which calls the joint iteration function. If apps have timers, then the JointApp runs periodically using the minimal timer of all the apps.

References

[BAG] Bundesamt für Gesundheit: Legionellose. https://www.bag.admin.ch/bag/de/home/krankheiten/krankheiten-im-ueberblick/legionellose.html. Accessed: 2022-06-09.

[Cro] CrossHair Documentation. https://crosshair.readthedocs.io/en/latest/. Accessed: 2022-06-09.

[CV22] Samuel Chassot and Andrea Veneziano. Svshi: Secure and verified smart home infrastructure, January 2022.