Extending the verification of SVSHI applications with time-sensitive constraints

Aymeri Servanin

June 24, 2022

Abstract

In this paper, we extend SVSHI's verification process to time-sensitive applications. Two approaches were explored, both of them using CrossHair, a python concolic execution tool. The first approach introduces the possibility to make application sleep. The second one converts functions into z3 expression to verify any properties. The last approach was taken as it keeps the event driven architecture.

Contents

Acknowledgments

I would like to thank Samuel Chassot and Professor George Candea for their guidance during this project. A special thanks to Ladina Roffler for participating in this project. Also all the SVSHI team of this semester (Loïc Montandon, Léo Alvarez, Isis Daudé) and the PhD students (Can Cebeci and Solal Pirelli) for their advice.

1 Introduction

The core statement of SVSHI (Secure and Verified Smart Home Infrastructure)[1] is to build a smart home hub using apps that are formally verified. SVSHI checks if the application and its stated properties never conflict with other applications. SVSHI performs verification by using CrossHair [2]. It is a python library that performs concolic ("a portmanteau of concrete and symbolic"[3]) execution to test the stated contracts. Before this paper, SVSHI used the Check function on contracts written by the app developers.
The contract would be enforced as pre and post-conditions. Therefore SVSHI checks if, after an execution, the application remains in a valid state. This could cause an issue when SVSHI is initialized or when entering a transition state.
To demonstrate this, let's take the example of the first SVSHI's whitepaper [1], "If the switch is on then the light is on". Once the button is pressed, a telegram will be sent to SVSHI and the state of the light is off but the state of the switch is on. This is not tested when formally verifying the application, as there can't be an invalid state before calling the application.
Another point we would like to extend is the compliance of smart homes with the laws. Because the goal of SVSHI is to make a smart home as safe as possible, it must comply with the local regulations. For example, there are regulations on boilers in Switzerland. They must be heated up to 60°C for at least one hour every day.[4]
With this regulation in mind, we attempt to make this rule added to SVSHI. This means that if the application that is verifying this rule is installed, we know there is at least one application that makes this smart-home compliant with this regulation.

2 Quick introduction to SVSHI internals definitions

This section is a summary of the important data structures and processes of SVSHI. More details can be found in [1].

Physical state A class that contains all the Group Addresses (GA). These Group Addresses represent the last known state by SVSHI at runtime and

the actual state when verifying the application.

App State A class that is used to hold values between two executions of the application.

InternalState A class that holds values that cannot be accessed by the developer.

SVSHI App structure and workflow The developer creates an application in SVSHI by providing a list of the devices in a JSON file. This file is used by SVSHI to generate the template of the application. It is made of two functions, the **invariant** and **iteration** function.
The **iteration** function is the core, where devices are manipulated. The **invariant** is where the properties of the application are stated. It must be satisfied every time an application is run.
Developers install their applications by compiling them using the CLI or GUI. This step ensures that there are no conflicts with other applications. If none of them occurs, then the application is installed. Otherwise, it is considered a "fail" and does not install. A detailed explanation of a failure is given in the next section.

Defining a "fail" using CrossHair A "fail" is when CrossHair finds a counterexample in the stated properties. That is when an invariant of an application is broken after the execution of another application.
For example, we define a function with a post condition that states the return is strictly greater than 0.

```
def f(i:int) -> int:
    """
    post: __return__ >0
    """
    if i>0:
        return i
    else:
        return -1*i
```

When running check, it gives a counterexample and we have this output message "error: false when calling f(i = 0) (which returns 0)" and a non-zero exit

code. We consider this application as a "fail". In SVSHI, functions, and conditions are more complex, but if a counterexample is shown the application does not install.

3 Related work

Regarding the verification of smart-home applications, most of the related work is already mentioned in the first SVSHI paper [1]. Modeling time-sensitive application in python or more generally verifying Python code is not widely covered. Regarding the time aspect, papers such as [5] uses UPPAAL [6] to verify part of Python's core.
For the verification of python code, only a few tools are available. SVSHI uses CrossHair [2] and can perform formal verification under specific conditions mentioned here. Additionally, there is Nagini [7], a front end tool for Viper. It stands for Verification Infrastructure for Permission-based Reasoning and "is a language and suite of tools developed at ETH Zurich, providing an architecture on which new verification tools and prototypes can be developed simply and quickly."[8]. This tool requires more advanced knowledge in making proofs, therefore a developer would have to be trained to use this tool. However, the goal of SVSHI is to make verification as easy as possible for the user.

4 Assumptions

Before starting to enhance the verification process, we need to define SVSHI's limitations in the verification process. We assume that SVSHI always holds the correct and latest value of the real world's physical state.
We need to assess first the differences between the verification, which simulates the environment and the real world. SVSHI can communicate with any KNX device which can have any behavior. Because of this multiplicity, at verification, we can only take a simplistic approach.
An application can set and read the values of devices. All devices receive this new value in a short time but some devices can take time to reach this new value. For example, if we set the room thermostat to be at 22°C, the target is set within seconds but we cannot expect the room to be at 22°C immediately. With this example in mind, we make the following assumptions.

Additionally, given the new structure, we assume the function to be idempotent. Therefore if it changes state, it should be enforced as long as required.

SVSHI's state is considered to be the correct one at all time We assume what SVSHI reads and writes on devices is the actual physical state. SVSHI isn't aware of the behavior of each device. For example, when SVSHI is setting a switch to be on, we assume it will be physically on. It may sound simple as stated, but this means we don't handle failing devices. Therefore if the switch is never on when it's being set to this value (it could be that the wires were swapped), we sadly can't do anything about it.
Taking our boiler example, the developer must be aware that it would take time to heat up. A boiler that is being set to 60°C for exactly one hour is unlikely to be compliant given that the operational temperature might be lower, thus taking time to heat up.
Furthermore, we only model explicit dependencies. For example, we can set a value of one device by reading the other one (an app makes a living room thermostat to match the kitchen's thermostat). However, if a device has an implicit impact on the other one we do not model it (if the thermostat has a temperature sensor, during formal verification it can read any values at any time even if after some time the thermostat should approximately match the thermostat).

Only SVSHI sets the value of the sensors If we don't control what is being set then we could not have any guarantee on the system.

Refreshing rate of the device is instantaneous Once the values are being set, they are not modified until an app changes them. This creates a gap with the real world. If we take the example of heating a boiler, we rely on what information the sensor provides us. Therefore if the sensor sends an update every hour, there will be a gap between the actual temperature and the last known one.

5 First approach - Adding sleep to keep track of previous events

Having now defined the assumptions, we can start finding a solution to extend the verification to time-sensitive applications. In this section, we describe how "sleep" is introduced in SVSHI. This allows to observe the duration of a device's state.

5.1 Basic application without changing SVSHI

The first approach we try with the existing SVSHI is an application that is overriding the behavior of other applications. The application would guarantee that the device is following the given regulations. It applies the regulations by altering the device's state.
Suppose we want a boiler to be at 60°C for one hour every day. Then one would write an application that is enforcing this by monitoring if the boiler has been set to 60°C for one hour continuously. If not, then after 23 hours have elapsed the boiler would be in an override mode by setting a flag and set the boiler to 60°C.
Code is available in the appendix A.
If an application is setting the boiler at a lower temperature than 60°C when the override flag is set, it would fail at compile time and wouldn't be installed.
The limitation of this application is that it is device-specific, is actively modifying the device and it would require creating an application or a template per device.

5.2 Creating an application that only monitors the device

We create an application that only monitors devices so that the implementation is not depending on one specific device. We can take the logic of our previous application but entering in a "fail" mode when the application was not satisfying the given constraint. Back to our example, instead of actively correcting the behavior of the device, we would just set a flag to make the application fail. On the invariant, if the fail flag is set then the application would not compile.
However, this would mean that the monitoring application would also fail,

as CrossHair would also test when the application enters the fail mode. We need to find a way to make other applications fail but not the monitoring application (while keeping the invariants to help verify the app in itself).
The approach we take is to split the invariants into two: one for the app that would help test and formally verify it . Another where it would fail the compilation of other applications that are not meeting the correct requirement. Then if the app is only monitoring, we set a condition before code generation. This specifies that the application is only monitoring. We use the JSON that stores parameters to generate the SVSHI application. This setting adds a new function called "invariant_rules_app" where it would be put as post conditions of all other applications that are not monitoring apps. This would work but only to some extent: a valid application would fail, and CrossHair would give a counterexample when the application would fail when switching from 60°C to a value below. This is because CrossHair is running on a single point in time and isn't aware of what happened before this.

5.3 Tracking previous executions

To be able to reproduce previous executions of the application, we introduce the ability to "wait" for a given task. This way we know for how long the status of the device was held.
The developer would write a "wait(time)" or "wait_sensor(sensor, condition, timeout)" that would pause the execution of the application until the wait is over. In the background, the application would return the value of the physical state to propagate the updated state and then go to sleep. The *wait_sensor* function puts the application into sleep until the given condition or timeout has been reached. This is helpful at verification time to model delay and at runtime device failures.
Because we are able to modify the code, at compile time we replace the definition of *wait* by increasing the time with the given argument.
However, CrossHair still gave us a counterexample as the counters of the monitoring application weren't updated. An example would be that at the start of the execution, the monitoring app is in an invalid state, then the boiler is at 60°C for one hour and then the app finished its execution. However, the counters weren't updated.

5.4 Updating the monitoring app at compile time

Given this limitation, before and after every *wait* we would call the monitoring app so that it updates its counter. Therefore, given a valid app, the monitoring app would register that the device state and for how long it is being held. Additionally, we need the application to share the same time. Calling the monitoring app every time a *wait* comes with a drawback: CrossHair has to do an extra task to solve the constraints of the monitoring app, even if this not necessary as it is tested before.

5.5 Adding time on SVSHI applications

We need that all applications have a shared notion of time. Otherwise, while performing the verification, multiple apps that are executed sequentially could be at different times.
Getting time was only possible by using an unchecked function, as imports in iteration were not allowed. For the developer, this usually means returning time.time()[9] in an unchecked function.
For the verification of time sensitive applications, we need to use one single shared variable for every app. Fortunately, we have an object shared by every app, the PhysicalState.
At verification time we have an int value that is determined by CrossHair.
Note: It was suggested by Samuel Chassot to create a class called InternalState to store variables that are used only by SVSHI and not the developers. Therefore we moved the time in this new class.

5.5.1 Defining virtual time

As CrossHair can choose any integer, we need to constrain it to sensible values. We choose to start time at 0. Therefore, time would be relative, starting at 0. Every time t is strictly greater than 0, this means the application was executed at time t+x.

5.6 Integration on the runtime function

Now that applications have the ability to sleep, we need to run them concurrently so that one application that goes to sleep doesn't block the execution of all applications (and SVSHI as well).
Additionally, every time a function enters a wait, it should return its updated

physical state. Therefore at runtime, instead of having the *wait* moving time forward, we call asyncio.sleep for the given duration and put the PhysicalState in a queue. Once an item is in the queue, we check that the invariants are satisfied and propagate the new state to KNX and we also update the copy of the physical state SVSHI keeps.

5.7 Conclusion of the first approach

We have a working solution that allows verification of time sensitive properties. However it has a few drawbacks: First, it does not follow the event-driven model that SVSHI is built on. Functions are expected to run and quickly return a value. Secondly it constrains the developer to think about the time being relative. This could cause confusion, especially since SVSHI can be started at anytime of the day.

6 Second approach - Modeling functions into z3 expressions

In this section, we discuss the second solution to verify time-sensitive applications. We model the iteration and invariant functions into z3 expressions using CrossHair. Then we use z3's SMT solver to see if the invaraiants constraints are satifsfiable given the iteration function. We also provide time through SVSHI's API.
With the previous solution being put aside, we redefine what we want to verify. We reformulate the verification of the boiler and found a mathematical formula : For all inputs, there exists a continuous time window of one hour where the boiler is at 60°C. The issue we face again is to give a certain duration of the state. Therefore we start to think about how we could represent these previous states.

6.1 Representing previous states by a list

CrossHair models the trace of the system by creating a symbolic list of valid traces to measure the duration of the device's state. We put this list and find a new execution that led to a counterexample. This would form a valid trace where during a specified timeframe the property would have never been satisfied.

This is also what Philip Schanely suggests, for a limited trace. [10] However constructing the list leads to path explosion, and running CrossHair with bigger lists quickly leads to timeouts.
We decide to not keep this approach as it would not scale.

6.2 Current limitations with CrossHair

Looking back at our formula it would be ideal if we could model the existential quantifier using CrossHair. However, it is not CrossHair's goal to model mathematical formulas, so we have to dig deeper to see if we could use CrossHair's engine to verify our properties. Currently our applications are built only with simple types (not complex classes that can be difficult to model) and we know CrossHair uses symbolic variables to execute the function. Additionally, CrossHair solves path constraints using z3.
z3 [11] is a SMT solver and we could use it to verify if our property is satisfiable. Therefore, if we're able to take all the path constraints and the symbolic return, we could convert them using z3 and thus be able to solve any equations in first-order logic from a given function.

6.3 Adding time to SVSHI to ease the verification

We add separate bounded variables for every time unit (minute,hour,etc..) through SVSHI's API.
When searching for a new solution, we tried to model a simple application that performs a task every day. Our original definition of time is a UNIX timestamp. Therefore to do a task every day, the function was using non-linear arithmetic to get the day. This led to z3 timing out as it is non decidable.
Therefore we need a way to avoid using non-linear arithmetic. We choose to split every time variable and bound them. This way we cover all possible times but without using modular arithmetic.
We now have multiple variables on the internal state: minute, hour, day, week, month. We did not include seconds as we do not see applications where one would have conditions on seconds.
The developer can get these variables through convenient functions of SVSHI's API. This is added to SVSHI and provided in SVSHI's API by Ladina Roffler.

Now we can convert our iteration function into a z3 expression where we can verify any property.

6.4 Modifying CrossHair

We need to extract two things to convert our function into a z3 expression: The path conditions and the return value. The idea is to use CrossHair *cover* function, which returns inputs of a corresponding path. A list of specific input is returned to get as much coverage as possible. Then we can manipulate the return and combine it with our property to be checked.
We still want to keep the basic structure of SVSHI application: the iteration where the devices are manipulated and the invariant where properties of the apps are defined. We check if the iteration function is valid by verifying it with the property of the invariant. The idea would be to have the following workflow in Figure 1.

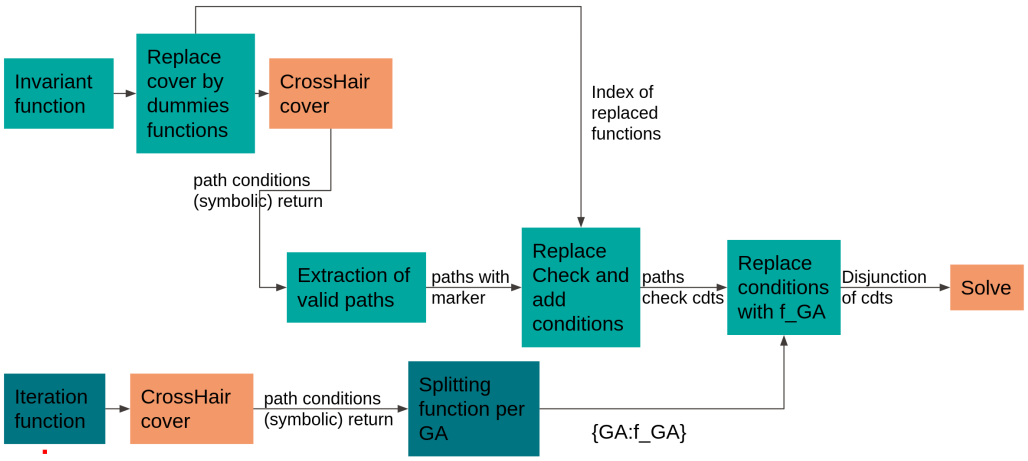

Figure 1: The structure of the new verification module

Additionally, instead of executing apps per GA or timer, we now make sure that all applications are always executed at the same time and in the same order. Before, the applications would be executed either periodically or when one of the devices has changed values.

6.4.1 Getting the paths

For the path conditions, Philip Schanely hinted here [10] that we could extract the solver state by getting its assertion list for each path before CrossHair switches paths. By copying the solver state into the result when the given path is covered, we now have all the paths of our function.

What remains is getting the symbolic return value on the given path. The big advantage of CrossHair is that it executes the function, therefore we know which variables are modified when getting the symbolic return.

6.4.2 Extracting the symbolic return

We now extract the symbolic values from CrossHair by collecting the return after one execution of the function in CrossHair. Extracting the symbolic return on a simple function is suggested by Loïc Montadon. He mentions this is possible to get it by intercepting the return value after the function is being executed. On a simple example, he shows us how to collect it by getting the "var" attribute from the return.
From this, we discover that the "var" attribute is actually when the function returns a SymbolicNumerAble, a CrossHair class that represent a symbolic number. However, when CrossHair has to realize a condition (it does not recognize an instruction or OPcode therefore it needs to give the variable a concrete value), we can only keep the concrete return. For example, this happens when the return is fixed or when encountering opcodes that are not managed yet by CrossHair (see [10] for an example of using "not" that leads to the realization of the variable).
Once we know to extract simple variables, we can then move on to what SVSHI currently returns. That is, for the iteration function it returns a dictionary of dataclasses (the appState, physicalState, and internalState). Therefore we need to extract the fields and recreate the dataclasses object by getting their symbolic or realized return value. For the invariant, it returns a Boolean only and the process is straightforward.

All the details of the modifications are available here [12].

6.4.3 Simple example

Now that we have modified CrossHair, we can show what it is capable of with a small example. Suppose we have the function

```
def solar_boiler_app(solar_boiler_app_app_state: AppState, physical_state:
PhysicalState, internal_state: InternalState):
    if 21<=svshi_api.get_hour_of_the_day(internal_state) :
        float_dev.set(60.0,physical_state, internal_state) #set 60.0 to GA_0_0_4
    else:
        float_dev.set(float(int_dev.read(physical_state, internal_state))
```

```
        -5.0,physical_state, internal_state)

    return {'physical_state': physical_state}
```

By running our modified CrossHair cover and extracting the attribute result of each PathSummary from the list of the PathSummary, we have:

```
[[21 <= time_hour_16], {'PhysicalState': {'GA_0_0_1':
GA_0_0_1_10, 'GA_0_0_2': GA_0_0_2_11, 'GA_0_0_3':
GA_0_0_3_12, 'GA_0_0_4': 60.0}}]

[[Not(21 <= time_hour_16)], {'PhysicalState': {'GA_0_0_1':
GA_0_0_1_10, 'GA_0_0_2': GA_0_0_2_11, 'GA_0_0_3':
GA_0_0_3_12, 'GA_0_0_4': ToReal(GA_0_0_3_12) - 5}}]
```

The first element of the list is the path constraint and the second is a dictionary of the physical state (the return values).
Here we can see that the boiler (GA_0_0_4) is either at 60 after 9 pm or equal to another sensor (GA_0_0_3) - 5 before. The other group addresses are unmodified and we have their z3 Constants as CrossHair created them.

6.5 From CrossHair to SVSHI

Now that we have all that is required to convert a simple function into a z3 expression, we can adapt it to SVSHI applications. We recall that the app is built in two parts: the invariant where properties of the application are stated and the iteration where the logic of the app is written.

To extract the paths, we use our modified CrossHair's cover function. First, by running it on the invariant function and then on the interation function.
On the iteration function, CrossHair outputs the path conditions and the symbolic return values after taking this path. For each path, we have the list of assertions of the solver. We make a conjunction of this to build a path condition. Then we create an expression built with If(pathCondition, symbolic return, (else)other path condition). We must split functions per GA as the If in z3py does not handle arrays. This expression is created for each return value and then stored. If we take our previous example, the function of GA_0_0_4 is:

```
If(21 <= time_hour_16,60,
   If(Not(21 <= time_hour_16), ToReal(GA_0_0_3_12) - 5, -1))
```

To gather the conditions of the invariant, we also use CrossHair cover function. Per path, we make a List of z3 expressions from the path and return the value to have the full condition. Additionally, we ignore paths that return a False value or a None.

```
def boiler_inv() ->bool:

    if 21<=svshi_api.get_hour_of_the_day(internal_state):
        return boiler.read(physical_state, internal_state)>60.0
    else:
        return True
```

Running cover outputs:

```
[[21 <= time_hour_16], {'ret': 60 < GA_0_0_4_13}]
[[Not(21 <= time_hour_16)], {'ret': True}]
```

Then transformed we have the list:

```
[[21 <= time_hour_16, 60 < GA_0_0_4_13], [Not(21 <= time_hour_16)]]
```

Then we replace the GA by their functions and convert the list into a disjunction:

```
Or(
    And(
        21 <= time_hour_16,
        #equiv. to 60 < f_GA_0_0_4 :
        60 < If(21 <= time_hour_16,60,
        If(Not(21 <= time_hour_16), ToReal(GA_0_0_3_12) - 5,-1))),
    Not(21 <= time_hour_16))
```

Finally, we use z3py's solver and output a counterexample if the formula is not satisfiable. Currently it displays this error:

```
counterexample [time_hour_16 = 21] for condition:
Or(
    And(21 <= time_hour_16,
```

```
        60 < If(21 <= time_hour_16,60,
             If(Not(21 <= time_hour_16),
                ToReal(GA_0_0_3_12) - 5,-1))),

    Not(21 <= time_hour_16))
```

The error is not extremely clear, here it states that after 9 pm the condition is false. This is because our boiler never goes strictly above 60.

6.6 Creating a user-friendly function

Once we have our invariant function represented as a z3 expression, we can start working on modeling a template that we would use to check any property for a given duration. Then we provide a simple function to the developers. Indeed, we want to make the verification process as easy as possible and therefore we do not expect developers to write z3 expressions. Then we create the function that converts the conditions to a z3 expression to finally integrate them on the invariant.

6.6.1 Definition for the developer

We create a function in SVSHI's API for the developer that keeps the condition as a high level. First, we define our condition. A developer wants to verify a condition for a given frequency and duration for every possible input. Given that time is redefined by SVSHI, to give how long should be the frequency and duration, we should also provide a way to indicate days, months, etc...
Therefore we created a class called DateObj, where we can easily instantiate duration. For example, Day(5) would be five days.
In this class, the amount of time is stored and in the superclass DateObj we also provide the name of SVSHI time variables. We store the name of the variables to create afterward a dictionary between the original name of the variable and CrossHair's. This is because CrossHair gives a different name to the variable, in the following format: originalVariableName_counter.
Now that we provided time, what remains is to get the condition we need to verify. We expect the developer to write the condition given the device they defined as it seems a straightforward procedure for them. It is a more challenging task for us, as we need to extract the GA of the condition for

each object. The extraction is done by running CrossHair cover. It is implemented by Ladina Roffler.
Finally, our function would look like this for an application developer :

```
svshi_api.check_time_property(frequency=Day(1),
                duration=Hour(1),
                property=a_sensor.read() > 5)
```

6.6.2 Internals of the check function

Now that we have a high-level definition of the function, we need to convert it to a z3 expression. First, we need to gather the variables names used by CrossHair and map them back to their original name. We do this by creating a dictionary between their name and the z3 expression after running cover.
To model the duration, we convert this expression "There exists a t, such that for all times between t and the duration, the property is true". In the "for all times", we also add smaller times unit of the duration. For example, if the duration is Hour(1), the z3 expression is:

```
Exists(t,
    And(And(t >= 0, t <= 22),
    ForAll(time_hour,
            Implies(
                    And(t <=time_hour,
                        time_hour <=t + 1),
                    OurCondition))))
```

Then we add the frequency, that is either "ForAll frequency" or if it is not occurring every time unit, we split it on multiple expressions to cover all. For example, if we have "frequency=Day(3)" that is every three day we have

```
ForAll(days,
        Or(Implies(1<=day<=3,condition),
           Implies(4<=day<=6,condition),
           Implies(day==7,condition))
```

The condition is the duration. In this example, we see one of the limitations of the current state of the verification. That is, we cannot check when it is the last day, then go back to the first one and second one to model a three day duration. Limitations will be explained later on.

6.6.3 Integration with CrossHair

Note: Code integration in SVSHI was done by Ladina Roffler. We only explain here the idea at a high level.
Once we have defined our function, we need to provide it to the developer. If they use the fully defined check function on the invariant, two issues arise: CrossHair will attempt to cover the paths of our check function and CrossHair is not built to add z3 expressions into its solver. Therefore we replace the check calls on the invariant function with a dummy variable to use as a marker of the check call. Here the idea is to track where *check_time_property* calls happen by making a simple path constraint that CrossHair will get during its execution.
To do this, we can add an internalState variable called c0 that is constrained per check call. Instead of having to go to check(), we replace it with a function that constrains c0. For example, this condition:

```
def invariant() ->bool:

    return svshi_api.check(frequency=Day(1),
    duration=Hour(1),
    boiler.read()>60.0)
```

Will be replaced by this:

```
def dummy_check(internal_state,v):
    return internal_state.c0 == 0

def invariant() ->bool:

    return  dummy_check(internal_state,v)
```

When we have a path constraint where it contains c0 == 0, we know it's the first check call.
Additionally, we still have to convert what the developer has provided as conditions. To make our lives easier, we can run cover on this fragment of function the same way we do it for the iteration function.
Once we have the marker, we can replace the dummy variable with the actual z3 expression of the mentioned condition.

6.7 A concrete example

Let's suppose we have a boiler that is plugged into the solar heater. The boiler can be heated by electricity using its heating element or by turning on

the circulator pump of the solar heater. We want the boiler to reach 45°C at anytime but never go above 65°C and prioritize the solar heating. We build an application that does this :

```
def iteration():
    if 65 >= solar_heater.read() >= 45:
        solar_heater_circulator.on()
        boiler.set(0.0)
    else:
        solar_heater_circulator.off()
        boiler.set(45.0) #GA_0_0_4
```

Then after installing this app, we notice that the boiler should be compliant with our local laws in Switzerland. That is, the boiler should be at 65°C for one hour [4]. We install the application to be compliant :

```
def invariant() ->bool:

    return svshi_api.check(frequency=Day(1),
    duration=Hour(1),
    boiler.read()>60.0)
```

See in D for the Z3 expression of the invariant function, where the condition is replaced with f_GA_0_0_4 $>$ 60. However when installing it, it outputs an error as the condition is not satisfied. (currently shows "solver unsat", but error message will be changed to Check condition "check(frequency=Day(1), duration=Hour(1), boiler.read()¿60.0)" is unsat, please install a valid app to satisfy this constraint") Therefore we rewrite our application to be:

```
def iteration():
    if 21<=svshi_api.get_hour_of_the_day() <=23:
        solar_heater_circulator.off()
        boiler.set(65.0) #GA_0_0_4
    elif 65 >= solar_heater.read() >= 45:
        solar_heater_circulator.on()
        boiler.set(0.0)
    else:
        solar_heater_circulator.off()
        boiler.set(45.0) #GA_0_0_4
```

See in C for the Z3 expression of the iteration function for GA_0_0_4's function. Then here the condition is satisfied and the new application is installed.

6.8 Limitations

With the new module, we cannot verify what is between two time units. This is because each time unit is independent and no links are represented. For example, we cannot verify that a switch is on for one hour if it is set between 2:30 pm and 3:30 pm.
The way we extract conditions on the check only allows variables that are defined globally. Therefore a developer cannot introduce variables that were defined in the function scope.

6.9 Future work

As mentioned in the related work, one could explore verification of SVSHI applications using the Nagini [7] tool. To use it with SVSHI's invariants, this would require manipulation of the code. However, if the new solution provided in this paper is not sufficient to verify the runtime module, it seems Viper could help. This would require an important rewriting of the code but will enhance the reliability of SVSHI.
Regarding the limitations, we cannot statically model the environment. One could create links between GAs, either by reading the KNX file and/or letting the user provide how devices react to each other. Another possibility is to use the KNX Simulator[13] when doing the compilation to give more trust to the applications. Furthermore, we could use the InternalState's time argument at runtime to be shared with the simulator. It would only be a simulation, however, it would help the user understands the link between the devices and the environment.

7 Conclusion

In this report, we proposed two approaches to model time-sensitive properties for extending SVSHI's verification process. We chose the last approach that converts functions into z3 expressions and provides time to the user through SVSHI's API. It was chosen as it keeps the event driven architecture of SVSHI.
The challenge was to sketch time and especially what happens with previous executions. Indeed, when using CrossHair and observing counterexamples it could not infer previous executions from the code. We end up with counterexamples when the path to go from the valid property to another one is

being taken. Even if the path is constructed with a sufficient duration.
We added sleep to make the execution "longer" in real-time and therefore would solve the issue. However, SVSHI applications should be executed quickly and thus we need to explore other solutions.
We decided to convert SVSHI applications to z3 expression so that we could model any properties. Then by using an SMT solver we know whether the condition is valid or not. This comes with a set of limitations that were previously mentioned, but it can also help the verification of the python modules in SVSHI.
Some of the limitations can be a reduced by using the simulator [13]. This introduces dependencies between devices and the environment, which can help the verification process.

Appendix

A Application that enforces a valid boiler behavior

```
from instances import app_state, BOILER_SENSOR


def invariant() -> bool:
    # Write the invariants of the app here
    # It can be any boolean expressions containing the read properties of
    ↪  the devices and constants
    # You CANNOT use external libraries here, nor unchecked functions
    set_to_60_deg_in_last_24h = app_state.BOOL_0
    return set_to_60_deg_in_last_24h or BOILER_SENSOR.read() >= 60

def iteration():
    # Write your app code here
    # You CANNOT use external libraries here, encapsulate calls to those
    ↪  in functions whose names start
    # with "unchecked" and use these functions instead

    # reading and defining constants
    period = 10  # time period in seconds when the app is ran

    last_time_set_to_60 = app_state.FLOAT_0
    time_at_60_deg = app_state.INT_1
```

```
    set_to_60_deg_in_last_24h = app_state.BOOL_0
    override_sequence = app_state.BOOL_1
    last_run = app_state.FLOAT_1
    if last_run < 1:
        last_run = unchecked_get_current_time(last_run)

    if BOILER_SENSOR.read() >= 60:
        time_at_60_deg+= unchecked_get_current_time(last_run) - last_run
        if time_at_60_deg >= 60:
            last_time_set_to_60 = unchecked_get_current_time(last_run)
            set_to_60_deg_in_last_24h = True
            time_at_60_deg = 0
    else:
        time_at_60_deg = 0  # reset the counter

    day_in_sec = float(24 * 60 * 60)
    time_delta = unchecked_get_current_time(last_run) -
    ↪  last_time_set_to_60

    if time_delta >= day_in_sec:
        set_to_60_deg_in_last_24h = False

    # assigning the constants into the app state
    app_state.FLOAT_0 = last_time_set_to_60
    app_state.FLOAT_1 = unchecked_get_current_time(last_run)
    app_state.BOOL_0 = set_to_60_deg_in_last_24h
    app_state.INT_1 = time_at_60_deg
    app_state.BOOL_1 = override_sequence


def unchecked_get_current_time(x:float) -> float:
    """
    post : __return__ > x
    post: __return__ > 0
    """
    import time
    return time.time()
```

B Example with waits

```
def wait(value, physical_state: PhysicalState) -> None:
    physical_state.time += value
    physical_state.time_state += value
```

```
def wait_sensor(sensor, physical_state:PhysicalState, fct, timeout:int,
↪  polling_frequency:int = 10)-> tuple[
    bool, bool]:
    return
    ↪  fct(sensor.read(physical_state)),(fct(sensor.read(physical_state))
    ↪  and physical_state.time_state >= timeout)
```

```
class GenericSetReadDPT9_boiler_rules_boiler_sensor():
        def set(self, temp:float , physical_state: PhysicalState):
            """
            pre:
            post: physical_state.GA_0_0_1 == temp
            """
            physical_state.GA_0_0_1 = temp

        def read(self, physical_state: PhysicalState) -> float:
            """
            pre:
            post: physical_state.GA_0_0_1 == __return__
            """
            return physical_state.GA_0_0_1
```

```
class GenericSetReadDPT9_boiler_set_temp_boiler_sensor():
        def set(self, temp:float , physical_state: PhysicalState):
            """
            pre:
            post: physical_state.GA_0_0_1 == temp
            """
            physical_state.time += 60*60

            physical_state.time_state = 0
            physical_state.GA_0_0_1 = temp

        def read(self, physical_state: PhysicalState) -> float:
            """
            pre:
            post: physical_state.GA_0_0_1 == __return__
            """
            return physical_state.GA_0_0_1
```

```
BOILER_RULES_BOILER_SENSOR =
↪  GenericSetReadDPT9_boiler_rules_boiler_sensor()
BOILER_SET_TEMP_BOILER_SENSOR =
↪  GenericSetReadDPT9_boiler_set_temp_boiler_sensor()


def boiler_rules_invariant(boiler_rules_app_state: AppState,
    boiler_set_temp_app_state: AppState, physical_state: PhysicalState)
    ↪  ->bool:
    set_to_60_deg_in_last_24h = boiler_rules_app_state.BOOL_0
    return set_to_60_deg_in_last_24h


def boiler_rules_iteration(boiler_rules_app_state: AppState,
    boiler_set_temp_app_state: AppState, physical_state: PhysicalState,
    ):
    """
pre: boiler_rules_invariant_rules_app(boiler_rules_app_state,
↪  boiler_set_temp_app_state, physical_state)
pre: boiler_set_temp_invariant(boiler_rules_app_state,
↪  boiler_set_temp_app_state, physical_state)
pre: boiler_rules_app_state.FLOAT_1 < physical_state.time
post: boiler_rules_invariant_rules_app(**__return__)
post: boiler_set_temp_invariant(**__return__)
"""
    boiler_rules_unchecked_get_current_time = physical_state.time
    last_time_set_to_60 = boiler_rules_app_state.FLOAT_0
    set_to_60_deg_in_last_24h = boiler_rules_app_state.BOOL_0
    time_at_60_deg = 60*60
    success,out = wait_sensor(BOILER_RULES_BOILER_SENSOR, physical_state,
    ↪  lambda x: x >= 60, timeout=time_at_60_deg)
    if success and out:
        last_time_set_to_60 = physical_state.time
        set_to_60_deg_in_last_24h = True


    day_in_sec = float(24 * 60 * 60)

    time_delta = boiler_rules_unchecked_get_current_time -
    ↪  last_time_set_to_60
    if time_delta > day_in_sec:
        set_to_60_deg_in_last_24h = False
```

```
    boiler_rules_app_state.FLOAT_0 = last_time_set_to_60
    boiler_rules_app_state.FLOAT_1 =
    ↪  boiler_rules_unchecked_get_current_time
    boiler_rules_app_state.BOOL_0 = set_to_60_deg_in_last_24h
    return {'boiler_rules_app_state': boiler_rules_app_state,
            'boiler_set_temp_app_state': boiler_set_temp_app_state,
            'physical_state': physical_state}


def boiler_rules_invariant_rules_app(boiler_rules_app_state: AppState,
    boiler_set_temp_app_state: AppState, physical_state: PhysicalState)
    ↪  ->bool:
    last_time_set_to_60 = boiler_rules_app_state.FLOAT_0
    set_to_60_deg_in_last_24h = boiler_rules_app_state.BOOL_0
    last_run = boiler_rules_app_state.FLOAT_1
    time_delta_above_24h = last_time_set_to_60 + float(24 * 60 * 60) <
    ↪  last_run
    return (last_time_set_to_60 >= 0 and last_run >= 0 and
    ↪  time_delta_above_24h ^ set_to_60_deg_in_last_24h and
            last_time_set_to_60<=last_run )

def boiler_set_temp_invariant(boiler_rules_app_state: AppState,
    boiler_set_temp_app_state: AppState, physical_state: PhysicalState)
    ↪  ->bool:
    last_set_at_60 = boiler_set_temp_app_state.BOOL_0
    return (last_set_at_60 and physical_state.GA_0_0_1 == 60
            ) or (not last_set_at_60 and physical_state.GA_0_0_1 == 0)


def boiler_set_temp_iteration(boiler_rules_app_state: AppState,
    boiler_set_temp_app_state: AppState, physical_state: PhysicalState):
    """
pre: boiler_rules_invariant_rules_app(boiler_rules_app_state,
↪  boiler_set_temp_app_state, physical_state)
pre: boiler_set_temp_invariant(boiler_rules_app_state,
↪  boiler_set_temp_app_state, physical_state)
post: boiler_rules_invariant(**__return__)
post: boiler_set_temp_invariant(**__return__)
"""
    last_set_to_60 = boiler_set_temp_app_state.BOOL_0
    BOILER_SET_TEMP_BOILER_SENSOR.set(60, physical_state)
    wait_sensor(BOILER_SET_TEMP_BOILER_SENSOR, physical_state, lambda
    ↪  x:x>=60, timeout=0)
    boiler_rules_iteration_no_cdt(boiler_rules_app_state,
                                  boiler_set_temp_app_state, physical_state)
```

```
    boiler_set_temp_app_state.BOOL_0 = True
    wait(2*60*60,physical_state)
    boiler_rules_iteration_no_cdt(boiler_rules_app_state,
                                  boiler_set_temp_app_state, physical_state)

    BOILER_SET_TEMP_BOILER_SENSOR.set(0, physical_state)
    wait_sensor(BOILER_SET_TEMP_BOILER_SENSOR, physical_state, lambda
    ↪  x:x==0, timeout=0)
    boiler_rules_iteration_no_cdt(boiler_rules_app_state,
                             boiler_set_temp_app_state, physical_state)
    boiler_set_temp_app_state.BOOL_0 = False
    wait(2*60*60,physical_state)
    boiler_rules_iteration_no_cdt(boiler_rules_app_state,
                                  boiler_set_temp_app_state, physical_state)


    return {'boiler_rules_app_state': boiler_rules_app_state,
            'boiler_set_temp_app_state': boiler_set_temp_app_state,
            'physical_state': physical_state}
```

Note : boiler_rules_iteration_no_cdt is boiler_rules_iteration without the docstring that contains the pre and post condtions.

C Z3 expression of f_GA_0_0_4

```
If(And(21 <= time_hour_16, 23 >= time_hour_16),
   65,
   If(And(Not(21 <= time_hour_16), Not(45 < GA_0_0_3_12)),
      45,
      If(And(21 <= time_hour_16,
             Not(23 >= time_hour_16),
             Not(45 < GA_0_0_3_12)),
         45,
         If(And(21 <= time_hour_16,
                Not(23 >= time_hour_16),
                45 < GA_0_0_3_12),
            0,
            If(And(Not(21 <= time_hour_16),
                   45 < GA_0_0_3_12),
               0,
               ToReal(-1))))))
```

D Z3 expression of the example check condition

```
[[ForAll([time_hour_16,
          GA_0_0_3_12,
          time_min,
          time_day,
          time_weekday,
          time_month,
          time_year],
         Implies(And(time_day >= 1,
                     time_day <= 7,
                     time_hour_16 >= 0,
                     time_hour_16 <= 23,
                     time_weekday >= 1,
                     time_weekday <= 4,
                     time_month >= 1,
                     time_month <= 12,
                     time_min >= 0,
                     time_min <= 59),
                 Exists(t,
                        And(And(And(t >= 0, t <= 22),
                                ForAll(time_hour_16,
                                       Implies(And(t <=
                                         time_hour_16,
                                         time_hour_16 <=
                                         t + 1),
                                         60 <
                                         If(And(21 <=
                                         time_hour_16,
                                         23 >= time_hour_16),
                                         65,
                                         If(And(Not(21 <=
                                         time_hour_16),
                                         Not(45 < GA_0_0_3_12)),
                                         45,
                                         If(And(21 <=
```

```
                    time_hour_16,
                    Not(23 >=
                    time_hour_16),
                    Not(45 < GA_0_0_3_12)),
                    45,
                    If(And(21 <=
                    time_hour_16,
                    Not(23 >=
                    time_hour_16),
                    45 < GA_0_0_3_12),
                    0,
                    If(And(Not(21 <=
                    time_hour_16),
                    45 < GA_0_0_3_12),
                    0,
                    ToReal(-1))))))))),
            And(time_day >= 1,
                time_day <= 7,
                time_hour_16 >= 0,
                time_hour_16 <= 23,
                time_weekday >= 1,
                time_weekday <= 4,
                time_month >= 1,
                time_month <= 12,
                time_min >= 0,
                time_min <= 59)))))]]
```

References

[1] S. Chassot and A. Veneziano, "Svshi - secure and verified smart home infrastructure," EPFL - DSLAB, 2022. [Online]. Available: https://github.com/dslab-epfl/svshi/blob/main/src/documentation/documentation.md

[2] P. Schanely, "Crosshair," GitHub, 06 2022. [Online]. Available: https://github.com/pschanely/CrossHair

[3] "Concolic testing," Wikipedia, 04 2022. [Online]. Available: https://en.wikipedia.org/wiki/Concolic_testing

[4] A. Cordin, V. Bernhard, and B. Stefan, "Hpt annex 46 domestic hot water heat pumps task 1 market overview country report switzerland," 2016. [Online]. Available: https://www.fws.ch/wp-content/uploads/2018/10/Market_Overview_Country_Report_Switzerland_Annex_46_DHWHP_Task1.pdf

[5] B. Kordic, M. Popovic, and S. Ghilezan, "Formal verification of python software transactional memory based on timed automata," *Acta Polytechnica Hungarica*, vol. 16, no. 7, pp. 197–216, 2019.

[6] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," 2006.

[7] M. Eilers, "marcoeilers/nagini," GitHub, 06 2022. [Online]. Available: https://github.com/marcoeilers/nagini

[8] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

[9] "time — time access and conversions — python 3.10.5 documentation," docs.python.org. [Online]. Available: https://docs.python.org/3/library/time.html#time.time

[10] "Using crosshair to convert simple functions into z3 expressions to verify time sensitive applications · discussion #165 · pschanely/crosshair." [Online]. Available: https://github.com/pschanely/CrossHair/discussions/165

[11] L. de Moura and N. Bjørner, "Z3: an efficient smt solver," vol. 4963, 04 2008, pp. 337–340.

[12] "Dslab's crosshair fork," 06 2022. [Online]. Available: https://github.com/dslab-epfl/CrossHair

[13] L. Alvarez and I. Daudé, "Smart home knx system simulator," EPFL - DSLAB, 2022.