SASIL: A Domain-Specific Language for Simulating Declarative Specifications of Scheduling Systems

Gahwon Lee

This document has been approved by my advisor for distribution to my committee

Department of Computer Science
The University of Texas at Austin
May 2020


**SASIL: A Domain-Specific Language for Simulating Declarative Specifications of Scheduling Systems**

Gahwon Lee

May 2020

**Abstract**

*SASIL* is a domain-specific language to describe and execute the specification of scheduling systems. The language interpreter operates on time-dependent behaviors and reactive events to simulate the described system. Furthermore, the interpreter is capable of selecting the appropriate actions needed to automatically schedule necessary components of the system to resolve requests. The DSL was tested by implementing an elevator control system, which is defined as a series of button requests and the requirements needed to complete each request. A multi-elevator system described using the language allowed the interpreter to complete requests 1.04 times faster on average than the traditional elevator algorithm.

1 Introduction

A specification describes what a system does, usually using formal methods. It is used to aid the implementation of the described system [4]. Although formal specifications can show that system designs are correct and verifiable, they are often cost-ineffective to design since they do not provide any actual implementation of the system. However, advances in formal methods have shown that it is possible to create an executable specification, which automatically synthesizes the implementation from the specification language [5].

In this paper, we explore the possibility of leaving out the algorithm from the specification of a scheduling system. The best schedule is automatically extracted by running a simulation of the system forwards in time to search for the most optimal way of scheduling requests. We developed the Specification with an Automatic Scheduler Integrated Language (SASIL), a domain-specific language and interpreter that simplifies the task of expressing the system as a simulation without explicitly writing out the algorithm needed to schedule requests.

SASIL is divided into data definitions and reactive transitions. The data definitions section describes the system as a hierarchy of types and time-dependent
values. By using SASIL’s flexible type system, the programmer can concisely write out the specific model to represent the system as data. The reactive transitions section describes the system as a set of events, reactions, and transitions. It describes how the data changes over time as new requests are created and completed.

To evaluate SASIL, we specify a multi-elevator control system without writing an explicit scheduling algorithm to map requests and elevators. We compare the SASIL scheduler with the elevator algorithm to check if the synthesized algorithm yields any speedup in request completion. Results show that SASIL can be used to produce a concise and correct scheduling algorithm. Although the elevator simulation ran slower than realtime, it followed an effective schedule to rapidly and fairly complete requests.

Overall, the contributions of this paper are:

- SASIL, a domain-specific language for describing schedulable systems.
- the SASIL interpreter implementation using the Enso language workbench, and how the implementation automatically generates an optimal schedule.
- a simplified method for programmers to describe what a system does declaratively by using SASIL.

2 Background

2.1 Scheduling Systems

SASIL simulates scheduling systems; we define a scheduling system as a system that consists of a stream of requests that it needs to complete in any order. If the system is represented as a sequence of state transitions, then a state entrance could create or complete a non-negative number of requests. By connecting all potential states with all legal actions the system can take at a particular state, the system can be represented as a state machine that isn’t bound by a specific order of request completion. In this directed graph, where states are vertices and transitions are edges with the time duration as cost, the shortest path from the request creation state to the request completion state is the most optimal schedule to complete a request, as shown by a direct application of Dijkstra’s algorithm. SASIL’s goal is to provide language-level constructs to concisely represent this system state machine as code. We show how we accomplished this goal in sections further below.

2.2 Elevator Control Systems

An example of a scheduling system is an elevator control system. The elevator system consists of two different types of requests: floor requests and car requests. A floor request is created when a person on a particular floor presses either the up button or the down button. It is completed when any elevator arrives at
the floor with its direction indicator light pointing in the same direction as the pressed button.

A car request is created when a person inside an elevator car presses any of the floor buttons. It is completed when this elevator arrives at the pressed button’s floor. Unlike a floor request, a car request does not require a specific indicator light direction. Both types of requests require a car to reach a floor to be completed.

The concept of indicator light direction is named committed direction [10]. Unless the elevator door is closed, the committed direction remains on without changing directions until the door fully closes again. Once closed, the elevator tends to move in the same direction as the committed direction. An exception to this case is when no requests are left pending in the system. In this case, the elevator remains stopped after closing its door. It remains stationary until a new request is created.

Traditionally, elevator systems used the elevator algorithm to complete requests [2]. Following this algorithm, the elevator would continue to move following its committed direction, occasionally stopping to complete requests involving the passed floor and the committed direction. It will stop following the committed direction when no more floors in front of the path can complete a request, in which case, the committed direction is reversed. This elevator algorithm proved to be easy to follow and fairly energy-efficient, but it failed to work well in taller buildings. This is because the elevator prioritizes the middle floors since it passes by them every time by definition of the algorithm [2].

More importantly, the elevator algorithm fails to scale when more elevator cars are added to the system. In this case, multiple elevators might compete to complete the same request, showing redundancy. A naive solution would be to assign each car to a specific cluster of floors, but practical optimizations involve communication between the cars to load-balance requests between each car.

Therefore, modern elevator systems use different scheduling algorithms, many of them being proprietary and hand-tuned to fit each building [2]. One popular algorithm is called the Estimated Time of Arrival Control, which let a computer assign the car that it heuristically determined would get to a request the fastest [2]. Another idea is to use machine learning to auto-learn a policy [2]. In either case, a computer is responsible for estimating the best way to map a request to a car.

2.3 Executable Specification and DSL

To formally verify that an elevator control system is correct, researchers would resort to writing a full specification of the system. A formal specification of an elevator system would include the algorithm to fairly distribute the requests [7]. However, formal specifications would prefer to describe the traditional elevator algorithm due to its relative ease of implementation and verification, straying away from more advanced algorithms [7]. As the algorithm increases in complexity, it would be harder to formally verify its correctness.
To reduce complexity, a specification could leave out the desired algorithm if it could be automatically generated from the rest of the specification. Mainly, the specification should be written in a domain-specific language (DSL) that restricts the language expressiveness to the domain of the specification, which allows the synthesis of new behavior. A DSL differs from a general-purpose language (GPL) mainly due to the fact that it is targeting a smaller and well-defined domain [9]. Executable DSLs would have an execution engine that would read in the code to produce behavior belonging to the domain. Modern examples of this behavior include SQL for databases and Excel formulas for spreadsheets. By restricting the domain, it is possible to express the specification more concisely than a GPL. Also, it is possible to reason about behavior by inferring knowledge from the DSL code.

For example, a DSL could be inferred as a graph with a starting node and an ending node. Then, the execution engine could use standard search techniques to figure out the shortest path from the start node to the end node. Finding the shortest path would be helpful in a state machine, for example, where the engine needs to determine the most optimal sequence of state transitions to reach a target state.

If an elevator control system is formulated in terms of a state machine, then the states would consist of at least the elevator location, committed direction, and door state. The goal of the DSL engine would be to find the shortest sequence of transitions that leads to a state that completes a request.

### 2.4 Time and Behavior

It is possible to represent the elevator control system as a state machine by describing the states as a behavior over time. [3] describes behaviors as flows of values, represented by the function \([\text{Behavior}_\alpha] = T \rightarrow \alpha\) where \(T\) is the current time and \(\alpha\) is a generic state. A behavior can be constant, meaning it can produce the same result regardless of time; or it can be time-dependent, meaning the resulting value varies over time. The SASIL interpreter treats constant behaviors as states and time-dependent behaviors as transitions. Specifically, it interprets time-dependent behaviors as a linear transition from one state to another.

Multiple behaviors can be combined into a single algebraic type. A product type would consist of multiple behaviors while a sum type would be an exclusive behavior among a set of alternatives. Combined, these types can describe a system's state as one varying value over time. This enables the execution engine to query the system as a searchable state machine.

### 2.5 Enso

DSLs can either be interpreted directly or transformed into a language targeting a more general domain. With SASIL, we decided to interpret the system simulation directly using the Enso language workbench. Enso automatically parses
the language into first-class Ruby objects, which can then be further manipulated using standard Ruby constructs. Thus, the grammar and schema are written in Enso while the simulation is written in Ruby. Having a custom interpreter for SASIL written in a GPL allows us to gain specific control over how the simulation works. Therefore, programmers writing SASIL code can focus on what the system does and leave how the simulation works to the interpreter runtime. Mainly, the interpreter deals with how a system should scheduling requests while the SASIL code describes when a request is created or completed.

3 SASIL

SASIL is a domain-specific language for describing scheduling systems. A programmer can use the language to write the system specification, and the interpreter can simulate the inferred system, following an automatically generated schedule. Additional user inputs such as button presses are provided externally to simulate user interaction with the system.

In this section, we share how we implemented the SASIL frontend and interpreter, and how it automatically schedules requests.

3.1 Enso Grammar and Schema

A typical SASIL file is divided into four sections: the data/type definition, global variable declaration, functions, and events. Data definitions describe the structure of all variables necessary to describe all potential states of the system. Global variables represent the entire system and its current state. Functions allow the programmer to refactor and reuse long expressions. And events handle reactive transitions between states. Figure 1 shows a simplified view of the top-level grammar parsed by Enso used to define all four sections.

```
start Language
Language ::= [Language] dataDefs:Data* vars:TopLevelVar* 
    functions:FunctionDecI|Impl* fors:TopLevelFor*
Data ::= [Data] "data" name:sym ":=" val:DataDef ";"
TopLevelVar ::= [TopLevelVar] "var" name:sym ":" data:DataRef 
    VarInit? ";"
FunctionDecI|Impl ::= FunctionDecl | FunctionImpl
TopLevelFor ::= [TopLevelFor] "for" args:TopLevelForArgs* 
    "{" whens:When* "}"
```

Figure 1: Top Level Object Grammar
3.2 Behavior and Types

SASIL is capable of describing time-independent and time-dependent behavior through its type system. The type `Between<A>` indicates that it can be a time-dependent behavior that has a value of type `A` changing over time. Figure 2 shows the simplified data definition of an Elevator, consisting of constant behaviors (`movementState` and `committedDirection`) and potentially time-dependent behaviors (`location` and `doorState`).

A variable of type `Between<A>` can accept either a constant behavior `A` or a time-dependent transition as its value. For example in figure 2, the field `location` can either be a particular floor or a transition from one floor to another.

```plaintext
data Elevator =
  location: Between <Floor>,
  doorState: Between <Closed | Opened>,
  movementState: Stopped | Moving,
  committedDirection: Up | Down | Idle;
```

Figure 2: Elevator Model

Product types are specified as records and have names for each field. A record instance’s field can be accessed by the standard dot operator (`.`) present in many object-oriented languages. Field access is evaluated at runtime regardless of type, and incorrect access is not revealed until evaluation.

State can be mutated with the standard assignment operator (`=`) present in many other languages. The assignment operator indicates that the left-hand side should be parsed by Enso as an lvalue, which ensures that the system knows which location in the environment memory to mutate.

A different operator is available to modify a time-dependent behavior over time. Figure 3 shows an example of the transition operator to close the elevator door. It takes in a target state, a sequential list of values to iterate over, and the time it takes to transition from one value to another. The operator starts from the current value of the variable and transitions to the target; if the target is more than a single step away from the current value in the given list, then the system automatically transitions sequentially from the start value to the end value, taking multiple steps in the list. For example, transitioning from floor 4 to floor 1 would resolve to transitioning from floor 4 to floor 3, floor 2, and floor 1 in that order, each step taking the full time duration given as the argument.

It is possible to defer the execution of assignment statements to happen after a transition finishes. This is done by placing the deferred statements within an `after` block. Deferred statements are executed as soon as the target transition completes, and they only run if the entire transition fully runs without another assignment or transition overwriting the variable. Internally, the deferred statements are placed in the events queue, which is described further in a section below.

The main idea behind these language constructs is to represent a system as a
set of states and transitions. A state would describe the system at a given time while a transition would indicate how the state changes as time increases. To simplify the implementation, we limit that a transition cannot be canceled in the middle of its transition. Thus, the transition operator allows the programmer to easily divide a long transition into multiple transitions, such as creating intermediate floor states between a long elevator movement from one floor to another.

### 3.3 Events

```python
for elevator in state.elevators, floor in state.floors {
    when elevator.buttons[floor] {
        output carButtonPressed();
    }
}
```

Figure 4: Event

An event prompts statements to execute when the state matches specific criteria. It can be used to trigger state mutations and transitions after a previous mutation occurs or a previous transition completes. Figure 4 shows an example event declared for when a specific boolean in a map results in a truthy value. The `when` indicates that the below statements should run as soon as `elevator.buttons[floor]` changes from false to true. Once the event runs, the statements are not executed until the evaluation of the `when` expression changes from false to true once again.
In the same figure, the `for` indicates that the event should run for every permutation of elevators and floors, with the current permutation assigned to the variables `elevator` and `floor`. If `state.elevators` has \( N \) items and `state.floors` has \( M \) items (if there are \( N \) elevators and \( M \) floors), then the event would trigger \( NM \) times at max.

An event cannot be triggered concurrently before a different event completes executing its statements. Thus, the interpreter defers triggering events until all assignments in the current event complete execution. An exception to the sequential execution are `after` blocks, which divide an event into multiple execution units. The statements within the block runs eventually, after the current event completes and potentially triggers events.

### 3.4 Request Scheduling

Requests are objects that can be created and completed at a set time. A programmer can write an event to trigger request creation when an external user input modifies a variable. The goal of the simulation is to complete as many requests as possible in the shortest amount of time. The request object type can be specified and customized by the programmer. The event to complete a request should also be written by the programmer.

The interpreter attempts to schedule requests by searching for the shortest path to complete all requests. The search itself is implemented as the Uniform Cost Search algorithm limited to a configurable depth. This algorithm generates a schedule to follow at the start of each event loop (after a state change). Its goal is to minimize the total transition duration and the total amount of transitions.

The language allows the programmer to specify places where the algorithm can select an edge to follow. This is done by involving the `select` function, which takes in an arbitrary number of arguments as input and returns one of the arguments as output. The interpreter is responsible for selecting the best argument to return.

### 3.5 Input

The system continues to execute while there exists a user input that can be processed. A user input is a command entered into the interpreter, and it can either be to set a variable, to sleep for a given time, or to exit early. The interpreter accepts user input from either STDIN or an input file, and the accepted string is parsed into commands from a different set of Enso grammar and schema. Input handling logic is shown below in Algorithm 1.

A completed request can trigger dependent user input. For example, the request for an elevator to arrive at floor \( X \) can make the simulation press an elevator button after the request completes. These chains of request dependencies are specified as user inputs. Figure 5 shows two different input chains. The symbol `elevator` within `|` indicate that the local variable `elevator` should copy its value from an external variable with the same name right after the request is
set ||{ state.floors[1].buttons[Up] = true; }, |elevator||{
  elevator.buttons[state.floors[2]] = true; }

sleep 50

set ||{ state.floors[7].buttons[Down] = true; }, |elevator||{
  elevator.buttons[state.floors[1]] = true; }
sleep 1

set ||{ state.floors[5].buttons[Up] = true; }, |elevator||{
  elevator.buttons[state.floors[6]] = true; }
sleep 50
exit

Figure 5: Example User Input File

completed. For example, this can be used to specify that the elevator completing
the initial floor request should get its car button pressed.

3.6 Simulation

The SASIL interpreter runs the system specification as a discrete event simu-
lation, meaning that the implementation treats all events as having the same
start time and end time. The implementation that we wrote follows Algorithm
1 and 2. The interpreter keeps a queue of events sorted by their priority (start
time). Until no events are left in the queue, the top of the queue is popped
and executed. An event can modify the system state, schedule other events, or
trigger when blocks.

Algorithm 1 Top Level of Simulation

procedure RUNSIMULATION
  for each input ∈ inputs do    // handle external user inputs
    if input is set then
      SET(input)
    else if input is sleep then
      RUNEVENTS(currentTime + input sleep duration)
    else if input is exit then
      report and exit
  end if
end for
end procedure

Before any statement is executed, a copy of the environment is made. This
copy contains the state of every global variable, requests, time-dependent behav-
ior, and currentTime at the time of the copy. After any global state is modified,
the interpreter checks every when block to check if any permutation should be
Algorithm 2 Handle Events

```plaintext
procedure RunEvents(stopTime)
    while currentTime < stopTime and ∃ event ∈ environment do
        event ← pop next event from environment
        if event happens before stopTime then
            currentTime ← event start time
            run(event)
        else
            currentTime ← stopTime
            break
        end if
    end while
end procedure
```

scheduled. A permutation for a `when` block is scheduled if and only if the `when` block guard evaluates to true in the current environment but evaluates to false in the previous copy. Scheduled statements are added to the queue, set to be triggered at `currentTime`.

When the interpreter executes a `when` block with a `select` function, it makes a copy of the environment for each argument of the `select` function. Each copy continues execution with its assigned argument returned from the function. After execution, if the copy has events remaining, the copy is enqueued in the search queue sorted by priority (start time of the first remaining event). Until a fixed depth, the search queue is evaluated similarly to the event queue: popping the first environment and running its first remaining event. Unless the popped environment has no more remaining events, it is pushed back into the search queue. While executing events in environments in the search queue, the interpreter keeps track of the environment that completes the most number of requests. In case of ties, it randomly chooses an environment. At the end of search, the `select` argument that created the best noted environment is returned in the original environment, which resumes the regular execution.

4 Evaluation

We present and discuss insights gained from implementing an elevator control system in SASIL. The implemented specification is included in appendix A.

The resulting elevator control system implementation only spans 200 lines of code (LoC). This is likely because the functionality implemented in SASIL describes what the elevator system does in response to events while the SASIL interpreter executes how the system schedules requests. For reference, the interpreter includes 3,200 lines of Ruby and 400 lines of Enso code. On Figure 6, the lines of code used to implement an elevator control system were compared between various open-source GPL implementations found on GitHub. Although our implementation has a significantly lower number of lines, it must be noted
that our implementation does not simulate some logic present in other simulators. Namely, it does not handle elevator capacity or buttons used to explicitly open or close the doors. It also only displays progress through the terminal and thus lacks graphical output.

<table>
<thead>
<tr>
<th>Project</th>
<th>Language</th>
<th>Extra Features</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our Implementation</td>
<td>SASIL</td>
<td></td>
<td>200</td>
</tr>
<tr>
<td>SASIL Interpreter</td>
<td>Ruby and Enso</td>
<td>DSL Parsing</td>
<td>3,600</td>
</tr>
<tr>
<td>Elevator Saga¹</td>
<td>JavaScript</td>
<td>Graphics</td>
<td>1,500</td>
</tr>
<tr>
<td>elevator simulator²</td>
<td>C++</td>
<td>Graphics</td>
<td>4,300</td>
</tr>
<tr>
<td>ElevatorSimulator³</td>
<td>C#</td>
<td>Graphics</td>
<td>3,000</td>
</tr>
<tr>
<td>Knuth’s Elevator Simulatør⁴</td>
<td>Go</td>
<td></td>
<td>700</td>
</tr>
<tr>
<td>ElevatorJS⁵</td>
<td>JavaScript</td>
<td>Bare Graphics</td>
<td>1,600</td>
</tr>
</tbody>
</table>

Figure 6: Lines of Code Comparison Against Open Source Elevator Simulations

To verify the elevator system implementation, we ran the interpreter using Figure 5 as user input. The system only had one elevator operating, and each transition was set to run for one second. The search algorithm was limited to 1,000 iterations. Detailed results are shown on Figure 7. An important part to note is that the elevator switched from targeting floor 7 at time 50 to targeting floor 5 at time 51. This shows that the automatic selector is capable to changing decisions to adapt to new input. Unfortunately, these user inputs took 125 seconds on average to fully run. This is due to how the search procedure makes an exponentially increasing number of copies of the environment. This is also due to the fact that the interpreter doesn’t compile or optimize repeated sections of the code.

<table>
<thead>
<tr>
<th>Request</th>
<th>Start Time (s)</th>
<th>End Time (s)</th>
<th>Duration (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floor 1, Up</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Floor 2, Elevator 1</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Floor 7, Down</td>
<td>50</td>
<td>61</td>
<td>11</td>
</tr>
<tr>
<td>Floor 1, Elevator 1</td>
<td>61</td>
<td>70</td>
<td>9</td>
</tr>
<tr>
<td>Floor 5, Up</td>
<td>51</td>
<td>53</td>
<td>2</td>
</tr>
<tr>
<td>Floor 6, Elevator 1</td>
<td>53</td>
<td>57</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 7: Individual Results to Figure 5

To test how effective the SASIL interpreter performs, we set up an exper-
iment consisting of varying the sequence of user inputs against the traditional elevator algorithm and the SASIL search procedure. Using the simulation’s output, we compared the average and maximum time it took to complete each type of request. Three different sequence of user inputs were tested: The Lobby Only test case consisted of people going from the first floor to the other upper floors. The Leaving Only test case consisted of people going from the upper floors down to the first floor. The Misc test case consisted of people going from any floor to any floor. Each test case had a person appearing with an average period of 8 seconds. Both test cases ran on a system with two elevators, each taking one second to complete any transition. Figure 9 shows the summarized results for each test case.

<table>
<thead>
<tr>
<th>Type</th>
<th>Average Time (s)</th>
<th>Maximum Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floor Request</td>
<td>4.3</td>
<td>11</td>
</tr>
<tr>
<td>Car Request</td>
<td>5.7</td>
<td>9</td>
</tr>
<tr>
<td>All</td>
<td>5</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 8: Results Summary for Figure 5

<table>
<thead>
<tr>
<th>Type</th>
<th>Elevator Alg.</th>
<th>SASIL Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg Time (s)</td>
<td>Max Time (s)</td>
</tr>
<tr>
<td>Floor Request</td>
<td>3.8</td>
<td>14</td>
</tr>
<tr>
<td>Car Request</td>
<td>6.8</td>
<td>9</td>
</tr>
<tr>
<td>All</td>
<td>5.3</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Elevator Alg.</th>
<th>SASIL Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg Time (s)</td>
<td>Max Time (s)</td>
</tr>
<tr>
<td>Floor Request</td>
<td>5.3</td>
<td>10</td>
</tr>
<tr>
<td>Car Request</td>
<td>10.5</td>
<td>12</td>
</tr>
<tr>
<td>All</td>
<td>7.0</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Elevator Alg.</th>
<th>SASIL Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg Time (s)</td>
<td>Max Time (s)</td>
</tr>
<tr>
<td>Floor Request</td>
<td>3.0</td>
<td>9</td>
</tr>
<tr>
<td>Car Request</td>
<td>5.3</td>
<td>8</td>
</tr>
<tr>
<td>All</td>
<td>4.0</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Elevator Alg.</th>
<th>SASIL Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg Time (s)</td>
<td>Max Time (s)</td>
</tr>
<tr>
<td>Floor Request</td>
<td>3.9</td>
<td>14</td>
</tr>
<tr>
<td>Car Request</td>
<td>7.2</td>
<td>12</td>
</tr>
<tr>
<td>All</td>
<td>5.3</td>
<td>14</td>
</tr>
</tbody>
</table>

Figure 9: Results Summary

also written in SASIL but without `select` functions
The Elevator Algorithm performs exactly the same as the SASIL search in the Lobby Only test case; we speculate that the reason results match is because all floor requests start from the lobby floor. Since car requests are only created once a floor request is completed, this causes the elevator cars to regularly return back to the first floor to handle incoming floor requests. The elevator algorithm follows this motion by definition.

On other test cases, we speculate that the SASIL search potentially delayed completing a floor request to favor an existing request. This caused the elevator algorithm to finish floor requests faster than the search when dealing with the Leaving Only test case. Combined with the Lobby Only results, this skews the combined section to have a faster average time duration for floor requests when completed by the elevator algorithm.

However, the SASIL search still performs better in average due to the fact that it has information about the search space. Thus, elevator cars can cooperatively distribute requests between themselves. In the combined average case, the SASIL search performs 1.041 times faster than the elevator algorithm.

Implementations in SASIL proved to be slower than realtime if the \texttt{select} function was used. The test cases for figure 9 all took around 550 realtime seconds to execute around 60 simulation seconds of content. We further discuss the work necessary to optimize the interpreter in the Conclusion section below.

5 Related Work

[1] also uses the shortest distance algorithm to schedule requests in an elevator control system. However, they are mainly focused on implementing a full simulation including the algorithm. Unlike their specific simulation, the SASIL interpreter can infer the shortest path for other systems described using the same language.

6 Conclusion

SASIL is a domain-specific language to describe executable specifications for scheduling systems. Unlike general-purpose specification languages, SASIL automatically infers the best schedule of handling requests by searching through simulation space. Since the interpreter handles how the system schedules requests, a SASIL programmer only needs to program what the system does in terms of reactive events.

We developed an elevator control system using SASIL to test its search algorithm. Results show that the SASIL interpreter completed a request 1.04 times faster in average when compared to the elevator algorithm. However, the interpreter in average took 550 realtime seconds to complete 60 simulation seconds of events.

Since our interpreter ran slower than realtime, future work is needed to optimize the simulation speed. This would include sharing environments between
the interpreter and the search algorithm, using the same instance without explicitly copying all data in the environment. Another would be to collect leftover garbage from local variables creating objects.

Acknowledgments I would like to thank my faculty advisor, Dr. William Cook, for the suggestion into this line of research and for the guidance throughout the process.

References


Appendix A  SASIL Specification of an Elevator Control System

data Direction = Up | Down;
data OptionalDirection = Direction | nil;

data Floor =
  const number : int<MinFloorNum, MaxFloorNum>,
  excludefromprint buttons : bool[Direction] = [];

data DoorState = Closed | Opened;
data Elevator =
  excludefromprint buttons : bool[Floor] = [],
  floor : Between<Floor>,
  committedDirection : Direction | nil = nil,
  doorState : Between<DoorState> = Closed,
  movementState : Stopped | Moving = Stopped;

data RequestCore =
  destination : Floor;
data FloorRequest =
  direction : Direction,
  using core : RequestCore;
data CarRequest =
  elevator : Elevator,
  using core : RequestCore;

data State =
  floors : Floor[int<MinFloorNum, MaxFloorNum>]
    = [Floor{number : key}],
  elevators : Elevator[int<0, NumElevators - 1>]
    = [Elevator{floor : floors[MinFloorNum]}];

////////////////////////////////////////
/// Global State
////////////////////////////////////////

var MinFloorNum : int = 1;
var MaxFloorNum : int = 5;
var NumElevators : int = 2;

var state : State = State{};

////////////////////////////////////////
/// Input
////////////////////////////////////////

for floor in state.floors, dir in Direction {
  when floor.buttons[dir] {
    schedule(FloorRequest{direction: dir,
      core: RequestCore{destination: floor}});
  }
}

for floor in state.floors, elevator in state.elevators {
  when elevator.buttons[floor] {

schedule(CarRequest{elevator: elevator,  
core: RequestCore{destination: floor}});
}
}

///////////////////////////
/// Output
///////////////////////////

// mapping from condition to event  
// (like lights, bell/sound, door open, etc)

for elevator in state.elevators {
    when elevator.movementState == Moving
        && elevator.floor is Floor {
        output floorNumberDisplay(elevator, elevator.floor);
    }
    when elevator.doorState != Closed {
        output elevatorDirectionDisplay(elevator,  
elevator.committedDirection);
        output arrivedElevatorDirectionDisplay(elevator.floor,  
elevator, elevator.committedDirection);
    }
    when elevator.doorState == Closed {
        output arrivedElevatorDirectionDisplay(elevator.floor,  
elevator, nil);
    }
    when elevator.movementState == Stopped
        && elevator.committedDirection == nil {
        output elevatorDirectionDisplay(elevator, nil);
    }
    when elevator.movementState == Moving {
        output elevatorDirectionDisplay(elevator,  
elevator.committedDirection);
    }
}

///////////////////////////
/// Core
///////////////////////////

for request in requests, elevator in state.elevators {
    when elevator.doorState != Closed {
        if completed(request) return;
        if (request is CarRequest && request.elevator != elevator)
            return;
        if elevator.movementState == Stopped
            && elevator.floor == request.destination {
            if request is CarRequest
                || request.direction == elevator.committedDirection {
                complete(request);
            }
        }
// turn off buttons
    elevator.buttons[request.destination] = false;
    request.destination.buttons[elevator.committedDirection] = false;
});

for elevator in state.elevators {
    when elevator.doorState == Opened {
        after 1 {
            after elevator.doorState <= Closed in DoorState for 1 {
                elevator.committedDirection = nil;
            }
        }
    }
}

for request in requests {
    @Select
    when !completed(request) {
        var execute: bool = select(true, false);
        if !execute return;

        var elevator : Elevator = select(state.elevators);
        if (request is CarRequest && request.elevator != elevator) return;

        // stopped on request floor
        if elevator.movementState == Stopped && elevator.floor == request.destination {
            if elevator.doorState == Closed {
                if request is CarRequest {
                    elevator.committedDirection = select(Up, Down);
                } else {
                    elevator.committedDirection = request.direction;
                }
            }
            elevator.doorState <= Opened in DoorState for 1;
        }
    return;
}

    if elevator.movementState == Stopped {
        if elevator.doorState == Closed {
            elevator.movementState = Moving;
            elevator.committedDirection = 
                elevator.floor.number < request.destination.number ? Up
            : Down;
            after elevator.floor <= request.destination in state.floors for 1 {
                elevator.movementState = Stopped;
                elevator.committedDirection = nil;
            }
        }
    } else { // if moving
if !(elevator.floor is Floor) return;

if elevator.committedDirection == Down
    && elevator.floor.number < request.destination.number
|| elevator.committedDirection == Up
    && elevator.floor.number > request.destination.number {
    // stop moving the wrong direction!
    after 1 {
        if elevator.committedDirection == Down
            && elevator.floor.number < request.destination.number
        || elevator.committedDirection == Up
            && elevator.floor.number > request.destination.number {
            elevator.movementState = Stopped;
            elevator.committedDirection = nil;
        }
    }
} else {
    after elevator.floor <- request.destination
    in state.floors for 1 {
        elevator.movementState = Stopped;
        elevator.committedDirection = nil;
    }
}