ElevatorLib: A Generic Library for Elevator Systems
A Software Engineering Case Study in PSF and Go

Bas van Berckel

June 17, 2019
Abstract

The Process Specification Formalism language, or PSF, provides a way to formally define a specification of any process, which can then be validated through the use of simulations and animations. In this thesis, we perform a case study of incorporating PSF into a software engineering process, by creating a library for elevator systems. We evaluate the impact this has on the process as a whole. We pay special attention to opportunities for generation and automation, to augment the viability of using PSF in a real software engineering project. We will conclude that PSF does indeed have a positive impact on such projects and that there are several opportunities for making the use of PSF easier and faster, by including generation and automation tools in several steps of the process.
A large part of all software engineering projects, be it commercial or academical, will start with a specification or at least a set of requirements. Depending on the paradigm the engineer or team of engineers adheres to, the first step will be to draw up a control graph, write tests or start defining models and views. No step is spent on the actual specification itself, even though this is the core of the project and determines nearly all work to be done. A better approach would be to formalize the specification after which it can be validated before any implementation begins. With this, any mistakes in the specification are caught earlier on in the process, instead of popping up halfway through. This has many advantages, as problems caught later on tend to be solved less elegantly, to fit within the work done so far. In general, solving such an issue later on in the project will take more time and potentially hold up more workers, resulting in higher costs. Having these issues smoothed out beforehand will prevent these extra costs.

One method for this formalization of the specification is with the use of Process Specification Formalism [MV90], or PSF. PSF builds upon the concept and language of process algebra, which is a way of describing a program or process in algebraic terms. Where process algebra is largely theoretical, PSF brings many practical aspects through the PSF Toolkit. This allows for simulation and visualization of the specification, to allow the engineer to validate his specification. Without implementing a single aspect, the specification can be tried against a number of scenarios the eventual product should be able to handle, which should bring to light any conceptual errors. After the specification is deemed valid, it can be used to lay the groundwork for the implementation, as the structure of parallel execution units has already been described, as well as any communication between these.

This thesis aims to leverage PSF and its Toolkit in a software engineering process, by designing a specification for a library for elevator systems. The library will enable a user to define a specification for an elevator system for any number of floors and elevators. After validating the library and the specifications it yields, we will implement it in Go. By creating the specification first and validating it before starting to implement it, we can ensure that the eventual implementation is also correct. We will make use of the foundations for Software Engineering with PSF as created by Bob Diertens [Die09]. With this case study, we can evaluate the usefulness of PSF in such a project, as well as experiment with a proof of concept for automation and generation within this process.

The rest of this chapter will introduce the concepts that come into play within this case study. Following that, Chapter 2 describes the specification we created for the elevator system, as well as the generation possibilities for the specification itself and the animation. Chapter 3 defines a possible step-by-step translation of this translation into Go, with a focus on consistency to allow for automation of this process. The final chapter is a discussion of the process, reflecting on what was achieved and what further projects might arise from this.
1.1 Process Specification Formalism

PSF has its roots in ASF, or Algebraic Specification Formalism. This formalism can describe abstract data types algebraically [BHK89], in the form of sets and functions. In PSF, this is found in the data modules. Along with the process modules, which contain process algebra, PSF can describe any process formally and in such a way that it can be compiled and simulated. This section describes the format and syntax of PSF used in this project. Full documentation can be found in [MV90].

1.1.1 Process Algebra

Process algebra, or more formally ACP, Algebra of Communicating Processes, is a theory and notation for describing parallel processes. Together with ASF, this forms the basis of PSF. A short overview of the syntax of ACP is given here, for more background and a full documentation we refer to [BK86].

![Table 1.1: Basic operators of ACP](image)

<table>
<thead>
<tr>
<th></th>
<th>(\cdot)</th>
<th>(\parallel)</th>
<th>(\mid)</th>
<th>(\partial_H)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sequential composition</td>
<td>parallel composition</td>
<td>communication</td>
<td>encapsulation</td>
</tr>
</tbody>
</table>

For this project, we make use of five ACP operators, listed in the table above. The alternative and sequential composition operators (also called sum and product) together describe the flow of a single process. Taking the abstract actions \(a\) and \(b\), we can form a simple process \(P\) as an example:

\[
P = (a + b) \cdot b
\]

As with most algebras, parentheses denote priority of operation. Thus, in this example, the process \(P\) consists of firstly either action \(a\) or action \(b\), followed by another action \(b\).

The parallel composition operator, or merge, combines two sequential processes into a single parallel process. At each execution step, the first action from the left side can be done, or the first action from the right side. Taking our previous process \(P\), we define a new process \(Q\) and merge them into process \(M\):

\[
Q = b \cdot a
\]

\[
M = P \parallel Q
\]

In this new process \(M\), a possible order of execution would be, \(a\) from \(P\), \(b\) from \(Q\), \(b\) from \(P\) and finally \(a\) from \(Q\).

In addition to simply having the processes run alongside each other, certain actions can also be defined to communicate together:

\[
c = a|b
\]

Meaning that actions \(a\) and \(b\) can communicate together, or execute simultaneously, which we call action \(c\). With this communication, it is still possible to run the previous example execution, but it is now also an option to combine the actions \(a\) and \(b\) into \(c\). For example: \(c\), i.e. \(a\) from \(P\) and \(b\) from \(Q\), \(a\) from \(Q\), then finally \(b\) from \(P\).

With the final operator, encapsulation, we can restrict execution options. In most cases, this means eliminating communicating actions from executing separately. In ACP, the user declares a set of atoms, usually called \(H\), which is passed to the \(\partial_H\) operator along with the process to encapsulate. Thus, if we fill \(H\) with \(a\) and \(b\) to encapsulate \(M\):
\[ H = \{a, b\} \]
\[ E = \partial_H(M) \]

We end up with a process \( E \) which has only one possible execution, namely \( c \) followed by another \( c \), as no individual atom is available anymore.

The actions \( a, b \) and \( c \) serve as the constants in this algebra, which in ACP are called atomic actions, or atoms for short, as they are “indivisible” process units. They can take any name and may also carry data, denoted as \( \text{atom}(\text{data}) \). The combination of an atom with a specific piece of data, is in itself an atom, thus when communicating with another atom both must have the same data. A special case is the deadlock, or \( \text{delta} \), constant. This constant represents the absence of possible atomic actions and can be inserted explicitly or result implicitly from execution. One instance where this might occur is when all atoms available are encapsulated, which would result in a deadlock.

### 1.1.2 Extensions to ACP

PSF comes with several extensions on standard ACP. Firstly, there are iteration based variants of the alternative and parallel composition operators, \textit{sum} and \textit{merge}. These take an iteration and a process as arguments, then expand into their respective compositions of the process with the iteration variable replaced:

\[ \text{sum}(\text{item in Set, a(item)}) \]

For a set (explained in 1.1.5) with items \( \text{A} \) and \( \text{B} \), this is equal to:

\[ a(\text{A}) + a(\text{B}) \]

The same applies to the merge operator.

There is also a conditional construct, which limits alternative cases to a certain condition. If the condition is not met, the process after it is replaced with a deadlock and thus cannot be executed:

\[ [\text{data} = x] -> X(\text{data}) \]
\[ + [\text{data} = y] -> Y(\text{data}) \]

In this snippet, at most one of the cases is available for execution, depending on what \( \text{data} \) is equal to. If, for example, \( \text{data} \) is equal to \( y \), the process is equal to:

\[ Y(\text{data}) \]

since the other case has been transformed to a deadlock and is no longer a possible choice.

Finally, \[\text{Die94}\] introduced some extra operators to PSF, of which \textit{priority} is used in this case study. This operator works similarly to the \textit{encaps} operator, taking one or more sets and a process as arguments. Any atoms in the first set take priority over atoms in the second set, which take priority over atoms the third set, etc. An atom having priority means that when it is available for execution, atoms with a lower priority cannot be chosen. Atoms that are not in any set, do not have a priority level and are always available.

\[ \text{HighP} = \{a\} \]
\[ \text{LowP} = \{b\} \]
\[ \text{prio}(\text{HighP} > \text{LowP}, \]
\[ (a + b + c) \]

This example process would, without priority, simply present a choice between atoms \( a, b \) and \( c \). With this priority defined, it only presents a choice between \( a \) and \( c \), since \( a \) takes priority over \( b \). \( c \) is not in any set and is always available.
1.1.3 Processes

In PSF, ACP terms are contained in processes. Processes can be used in other processes, similar to atoms. They can be viewed as simple placeholders to be replaced by an actual process. In the following example, process P1 is equal to P2:

1. \[ P_1 = a \cdot (a+b) \cdot a \]
2. \[ Q = a + b \]
3. \[ P_2 = a \cdot Q \cdot a \]

A process can also be recursive, even with itself, to produce possibly infinitely many processes. A common trope is the simulation of worker cells passing around products. A simple example is the following “factory” which produces a product A. The first part of the product is made in WC1, which passes it on to WC2, which finished the product. These are separate entities that can run in parallel, but WC2 can only work when it has received a half-product from WC1:

1. \[ WC_1 = \text{build}(\text{half}(A)) \cdot \text{send}(WC_1, WC_2, \text{half}(A)) \cdot WC_1 \]
2. \[ WC_2 = \text{recv}(WC_1, WC_2, \text{half}(A)) \cdot \text{build}(A) \cdot WC_2 \]
3. \[ \text{Factory} = WC_1 || WC_2 \]

Finally, we define the communication:

1. \[ \text{comm}(WC_1, WC_2, \text{half}(A)) = \text{send}(WC_1, WC_2, \text{half}(A)) | \text{recv}(WC_1, WC_2, \text{half}(A)) \]

1.1.4 Modules

In PSF, everything is contained in a module. These are based on the modules in ASF and largely follow the same structure. We will not go through all parts of a module, but only discuss the main aspects. There are two types of modules, the Data module and the Process module. While there is some overlap, these modules serve different purposes. The Data module is passive and only contains definitions. While process modules may also contain some definitions, the main part of the Process modules is the processes, as described in 1.1.3.

1.1.5 Data and Sets

Data in PSF makes processes flexible. The Factory process described above could only produce one product, but in PSF it is possible to define a set of products in a Data module. With such a set, operators from the PSF extensions become available. To expand the Factory to also produce a product B, we define a data module that contains our products:

1. \[ \text{data module} \ Products \begin{align*}
2 & \begin{align*}
3 & \text{begin} \ \
4 & \text{sorts} \ \
5 & \text{PRODUCT} \ \
6 & \text{functions} \ \
7 & A : \text{PRODUCT} \ \
8 & B : \text{PRODUCT} \ \
9 & \text{half} : \text{PRODUCT} \rightarrow \text{PRODUCT} \ \
10 & \text{end Products} \end{align*}
\end{align*} \]

This creates a sort, or data type, PRODUCT, which has three functions that yield it: A and B which are simple PRODUCTs, and a modifier half, which takes another PRODUCT and yields a PRODUCT, to signify a half-product. Our previous Factory can then be expanded to produce both of these:
process module Factory
begin
  processes Factory, WC1, WC2
  definitions
    WC1 = sum(p in PRODUCT, build(half(p)) . send(WC1, WC2, half(p))) . WC1
    WC2 = sum(p in PRODUCT, recv(WC1, WC2, half(p)). build(p)) . WC2
    Factory = WC1 || WC2
end Factory

For brevity, some parts are left out of these modules. A full working version can be found in appendix A. This module takes the processes defined earlier and wraps them in sum operators. This creates an alternative composition of these processes for either product. However, because the variable is the same throughout the process, the product can’t suddenly change into the other product.

1.1.6 Equations

PSF also provides a feature to manipulate data, through a Term Rewrite System. This is done through equations in the data modules of PSF:

data module Products
begin
  variables
    p : -> PRODUCT
  equations
    [othp] otherProduct(A) = B
    [othp] otherProduct(B) = A
    [full] fullProduct(half(p)) = p
end Products

With this set of equations, we can transform products into other products. When used in a process definition, they are matched to the left side of the equation, then replaced by the right side. When a variable is used in the equation, this is instantiated with the value passed in the process definition. The names in square brackets are simply for debugging purposes and do not have any effect on the process itself.

1.1.7 Variables

The last aspect of PSF relevant to this project is variables. Variables can be used to define multiple variations of a process, similar to the data in atoms. We could define several different processes that do the exact same thing, or we can define it once, with a variable:

process module P
begin
  variables
    id : -> PRODUCT
  definitions
    P(id) = a(id) . P(id)
end P

This trivial process does a single atomic action and then repeats itself. Due to the addition of the variable id, there are now as many varieties of this process as there are varieties of the data type of id.
1.2 PSF Toolkit

The PSF Toolkit is a variety of programs and libraries that provide a way to implement and validate a specification. Its core functions consist of compiling, simulating and animating any PSF specification. The toolkit is mostly based on the Tool Interface Language (TIL), an intermediate language between all these tools [MV89]. A TIL specification consists of tables, in the form of keyed tuples, containing all the components of the PSF specification. The toolkit’s compiler will transform the PSF specification into TIL, from which the simulator can run a simulation by extracting the process status, i.e. the possible atomic actions to run at any given point in the process. On a certain subset of specifications, an animation can be automatically generated, otherwise this can be supplied manually to work in conjunction with the TIL specification.

1.2.1 Compilation

PSF compilation, like most other compilations, consists of several steps to eventually reach a specification in TIL [Vel90]. The first step is to collect and organize all the necessary modules. The specified files are scanned for defined modules and imports, then these imports are resolved by matching them with modules defined elsewhere. If the imports are not resolved with the given files, the supplied libraries are used to fill the gaps. Some libraries are shipped with the PSF Toolkit, but project-specific libraries can be added.

These modules are placed in separate files, which in turn are parsed separately into MTIL, the first of two intermediate languages. These Modular TIL files are then normalized to a single Intermediate TIL file. This step consists of resolving all the dependencies of an MTIL file with the corresponding (already normalized) ITIL files. The final step is to flatten this single ITIL file into the resulting TIL file. Two optional steps may follow, which respectively transform sorts into sets or check the defined equations for a valid Term Rewrite System, both to ensure the simulator can successfully run the simulation.

1.2.2 Simulation

The TIL file can be read by the simulator, which can extract the possible atoms at any given point, considering limitations such as prioritization and encapsulation, allowing the user to step through the program atom-by-atom, in several ways.

The user chooses a starting point, i.e. the main process, after which the possible atoms are shown in the action choose-list. If a deadlock is reached, meaning no atomic actions are possible, this list is empty and DEADLOCK is shown in the Display screen. The user may step through the program manually, by selecting an atom, or they can opt to let the simulator choose randomly. Both are useful for validation, by simulating a specific scenario or reaching unexpected edge cases, respectively. Figure 1.1 shows a simulation of the simple factory process from the previous sections, after building both products.

The simulator offers some features to allow for easier validation and debugging, such as process tracing, breakpoints and status insight. These enable the user to get a closer look at all the behaviors their specification may show.

Figure 1.1: PSF Simulator running the example specification.
1.2.3 Animation

While simulation itself allows a user to experiment with their specification, animation provides a better understanding, by visualizing states and actions which, depending on the process, may resemble the real world implementation far more than a text-based simulation. Figure 1.2 shows the animator running the same process as in Figure 1.1.

Animation is done in Tcl/Tk, assisted by a library of functions provided in the PSF Toolkit [Die09]. The structure of an animation consists of an initialization, followed by two functions, for generating animation actions and simulation choices, of which the latter is optional. In short, the simulator calls the action function whenever an action is executed. This function should then perform the appropriate animation steps and/or status updates. If specified, the choose function is called afterwards, once for each (newly) available action. The function should add the action to the appropriate object, allowing the user to click an object and select an action, for a more intuitive user interface.

The Anim library provides a variety of objects, mostly basic shapes, which can be moved, altered or annotated when required by an action. These features combined allow for the inclusion of moving parts, buttons and many other aspects of physical or digital processes. In the initialization section, these objects are defined with a name, position, shape and size. Later in the choose and action functions, these objects can be referenced by their name to manipulate them.

The toolkit also provides a tool to generate a simple, generic animation that shows the different processes and the communications between them. This tool does not always work and may break with more advanced functions of PSF. It does allow for faster prototyping or can provide a base for manually created animations.

1.3 Go

For the implementation side of this case study, we will use the programming language Go, or Golang, a modern language with roots in C. The discerning feature of Go is its native support for concurrency and interprocess communication, making it a perfect fit for translating a PSF specification. This section will provide an overview of some features of Go, mainly those related to concurrency. The full specification is available at [Aut18].

1.3.1 Concurrency

Go natively provides several features to support concurrent programming, of which the most important are Goroutines, channels and the select statement.

Goroutines are similar to threads or subprocesses in other languages that support parallel programming. Unlike most of those languages, goroutines are a language construct available without the need for an external library. Any function or method can be executed in a new thread, called a goroutine. No special declaration needs to be made for the function, it only needs to be called with a special keyword, `go`. 
func hello() {
    // Print "hello" to the stdout stream
    Println("hello")
}

func main() {
    // Launch a new goroutine that runs the hello function
    go hello()
    // Print "world" to the stdout stream
    Println("world")
}

In the code snippet above, the hello function is executed in a new goroutine, in parallel to the original goroutine where it was called. Depending on the execution order of the goroutines, this program would print either "helloworld" or "worldhello".

In a concurrent program, well-defined communication is essential. Go provides this in the form of channels. Channels are the only communication line between different goroutines and can transmit any data of the specified type up- or downstream. The writing and reading of these channels are blocking operations, which provides an easy way to sync up goroutines. While we won’t go into all features of Go channels, the following example shows the basic use of a channel, to return data from a goroutine to the original goroutine:

func hello(ch chan string) {
    // Send "hello" over the channel
    ch <- "hello"
}

func main() {
    // Create a new channel
    ch := make(chan string)
    // Launch a new goroutine for the hello function, passing the channel as argument
    go hello(ch)
    // Receive the result from the hello function
    result := <-ch
    // Print both the result and "world"
    Println(result, "world")
}

In this example, instead of having the hello function print its own result, it sends it through the channel that was passed as an argument. The original goroutine receives this result, which is a blocking operation, thus it will not continue execution before it has received its result. Because of this, the output is now deterministic and will always be "hello world".

Go also provides the select statement. This construct is very similar to the switch statement, but instead of operating on comparisons, it operates on channels. Each case has a channel operation, i.e. a send or receive operation, and the select statements waits until one or more of the operations can be executed. It chooses randomly which of the available operations is executed, after which the (optional) code within that case is executed. This construct allows for communication with multiple goroutines, without having to choose which to communicate with at what time. This final example shows another iteration of the hello world program, that randomly displays either a lowercase or uppercase of the phrase:
Here, the original goroutine launches two other routines, who both try to write their result to their respective channels. However, due to the select statement randomly choosing one of the cases, only one result is received and printed. Thus, this program will output either "hello world" or "HELLO WORLD" randomly.
CHAPTER 2

Specification

For this case study, we experiment with the idea of a generic specification, in the form of a library and a single configuration and instantiation module. With this, it is possible to validate the specification for any number of process setups. In the case of the elevator, this means that the number of floors and elevators can be altered without having to edit the specification. In other words, this specification will work in any situation where an elevator system needs to be constructed.

Other examples of where a generic specification may be useful are processes that are network based, having variable numbers of clients and servers, which can now be accounted for and tested with a single change in the configuration. By providing such a simple interface to the variables of the specification, as opposed to creating a new module with imports and bounds, generation and automation are also made much easier. This could also have applications of integrating variables into the simulator, allowing the user to test multiple configurations without leaving the toolkit GUI.

The chapter contains a description of the library created for this specification, ElevatorLib, as well as the configuration and instantiation module. After that, we look at the process of validating this specification and the scenarios it needs to be able to complete. Finally, we review the possibilities for automation and generation of specifications and validation thereof.

2.1 ElevatorLib

2.1.1 Modules

The library consists of the following modules:

- **IDs** and **IDsets**
  These two modules provide the identifiers for all components, as well as equations to find relations between the components. The modules are separated into a data and a process module, to be able to make use of all features.
  **IDs** contains the sort ID, as well as all functions that make up the identifiers. The format of these is expanded on in a later section. It also contains all the equations needed for the logic in the functional components. These include comparison of floors, to see which is higher or lower, for example.
  **IDsets** contains the sets of identifiers to differentiate between types of components. It uses the configuration values to enumerate the different values in the identifiers, resulting in the sets of all elevators, floors and buttons. It also contains logic to exclude buttons on the top and ground floor, as these floors wouldn’t have an Up or Down button, respectively.

- **Commands** and **Control**
  The Commands and Control modules contain all the data and communications that can occur between the components. Commands contains the functions for all types of commands, while Control contains all the functions to send these commands. The available
commands and communication functions are explained further in a later section. Finally, it also contains the push atom, used to simulate a user pushing one of the buttons. The Control module also exposes the set of partial communication atoms, which are to be encapsulated, as well as the set of full communication atoms that are to receive priority.

- **Direction and Movement**  
The atoms simulating movement are provided in these two modules. **Direction** contains the Up and Down directions; **Movement** contains the functions for elevators moving and doors opening and closing. The movement is split up into departing and arriving atoms, to allow for input when the elevator is in motion. Similar to the Control module, it exposes sets for encapsulation and prioritization. This module has two separate priority sets, to give priority to the doors opening over moving to the next floor.

- **ElevatorButton and FloorButton**  
These are the two types of buttons and the first real functional modules. Both buttons operate in a similar way. In their default state, they wait for a push atom, after which they send commands to the appropriate components. The FloorButton sends a Call command to one of the doors on its floor, prompting the FloorDoor module to summon an elevator. The ElevatorButton sends an Expect command to the specified floor, so that it knows to open its doors when the elevator arrives. It also sends a Goto command to the elevator itself, to make it start moving or, if it was already in motion, update its final destination if needed. After this, both buttons go into their active state, awaiting the Reset command. This structure is analogous to the button lighting up until the requested atom has been performed.

- **Elevator and FloorDoor**  
These modules represent the main components of the system, the elevators and the floors. These go through various states, each with different available atomic actions, which we will go into further in the next section.

- **ElevatorSystem**  
This is the module that ties the whole system together. It contains the main process which merges the processes of the previous modules. It also accounts for the priorities of the different atoms and encapsulates any partial communications. The processes are merged based on the sets of IDs formed by the IDsets module.

2.1.2 States

Being the main components, with the most complicated processes, the FloorDoor and Elevator modules will be explained in detail in this section. The PSF processes stand for the states that the components can go through, which are listed below.

**FloorDoor**

The FloorDoor module, representing a single door of each floor for a specific elevator, contains states to signify an elevator that is expected to arrive, as well as states for when an elevator is currently there.

- **Idle**  
The default state. In this state, no action is being performed or expected to occur. The system is waiting for input from either Floor buttons or Elevator buttons, to go to the Waiting or Expecting states respectively.

- **Calling**  
Calling is a simple state that tries to call an elevator to come, or accepts an already arriving elevator and opens the doors.
• **Waiting**
  A floor in the *Waiting* state has had one or both of its buttons pressed and is now waiting for an elevator to arrive. Two variations of this state exist, either for a single direction requested, or for both. If an elevator arrives for a certain direction, the floor “drops” to a *Waiting* state for the reverse direction. If an elevator is requested for the direction it was not yet waiting for, it will transition to the *Waiting* state for both directions.

• **Expecting**
  Expecting is the state in which an elevator button was pressed and an elevator is coming to that floor. The state is different from the *Waiting* states in what happens after an elevator arrives. Again, two varieties exist, but for a slightly different purpose. The regular *Expecting* state accepts an elevator in any state and will go to the *Idle* state after an elevator arrived. The *Expecting* state with a direction will do the same if the elevator arrives for the specified direction, if it arrives in the opposite direction it will switch to the *Waiting* state for that direction.

• **Open** and **OpenToWaiting**
  An elevator is currently at this floor with the doors open. A reset command is sent to the appropriate direction button if it is waiting for such a command. After the doors close, the floor goes into an *Idle* state, unless the *OpenToWaiting* variant was used, in which case it will go back into the *Waiting* state for the opposite direction.

**Elevator**

The Elevator modules contains states for when it is in motion and for when it is at a requested floor.

• **Idle**
  The default state. It is waiting at a certain floor for input, either from a *FloorDoor* summoning it, or from its own buttons to go to a certain *FloorDoor*. Both will send the elevator into the *Arriving* state. The doors can also open if the *FloorDoor* is opening, when a user pressed the button on the current floor.

• **Moving**
  A small state to signify motion with the two movement atoms *depart* and *arrive*. It switches to the *Arriving* state after motion is completed.

• **Arriving**
  This state branches in two, depending on the variables of the state. If the elevator can be opened, they are opened. If the elevator is at its destination, it will go into the *Idle* state to wait for its next command. If it is not yet at its destination, it will continue moving. This can only be done if the doors cannot be opened, due to the priority of atoms. This state is used for stops earlier than the final destination as well as for an elevator that just received a command to go to another floor and is preparing to start moving.

• **Open**
  The doors are opened, during which time the elevator may receive commands to go to elevators that lie in its current direction. If no (further) commands are received, the doors close and it goes into the *Closed* state.

• **Closed**
  In this state, the elevator decides whether to continue moving or go into the *Idle* state, depending on the destination.
2.1.3 Communication

Communication is done in the form of simple commands with no data, except for the Call and Summon commands which carry a direction. Some commands have already been mentioned in the previous sections, but for completeness they are listed below:

- **Call**
  This command is sent from a FloorButton to one of the FloorDoors on the floor it is on. It prompts the FloorDoor to summon an Elevator.

- **Summon**
  The Summon command is the equivalent of the Call command, but from a FloorDoor to an Elevator.

- **Expect**
  The reverse of the Summon command, it is sent from an ElevatorButton to a Floor, telling it the elevator will be arriving, so that it is ready to open doors when needed.

- **Goto**
  When an ElevatorButton is pressed, it also sends the Goto command to the Elevator, so that it will start moving, or update its destination when necessary.

- **Reset**
  The only command used between two pairs of components. The Reset command can be sent to either type of button, signifying that the requested action has been fulfilled.

Communication works based on the IDs, which in turn are formed in such a way that they contain all the static information about the component they identify. Specifically, the IDs of Buttons contain the IDs of the components they are linked to. With this, some communications can happen with just one ID, instead of the common two IDs for sender and receiver. For example, the Goto command is sent with just the ID of the ElevatorButton, which contains both the ID of the Floor and the Elevator that it is used in. Communications between Floors and Elevators is still done with the regular two IDs.

Due to the way communication works in PSF, we can simulate a queue of commands simply by receiving them only at times where they are useful. This way, a Button for a floor that cannot be served at a certain moment can still be pressed, only sending the actual Goto command when the elevator is ready to travel in that direction. This prevents the need for multiple separate control units to receive and send commands in a specific order and keeps the specification slightly simpler.

2.2 Configuration

To configure a specification for a specific elevator system, a configuration file must be created. This contains several values that identify the number of floor and elevators in the system. A configuration module for a system with two elevators and four floors looks as follows:

```plaintext
1 process module Config
2 begin
3   exports
4     begin
5       sets
6         of NATURAL
7           FloorNumbers = {nat1, nat2, nat3, nat4}
8           GroundFloor = {nat1}
9           TopFloor = {nat4}
10          ElevatorNumbers = {nat1, nat2}
11     end
12   imports
13     Naturals
14 end Config
```
The *FloorNumbers* and *ElevatorNumbers* sets enumerate the identifying numbers for the two types of components. These must be sequential natural numbers starting at one. Natural numbers in the PSF library have the form `natX`, to prevent collision with single digits. Because the PSF simulator is not capable of set formation or manipulation beyond simple enumerators, the entire set must be specified instead of single numbers. For this same reason, the Top and Ground floor must be separately specified, which are used for the generation of the FloorButtons, or more specifically, lack thereof.

To instantiate the specification, a second module is created. This module exists to import the ElevatorSystem module, as well as provide a place to name the specification. This is done as follows, for a system called *TwoElevatorsFourFloors*:

```plaintext
process module TwoElevatorsFourFloors
begin
    imports
        ElevatorSystem
    processes
        TwoElevatorsFourFloors
definitions
    TwoElevatorsFourFloors = ElevatorSystem
end TwoElevatorsFourFloors
```

### 2.3 Validation

#### 2.3.1 Animation

Animation for the elevator specification consists of four parts. Two are the functions required by the toolkit, `ANIM_action` and `ANIM_choose`. The third is another function, `ID_to_item` which is used to transform the IDs used in the specification to simpler IDs usable in Tcl/Tk. Finally, all the animation components must be defined, including the items for the doors, elevators and buttons, as well as lines to show communication paths.

The first three are invariant under the configuration of the system, the latter component changes depending on the number of floors and elevators. Due to the genericity of the IDs, the function `ID_to_item` can easily translate between the varying specification and the animation, making the functions fully generic as well. The workings of these functions is described below:

- **ID_to_item**
  This function takes an ID in the format of the specification. Using regular expressions, it can detect which component type it identifies, as well as extract the information that identifies the exact component. To handle IDs that contain other IDs, the function recurses. It returns a string that matches the original identifier. For example, `FLOOR(nat1)` would translate to `FLOOR1`, and `ELEVATORBTN(ELEVATOR(nat1), FLOOR(nat1))` would result in `ELEVATOR1FLOOR1`. Additionally, it can take a second parameter, 0 or 1, when only the first or second part of an ID is needed.

- **ANIM_action**
  This function only needs to act on atoms related to movement, i.e. doors opening and closing and the elevator going up and down. It uses regular expressions to match these atoms. It also extracts the elevator or floor that is involved and the direction of movement. Using this, it calls the Move function provided by the animation library, to move the item to the correct position. For a moving elevator, it also updates the indicator at the top of the elevator shaft.

- **ANIM_choose**
  This function operates on all atoms and also uses regular expressions to extract the component with which the atom is associated. It then links the atom with the item for that component, using the `ID_to_item` function to get the proper identifier.
The definition of the objects that the animation will work on is not generic and needs to be altered for each permutation of configuration values. Because of the simple naming schematic we used for this project, it is relatively simple to do so. For this animation, each elevator shaft consists of an object for each floor, represented by a door, connected by a passageway. The two types of buttons are in yellow/grey, depending on whether they have been pressed or not. Below each shaft, there is a keypad containing the buttons inside that elevator. On each floor, there is a single set of buttons for the up and down direction, that operates on all elevator shafts. Above each shaft, there is an indicator to show which floor that elevator is currently on. Altogether, an animation for two elevators and three floors looks as in Figure 2.1.

Figure 2.1: Animation of a system with two elevators and three floors. The elevator on the left is at floor 2, the elevator on the right is moving towards floor 2.

2.3.2 Scenarios

In order to test the system beyond random execution to find deadlocks, we create certain scenarios that we expect to occur in the real world. These scenarios allow us to test for actual functionality and behavior, in order to expose flaws in logic and structure. For this project, we divide the scenarios into two categories: Perfect Users and Typical Users. The first category contains the basic features as well as some more complicated scenarios, but limited to users that adhere to the “rules” of the system. In short, this entails a user summoning an elevator, entering and selecting a single destination, then exiting. The other category contains more difficult scenarios, which may not fall under the intended use of the system but will certainly occur in typical use. This includes choosing destinations that are not in the direction of the elevator, choosing destinations at unexpected times and so on. For this case study, we do specify these harder scenarios, but they are not necessary for completion of the project, due to the large amounts of extra complexity these will incur, in some cases. Any production elevator should, naturally, be able to handle these scenarios, but their individual solutions are not particularly interesting for this thesis.

Perfect Users

For the perfect world scenarios, we start of with the most basic of functionalities of an elevator. A user should be able to summon an elevator and, if needed, move to the specified floor. Our first two scenarios are therefore these:

- A user presses a button on any floor. Any elevator moves towards this floor and the doors open, indicating the direction corresponding to the user’s choice, allowing the user to enter.

- A user has entered the elevator and presses a button for any floor that lies in the chosen direction. The elevator should close its doors, move towards the specified floor and open its doors.

With these two scenarios, we can confirm that a basic functioning elevator system has been specified. However, a building does not have multiple elevators to accommodate a single user at once. Scenarios should be specified to simulate multiple users at the same time:
• An elevator is in motion. A user presses a button on a floor between the elevator and its
destination. If the direction of the selected button matches that of the elevator, it should
stop and allow the user to enter.

• The elevator is open, on any floor. Multiple destinations that lie in the elevator’s direction
are selected. The elevator should move towards the closest floor first, then continue to the
next, until it reaches the furthest floor selected.

• Two users arrive on the same floor and select a different direction. Two elevators should
arrive, or the same elevator should arrive twice, one for either direction.

Typical Users

In the real world, users will not always behave as intended, which may complicate the system
when accounting for this. For this system, an example would be requesting floors that lie in the
opposite direction. We define some scenarios to test for such cases:

• The elevator is in motion. It is traveling in a specified direction. A user in the elevator
presses a button for a floor that lies in the opposite direction. The elevator should continue
its path in the current direction, until the last requested floor has been reached. After this,
it should start traveling in the opposite direction to visit the requested floor.

• The elevator is open, for a specific direction, but has no further requests. A user requests
a floor that lies in the opposite direction. The elevator should close normally and switch
to the opposite direction before beginning its journey towards the requested floor.

2.4 Generation

Because of the simple and static format of the configuration and instantiation modules, generation
of specifications for any elevator system is trivial. Animation is slightly more complicated to
generate, due to the various items that must be instantiated. However, a single script can
perform both these tasks, allowing the user to generate an elevator system of any desired size
with a simple command. Figure 2.2 demonstrates a generated animation of a large system.

![Figure 2.2: Initial state of a 8 elevator, 6 floor system animation](image)
To see if the specification we created is useful in the software engineering process, we need to translate it to a regular programming language. As mentioned earlier, Go is specifically useful for this, due to its native concurrency features. To best evaluate the feasibility of including PSF in the software engineering process, we try to keep the implementation close to the specification, transcribing it line for line whenever possible. Naturally, not all features of PSF lend itself to be incorporated into Go, or programming in general, and we will discuss these separately.

3.1 Specification Translation

For this proof of concept, the implementation is done for a single configuration of the elevator system. Complications arise when translating the sum operator into Go, due to the nature of the select statement. A possible solution to this is discussed in 4.4. As one of the aims of this case study is to see if generation is possible, the specification was translated consistently, by defining common structures for recurring bits in the specification. The module structure was also retained, thus a potential generator should only have to translate a single module at a time, with a few exceptions. This section goes through these so called compilation schemes.

Starting with the simplest construct in PSF, the sequential composition, this is trivial to translate. Since Go is an imperative language, sequential composition is equal to adding a new instruction. The following example demonstrating the translation serves mostly to demonstrate the structure of the examples for more complicated schemes:

<table>
<thead>
<tr>
<th>PSF</th>
<th>Go</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1 . p2</td>
<td>p1()</td>
</tr>
<tr>
<td></td>
<td>p2()</td>
</tr>
</tbody>
</table>

To implement atoms, we can use functions. These functions will take the same arguments as their PSF counterparts. Two variations of these functions are necessary, depending on whether the atom is part of a communication. If not, the function does not need to perform any action and simply returns. If it does communicate, it should return a channel which is specifically created for that communication. All these channels should be created at the start of the program and stored in a dictionary based on the parameters of the atom. To discern between sending and receiving atoms, the definition of the communication is used. In the following communication:

```
send(id, data) | recv(id, data) = comm(id, data)
```

send(data) is the sending atom and recv(data) the receiving atom. The functions for both of these will look the same, but the usage is different. The function should look like this:
func send(id, data string) chan bool {
    return commChans[id][data]
}

This will return a boolean channel specific to the \textit{comm} communication and the id and data parameters, thus unique to this atom.

Atoms can also take data as an argument, which must be translated as well. In PSF, this data is a function in itself, which can optionally take parameters. To translate this to Go, we use constants and functions, based on whether it takes arguments. If the data does not take an argument, we simply define a string constant with the same name. If it does take arguments, we create a function which takes as many arguments and returns a string:

PSF
\begin{verbatim}
1 sorts
2 DATA
3 functions
4   data : \rightarrow DATA
5   argdata : ID \rightarrow DATA
\end{verbatim}

Go
\begin{verbatim}
1 const data = "data"
2
3 func argdata(id string) string {
4    return "argdata" + id
5 }
\end{verbatim}

This ensures that all atoms with their respective data have a unique identifier. By differentiating with constants and functions, the notation stays exactly the same as it would in PSF:

PSF
\begin{verbatim}
1 p1(data)
2 p2(argdata(x))
\end{verbatim}

Go
\begin{verbatim}
1 p1(data)
2 p2(argdata(x))
\end{verbatim}

The second operator construct is the alternative composition, which introduces some complications and the first restriction on the specification. In the simplest case, it can be handled with an if statement:

PSF
\begin{verbatim}
1 [ data = x ] \rightarrow
2    p1
3 + [ data = y ] \rightarrow
4    p2
\end{verbatim}

Go
\begin{verbatim}
1 if data == x {
2    p1()
3 } else if data == y {
4    p2()
5 }
\end{verbatim}

Here, the conditional construct is replaced with an if statement with the same conditions. This only works when the processes are simple atoms that do not communicate. If they do, another scheme must be applied, which is the first to make use of Go specific features. If an atom communicates, it can only be executed if its counterpart is also available for communication. This can be emulated with the select statement of Go, which only allows a certain case to be executed when the channel can be read from or written to. A single communicating atom can then be transformed as follows:

PSF
\begin{verbatim}
1 recv(id, data)
2 send(id, data)
\end{verbatim}

Go
\begin{verbatim}
1 <-recv(id, data)
2 send(id, data) <- true
\end{verbatim}
The sending atom is translated to a write operation and vice versa. If these were in different goroutines, they would communicate with each other, like they would in PSF. In an alternative composition, these would be translated as follows:

**PSF**

1. `recv(id1, data)`
2. `+ recv(id2, data)`

**Go**

1. `select {
   2   case <-recv(id1, data):
   3   case <-recv(id2, data):
   4 }`

Depending on the id of the goroutine that does a send action, one of these cases will be chosen, which is the same behavior as in PSF. If the alternative composition is between multiple sequential atoms, the first atomic action for each is used as the case atom, the rest is in the code block for that case:

**PSF**

1. `(recv(id1, data),
   2   send(id3, data))
   3   + (recv(id2, data),
   4   p1(id2, data))

**Go**

1. `select {
   2   case <-recv(id1, data):
   3   send(id3, data) <- true
   4   case <-recv(id2, data):
   5   p1(id2, data)
   6 }`

This also introduces the first restriction of this compilation scheme. If one of the atoms available to choose from is a communicating atom, all others must be as well. The subsequent atoms do not need to be communicating, like `p1` in the example above. There is one exception to this, depending on the desired behavior. There can be a single option that is not communicating, which serves as the fallback. However, if none of the communications can be executed, the fallback is executed immediately, unlike PSF, where other actions can take place that might “reveal” the communication first. This fallback can be reached with the default case of the select statement, which is chosen when no cases can be executed:

**PSF**

1. `(recv(id1, data),
   2   send(id3, data))
   3   + p2`

**Go**

1. `select {
   2   case <-recv(id1, data):
   3   send(id3, data) <- true
   4   default:
   5   p2()
   6 }`

When this is not the desired behavior, all cases must be part of a communication. In PSF, it is also possible to place a communicating atom behind a condition, resulting in a deadlock, i.e. an unavailable option. In Go, we simulate this behavior with an assisting function, `CondChan`:

1. `func CondChan(cond bool, ch chan bool) chan bool {
   2   if bool {
   3     return ch
   4   } else {
   5     return nil
   6   }
   7 }`

This function takes a boolean and a channel, then returns the channel if the boolean is true. If not, it returns `nil`, the NULL value in Go. When `nil` is a case in a select statement, Go ignores it and only looks at the other cases. We can now translate a construct like this:
PSF
1  [data = x]  →
2    recv(id1, data)
3  +  [data = y]  →
4    recv(id2, data)

Go
1  select {
2    case CondChan(data == x, recv(id1, data)):
3    case CondChan(data == y, recv(id2, data)):
4  }

Now, the atom is only an option if the condition evaluates to true. If none of the conditions are met all cases are nil and Go falls into a deadlock, which is the same behavior as PSF would have.

Processes in PSF are translated to functions, with the process itself as the function body. This lends a nearly one-to-one translation, only the names need to be changed to prevent clashing. Since PSF can have overloaded processes, but Go does not, some steps need to be taken to ensure unique names. Each process has the module name prepended, which ensures uniqueness between modules. Within a module, processes with the same name can still exist with different arguments. For this, clashing function names get appended with the number of arguments:

All processes are now uniquely named with a simple renaming algorithm.

The parallel composition operator is trivial to translate due to Go’s concurrency features. By calling each process on either side of the operator with a go operator, goroutines are launched for each, which then run the processes in parallel: (the renaming of processes is left out for simplicity)

The iterative merge operator is also easy to translate, by wrapping the goroutine call with a for loop:
Finally, the last construct that can be translated consistently is the priority operator. While there is no way to do this for the program overall, like the specification has, it can be applied to each alternative composition. The select statement’s default operator plays a key role in this, providing multiple levels of choices:

PSF
1. priority (recv > send)
2. recv(data) + send(data)

Go
1. select {
2.     case recv(data):
3.         default:
4.             select {
5.                 case recv(data):
6.                 case send(data):
7.             }
8. }

(Note that the notation in the PSF part is not correct, it was adapted for conciseness). This select statement will first try to execute the recv atom, if that is not available, it will fall through the default case. Here, it can perform either atom, depending on which is available.

3.2 Implementation specific features

While the specification and its direct translation only deal with the core process of the system, a real system will be much more complicated. A significant difference between a simulation of PSF and running the actual implementation lies in the fact that the simulation is in discrete time, while the implementation is in continuous time. We discuss some of the impact this has on the process in this section. We also look at what is needed to have this core process control an actual system, through the use of drivers.

3.2.1 Continuous time execution

When simulating in PSF, the process is indifferent towards the time that passes between two atomic actions. In our implementation, the time each atom takes to execute changes how the process overall looks. This would become even more complicated when linking the system to actual hardware, for example the doors might be ajar, lengthening the time of completing for the open atom. Another difference is in the selection of atoms. In a PSF simulation, barring any prioritization, we can choose to ignore some atoms for several steps by simply choosing other atoms. In the implementation, choices are made as they are presented, whenever at least one of the options is available. While in most cases this is exactly the intention, it is something that must be kept in mind, both when creating the specification and when creating or validating the implementation.

3.2.2 Drivers

To get actual functionality in our implementation, the atoms need to perform their respective actions. To keep the process part of the implementation as clean as possible, this functionality is abstracted into a driver interface. All non-encapsulated atoms have a driver counterpart, while the ElevatorLib package takes care of calling these driver functions; every atom’s function calls their respective driver function before returning. It passes the same arguments it received to the driver function. If the atom is part of a communication, only the sending atom’s function calls the driver function. The driver only has a function for the communication, not for the separate (encapsulated) atoms.

For this proof of concept, all driver functions print their information and then wait for two seconds. Additionally, the push function has an infinite loop that has a 2% chance of breaking every second, to simulate users pushing buttons at random times. This allows for a simple validation of the implementation. Any more complicated functionality, such as animation or driving hardware object, can be added by extending the functions in the driver package.
3.3 Generation

While this implementation was done by hand, nearly all constructs could be consistently translated. This opens the door towards generating, or transpiling, a PSF specification into Go. Some restrictions will likely apply to the PSF specification, such as the inability to combine communicating and non-communicating atoms in an alternative composition. Some features from PSF will require a convenience library in Go, to include functions for conditions on communicating atoms and the \textit{sum} operator. The Future Work section contains more details on a possible solution for this.
CHAPTER 4

Evaluation

To evaluate the usefulness of PSF in the software engineering process, we will look at how it has helped in creating the implementation. Firstly we will look at how successful we were in creating a generic library for elevator systems in PSF. We will also look at how our results possibly impact the general software engineering process. For this particular case study, one of the points of focus was the feasibility of generation and automation of specifications and eventual implementations, as this would raise the viability of incorporating PSF into the software engineering process. Finally, some possible future work that has risen from this project will be laid out in this last chapter.

4.1 ElevatorLib

We have been successful in creating a generic library for specifications of elevator systems. The library takes two configuration values in the form of sets of floor and elevator numbers. From this, it generates a complete specification with the provided amount of floors and elevators. It results in working specifications for systems with a single elevator on two floors, up to any number of floors and elevators that fit in the computer’s memory. The specifications are able to run all of the scenarios we constructed, but there are still differences with systems that are in use in the real world. Some examples of this are emergency buttons, buttons to open and close the doors and smart scheduling system that can provide optimizations in which elevators are sent to which floors. While also a very interesting topic, we left such aspects out of the scope of this project, as we did not deem those relevant for what we set out to explore.

4.2 Software Engineering with PSF

As [Sta08] and [Sch16] have shown before us, PSF can be of great use in a Software Engineering process. In this case study we looked at elevators, a seemingly simple system that becomes very complex due to the many parts, especially in the case of a system with multiple elevators. Due to the focus on communication within PSF, it becomes much simpler to model such a complex system. When PSF would not be used, a lot of time would need to be spent on the implementation before even the core process could be properly checked and validated. Now, by creating a formal specification first, we can move to a testing phase much earlier, saving a lot of time and money. The question arises, though, if the time spent on creating such a specification is not lost time. As we have seen, once the specification has been created and validated, it can relatively easily be converted to a boilerplate implementation in Go, by following the compilation schemes we devised. After this only the “drivers” need to be created, which in most cases would also need to be done when working without PSF. In conclusion, PSF can definitely be a useful addition to the software engineering process, especially where parallel processes are concerned, possibly even saving on time and costs.
4.3 Generation and Automation

In this case study, we have encountered multiple possibilities for incorporating generation and automation in different parts of the software engineering process. Firstly, in the specification itself, we have created a generic library which can yield arbitrarily large specifications depending on a small amount of configuration settings. An example of a field where such a generic form of specification might be useful, is networking. By creating a specification for a generic network, to which we can add any amount of clients and servers, we can easily validate a networking protocol for various setups.

To perform this validation, an animation can be created. Like the generic specification, the animation can also be made generic. In this case study, the functions of the animations were agnostic towards the number of elevators and floors, meaning they would only need to be created once for all possible specifications created with the library. The only part that would need either manual work or a script for generation, would be the instantiation of all the animation objects.

Finally, the implementation can likely also be automated, at least when done in Go. Although we were not able to devise compilation schemes for all possible constructs in PSF, we expect that with the creation of a Go library for PSF, this may be possible in the future.

4.4 Future Work

During this project, we encountered some possible opportunities that we were not able to fulfill yet. These relate to the automation of the usage of PSF, from creating and validating the actual specification to implementing it in Go. The first is a possible enhancement for the PSF Toolkit simulator. In this case study, we devised a concept for working with configuration values to alter the specification itself. To ease validating such a construction, the simulator could incorporate the settings for these values, which would entail recompiling the specification and running the simulation again.

Another possible future project is a transpiler from PSF to Go. This project has laid some possible groundwork for such a project, but various constructs still need translations. This project includes a beginning for a Go library that would assist in this translation, amongst which is a function that emulates some simple cases involving the \texttt{sum} operator. Go does not have a native option for iteratively defining a select statement, but the \texttt{reflect} package provides a possible solution for this. Complications arise, however, when a \texttt{sum} operator is placed in an alternative composition, as it would now need to extend its select statement with the other cases. The solution chosen in this project can not account for such cases, but we do expect it to be possible, although it might require a low level extension to Go.

4.5 Conclusion

In this project, we were successful in creating a library for specifications of elevator systems of any size. We were able to validate these specifications through the use of the PSF Toolkit, with simulations and animations. After confirming that the system works as intended, by running it through various scenarios, we were able to implement the specification in Go, by using a system of compilation schemes for most parts of the specification. This opens the door for some possibilities for automation within the process of software engineering with PSF. From this case study, we conclude that PSF is a useful addition in the software engineering process and that automation is possible to make the use of PSF easier.
Bibliography


APPENDIX A

Example PSF specification

```plaintext
data module Data
begin
exports begin sorts
HALFPRODUCT, PRODUCT, ID
functions
WC1 : -> ID
WC2 : -> ID
A : -> PRODUCT
B : -> PRODUCT
half : PRODUCT ->HALFPRODUCT
end
end Data

process module Factory
begin imports Data
atoms
build : PRODUCT
build : HALFPRODUCT
send : ID # ID # HALFPRODUCT
recv : ID # ID # HALFPRODUCT
comm : ID # ID # HALFPRODUCT
processes Factory, WC1, WC2
sets
of atoms
H = {send(s, r, p), recv(s, r, p) | s in ID, r in ID, p in HALFPRODUCT}
communications
send(s, r, p) | recv(s, r, p) = comm(s, r, p) for s in ID, r in ID, p in HALFPRODUCT
definitions
WC1 = sum(p in PRODUCT, build(half(p)) . send(WC1, WC2, half(p)) ) . WC1
WC2 = sum(p in PRODUCT, recv(WC1, WC2, half(p)) . build(p) ) . WC2
Factory = encaps(H, WC1 || WC2)
end Factory
```