Purely Event-driven Programming in PSF and Go

Bas van den Heuvel
10343725 — vdheuvel.bas@gmail.com
Supervisors: Dr. ir. B. Diertens and Dr. A. Ponse

August 29, 2018

1 Introduction

The Purely Event-driven Programming language (PEP), described in [7], remained merely an abstract concept, until its semantics were formalised in [8]. The paper describes the different aspects of PEP, such as state machine instances and Machine Control, by formalising their processes using ACP (the Algebra of Communicating Processes; see [1, 2, 5]).

However, due to the large amount of concurrency in the design of PEP, it was unclear whether its intrinsic ACP designs were correct. Therefore, I set out to test the specifications in two settings. The first setting is PSF (the Process Specification Formalism; see [3, 10]), in which it is possible to directly express the ACP specifications. Its toolkit (see [4]) supports concurrent execution, but not parallelism. Therefore, the second setting is Go (Google's programming language; see [11]). Go has fairly easy-to-program parallelism and synchronous channels for communication, making it a suitable programming language to implement the PEP specification.

The experiment involved implementing the example problem in [8, Appendix D] in PSF and Go. This document explains what parts of the original specification had to be changed, due to errors that were discovered during this implementation phase. Then it explains how certain assumptions about specification in PSF were wrong, and how these problems were solved. Next, it is described how the specification was transformed to an implementation in Go. Finally, there are some conclusions, mainly on the usefulness of this type of work for programming language design.

2 Discovered design errors and their corrections

Especially the implementation in Go uncovered some errors in the original specification of PEP. Because execution of the intrinsic PSF specification was quite time-consuming due to amount of work the term-rewriter had at each step, some errors were not found in this setting. It might be possible to put the PSF toolkit to better use by creating abstract specifications of different parts of the design of PEP and individually testing them. The fast, parallel execution of the program in Go had some crashes due to deadlock and some unexpected behaviour. These design errors are described below, followed by their solutions.

2.1 Deadlock

While implementing the designs in Go, a deadlock was discovered in the processes that communicate events between Machine Control and machine instances by merely running a simple test program. This test program was then implemented in the PSF version of PEP, and it was verified that the specification indeed contained a deadlock.

The deadlock occurs in M_{id}^{out}, M_{id}^{in}, and MC^{in}, all in the same manner. This example is on MC^{in}, but the situation can be replicated to any communicator process. The communicator receives an event from a machine instance, while in the meantime the scheduler decides that all instances are halting and that it's time to halt the program. It sends a halt request to Machine Control's queue, which halts before the communicator can enqueue its event. Now the communicator is stuck with the event, not being able to halt itself. Deadlock occurs.

The problem has been resolved by giving each communicator process the option to halt even after receiving an event. This has been dubbed *fail-safeness*.

2.2 Special id 0

The Go implementation brought to light another design error. The first machine, which is without context, is started with $ctxid = 0$. Every machine instance is initiated with a default reaction to halt once its context halts. Once a machine has sent a halt request to Machine Control, Machine Control distributes a destination-less halt event to all running instances. This has the consequence that once a child instance halts, the initial, context-less, instance will receive the halt event. It will consult its reaction table and find a reaction to a halt event from any instance, since reactions with sender id 0 are defined to be reactions to events of the given type from any sender. Therefore, acting according to design, the initial instance will halt as well, causing its children to halt, causing their children to halt, and so on. Obviously, this behaviour is undesired.

The problem has been resolved by checking the instance's id before starting its reaction table with the default halt reaction. If the id is 0, the reaction table starts empty, otherwise it will include the reaction to its context.

3 Wrong assumptions about PSF and their corrections

During the initial specification phase, some assumptions were made about what was possible with the PSF toolkit that turned out to be wrong: sums over infinite series, sums over internal data structures, and the usage of -1 as a special id. This section describes those assumptions, and explains either how the implementation in PSF has been changed to solve these problems or how the original specification has been changed to respect the possibilities of PSF.

3.1 Sums over infinite series

The specification contained a lot of sums over infinite series, e.g. the natural numbers. Although it is possible to specify this using the PSF toolkit, it is not possible to compile/run such a specification. This is because the toolkit's sum tool needs to rewrite the sums over lists to sums of concrete terms. An example:

$$\sum_{i \in \{1,2,3\}} i \Longrightarrow 1 + 2 + 3$$

If the process expansion tool would try to "unfold" sums over infinite series, it would obviously be doing so forever. That's why the PSF toolkit refuses to do this. Therefore, the specification in PSF is limited to sums over finite series. For PEP in PSF, this has the concrete consequence that only a limited amount of instances can be started. The PSF implementation does use the natural number series $\mathbb{N}$ for ids, but it is defined with a maximal id m:

$$\mathbb{N} = \{i \mid 0 \leq i \leq m\}$$

3.2 Sums over internal data structures

Machine Control's scheduler process was specified with sums over data structures that were parameters of the process. However, those data structures are merely processes or functionality defined *using* PSF. Only enumerations defined as part of the specification can be summed over. These sets cannot be altered by processes, only by the programmer. Therefore, these sums over internal data structures are not supported. My solution was to implement these sums using tail recursion. Given some non-empty list ℓ and process P using some element $e \in \ell$, the sum $\sum_{e \in \ell} P$ can be implemented using tail recursion as such:

$$\begin{aligned} Sum(\ell) =& SumRecurse(\text{first}(\ell), \text{tail}(\ell)) \\ SumRecurse(e, \ell) =& [\ell \neq \epsilon ?] :\rightarrow SumRecurse(\text{first}(\ell), \text{tail}(\ell)) \\ & + P \end{aligned}$$

3.3 Special id -1

In the original specification, the id -1 was reserved for special cases, such as the context of the initial machine instance, the sender id of a sender-less event reaction, and the destination id of an event without destination. However, in PSF it is not trivial to add -1 to the set of natural numbers. Therefore, the choice was made to change the specification to reserve 0 as the special id. This turned out to be a simple change: the initial instance was already started with id 1, so it was only necessary to change all the -1s to 0s.

4 Go and the deviant implementations of PEP

It is not possible to have a perfect one-to-one correspondence between an ACP specification and an implementation in Go. This section describes the basics of concurrent Go programming, and how this was used to implement the PEP program. There are also a few comments on reducing overhead and increasing code simplicity.

4.1 PEP in Go

An implementation in Go is different from a specification in ACP and PSF. In ACP, a merge of concurrent processes can be rewritten to a non-deterministic choice of actions from those processes until both processes either are terminated or deadlocked. In Go we have the opportunity to run the merged components simultaneously, using *Goroutines*. Goroutines are a lightweight version of threads, that, provided the proper hardware, can run on separate processors. Thus, instead of ACP's consecutive concurrency, in Go we have *actual parallelism*.

The next step is to provide an implementation of ACP's communicating actions. For Goroutines to communicate, Go uses *channels*. Channels are bidirectional and can be passed to Goroutines as arguments. These channels are typed, meaning that they can only transfer values of a certain type. It is obvious that each channel has its own performance overhead. However, this project's purpose is not to give an efficient implementation of PEP in Go, but to test the specifications in ACP. Therefore, in order to stick to the ACP specifications as closely as possible, each communicating action was implemented with its own channel. Actions without arguments were implemented with boolean channels, single argument actions use the corresponding Go type, and multiple argument actions use a struct collecting the necessary argument types for their respective channels.

The process starting a Goroutine doesn't wait for the routine to finish once it's finished itself; Goroutines are detached. Now consider the following ACP example, where P, Q and R are arbitrary processes:

$$(P||Q) \cdot R$$

In this example, although the order doesn't matter, both P and Q have to be finished before R can start. Not waiting for merged processes to finish could lead to unexpected behaviour.

Thus, in order to be able to rely on the behaviour of our specification in ACP, it was necessary to ensure all merged Goroutines are finished before moving on. This was done using *Wait Groups*. A Wait Group is a thread-safe counter. When a Goroutine is started, the counter is incremented, and a reference to it is sent to the Goroutine as an argument. Once the Goroutine finishes, it decrements the counter. After starting all merged Goroutines, the calling process waits for the Wait Group to become zero, before it continues. In other words, a process *synchronizes* with its child processes at Wait Group wait points.

Although Go introduces actual parallelism to PEP, the drawback is that there is a lot of overhead. Each machine instance consists of five Goroutines (the tables are not implemented as separate processes; see Section 4.2). Machine Control consists of three Goroutines. Every communicating action needs its own channel, so in between all of these processes are many channels. A queue has for example four channels: one to enqueue, one to dequeue, one to announce the queue is empty, and one to halt. On top of all of that, every Goroutine needs a reference to the Wait Group of its calling process.

4.2 Non-concurrent lookup tables

In Go, lookup tables aren't used with pure functions. Instead, you directly change the lookup tables. Although it is still *possible* to implement the tables as concurrent processes, it is not *necessary*. Even though the PEP implementation in Go would be one step further away from a perfect one-to-one correspondence to the ACP specification, the choice was made to utilise the impure way the tables work. The main reasons for this are simplicity of the Go code and limitation of concurrency overhead, as described above.

4.3 Equivalent event handlers

During implementation, it became clear that the three event handlers, MC^{in}, M_{id}^{out}, and M_{id}^{in} are equivalent. Therefore, to preserve simplicity of the Go code, they are implemented by one single function. The distinction is in what channels are passed as arguments. The incoming event handler M_{id}^{in} uses the `distrib` channel for input of events, and the incoming event queue's `enq` channel for output, where the outgoing event handler M_{id}^{out} uses the outgoing event queue's `deq` for input and the `out` channel for output.

5 Conclusion

Both implementations contributed greatly to verifying and improving the programming language's formal specification. Some major errors were found and corrected, and some details of the specification brought limitations in one setting, while they brought improvements in the other. The work reported in this document resulted in an improved version of [8]. Both implementations are available at the project's Github repository (see [6]).

PSF was mainly useful for getting all the notations right and for finding limits of formal specifications in ACP. The recursive sums described in Section 3.2 are an example where the specification could be simpler, but had to be changed in order to accommodate for PSF's limitations. PSF does, however, force you to consider details one might have missed beforehand, and to rethink certain design choices. This contributes to a better substantiation of the choices that were made in the design of the programming language.

Go was especially useful, because of the concurrent nature of PEP. It is therefore advised to utilise the Go language to simulate the concurrency of a programming language in design. Although a perfect correspondence was not reached, it was possible for the implementation to closely follow the specification. Even though there is a lot of overhead due to the amount of Goroutines and channels necessary, because of its parallel execution, the Go implementation is more suitable to test PEP programs than PSF. It would be interesting to test this claim more thoroughly by diagnosing the Go implementation's performance in different situations.

It would be very interesting to consider Go for implementing a compiler for PEP. After a thorough round of optimisations, mainly focussing on reducing the overhead of channels, it could be used as an intermediate step in the compilation phase. Once an original syntax has been parsed, it can be put into a template Go implementation, and then compiled to an executable. This would spare the implementer from having to implement parallelism. Why would one reinvent the wheel? Instead, the implementer could focus on providing the language with debug tools, such as Message Sequence Charts (see [9]) to gain insight into the flow of events.

References

[1] J.A. Bergstra and J.W. Klop. "Process Algebra for Synchronous Communication". In: *Information and Control* 60.1-3 (1984), pp. 109–137.

[2] J.A. Bergstra, A. Ponse, and S.A. Smolka, eds. *Handbook of Process Algebra*. Amsterdam: Elsevier, 2001.

[3] B. Diertens. *What is PSF?* Nov. 23, 2005. URL: `https://staff.science.uva.nl/b.diertens/psf/whatispsf.html` (visited on 02/02/2018).

[4] Bob Diertens. "Software Engineering with Process Algebra". PhD thesis. University of Amsterdam, 2009.

[5] Wan Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science, An EATCS Series. Available at `http://www.cs.vu.nl/~wanf/BOOKS/procalg.pdf` (draft 2nd edition). Berlin: Springer, 2000.

[6] B. van den Heuvel. *Purely Event-Driven Programming GitHub repository*. URL: `https://github.com/klaplong/pepsim` (visited on 06/03/2018).

[7] B. van den Heuvel. "Purely event-driven programming: A programming language design". Bachelor Thesis. University of Amsterdam, June 2016. URL: `https://esc.fnwi.uva.nl/thesis/centraal/files/f522241892.pdf`.

[8] B. van den Heuvel. *The process of purely event-driven programs*. Feb. 2018. arXiv: `1803.11229 [cs-pl]`.

[9] S. Mauw and M.A. Reniers. "An Algebraic Semantics of Basic Message Sequence Charts". In: *The Computer Journal* 37.4 (1994), pp. 269–277. URL: `http://dx.doi.org/10.1093/comjnl/37.4.269`.

[10] S. Mauw and G.J. Veltink, eds. *Algebraic Specification of Communication Protocols*. Vol. 36. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.

[11] *The Go Programming Language*. URL: `https://golang.org` (visited on 06/03/2018).