Concurrency issues in the iTasks3 framework.

Dennis Brentjes

June 25, 2012

Abstract

In this paper an solution for multi-threaded handling of iTask web-requests is being proposed. This solution preserves much of the original iTask system and aimed to preserve the uniqueness constraint present on the "World" object. There are however several issues with this solution that need to be solved before attempting implementation. This paper should therefore be considered as a proof of concept than a detailed guide about how to proceed with a multi-threaded iTask web-server.

Contents

1 Introduction

Web-based application have always lend them self well for multi-threaded processing. Pure functional languages like clean also lend them self well for multi-threading the functions. However these two concepts have not been merged and this what I try to do in this paper.

We first discuss what the points of interest are regarding the multi-threading of the framework. Using UPPAAL models to discover critical sections and check for system invariants.

Then using the results from the UPPAAL models to build a solution. Keeping as much of the properties that Clean has and keep it as easy to use as possible as The skill required to define workflows should be as low as possible.

2 Literature

2.1 Clean

The programming language Clean is a pure functional language currently being developed in Nijmegen, the Netherlands. Apart from being pure it can also switch between evaluating strict or lazy, support higher order functions, and implements generic and dynamic typing.

In this paper we will heavily rely on dynamic and generic typing to generate the needed boilerplate for the user to easily interact with the new features. The generics are mainly used to accept every type to be put into a dynamic. And dynamics are used to serialize functions and data to file, to be able to communicate using while the file system. How the new features use the dynamics and generics is described later in the paper, the inner workings however are not. They are described extensively in this paper by P. Achten et all [1].

2.2 ITask

"ITask is a scientific prototype of a Work-flow Management System (WFMS). It is also a real-world application that deploys and coordinates contemporary web technology" [4] ITask is a programming library made in and for the pure functional programming language Clean.

The general idea of the iTask WFMS is to provide a "as simple as it gets" powerful work-flow definition language. Most contemporary WFM systems use special purpose specification languages which are less expressive than needed in some cases. The iTask system tries to solve this by using a programming language as a base for their WFMS as this grants them some features for free. features like general recursion, powerful abstract types, and function abstraction. They use a functional programming language because this gives them higher order functions and a more expressive language for the set-domain in which the iTask system operates. It also gives them fast compiler and interpreter technology.

The framework in its original state does not provide a multi-threaded web-server to serve contents to its users. When I am finished it will, so this system can be deployed in an environment with a large number of users. The lack of multi-threaded serving of content usually is not a problem if the tasks are performed linearly, but as the user has the ability to grant either tasks to other users, leave the task to be completed by somebody else or to be finished at a later time this causes some issues. The iTask system uses a disk cache to store user sessions before they are completed and serves them again when needed. It does not however supply the programmer with synchronisation primitives to be able to access this cache concurrently. This is why this paper will try to identify the critical sections in the iTask framework and find a suitable solution to these concurrency issues.

2.3 Uppaal

Uppaal is a toolbox for modelling, simulation and verification of state machines, therefore anything that can be modelled with state machines.[5] The toolbox consists of 3 parts. One of these is a modelling language in which to model your system, In either a graphical or scripting fashion. The second part is a simulator in which you can quickly inspect your models behaviour. Finally part three is the model verifier in which you can check for invariants in the entire model. The verifier then performs an exhaustive search on the statemachine or model that you created. It then reports its findings back to you. This allows you to make a model of a system that heavily depends on timing and simulate during the modelling process to look for model flaws before the model checker is used. This last observation is quite vital to using the Uppaal tool, as checking your model with the verifier is very computational expensive. You on the other hand can exploit what you already know about the model to find flaws faster and then later check if you missed a situation in which an invariant no longer holds.

Uppaal offers both a WYSIWYG (what you see is what you get) editor and a textual interface to input your models. The internal compiler transforms any graphical part of the model to the textual which can then be parsed and verified by the model checker. The Uppaal description language is also very flexible as it could be adapted to support models which can be transformed to timed automata. If parts of the uppaal system need te be adapted, they can be. This enables the modeller to input a model that isn't a pure timed-automata and have the system do all the necessary conversions for the model checker as described by [5].

The model checker is designed to check for system invariants and reachability properties. Checking if 2 instances of an automata can be in the same state or if some variable in one of these automate's state are the same as in the other or any other properties expressible in logic. In this paper we will express the invariants of the iTask system into logic and variables used in the model. The simulator will try to find violations of the constraints after which these violations should be mended using standard techniques for concurrency issues.

Another nice feature of Uppaal is that the verifier can generate a trace to a state which violates your conditions in the verifier. This trace can help you identify and mend the problem in the system.

2.4 Concurrent programming

A lot of research has already been done on concurrent programming. Consequently a variety of interprocess communication and synchronisation methods have been developed in the past. The challenge in concurrency is no longer to correctly implement a solution, although this of course is a requirement, but focuses more on having the right solution for the right problem. Choosing one solution over another can have dramatic effects on your performance and or maintainability of your code base [2].

For instance a solution could be to distinguish between types deciding to allow write operations on atomic types, types no bigger than the size of a register or word size of the system you are working on. The gain in performance will be quite noticeable as you do not need to ask for the state of the variables. However this solution requires you to have a deep understanding of the types the read/write method is invoked with, and make a distinction which varies from system to system. This leads to unmaintainable code as this requires a rewrite or at least code addition every time you want to deploy your software on a new system.

A more universal solution to implement synchronisation, like using a monitor or a semaphore is much slower. A semaphore knows nothing about the data it guards. It just knows it can allow one caller at a time to manipulate data and to tell any subsequent caller to wait until the first caller says he has finished manipulating the data. As abstract as it is it it does not allow for much optimisation. All callers have to ask the semaphore if it's all right to approach the data even if has been available for a long time or if it is an atomic data type. Depending on how long it takes for the semaphore to respond this could take up valuable computation time. We should keep this in mind when choosing our solutions and find a balance between abstraction and performance.

3 ITask models in Uppaal

To show that there is a need for synchronisation in the iTask framework this paper will attempt to create a simple model in Uppaal. This model will model will consists of a small part of the framework and show iTask programs requires synchronisation when concurrency is allowed.

3.1 Adapting Uppaal

The iTasks 3 WFMS has a document that describes the semantics in the language Clean*. This programming language is used as it is more readable for the Haskell community as Haskell is the standard in the functional language world. This document comes in handy when making an implementation of the iTasks system. Part of this research will be to model a small part of the Itask framework in Uppaal. By doing this I hope to prove that certain actions should be made atomic before multi-threading can be applied to iTask in its current form.

The tool I'm using for this is called Uppaal, this tool has a big advantage as it can automatically check for any condition you want it to check over several automata at once. This allows you to specify a certain property, make a model, and then verify if the condition is satisfied in the given model. If this is not the case you can ask the verifier to create a trace that leads to this violation.

What properties are we looking for? Well the properties are hidden inside the semantics document. Two examples are "two tasks can never have the same Task-id" or "two tasks can never have the same time stamp". Before we can actually check if these properties would be violated by our new environment we need to overcome some minor technical issues.

Uppaal has a basic definition language which is a subset of the C programming language with added domain specific types such as clocks, states etc. This being said it does not have interface to define algebraic data types, so this needs to be worked around.

3.1.1 Enumerations

Even though the highlighted keyword enum in the declaration language suggests it, Uppaal does not contain enumerations. Also the most basic algebraic type definition in Clean only contains possible labels. This is equivalent to enumeration types found in other languages. The language does contain "const" and bounded integers so it is easy to add your own enumerations and they to make them readable by using "typedef" to hide the underlying bounded integer type.

To give an example related to our domain, here is how I modelled the "Stability" type. In Clean* it is defined as follows.

```
:: Stability    =        Unstable | Stable
```

Listing 1: algebraic type definition

In the model definition language of Uppaal this looks as such, listing 2.

```
typedef int[0,1] Stability;
const Stability stability_UNSTABLE = 0;
const Stability stability_STABLE = 1;
```

Listing 2: Uppaal enum emulation

3.1.2 Records

Thankfully records are convertible to C-type structures. So records like State and World are represented as structures with data members. And Uppaal supports the use of C-type structures so we can port these types straightforwardly. Furthermore in the next few models we only model the structures members we need, to clarify what variables matter.

3.1.3 Function calls

Uppaal does allow for function definitions in the declaration language, but only if the function is to be considered an atomic operation. In this paper we need to be able to split the functions we model up into multiple state machines to keep things manageable. This requires us to do some extra work regarding emulating function calls and returns. Currently there is no way in Uppaal to directly pass values from one state machine to another. For this purpose I have added more variables to the global state to emulate these features.

Firstly we need to represent the function arguments and get these into our model. We do this by making a structure containing all the argument members. Then we declare an array of size two of the made type above to store the arguments with which we call the function. When the function gets called this state machine, representing the called function, will access this array to get the arguments on which to operate. But we haven't covered the calling mechanism yet. This is handled by the only synchronisation method built in Uppaal.

This synchronisation method uses a type of variable called a channel and on this channel there are 2 operations. The "?" and "!" operation. The "!" sends a signal over the channel and all state machines that have the corresponding "?" operation in their next transition are allowed to make that specific transition. Alternatively the "?" operation blocks until a corresponding "!" operation is performed.

These synchronisation operations are not the only operation the state machines can perform on a state transition. So on the same edge you would do the array manipulations we talked about earlier. To be more precise; on the edge containing the "?" synchronisation operation the machine accesses the "stack" array to get his function arguments. Here is an example of the system in action.

The function "evaluate Task" in the iTasks framework takes a "Task" and a "State" as argument. So there is a structure containing just that and just for convenience we name the structure "arg_evaluateTask" in listing 3.

```
//***** Argument structs *****
typedef struct
{
    Task_t ta;
    World_t world;
} arg_evaluateTask;
```

Listing 3: Example function argument struct

Then we create the array for the function arguments. In this model there are two threads as this is enough to verify if our properties hold. The two threads take up most of the available resources during the validation phase so more threads would simply not be possible at this time. This also means that there have to be two places for us to store each thread's function arguments. As function arguments are pushed to the stack prior to calling the function we call these array variables stacks.

```
//***** Function stacks *****
const int TNR = 2;
arg_evaluateTask     stack_evaluateTask[TNR];
```

Listing 4: The function "stacks"

Then we have the channel type variable to facilitate the synchronisation routines of Uppaal.

```
//***** Channels / function calls  *****
chan call_evaluateTask[TNR];
```

Listing 5: Function calling in Uppaal

Following are two pictures (figure 1 and figure 2) of the state machines utilizing this function call method.

As you can see this test state machine "calls" the other state machine shown below

In these examples you can also see that we use the same mechanic to transfer return values from one state machine to another. We will call these places a return point from now on.

Figure 1: A state machine calling another state machine as if it were a function.

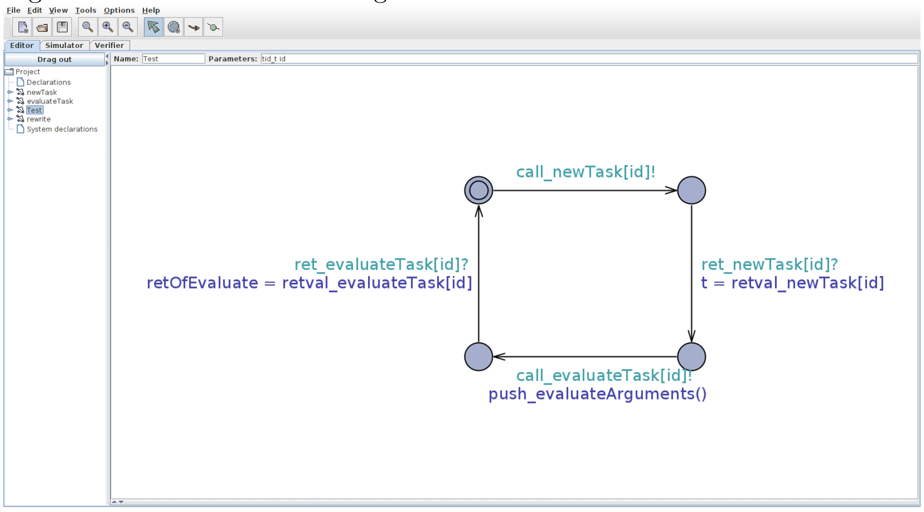
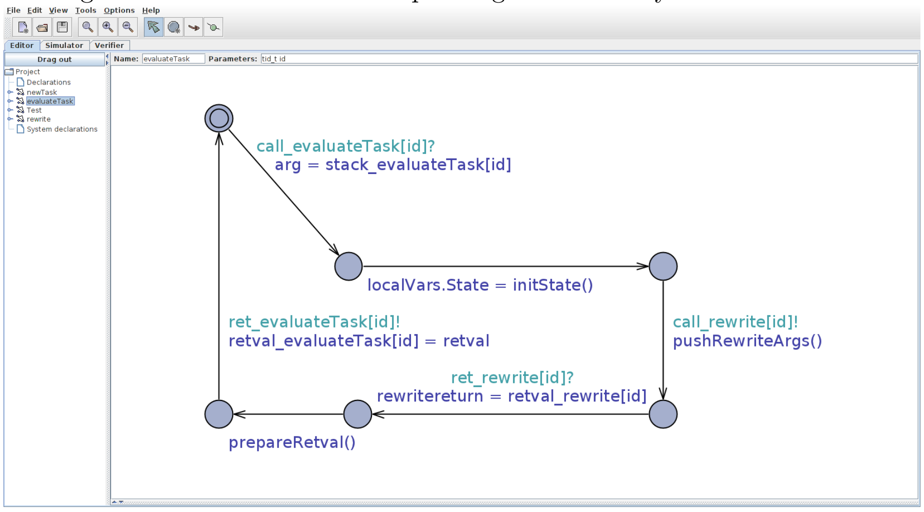


Figure 2: A state machine Responding to the call by the caller above.



3.1.4 Limiting the search space

Since the models we built are big we need a way to keep the search space as small as possible, but we should do this without limiting the model to not enter states it could enter under normal circumstances. Fortunately Uppaal allows for bounded integers. We use these bounded types to model things like timestamps which would eventually also run out if we used normal integer types as these have an implicit maximum value dependent on the system architecture. For example if they get bigger then the register size of the current host computer, or in this case Uppaals max int (16 bits signed integer). But this simplifies and speeds up our model checking.

This however has the troubling side-effect that it introduces deadlock as the bounded variable cannot grow larger than its upper bound. These issues would also be present in the real system but would not emerge as quickly as the integer sizes would be much larger. That being said we still want to check if our model deadlocks on other parts as a result of our concurrent programming. So we allow deadlocks only if these bounded variables in the global state have met their upper bound. This allows us to generate traces of deadlocks that occurred because of flaws in the model instead of inherent properties of computer memory, or lack of it.

3.1.5 Uppaal's use

Although I was originally planning to validate the whole system in this manner, this was not possible. The search space grows exponentially, so every state you add increases the computation capacity needed to validate the system. This resulted in using this as a tool to prove parts of the system and use the "Referential transparency" property of Clean. In short the referential transparency means: "a function called with the same arguments will always return the same value, no matter the state of the program".

So if I prove a certain property holds within a certain function, no matter what the state of execution of another function would be, it would not alter the way this function interacts with his arguments and thus yielding a correct result all the time.

After modeling a few functions however I discovered that the primary issues of the current system is not being able to share the world in a safe manner. When we can do this we can multi-thread the whole system. The next sections will try to give a solution to this problem.

4 A general extension for threading

In this section we will discuss what solutions are possible to the problems found by the Uppaal models and how to implement these solutions. This will give us a generic toolbox to use within the iTask system. But firstly we will discuss it outside this setting to focus on the solutions that are being introduced instead of the framework itself.

4.1 Points to change

We will discuss the weak points in the system as identified by making Uppaal models using the methods clarified in section 3.1. Following is a list of things to tackle when making the system multi-threaded:

- State object "World". This "World" is essentially a variable all Tasks can access. This is exactly the issue I identified in section **??**.
- Uniqueness constraint of the "World" variable. The world can be copied but it will be a separate instance as of the uniqueness constraint.
- Shared data of the iTask framework (RWshared). Also these can be spawned and shared within the world and we should be able to access them from within tasks.
- Inter-thread communication. The threads need a way to communicate with each other now facilitated by the world but this will no longer be accessible in the new system as it was in the old system.

4.1.1 State object world

Clean uses a state object world to interact with the the outside "World". This state is extended in the iTask framework and is called "world". This is an abstraction for everything the underlying operating system supplies as a service to it's programs. Think of input- and output operations as a prime example. To be able to create a socket we need this world object. This means all access to this world should be regulated by some kind of mutex lock or only a single thread should have access to this object.

4.1.2 Uniqueness constraint of the "world" object

In Clean their is a concept of unique objects. These objects may be duplicated but this will create a second distinct object. The thing I need in this paper are multiple references to the same object. For example; when opening a file in Clean you get a object of type "Unique File" (transcribed as "*File"). If you could duplicated the reference to a file currently in state A. 2 functions could get a File in state A while function 1 might have written something in the file. This first function could close the file while the second files still assumes it has an open file handle. These inconsistencies are prevented by this uniqueness typing, but is exactly what I need for my solution.

The world object is an example of such an uniqueness typed object. But as stated in section 4.1.1 we either need to regulate access to the State by mutual exclusion or only give a single thread access to the world object.

The most common solution in imperative programming is by using a "mutex" lock. These mutexes will only grant the first thread that accesses it and will sleep the other thread until the first thread releases it's lock on the mutex. But in the context of pure functional languages this means we have to represent them without using global state. As pure functional languages do not allow global state all information must be be passed to the function. But the callers in this case are the threads, and if they would know what to pass to this mutex function, which in turn would use this information to determine whether to grant access or block the incoming thread, we wouldn't need to ask the if we are allowed to access the data. As the threads would have the information needed to determine if the data is safe to access. As we know this is not the case, therefore implementing a mutex in the classic way is impossible in Clean. This also rules out other standard solutions like Spin-locks and wait-notify synchronisation primitives.

We will come back to the solution in our abstract and real world solution sections.

4.1.3 Shared data of the iTasks framework

This problem boils down to the unique world problem. Both threads need access to the world object to be able to either read or write data from or to the RWShared data types. So if we solve the issue of the unique world we solve this issue as well. There is however the possibility for have RWShared object not located in the world but more local, but as it is possible to also store them in the world my solution has to be universal enough to accommodate them.

4.1.4 Inter-thread communication

The pure functional nature of Clean makes coupling separate parts together difficult. You can not have a common middle ground as it has to be contained in one of the separate units, in this case threads. A Mailbox system in Clean is impossible to make because of the lack of locking mechanisms. So one never knows if it is safe to push a message to the receiver, and there is nowhere a message could be send to that is available to both parties.

Yet again the solution is there, but just not in the Pure functional world Clean lives in.

4.2 Abstract solutions

All the problems mentioned above can be tackled by a single solution. We make an abstract State "mutator". The State in this case being the world, but I'll refer to State as this solution is generic and does not only work on "Worlds". This will be a separate thread and is the only thread that has access to the world object. This thread listens to some abstract queue that provides it with functions to transform the world and sends the return-value of these functions back to the sender via another abstract queue.

These queues are not unique and will not be pure in the sense of pure functional as they depend on the state of the files used which could be altered at any entity that misuses the system. These dirty object however are the only dirty object this solution will introduce. And by regulating and keeping the interface to these objects to a minimum one could use other methods to prove nice properties that are beyond the scope of this article.

In this chapter semantic functions will be framed and sample code will be "inline" with the document. This is common practice in this particular academic world and I will conform to these standards.

```
::Pipe           = Pipe Dynamic

WriteToPipe      :: Pipe a -> Pipe | TC a
ReadFromPipe     :: Pipe -> (Dynamic, Pipe)
DynStApply       :: Dynamic *State -> (Dynamic, *State)
Match            :: Dynamic -> Maybe a | TC a

::ReturnVal      = Ret
::Void           = Void
::Func           = Func
```

Listing 6: The Pipe typedefinition.

This algebraic data-type is representing the abstract queues we introduce. This type does not have any real object backing it up right now but lets assume for now such a type exists and we'll discuss concrete implementations in section 4.3

Next up writing and reading from a pipe and restricting its use.

```
WriteToPipe :: Pipe a -> Pipe | TC a
WriteToPipe p v = sWriteDynamic p (dynamic v :: a^)

Match :: Dynamic -> Maybe a | TC a
Match (x :: a^) = Just x
Match other     = Nothing

ReadFromPipe :: Pipe -> (Dynamic, Pipe)
ReadFromPipe p  = sReadDynamic p

DynStApply :: Dynamic *State -> (Dynamic, *State)
DynStApply (f :: *State -> b) st = (dynamic (f st) :: b, st)
DynStApply _ _  = abort "DynStApply: arguments of wrong type."
```

Listing 7: The pipe API, containing all neccesary functions to operate with the new threading model.

In this example I have chosen for an generic interface for both reading and writing pipes. Maybe it would be better if the interface was more limited, but for now it is easier to talk about a small powerful set of tools and focus on solving the issues.

The functions sWriteDynamic and sReadDynamic are not specified further in this document. They will be modified versions of the read and write dynamic discussed in P. Achten et all [1]. These variant would be made to work on any type underlying the actual Pipe type. The case of shared files will be discussed in section 4.3

When writing something to a Pipe it can be either be of type tuple with Ret or Func as first member. (Ret, Void) is for function normally returning a State but this is handled by the masterthread and thus cannot be returned. So it just returns Void. Any other Ret tuple wil contain any arguments returned beside State. So a function normally returning (Int, *State) will send a (Ret, Int) tuple over the pipe. Last of the (Func, f) tuple will be send if a function requests something of the world.

In the next block some convenience functions are denoted that you would want to implement when holding a shared data-source in the proposed manner So I will add them to the semantics description.

```
loopForInput :: *State [(Pipe, Pipe)] -> *State
loopForInput st x
#(st, x) = iterate st x []
| x == []          = st
| otherwise        = loopForInput st x

iterate :: *State [(Pipe, Pipe)] [(Pipe, Pipe)]
                                        -> (*State,[(Pipe, Pipe)])
iterate st [] ops        = (st, ops)
iterate st [(pin, pout) : ps] ops
#(df, pin)                         = ReadFromPipe pin
#(dv, st)                          = DynStApply df st
#mv                                = Match dv
|mv == Nothing                     = iterate st ps [(pin, pout) : ps]
|Ret == fst(fromJust mv)
         | Void == snd (fromJust mv) = iterate st ps ops
         | otherwise
                 //Do the needed computations on the return value
                 = iterate st ps ops
|Func == fst(fromJust mv)
        #(Just(Func, v))           = mv
        #pout               = WriteToPipe pout (Ret, v)
= iterate st [(pin, pout) : ps] ops
```

Listing 8: Convenience functions for handling input.

I would like to note that we use dynamics to store and somehow write the data to and read from the pipes. This allows us to send functions and this is exactly what we want to do, as we made a separate thread handling all the State transformations. So the task of this thread is to check if there is information on one of his incoming pipes and match that with a function that takes a $*State$ and returns a generic type a. Then he transforms the State and writes the generic outcome of type a back on the corresponding Pipe. Lets have a look in an abstract way.

```
work :: *State -> (Int, *State)
work st
# value           = st.value
# st              = {st & value = value + 1}
= (value, st)

futureInt :: Pipe -> (Int, Pipe)
futureInt p
#(val, p)         = ReadFromPipe p
#val              = Match val
|val == Nothing   = futureInt p
#(Just (Ret v))   = val
|otherwise        = (v, p)
```

```
threadedFunction :: Pipe Pipe -> (Pipe, Pipe)
threadedFunction pout pin
#pout           = WriteToPipe pout (Func, work)
#(val, pin)     = futureInt pin
#pout           = WriteToPipe pout (Ret, val)
=(pin, pout)

startProgram :: *State -> *State
startProgram st
#pin1    = Pipe (dynamic 0)
#pout1   = Pipe (dynamic 0)
#pin2    = Pipe (dynamic 0)
#pout2   = Pipe (dynamic 0)
#(pin1, pout1)  =      startThread (threadedFunction pin1 pout1)
#(pin2, pout2)  =      startThread (threadedFunction pin2 pout2)
= loopForInput st [(pin1, pout1), (pin2, pout2)]
```

Listing 9: Example usage.

In which "st" contains an unique "Filesystem" and startThread an abstract function not further specified that runs the given function with parameters in a separate thread.

By showing this piece of example code I hope I have shown how the tactic works. But it might not be self-explanatory. The program starts by setting up so called pipes and giving them to threaded functions so they can get on their way. These threaded functions can go on their way and start executing the real program. As soon as they hit a function call they wrap the function and partially apply it until it only needs a *State variable. At this point the threads sends it over the outgoing pipe it received earlier from the thread now holding the *State variable. This program reads the wrapped function from the Pipe applies it to the State and returns whatever value it is back on his outgoing pipe corresponding with the incoming pipe on which he received his partially applied function. From the moment the threaded function sent his request it loops until he receives the answer to his request regarding the *State variable. This might be a simple True/False value or even Void, or an action event from a socket. The threads knows what it expects and can check this value to be of that type. If so it accepts the value as being correct and moves on to the next function call requiring the *State.

4.3 Real world solutions

Now that we discussed what we need for this approach we will have a look at how we could implement this using the Clean language constructs. As we used some non-existing language constructs to define the solution, but this does not mean it can not be implemented in real Clean. We start by redefining the Pipe. Pipe did not have any physical Clean data structure backing it in the previous definition of the system, therefore it is impossible to use this type to store any data as we assumed it could in the previous definition. So let redefine it so we can work with it properly.

The Clean language usually has Files that are uniquely typed. Any variable can be typed "unique" and in this case the uniqueness makes the underlying data consistent as only one read/write operation can be carried out at any given time. But this is exactly what we do not want or at least what we can not have. As we cannot have 2 references to the same unique file object between two threads. Fortunately the Clean language also provides so called shared files to which we can have multiple references. So our redefinition of the Pipe will be as follows.

```
:: Pipe :== File
```

Listing 10: Our real world Pipe type definition

As you can see the file misses uniqueness ("*").

As defined in listing 7 we need a read- and write dynamic operation on these pipes.

The read operations have already been made as discussed in this paper by P. Achten et all [1]. These read operations are defined using a string and a file system. But as I am using shared files in this section, these functions should be redefined using a shared file. This will not be a problem as the underlying function calls opens a file anyway and does the operations on it to extract a dynamic. The Function only needs to be exposed or possibly be defined for shared files.

The write operation specified in 7 is a different story. There is no write operation on shared files at this moment in time. This is for very good reasons e.g. to prevent corruption of data that is bound to occur if used in normal file interactions, however file access in this new framework is very restricted in the sense that only 1 thread writes 1 file. And then waits until the other thread acknowledges that he received that data. So no simultaneous writes are possible. To stress that this should be impossible one could give the "WriteToPipe" some special semantics. I would suggest to make it truncate the file and then write it's contents. This way you guarantee that if there was some information that didn't get processed by the master-thread (The thread which holds the world and spawned the other worker threads) first gets deleted. Remember this is not possible as the thread writes and waits for it to be processed, but this is just be sure So the master-thread will be protected from receiving 2 request in the time he actually can only process 1 Read.

Another fail-safe could be some way to enforce that only one request is made at a time, without limiting the worker thread too much. But if the user abides by the rules this won't be a problem so I won't discuss possible solutions. This could be achieved to enlarge the responsibility of the "future" functions. These functions are implemented by the end-user him self in this example. However these functions can be generated for him by using generics but this makes the example more difficult to understand so I left it out for simplicity. We could implement a generic "future" function as follows

```
future :: Pipe -> (a, Pipe)
future pipe
#(val, pipe)             = ReadFromPipe pipe
# val                    = Match val
| val == Nothing         = future pipe
# (Just (Ret v))         = val
| otherwise              = (v, pipe)
```

Listing 11: A generic future

Another interesting proposed improvement upon the system that would make matters a bit more complicated is by merging the request to the master-thread and receiving an answer from this master-thread. One could easily expand the above "Future" function to include sending a function to the master thread but this would have to be done at the cost of readability. This also solves the issue of having two requests on one shared file. As the only way to request information is to enter the Future function and this Function blocks until it receives an answer. If the user somehow evades using it in this way every write should still truncate the file as to invalidate the current request and to prevent the system from going into a deadlock.

Another implementation detail that should be addressed is that all the operations which have something to do with the state will be very cpu intensive as of busy waiting. Time after time we are retrying to see if there is any information available. It is completely safe to do so but it is bad practise to waste cpu cycles, checking if you can proceed with the execution. Some notification mechanism could be developed for this purpose, but this is beyond the scope of this paper and would probably be hard to implement using the file system which I used for my proof of concept. Of course others are free to pick this up and implement a better system that supports notification of threads.

4.4 An application of the semantics in iTask

To effectively use the new functions in iTask we need to adapt the dispatching of requests. This function in embedded in the http-server of iTask should be adapted to fit our new needs.

```
http_startServer options handlers world
//Start the listener
# (listener,world)      = startListener (getPortOption options) world
//Enter the listen loop
#(suci, pin, world)             = sfopen "mainpin" 1 world
#(suco, pout, world)            = sfopen "mainpout" 1 world
| not suci && suco      = abort "Pipe creation failed"
#(pin, pout) = startThread (loop options handlers listener [] [] [] pin pout)
= loopForInput world [(pin, pout)]
```

Listing 12: Rewritten http startServer

This function will process all the world requests made by its descendants including all requests to make more pipes. This allows its descendants to spawn more threads to achieve true multi-threaded processing of requests. All the functions needing the world should partially apply the function until the only argument left to apply is the world and then write it to the pipes.

Further down in to the system every action request from the web server users will trigger a new thread to handle this request. In addition a second thread will be spawned to handle the next requests. This will result in a complete chain leading to the original thread holding access to the world to handle all the world access and then push it down the pipes into the system to where the request was made. The system as it now stands is not very manageable and needs to be improved upon if it were to be implemented. For now however it is a proof of concept and how it could be improved will be discussed in section 5.

```
loop :: [HTTPServerOption]
[(!(String -> Bool),!(HTTPRequest Pipe, Pipe -> (!HTTPResponse, Pipe, Pipe)))]
TCP_Listener [TCP_RChannel] [TCP_SChannel]
[(HTTPRequest,Bool,Bool,Bool)] Pipe Pipe -> Pipe Pipe
loop options handlers listener rchannels schannels requests pout pin
...
//Process a completed request
| method_done && headers_done && data_done
        ...
        // Create a response
        #pout                   = WriteToPipe pout (Func, (sfopen "loopPin1" 1))
        #(looppin, pin)         = future pin
        #pout                   = WriteToPipe pout (Func, (sfopen "respPin1" 1))
        #(response, pin)        = future pin
        #pout                   = WriteToPipe pout (Func, (sfopen "loopPout1" 1))
        #(looppout, pin)        = future pin
        #pout                   = WriteToPipe pout (Func, (sfopen "respPout1" 1))
        #(responsepout, pin)    = future pin
        #(looppin, looppout)    = startThread (loop ... looppin looppout)
        #(r..pin, r..pout) = startThread (http_makeResponse request handlers
                          (getStaticOption options) responsepin responsepout)
        #(pin, pout) =  redirector [(looppin, looppout),
                                    (responsepin, responsepout)] pin pout
        #pout     = WriteToPipe (Func, (debug "Generated_response:" options)) pout
        #(_, pin) = future pin
        #world    = WriteToPipe (Func, (debug response options)) pout
        #(_, pin) = future pin
        ...
...
where
        redirector x pin pout
        #(x, pin,pout) = iterate2 x [] (pin, pout)
        | x == []        = (pin,pout)
        = redirector x pin pout

        iterate2 [] x pin pout  = (x, (pin,pout))
        iterate2 [(lpin, lpout) : ps] ops pin pout
        #(dmf, lpin)             = ReadFromPipe lpin
        #(mf)                    = Match dmf
        | mf == Nothing          = iterate2 ps [(lpin, lpout) : ops] pin pout
        | Ret == fst (fromJust mf)
                | Void == snd (fromJust mf) = iterate2 ps ops pin pout
                | otherwise //do the needed computations
                        = iterate2 ps ops pin pout
        | Func == fst (fromJust mf)
                # (Just (Func, f))      = mf
                # pout          = WriteToPipe pout (Func, f)
                #(v, lpin)      = future pin
                #(lpout)                = WriteToPipe lpout (Ret, v)
        = iterate2 ps pin pout
```

Listing 13: Rewritten loop

Like a mentioned earlier this solution is far from perfect and serves as a proof of concept. People can rework and improve this to make it easier to use and more efficient to execute. In the next chapters we'll discuss other solution that have been proposed and the points to improve in this solution

5 Discussion

The solution proposed in Section 4.2 is by no means optimal. Using the file-system for interprocess synchronisation is inherently slow. This paper does provide a proof of concept that a system that synchronizes between pipes is possible in the pure functional language clean if you are prepared to step out of this pure world for small amounts of time. By consequently limiting how this Pipe could be accessed by either describing the usage or by enforcing it trough semantics, one could still use this system to prove certain properties as you limit the states the program can be in.

Another downside of this solution that it brings clutter to the program. It started as a nice clean idea in my head but resulted into a mess as I had to disambiguate between results and functions, I even had to introduce a void type. This needs to change if it would ever be implemented. This could be a topic for later research.

There is however another type of solution. This solution was proposed by László Domoszlai and Rinus Plasmeijer in [3]. This solution involves roll-backs. In a nutshell the method pushes as much work as possible to javascript and thus the client. It also gives it a world object as is and tries to keep the world consistent with the original world by performing a roll-back when it conflicts. Although elegant there is no guarantee that if steps that can lead to a inconsistent state will not be taken time and time again. Thus introducing a live-lock as any algorithm that terminates probabilistic. Of course the chances of this actually happening infinitely are very slim and one could fall-back to interleaved execution if one would loop too many times, but This didn't seem like an attractive solution to me.

Another reason I did not reuse any of his work is that we worked in the same time frame as each other but at different phases of the actual report. So at the time this paper came out I couldn't incorporate this any more in my work as I spent all of my time finding a solution and am under serious time constraints.

6 Conslusion

The solution proposed in this paper can be made to work, but as I went on with this topic I stumbled upon more elegant solutions either proposed by colleagues or by making discoveries inside the Clean language itself. Therefore the solution isn't as elegant as it could be. There is also room for improvement within this solution. For instance the call-stack increases by one with each request and this could turn ugly if left unfixed.

As a proof of concept this is fine but this solution should not be implemented without improving on the key areas both discussed here and in the discussion (section 5).

References

[1] Peter Achten, Artem Alimarine, and Rinus Plasmeijer. When generic functions use dynamic values. In *Proceedings of the 14th international conference on Implementation of functional languages*, IFL'02, pages 17–33, Berlin, Heidelberg, 2003. Springer-Verlag.

[2] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, March 1983.

[3] L'szl' Domoszlai and Rinus Plasmeijer. Client-side evaluation for itask3 using generalized client-embedded micro applications.

[4] Jan Martin Jansen, Rinus Plasmeijer, Pieter Koopman, and Peter Achten. Embedding a web-based workflow management system in a functional language. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, pages 7:1–7:8, New York, NY, USA, 2010. ACM.

[5] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1:134–152, 1997. 10.1007/s100090050010.