Thales Questionnaire (Translated into English)

So, as I mentioned before, let’s clarify why we are here. We have this paper, which was not rejected due to the content itself, but because of its structure. The reviewers are essentially asking for a more formal validation of the decisions we made.

We worked on this with [Expert 1], though he cannot be involved directly… well, reviewers could even be among us here. We’ve got the CTO, the PNR, the UR – serious profiles. It’s easier for us to validate this way, otherwise we’d have to involve ESA, which I wouldn’t even know how to reach.

The main challenge is that we’re dealing with a static scheduling system, and evaluating the insertion of a task in such a system, and its impact, is non-trivial. The work required for this hypothetical designer to understand what’s happening is far from easy.

What is the goal? The Digital Twin – for those unfamiliar – is not simply a digital replica or a simulator. It’s something more intelligent, able to model reality, harmonize its aspects, and suggest changes when something does not match expectations. It’s not only based on the model, but also on actual data from the Physical Twin. If a simulator is passive, the Digital Twin is proactive.

How does it work? You can use it from the outside to generate a solution, implement it in the multicore system, start using it, and after a while collect data on what was tested. You feed that back into the system, and it updates the model to reflect aligned behavior.

What we did was: start from problem definition, build the tool, create a graphical interface for configuration, and highlight potential edge cases – especially related to deadlines. We noted, for example, that the sum of deadlines exceeds the scan period of 129ms. Analyzing real data, we observed that in some configurations the system produces many context switches. But if tasks A and B are launched in a specific way, these disappear.

This could be framed as a performance constraint. So we built a small scientific model of the scheduler. In the end, we opted for a bilevel optimization method, with differential evolution at the outer layer and quadratic programming at the inner layer, to ensure priority ordering.

We're dealing with a more general class of mixed-integer nonlinear problems. Simplex wouldn't be enough.

Today, we're asking for feedback on the full loop, to assess whether this full loop makes sense in a specific context. In fact, one reviewer asked us to choose a specific context and validate our solution against that. Among several options, the one that seemed more manageable was this: not rescheduling based on satellite physical conditions (like being in shadow), because that has been extensively studied. But rather a context where, say, the satellite is launched and then leased out on demand to whoever needs it. So rescheduling happens because a client (say, Leonardo) buys it for two hours, and during that time a task order must be uploaded.

Doing this manually is possible, but automation would make it smoother. Initially, our goal was not to adapt the scheduler at runtime due to dynamic conditions, but rather to enable analysis and configuration when a new demanding task is added.

We are now three profiles in the loop: one with limited knowledge, one with intermediate expertise, and the domain expert – the one who might say "I'll just do it in Excel anyway."

When we initially got feedback, no one really contested the architecture itself. The core idea was well received. What they pointed out – as the software engineering community always does – is not the claim, but the method used to justify it. Because there are rarely true domain experts in software engineering conferences. So we were told to validate it against a specific use case – which was this infrastructure-as-a-service scenario in orbit – and to ask domain experts whether we had missed the mark.

There should be nothing active there. I'm going to grab some water. These are not the actual task timings, these are just the fences. We're here, you must operate within this space. If you go beyond it, it gets blocked. When Puzzle starts, it works within this window. It might finish earlier. So what happens then? It ends early and starts again. If I exploit this, it creates a mess.

There's a priority mechanism: nominally all tasks have a priority of 139. This pump, as you can see, hasn't been added yet because it's still under discussion. And this one has a priority of 142. There's a rule that it can only be blocked by 128. So yes, 139 could be blocked by 133, 134… the lower the number, the higher the priority.

Any anomaly is just a parameter, but basically what the self-healing does, when it comes here, first of all, it's not the self-healing that checks for anomalies. Actually, what checks the fences is the BUSB. At every BDS, it checks the task descriptors to see which tasks are ongoing and maybe outside their fences.

And based on that, it can generate different types of errors. I didn't quite understand this part though: if a process is running and it crashes because of the fence, wouldn't it be faster to decide if it's overlapping with another process?

It's just a task. But no, because the BUSB handles that. The task that keeps things safe is the BUSB, because it checks what's happening every BDS cycle. It always runs. The SFDIR, I don't even remember why… There's the BUSB task. It's not a continuous thread, it runs at each BTS for a maximum of 1.5ms. If it exceeds that, it triggers a crash.

Let's keep in mind we're recording. It's not a military discussion, even though it may sound like one.

What we've done by working on this is to dive into a specific task that we understood more deeply. Our goal was to provide structure, something basic. Quick recap: a while back, in the ASTRA project, there was a formal requirement to adopt a Digital Twin. That's where the idea came from.

Given how manual and expertise-heavy scheduling is, especially if the business mission changes, like when you launch a satellite nearly empty and dynamically add services depending on ESA or other third-party needs, we thought: let's make a small tool that automates what's currently done manually.

So this becomes a decision-support system. It lets you define specs, run optimizations, and validate whether they're acceptable. If so, you go ahead. If not, you refine.

Scheduling hasn't traditionally been seen as very demanding. It's platform software—it doesn't change often. It runs the same tasks for a decade. In fact, this tasking was done on a 66 MHz processor, and even then we didn't have task overruns. The difficulty in scheduling was deciding exactly when to acquire data from the flight units (gyroscope, magnetometer, etc.) within the time slice.

All CAN or 1553 transactions were spread across the time slice, and the challenge was acquiring the data at the right time so that the attitude control algorithms could propagate them correctly. This is something you can’t detect from telemetry, because it happens at the CAN/1553 level.

Originally we had nine tasks, plus a tenth for scrubbing. Over the mission, new requirements emerged—like managing two onboard cameras, one for visible spectrum and one for UV. They needed to work in parallel and process images onboard (e.g., filtering environmental noise, compressing images).

Given the constraints (50 MHz CPU, 2MB RAM), the software had to coordinate all this. Tasks start based on events and then self-regulate. Now, what the decision-support tool does is: you define the tasks, their functional dependencies, expected execution times, and the tool computes offsets (i.e., the delay from time zero of the scan window) and the priority (0 to 255) for each one.

The optimizer minimizes preemptions and maximizes earliness.

Originally only one task (the control loop) preempted others. Later we added the CMG, which needs to be commanded precisely. The bus task couldn’t be overloaded—it must remain lightweight. So now we made this configurable too, and the tool could also estimate the temporal cost of an architectural change.

This is already supported in the simulation output, which includes a full visual execution trace with red lines where preemptions happen. Everything is highly configurable. You can either provide a shadow plan or let the tool compute priorities and durations to generate everything.

Currently, it always assumes a maximum of 15 BTS cycles. The tool is tailored to our case, a prototype, but the underlying metamodel is generic and can be extended. In fact, the simulator doesn’t really “know” what BTS means. It just schedules according to time units (e.g., 0.1ms granularity).

We built the GUI because we realized that Excel wasn’t enough—visually it may look similar, but behind it there’s now a structured model. You define tasks like in Excel, but with an actual structure underneath.

You can also specify that a task depends on another, thus defining a precise structure. The resulting model is always conformant to the metamodel: you can't input something invalid. For example, if priorities must be between 0 and 50, it will enforce that. It will also walk through the dependency graph.

This model supports dependencies without requiring priority values. Of course, execution dependencies don't imply priority, so a higher-priority task might still interrupt. That's actually one of the key benefits. For example, CMG might have the highest priority overall, even if another task doesn't depend on it.

So the solver selects priorities to meet the defined objectives. The model also supports periodicity: if a task must run four times, it's instantiated four times. Functional dependencies allow you to declare that Task A produces data for Task B, even without setting explicit priorities.

These are complementary constraints. Priority defines ordering, dependencies define causality. Fences further refine the constraints (e.g., earliest start, latest finish).

If you include all of this—start time, end time, dependencies—then it becomes computationally challenging. Still, having this structure allows reasoning. For instance, one constraint was to keep CPU usage below a certain percentage. That's hard to define precisely, but we know the system should not be overloaded.

CPUs don't throttle like modern desktop processors. If the processor is working, it's fully on. In one case, ESA asked us for CPU usage, and we replied 100%. Panic ensued. But it was accurate: the scrubbing task fills all the idle gaps, so technically, the CPU is always doing something.

But components shouldn't be pushed to 100%. Thermal constraints apply. CPUs used in space don't behave like Intel chips—they don't downclock. So yes, the schedule was patched together, mostly to enforce thermal margins and safety.

We know that our human-based scheduling leaves huge margins. Something like Puzzle and 7CM are wildly overestimated. We used upper bounds for all process execution times, ignoring interplay and anticorrelations. Ideally, the tool could also help with that kind of interplay analysis.

In fact, one idea is to use the tool to investigate issues with high-priority tasks. Execution times recorded in the system include wall time, not pure CPU time, because time measurement happens inside the task. So the measurement is affected by preemptions.

The tool reconstructs CPU time from wall time, assuming no “phantom tasks.” It’s not perfect, but it works under those assumptions. Some tasks, like BUSP, are scheduled like any other. Others, like background tasks, run in all leftover time slices with the lowest priority. These are for scrubbing (i.e., refreshing memory) and telemetry.

There used to be a telemetry task for live data, but that’s now gone. At this point, we just want to understand whether the loop we’ve proposed makes sense. That’s why we’re here. The loop is: the Digital Twin proposes a new schedule, it’s validated by a supervisor, and then uploaded to the Physical Twin (i.e., the satellite).

Once executed, telemetry flows back down, and the Digital Twin evaluates whether the behavior aligns with expectations.

If something fails, it’s like sending a command saying “destroy me.” But thanks to the model structure, the optimizer will never propose an obviously infeasible configuration. Sure, the user might misconfigure something, but that’s a different matter.

There are two main failure modes: either an error in specification or a numeric error in the solver. But if that happens, you’ll see it immediately in simulation—the tool shows red flags or anomalies. The idea of scheduling memory bandwidth (i.e., the bus) in addition to CPU is still pending.

This is all solvable. Static checks can be added to the solver, like verifying that all fences across all tasks are schedulable within the scan cycle. You can also add domain-specific rules. Now, shall we move on to the seven questions?