Analyzing spreadsheets for parallel execution via model checking

Thomas Bøgholm, Kim G. Larsen, Marco Muñiz, Bent Thomsen and Lone Leth Thomsen

Department of Computer Science, Aalborg University, Denmark

Abstract. In this paper we briefly report on work in the Popular Parallel Programming (P3) project where we follow in the footsteps of Bernhard Steffen using the idea of program analysis via model checking and abstract interpretation. The programs we analyze are spreadsheet programs, which for long have been identified as an ideal programming model for parallel execution. We translate spreadsheet programs into Timed Automata Models, which may be analyzed by the UPPAAL model checker and its derivatives, with the purpose of finding schedules for parallel execution. In this paper we mainly focus on the techniques and scalability issues of various variants of UPPAAL, but also report briefly on the performance results achieved through the parallelization.

1 Introduction

Mani Chandy noted as early as in 1985 in his keynote at the fourth annual ACM symposium on Principles of distributed computing (PODC '85) entitled *Concurrent programming for the masses* that a programming model based on spreadsheets would reach a much wider audience and should be much easier to parallelize than the traditional programming model(s) [7]. However, in the three decades that have passed since this keynote only sporadic efforts have been made in this area [21, 2, 11, 4].

To realize the idea of parallel programming via spreadsheets, it is necessary to adapt and further develop program analysis techniques to the spreadsheet programming model to identify the parts of a program that can be executed in parallel and subsequently find schedules for their execution.

The idea of program analysis via model checking was pioneered by Bernhard Steffen and presented in his 1991 paper entitled *Data Flow Analysis As Model Checking*[20] and further elaborated with David Schmidt in the paper entitled *Program Analysis as Model Checking of Abstract Interpretations*[18].

The Popular Parallel Programming (P3) project[1] set out to follow in the footsteps of Bernhard Steffen by using the idea of program analysis via model checking and abstract interpretation to investigate various approaches to parallelizing the execution of spreadsheet programs based on the open source spreadsheets Corecalc and Funcalc[2] implemented in C# and thoroughly described in

[1] `https://www.itu.dk/~sestoft/p3/`

[2] `http://www.itu.dk/people/sestoft/funcalc/`

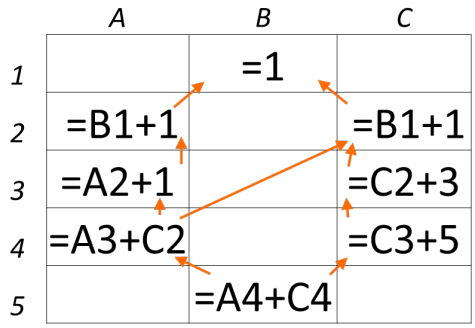

Fig. 1: Example spreadsheet

[19]. The P3 project views spreadsheets as a dataflow language with the purpose of improving compilation of dataflow languages to shared-memory multicore machines, partly by drawing on recent advances in static execution time estimates based on abstract interpretation [6] and scheduling techniques based on timed automata [10, 12, 3, 5, 16, 1].

2 Spreadsheets and dataflow

To see how spreadsheets can be viewed as dataflow programs we first look at an example. Figure 1 shows a small spreadsheet with 8 active cells, A2, A3, A4, B1, B5, C2, C3 and C4. Only the formulae are shown, whereas in a normal spreadsheet application the results of the computations would be shown.

B1 is a *data cell*, where the remaining cells are formulae directly or indirectly depending on B1. In a small spreadsheet like this, it is easy to see that the calculation of the value of the formula in cell B4 depends on the value of the formulae in cell A4 and C4. The formula in cell C4 depends on cell C3, which in turn depends on cell C2, depending on cell B1. Similarly cell A4 depends on cell B1 and A3, which depends on A2, which depends on B1. This dependency relationship is depicted by the orange arrows in Figure 1. Thus to calculate the results presented in Figure 1 a dataflow in the reverse order of the dependency relationship is needed, i.e. data from cell B1 flows into the formulae in cell A2 and C2. A2 flows into A3. C2 flows into C3 and A4. C3 flows into C4 and finally A4 and C4 flow into B5.

Based on the dependency relationship one can construct a schedule for executing the formulae in parallel on a dual-core machine such that the needed dataflow between cells is upheld. One schedule could be on CPU 1 calculate cell B1. Then in parallel calculate cell A2 on CPU1 and cell C2 on CPU2. Then in parallel A3 on CPU 1 and B3 on CPU 2, followed by cell A4 on CPU 1 and B4 on CPU2 in parallel. Finally cell B5 can be computed on CPU 1. This schedule would require 5 time units, assuming that the calculation of each cell takes 1 unit. A sequential calculation would require 8 time units.

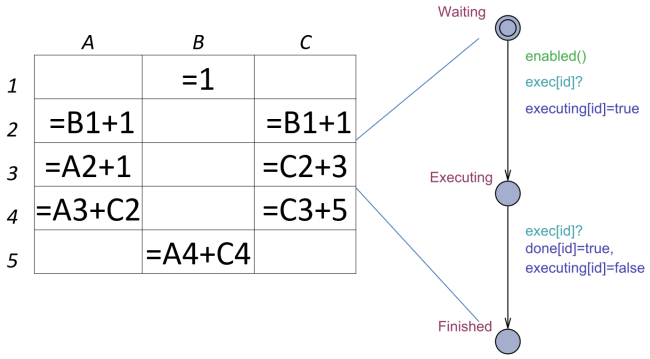

Fig. 2: Example spreadsheet with TA for cell C3

The example in Figure 1 is small enough that the dependency relationship and dataflow can be inspected or even constructed manually. However, the relationship quickly becomes difficult to keep track of manually.

3 Generating Timed Automata models from spreadsheets

In this section we show various translations from spreadsheet into Timed Automata (TA) which in turn may be analyzed with various variants of the Uppaal model checker. Timed Automata and extensions such as Priced Timed Automata, together with model checkers, especially the Uppaal model checker, have for more than a decade been used to solve scheduling problems by a reformulation as reachability problems [10, 12, 3, 5, 16, 1].

We regard each cell as a task which is translated into a separate process in Uppaal. Similarly, we generate a process for each computation unit, i.e. CPU, and the scheduling algorithm. These processes are then composed in parallel into a single model. Processes synchronize using channels, and have access to a number of functions, which allows for expressing more complex functionality in a small C-like language.

Figure 2 shows an example spreadsheet with a Uppaal TA task model for cell C3. The general idea is to translate each cell in a spreadsheet into such models which are then combined into one TA with dependencies between tasks. The task model consists of three locations: *Waiting, Executing, Finished*, which represent the three states of a task. These three states are linked through two edges: The first edge, from *Waiting* to *Executing* contains a *guard*, *synchronization*, and an *update*.

The guard `enabled(id)` is a call to the function `enabled(job_t id)`, this must evaluate to true for enabling the transition to the *Executing* state. This edge is synchronizing on channel `exec[id]?`, a receiving synchronization indicated by `?`. The `id` of the process is used as index into the channel array `exec[job_t]`. Last, the update `executing[id]=true` atomically updates the executing flag for the current task, indicating that it is currently executing. The second edge from

the *Executing* state to the *Finished* state is enabled when the channel `exec[id]` is signaled. Taking this transition sets the done flag for this process and unsets the running flag.

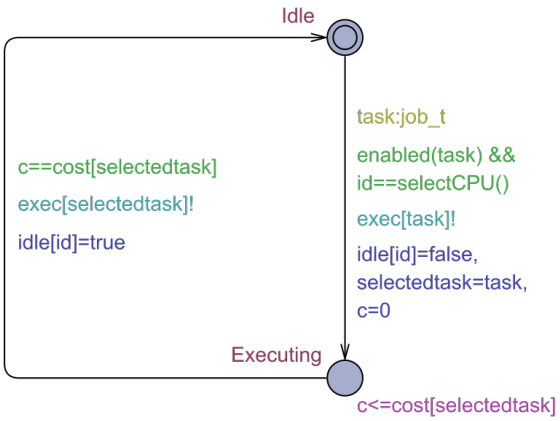

Fig. 3: CPU template

Figure 3 shows the model of a CPU which consists of two locations, *Idle* and *Executing*, representing the two states of a CPU in our model. CPU is a template parameterized with id of type `cid_t`, an integer subtype which ranges from zero to the number of CPUs. Additionally, the CPU template has two locally defined state variables: `clock c`, a clock variable for recording execution time spent in location *Executing*, and `job_t selectedtask`, representing the task this CPU is currently executing. In the *Executing* location, the invariant `c<=cost[selectedtask]` limits the time spent in this location to the execution cost of the selected task.

In Uppaal the system declarations for the resulting model then consist of:

```
1  Sheet1_A2 = Task(1);
2  Sheet1_A3 = Task(2);
3  Sheet1_A4 = Task(3);
4  Sheet1_B1 = Task(4);
5  Sheet1_B5 = Task(5);
6  Sheet1_C2 = Task(6);
7  Sheet1_C3 = Task(7);
8  Sheet1_C4 = Task(8);
9  Cpus(const cid_t c) = CPU(c);
10
11 system Cpus, Sheet1_A2, Sheet1_A3, Sheet1_A4, Sheet1_B1, Sheet1_B5,
       Sheet1_C2, Sheet1_C3, Sheet1_C4 ;
```

Line 9 creates process instances of the CPU template for each value in *cid_t*, named Cpus. Lines 1-8 create instances for each task, with a name representing the cell in the spreadsheet. Each instance is created separately in order to give identifiable names for each task in the resulting trace. The fastest trace will be the optimal schedule. Here fast refers to lowest global clock value, not the length

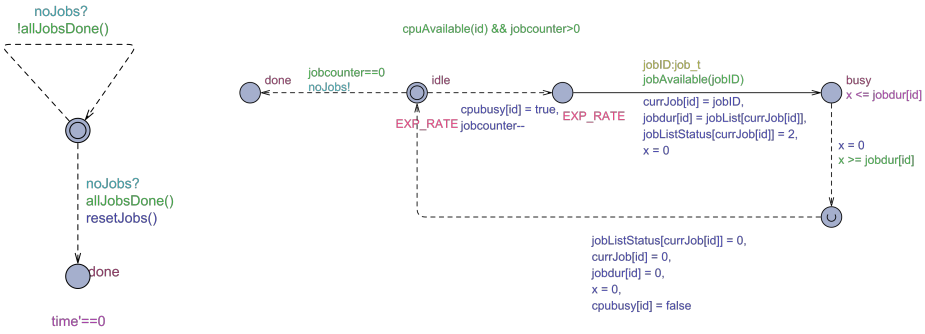

Fig. 4: Stratego Model. Left) Environment indicating if there are pending jobs. Right) CPU-Model where controllable choices are among available jobs.

of the trace, the latter being shortest trace in UPPAAL. The number of clocks in this model is the number of CPUs plus one, for recording the global clock. Unfortunately this approach does not scale beyond toy-like spreadsheets like the one depicted in Figure 2. This is not surprising as e.g. [13] reports on various task graph scheduling examples with up to 16 tasks. Larger examples quickly run into the state-space explosion problem.

However, often we do not need the optimal schedule, but *a good enough* schedule is sufficient. Such schedules may be explored by UPPAAL-STRATEGO [8] which can be used to model $1^1/_2$-player games where the opponent is stochastic. Given a game UPPAAL-STRATEGO can synthesize near-optimal strategies for complex systems. It has successfully been used for controlling floor heating systems [14] and controlling traffic lights [9].

In a nutshell UPPAAL-STRATEGO synthesizes near-optimal strategies by: staring with a uniform distribution over the controllable choices, generating runs, evaluating how good are these runs, refining the distribution on the controllable choices via learning algorithms, and iterating.

Figure 4 illustrates the UPPAAL-STRATEGO model for 1-CPU. It is a game between the scheduler and the environment which includes the jobs to execute. The solid arrows are the scheduler choices whereas the dashed arrows correspond to the environment choices. Note that the solid arrow has a select statement jobID:job_t which is equivalent to the enumeration of all tasks with one solid arrow for each task. The intuition from the model is as follows. First at location idle a delay is chosen from the exponential distribution with rate EXP_RATE, if there are no jobs left location done is reached, otherwise if a CPU and a job are available a job has to be taken. A job is taken by executing one of the solid arrows induced by jobID:job_t leading to location busy, the CPU stays at this location for the duration of the job. When the job duration has elapsed the environment sets the status of the job to 0 indicating that the job is done, and returns to the initial location. Table 1 shows the results of using UPPAAL-STRATEGO from Figure 4 in different spreadsheets. In the generated models, all cell computing costs are random values using same initial seed. These costs will in future implementations

be replaced by cost inferred based on abstract interpretations of the execution time for cell formula [6].

Model	CPUs	Cost	Greedy	Time Sec.	Speedup %
Supportgraph 31 cells, 119 dependencies	2	7.047	6.959	42	-1.20
	4	4.884	4.981	45	1.95
	8	4.611	4.611	62	0.00
	16	4.611	4.611	59	0.00
Example 115 cells, 65 dependencies	2	24.890	25.411	1.627	2.05
	4	12.581	13.375	1.373	5.94
	8	6.484	6.993	2.202	7.28
	16	3.450	4.035	199	14.50
Formulacopies 73 cells, 255 dependencies	2	16.051	16.087	253	0,09
	4	8.522	8.181	10	-4,17
	8	4.650	4.307	216	-7,96
	16	3.900	3.982	683	2,06

Table 1: Results for UPPAAL-STRATEGO

4 The dependency scheduler

The dependency scheduler is a generic task scheduler for the .Net platform, originally developed by Nichlas Korgaard Møller as part of his MSc [17]. The dependency scheduler takes as input a set of tasks and a description of their dependencies. The scheduler will then execute these task in such a way that if a task depends on other tasks, it will only execute when these tasks have completed, e.g. if we have three tasks A, B and C, where task A and task B do not have any dependencies and task C is dependent on task B, then the scheduler will execute task A and task B concurrently, and when task B finishes, task C will start. The dependency relationship is described via a dependency graph which may come from a task dependency analysis produced by (various versions of) UPPAAL.

The dependency scheduler uses the thread pool from Microsoft .NET library, which is used to keep track of the threads, managed by .NET. All threads are created from system start up, so no additional time has to be spend on creating new threads during execution.

The dependency scheduler will first start all tasks without dependencies. Tasks will signal when they finish and as soon as tasks dependent on finished tasks are ready for execution, they will be released. Thus a kind of wave of tasks goes through the set of tasks until all tasks have been executed. For spreadsheets, this wave will follow the dataflow based on the schedule of tasks calculated by UPPAAL.

The dependency scheduler has now been fully integrated into the Funcalc platform and can take as input cell formulae wrapped as .Net tasks together with a schedule produced by UPPAAL. For efficiency reasons *null* and *constant data cells* will not be included in the dependency scheduler.

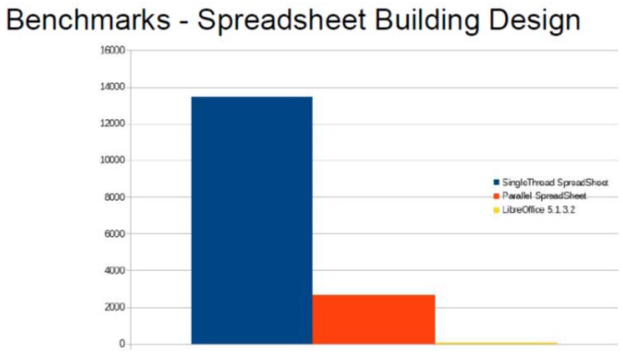

Fig. 5: Benchmarks – Building Design

We have carried out a small performance study based on two spreadsheets, the *Building Design benchmark* and the *Ground Water daily benchmark*, from the LibreOffice Benchmarks [15] developed in connection with a study of parallelisation of the LibreOffice spreadsheet on AMD GPUs [4]. These spreadsheets have about one million data cells and about 50.000 formula cells. However, they differ slightly in the complexity of the formulae and the dependencies between cells.

Figure 5 shows the average execution time of ten runs of the Building Design benchmark using the original sequential version of Funcalc, the version of Funcalc with the dependency scheduler and an execution with the LibreOffice GPU accelerated version. Lower is better. As can be seen LibreOffice is the fastest executing at 84.86ms, then Parallel Spreadsheet with 2668.8ms and then Singlethread with 13463.4ms. So we obtain approximately a five-fold speedup on a 6 core machine. The benchmark was executed on an I7-5930k 6 core machine with a 3.5 GHz clock and 32 GB DDR4 RAM; the program was executed in 32bit mode limiting memory usage.

Figure 6 shows the average execution time of ten runs of the Ground Water daily benchmark. Again Lower is better. On this benchmark the Parallel Spreadsheet is the fastest at 7142.7ms, then LibreOffice with 15959.97ms, then Singlethread with 38327.9ms. In this benchmark we see at 5.3 time speedup over the sequential version.

5 Conclusions

The Popular Parallel Programming (P3) project has been inspired by Bernhard Steffen to use the idea of program analysis via model checking and abstract interpretation. We translated spreadsheet programs into Timed Automata Models, which were analyzed by the Uppaal model checker and its derivatives, with the purpose of finding schedules for parallel execution. Execution time for each formula in the spreadsheet is estimated using abstract interpretation. We mainly focused on the techniques and scalability issues of various variants of Uppaal,

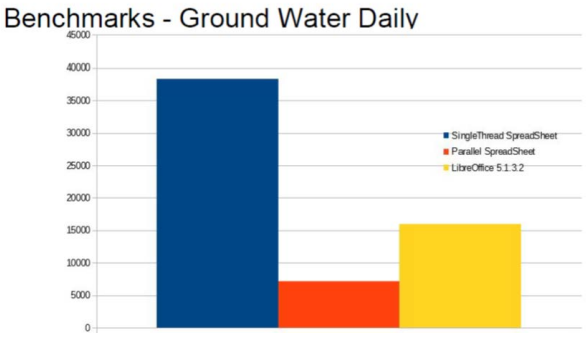

Fig. 6: Benchmark – Ground Water daily

but also reported briefly on the performance results achieved through the parallelization analysis. On some benchmarks the parallel version of Corecalc and Funcalc gain over a five-fold speed up on a six core machine.

References

1. Abdeddaïm, Y., Kerbaa, A., Maler, O.: Task graph scheduling using timed automata. In: Parallel and Distributed Processing Symposium, 2003. Proceedings. International. pp. 8–pp. IEEE (2003)
2. Abramson, D., Roe, P., Kotler, L., Mather, D.: Activesheets: Super-computing with spreadsheets. In: 2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference. pp. 22–26. Citeseer (2001)
3. Alur, R., La Torre, S., Pappas, G.J.: Optimal paths in weighted timed automata. In: International Workshop on Hybrid Systems: Computation and Control. pp. 49–62. Springer (2001)
4. AMD: Collaboration and open source at amd: Libreoffice (2015), `https://developer.amd.com/collaboration-and-open-source-at-amd-libreoffice/`
5. Behrmann, G., Fehnker, A., Hune, T., Larsen, K., Pettersson, P., Romijn, J., Vaandrager, F.: Minimum-cost reachability for priced time automata. In: International Workshop on Hybrid Systems: Computation and Control. pp. 147–161. Springer (2001)
6. Bock, A., Bøgholm, T., Sestoft, P., Thomsen, B., Thomsen, L.L.: Concrete and Abstract Cost Semantics for Spreadsheets. Technical Report, TR-2018-203, IT University, Denmark (In preparation, 2018)
7. Chandy, M.: Concurrent programming for the masses (invited address). In: Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing. pp. 1–12. PODC '85, ACM, New York, NY, USA (1985). https://doi.org/10.1145/323596.323597, `http://doi.acm.org/10.1145/323596.323597`
8. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal stratego. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 9035, pp. 206–211. Springer Berlin Heidelberg (2015), `http://dx.doi.org/10.1007/978-3-662-46681-0_16`

9. Eriksen, A.B., Huang, C., Kildebogaard, J., Lahrmann, H., Larsen, K.G., Muniz, M., Taankvist, J.H.: Uppaal stratego for intelligent traffic lights. In: Proceedings of the 12th ITS European Congress, Strasbourg, France, 19-22 June (2017)
10. Fehnker, A.: Scheduling a steel plant with timed automata. In: rtcsa. p. 280. IEEE (1999)
11. Hirsch, A.: Compiling and optimizing spreadsheets for FPGA and multicore execution. Ph.D. thesis, Massachusetts Institute of Technology (2007)
12. Hune, T., Larsen, K.G., Pettersson, P.: Guided synthesis of control programs using uppaal. Nord. J. Comput. **8**(1), 43–64 (2001)
13. Jørgensen, K.Y., Larsen, K.G., Srba, J.: Time-darts: A data structure for verification of closed timed automata. arXiv preprint arXiv:1211.6195 (2012)
14. Larsen, K., Mikucionis, M., Muniz, M., Srba, J., Taankvist, J.: Online and compositional learning of controllers with application to floor heating. In: Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16). LNCS, vol. 9636, pp. 244–259. Springer-Verlag (2016)
15. LibreOffice: Libreoffice benchmarks (2011), `https://gerrit.libreoffice.org/gitweb?p=benchmark.git;a=tree`
16. Maler, O.: Timed automata as an underlying model for planning and scheduling. In: Proceedings of the 2002 International Conference on Planning for Temporal Domains. pp. 67–70. AAAI Press (2002)
17. Møller, N.K.: Pre-analyses dependency scheduling with multiple threads (2016)
18. Schmidt, D., Steffen, B.: Program analysis as model checking of abstract interpretations. In: International Static Analysis Symposium. pp. 351–380. Springer (1998)
19. Sestoft, P.: Spreadsheet Implementation Technology: Basics and Extensions. The MIT Press (2014)
20. Steffen, B.: Data flow analysis as model checking. In: Proceedings of the International Conference on Theoretical Aspects of Computer Software. pp. 346–365. TACS '91, Springer-Verlag, Berlin, Heidelberg (1991), `http://dl.acm.org/citation.cfm?id=645867.670930`
21. Wack, A.P.: Partitioning dependency graphs for concurrent execution: a parallel spreadsheet on a realistically modeled message passing environment (1996)