Developing a Scheduler with Difference-Bound Matrices and the Floyd-Warshall Algorithm

Lorenzo Ridi, Jacopo Torrini, and Enrico Vicario, Università di Firenze

// A study of difference-bound matrices and the Floyd-Warshall algorithm in the development of an online scheduler provides the backdrop for a reflection on software practice and algorithmic theory. //

A SURPRISING VARIETY of applications can come from common classic algorithms. The Floyd-Warshall algorithm[1] is one of the most widely adopted solutions for the all-shortest-paths problem on directed graphs. In one of its many applications, Floyd-Warshall is used for solving systems of inequalities called *difference-bound matrices* (DBMs).[2,3] In turn, DBMs form the basis of well-known verification and model-checking tools such as Uppaal,[4] Kronos,[5] Romeo,[6] Tina (Time Petri Net Analyzer),[7] and Oris,[8] which support modeling and analysis of concurrent systems with nondeterministic timers.

We report our experience in the development of an online job scheduler using DBM data structures and the Floyd-Warshall algorithm. In our setting, jobs have a predefined, deterministic execution time, but bounded slack delays can control their interarrival time. The delays must satisfy precedence and mutual-exclusion constraints and still minimize the overall completion time. Although the problem formulation can fit a variety of applications, we addressed it concretely when engineering a real industrial system (see the "Online Scheduling in a Biological Analysis System" sidebar).

In principle, we could cast the problem as a case of real-time model checking by

- representing concurrent sequences and constraints through nondeterministic timed formalisms, such as timed automata (TA) or Time Petri Nets (TPNs), and
- resorting to general, well-established solution techniques and tools.

This approach could help with understanding the problem and with offline analysis. However, it would fail to incorporate the problem's specificity and would be far more complex than an online operation can usually handle.

Abstract-Problem Formulation

As a first step, the physical problem needs a logical formulation that captures characterizing traits but omits unnecessary details. This not only supports requirements validation but also opens the door for solutions to individual problems through the knowledge of a wider community, which is accessible through scientific surveys, numerical recipes, digital libraries, or even general-purpose search engines. In this process, it's crucial to find search

0740-7459/12/$31.00 © 2012 IEEE

ONLINE SCHEDULING IN A BIOLOGICAL ANALYSIS SYSTEM

We concretely deployed the algorithms we discuss in the main article on the online scheduler of an electromechanical system for immunoenzymatic analysis manufactured by BioMérieux.

The system executes multiple concurrent analyses, each comprising a sequence of reactions on a biological sample moved across subsequent cells through a moving pipettor (see Figure A). At each sequence's end, a moving read head acquires the analysis results. Both the pipettor and read head are shared among all concurrent analyses, and their operations have deterministic durations. Each reaction has a deterministic duration, resulting from biological protocols; however, we can add a bounded slack time between subsequent reactions of each analysis. The scheduler must determine the slack times so as to minimize the completion time for the overall set of analyses, while preventing situations in which two analyses need the same moving mechanical device at the same time.

The scheduling algorithm must run on the embedded hardware shipped with the system (in the present version, a 200-MHz CPU with less than 32 Mbytes of memory and no more than 6 Mbytes allocated to the scheduling component). This situation constitutes a major challenge. Moreover, to guarantee adequate responsiveness for the system's operators, the algorithm must provide a conclusive (though not necessarily optimal) schedule within a few dozen seconds. Typical values for the problem size are 6 to 12 parallel analyses, each including from 8 to 12 pipettor and read-head actions. In the formal representation of the problem, the problem could involve determining the release times for 12×12 jobs, each depending on the number of slack times, which ranges from 1 to 12.

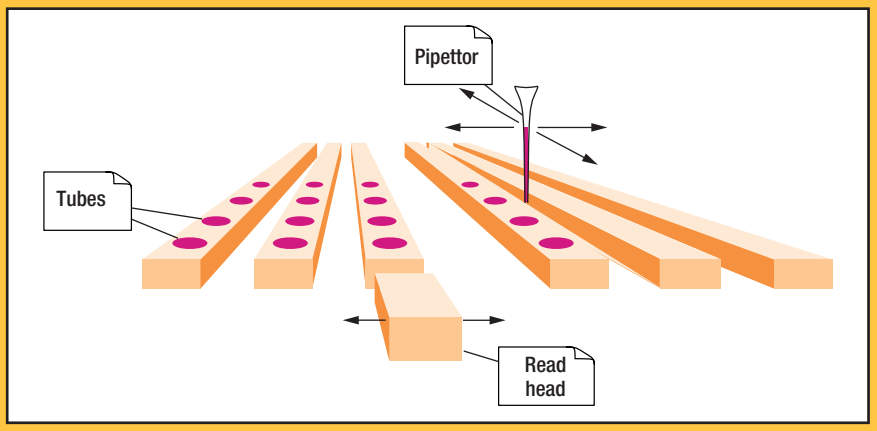

FIGURE A. A scheme of the electromechanical system for immunoenzymatic analysis. Each slot comprises several test tubes on which the pipettor and read head can act simultaneously.

keywords that capture the problem and comply with the community's specific terminology. In our case, the search phrase "job shop scheduling" provides an agreed notation for a first abstraction of our problem.

This abstraction involves a set of n nonpreemptible jobs $J_1 \ldots J_n$, with release times $r_1 \ldots r_n$ and deterministic execution times $e_1 \ldots e_n$, statically assigned to m identical, independent machines $M_1 \ldots M_n$ and subject to two types of constraints. First, for any two jobs J_i and J_k, a precedence constraint might require that the delay between the completion of J_k and the start of J_i is within the range of d_{ik}^- as a minimum and d_{ik}^+ as a maximum:

$$d_{ik}^- \leq r_i - (r_k + e_k) \leq d_{ik}^+.$$

Second, a mutual-exclusion constraint might require that the execution periods of two jobs J_i and J_k don't overlap:

$$r_i \geq r_k + e_k \vee r_i + e_i \leq r_k. \quad (1)$$

A schedule satisfies this kind of constraint if J_i starts after the end of J_k or vice versa.

The scheduling problem determines $r_1 \ldots r_n$, subject to all prescribed precedence and mutual-exclusion constraints, so as to minimize the completion time of the overall set of jobs. Figure 1 illustrates this problem.

A State-Space Analysis Approach

In the next step, we focus our search for tools and technologies in the context of the scheduling community, seeking those

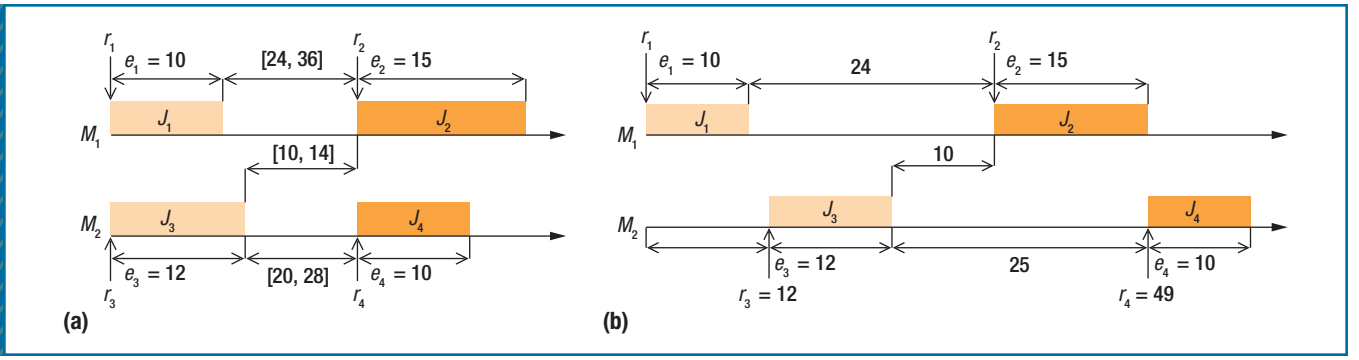

FIGURE 1. The scheduling problem. (a) Four jobs (J_1–J_4) allocated on two machines (M_1 and M_2). The variable e indicates an execution time, and r indicates a release time. J_2 has a mutual exclusion with J_4. It also has a precedence constraint that binds its start time to the completion times of J_1 and J_3 falling in the intervals [24, 36] and [10, 14], respectively. A precedence constraint also binds J_4 to start with a delay in the interval [20, 28] after J_3 completes. (b) A schedule that minimizes the overall completion time.

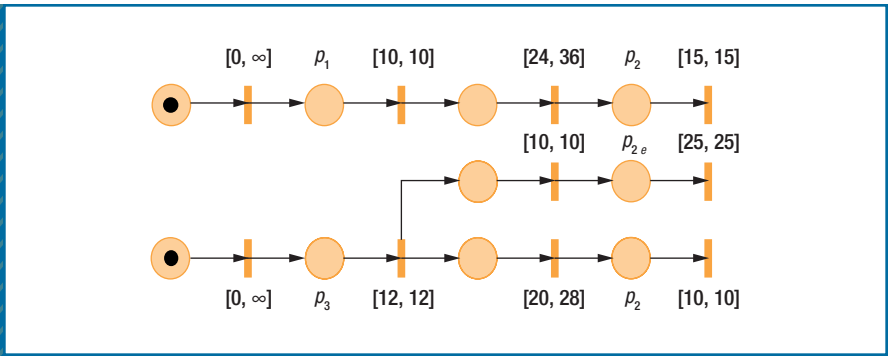

FIGURE 2. A Time Petri Net that encodes sequencing and timing constraints of the scheduling problem in Figure 1. A scheduling execution is correct if and only if it can remove all tokens from the model without visiting any state where place p_2 has a token while p_{2e} is empty or where both p_2 and p_4 contain a token.

that fit the specific needs of the problem. In doing so, some aspects of the formulation are more selective than others.

In particular, a salient trait of our specific problem is the concurrent progress of multiple activity threads with nondeterministic delays that take values within bounded intervals. This narrows the search to a family of tools based on formalisms, such as TA or TPN, that abstract jobs, slack times, precedence constraints, and mutual-exclusion constraints into logical locations, transitions, preconditions, and postconditions. Figure 2 illustrates this concept using a TPN.

Following this approach, the scheduling problem involves the identification of a witness satisfying a real-time model-checking formula[9] that selects all behaviors that reach a completion state within a maximum time T, without visiting any intermediate state that violates some mutual-exclusion constraint. Various tools—and, notably, the Uppaal environment[10,11]—support polynomial iteration providing the optimal value of T. Oris solves the optimization problem in a direct manner by evaluating the minimum and maximum time of each sequence of transitions that satisfy the expected sequencing constraints.[3]

Unfortunately, these techniques face the infamous problems of state-space explosion (the number of states grows exponentially with the number of jobs) and the curse of dimensionality (the size of data structures encoding each state grows quadratically with the number of sequences). This results in space and time complexity that severely limits the number of manageable sequences and jobs, and that's unaffordable for an online scheduler, especially if it's supposed to be embedded in an industrial system.

Results we obtained through Oris illustrate this problem. For a relatively small case with six sequences and 21 jobs with mutual exclusion between any two jobs, our analysis identified a huge state space of 255,128 state classes, with 4,216 acceptable paths. On a dual-core 2-GHz desktop computer, this took about 20 minutes. For the target platform of our system under development, the estimated execution time was about 10 times higher, and the analysis exceeded the 6 Mbytes of space allocated to the online scheduler.

Direct Exploitation of DBM Structures

Although a general-purpose tool doesn't provide the necessary performance, it can at least provide the functionality to solve the scheduling problem. This suggests we can still obtain our objective through a direct, tailored exploitation of the algorithmic basis for timed

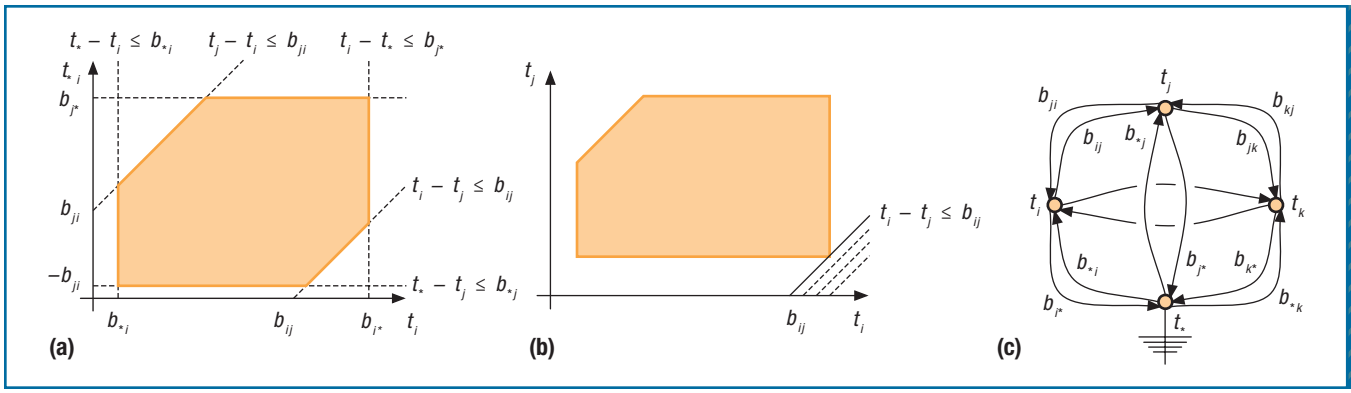

FIGURE 3. Graphical representation of difference-bound matrices (DBMs). (a) A 2D DBM zone for variables t_i and t_j. (b) A DBM in which the constraint $t_i - t_j \le b_{ij}$ (where b_{ij} are given coefficients) isn't effective and can thus take infinite different but equivalent values. (c) A graph-theoretical representation of a DBM for t_i, t_j, and t_k with an additional fictitious ground variable $t_* = 0$.

verification. In our example, the basis consists of DBM data structures and the Floyd-Warshall algorithm, and it's simpler than it might appear. This allows us to strip down the identified general-purpose theory to meet our specific problem's efficiency and portability requirements.

DBMs and the Floyd-Warshall Algorithm

A DBM is a system of linear inequalities that constrain single variables in a set $\tau = \langle t_1 \ldots t_N \rangle$ and their differences within the limits identified by given coefficients b_{ij}. We express this formally as

$$D = \begin{cases} t_i - t_j \le b_{ij} & i, j \in [1 \ldots N] \cup *, b_{ij} \in \mathbb{Q} \\ t_* = 0 \end{cases}.$$

The set of solutions of a DBM identifies a *zone*, a linear convex polyhedron whose projection on any two dimensions t_i and t_j takes the shape of Figure 3a.

When one or more inequalities aren't effective (that is, they can be removed or loosened without changing the set of solutions of the DBM), as in Figure 3b, different values of some of the coefficients will encode the same zone. To disambiguate this case, we define a unique normal form as the representation in which all inequalities are maximally tight. This occurs if and only if the following triangular inequality holds:

$b_{ij} \le b_{ik} + b_{kj}$ for all i, j, k.

When a DBM is represented in normal form, the value of b_{ij} directly represents the maximum delay between t_i and t_j. So, the minimum and maximum admissible values for t_i, which are attained in the bottom-left and top-right corners, have the values $-b_{*i}$ and b_{i*}, respectively.

Computing the normal form becomes the core step in DBM-zone manipulation; this is where the Floyd-Warshall algorithm comes into play. We can look at a DBM as a complete directed graph in which inequality bounds become arc weights (see Figure 3c). Computing a DBM's normal form then corresponds to evaluating the shortest path between any two graph vertices.

The assumption that a DBM isn't empty rules out cycles on which the sum of traversed edges is a negative value. So, we can derive the normal form through N repeated applications of the well-known Dijkstra algorithm, with a total complexity of $O(N \cdot N^2)$. The Floyd-Warshall algorithm provides a direct solution that still runs in $O(N^3)$. However, it performs much simpler operations than Dijkstra within its inner cycle and is, on average, more efficient with dense adjacency matrices, as in the case of DBM zones.

```
1   begin
2     for [i, j] ∈ [1, N] × [1, N] do
3       b_ij^N = b_ij ;
4     end
5     for k = N → 1 do
6       for [i, j] ∈ [1, N] × [1, N] do
7         b_ij^(k−1) = min{b_ij^k, b_ik^k + b_kj^k};
8       end
9     end
10  end
```

FIGURE 4. The Floyd-Warshall algorithm. The algorithm solves the all-shortest-paths problem on a directed weighted graph; in our case, it evaluates the normal form of a difference-bound matrix zone.

Figure 4 shows the Floyd-Warshall algorithm. Its core invariant is the variable b_{ij}^k, which encodes the shortest path from j to i under the restriction that only vertices with an index higher than k can be visited as intermediary nodes. So, we can assign to b_{ij}^N the initial values of b_{ij}, which is performed in the first **for** loop. It's also clear that b_{ij}^0

comprises the problem's final solution, which we attain by decreasing k from N to 1 through the second **for** loop. The algorithm's real magic is how the nested double loop on i, j derives b_{ij}^{k-1} from b_{ij}^{k}.

Understanding an algorithm's inner mechanism can even become an appealing challenge, if the goal is to optimize the algorithm and its application with respect to the problem at hand. Next, we show how we did this.

Optimizing through Adaptation

Both the precedence and mutual-exclusion constraints are encompassed in the structure of a DBM's linear inequalities. However, the mutual exclusion in Equation 1 involves choosing between two solution spaces. According to this, the set of solutions is the union of a set of DBM zones, which, unfortunately, isn't a DBM zone.

In principle, we could resolve the problem by considering all possible orderings in mutual exclusions and representing, for each of them, the resulting set of constraints in a DBM whose normalization shows each constraint's minimum feasible completion time. However, if we did this, the number of orderings would grow exponentially with the number of mutual-exclusion constraints.

To optimize the search, we organize the set of alternative orderings in a binary tree. The new algorithm iteratively constructs candidate solutions by incrementally adding mutual-exclusion constraints and verifying that the temporal parameters conform to the constraints imposed by the problem. In this setting, DBM structures encode the set of constraints that determine precedence and exclusions among jobs, while repeated invocation of the Floyd-Warshall algorithm keeps DBMs in their normal form to allow efficient manipulation and comparison of the results.

The algorithm initially builds a DBM zone D_{prec} representing only the precedence constraints that, by construction, must be satisfied by every feasible schedule. Then, it adds a mutual-exclusion constraint for each pair of conflicting jobs, using a queue to store intermediate solutions. At each step, it checks whether the added constraint produces an unfeasible schedule; if so, it rolls back to a previous legal configuration. When the algorithm has resolved all conflicts, and if the corresponding DBM isn't empty, we've found a valid solution to the scheduling problem.

Once we have organized the search as a tree traversal, four different approaches can improve efficiency. The basis of this optimization process is the understanding of the factors determining the overall algorithm's complexity.

> The basis of this optimization process is the understanding of the factors determining the overall algorithm's complexity.

In our case, the number of nodes in the tree grows exponentially (as $O(2^E)$) with the number of mutual exclusions E, whereas the cost for each node's construction grows polynomially (as $O(N^3)$) with the number of jobs N.

Heuristic search. The algorithm's behavior is determined largely by the order in which it traverses the search tree. We can limit the number of visited vertices by adopting some search heuristics, provided we're willing to accept suboptimal solutions. This reduces the complexity's exponential factor, although only in a "best effort" manner.

Good heuristics should always provide choices that will likely lead to feasible solutions without adding too much overhead to the search algorithm. We defined two simple heuristics that drive the algorithm's behavior according to the conflicting values of *feasibility* and *tightness*.

Feasibility explores loose configurations that most likely can be extended up to completion without encountering unfeasible solutions. Intuitively, we can achieve this by selecting the mutual-exclusion constraint that leaves a greater degree of freedom to constrained jobs J_i and J_j in the resulting configuration. That is, we choose the constraint for which the distance between $|r_i - r_j|$ can range over the largest interval. We call this the *longest-interval heuristic*.

Tightness aims to obtain compact schedules, even at the risk of encountering more unfeasible solutions. So, we select the constraint that creates the shortest distance between r_i and r_j. We call this the *shortest-interval heuristic*.

Time interval analysis. Another refinement aims to reduce E. Although this optimization doesn't affect the algorithm's theoretical complexity, in most practical cases it can drastically reduce computation time.

Some of the alternative choices for the mutual-exclusion constraints to be added at each iteration are already resolved by constraints of D_{prec}. For instance, if two events requiring the same resources are explicitly constrained to occur in sequence, the mutual-exclusion constraint is redundant and can be ignored. In the job-shop-scheduling application, this determines a strong speedup because jobs represent system actions that are naturally structured along linear sequences.

The warm-restart Floyd-Warshall algorithm. This optimization reduces the

procedure's asymptotical time complexity by optimizing its core engine, the Floyd-Warshall algorithm.

Every time we add a mutual-exclusion constraint, normalization of the resulting DBM zone presents a peculiarity enabling a specific algorithmic refinement. In fact, we obtain this zone by adding a single constraint to the zone extracted from the queue, which is already in normal form. This implies that the normalization affects only some of the coefficients. According to this, we can rewrite the Floyd-Warshall algorithm to run in time $O(N^2)$ when it operates on a DBM whose only coefficient b_{ij} is modified (see Figure 5).

Event aggregation. This optimization reduces the cost of constructing each search tree node, when the problem comprises jobs with immediate delays (that is, jobs that are constrained to run immediately, one after the other). This can happen if and only if we set a precedence constraint between J_i and J_j such that $d_{ij}^{+} = d_{ij}^{-} = 0$. In this case, we can merge J_i and J_j into a single job J_{ij} with $r_{ij} = r_j$ and $e_{ij} = e_i + e_j$. This results in a DBM matrix reduced by one row and one column.

Low-level code optimizations. The search algorithm's core fits in a few dozen lines of code. To successfully deploy a production version on the target platform, we implemented a lightweight ANSI-C library. It provides accessory functionalities such as I/O management, execution time control, and online switching among different heuristics, for a total of approximately 2,600 LOC.

Owing to the severe hardware limitations, we adopted several code optimizations by dirty-working selected "hot" lines of code to further conserve space and time. However, it's crucial that such low-level refinements come at the end of the optimization because we know that making a correct system run fast is easier than making a fast system run correctly.

```
1   begin
2     for k = 1, …, N do
3       if b_kj > b_ki + b_ij then
4         b_kj = b_ki + b_ij;
5         for l = 1, …, N do
6           b_kl = min{b_kl, b_ki + b_ij + b_jl};
7         end
8       end
9     end
10    for h = 1, …, N do
11      b_ih = min{b_ih, b_ij + b_jh};
12    end
13  end
```

FIGURE 5. The optimized Floyd-Warshall algorithm. This algorithm runs in time $O(N^2)$ against the $O(N^3)$ of the original formulation, assuming that coefficients of the DBM taken as input are already in normal form (coefficient b_{ij} is the only exception).

Performance Testing

We performed experiments on a desktop computer (with a dual-core 2-GHz processor). This implementation performed roughly 10 times faster than one on the target embedded platform (see the sidebar). That is, an execution took about 10 times longer on the target platform than on the desktop computer.

We ran our search algorithm on a test suite comprising three realistic examples (AP1, AP2, and AP3) taken from real-world biological analyses (and actually used in the system's production version). The protocols consisted of 18 to 24 pipettor actions, eight read-head actions, and nondeterministic delays.

We corrected experiments by incrementally adding proposed refinements to the search algorithm, which we ran using both the shortest and longest-interval heuristics. Figures 6a and 6b show the results, which compare extremely favorably to the 20 minutes the model-checking approach required.

As we expected, the successive refinements produced better results. In particular, the deeper the search tree is, the more relevant both the pruning performed through time interval analysis and the speedup of the Floyd-Warshall algorithm become. Another interesting result comes from the comparison between the longest- and shortest-interval heuristics. On realistic test cases, the latter performed better than the former, for both performing a full search and finding the best solution. (On the selected cases, the shortest-interval heuristic always found the best solution on the first try).

Lessons Learned

From a product perspective, our experience has provided a recipe for solving scheduling problems that can arise in a variety of cases, including production scheduling, personnel management, or supply-chain optimization. From the process perspective, our experience has taught us the following three lessons.

First, seek an abstraction that can achieve the needed functional capability. In our case, we had nondeterministic timers taking values within dense intervals. This situation can be cast into the expressive capabilities of a model such as a TPN or TA, or even in the practical function of some supporting tool such as Oris or Uppaal.

Second, try to manage nonfunctional requirements on the basis of the selected abstraction. To this end, you need to understand the inner mechanisms on which functional capabilities are developed. If you start from a good abstraction, these mechanisms often turn out to be much simpler than they appear. In our case, nonfunctional requirements

FIGURE 6. The results of experiments using the (a) shortest-interval and (b) longest-interval heuristics for analytical protocols AP1, AP2, and AP3. For each analytical protocol, results appear in the order of increasing optimization level: basic, time interval analysis (TIA), TIA + warm restart (WR), and TIA + WR + event aggregation (EA). The orange bar shows when the search algorithm found the first solution, the blue bar shows when it found the best solution, and the green bar shows the time required for exhaustive analysis.

mainly addressed integrability and performance, for both of which the tool solution wasn't applicable. In the end, we needed to change the level of abstraction so that we wouldn't see any more TPNs, TAs, or their supporting tools, but rather their algorithmic foundation, which is DBM data structures and the Floyd-Warshall algorithm.

Third, implement your solution using an abstraction-driven development approach, in which you see algorithms and data structures more than classes and methods. In particular, this unveils the solution's inherent complexity and drives the focus of optimization.

In our case, the number of mutual exclusions determined the search tree's depth, exponentially impacting the exhaustive search's complexity. Applying the heuristics let us prune the explored tree, but (as we mentioned earlier) just in a best-effort sense. The worst case (and the exhaustive search's cost) still remains exponential. So, putting effort into refining a heuristic surely brings some benefit, but the first step to adequately decreasing the complexity is to reduce the number of exclusions.

Also, the cost of each algorithm step was N^3, owing to Floyd-Warshall complexity. Toward that concern, event aggregation let us decrease N. However, we achieved a structural complexity reduction only when applying the warm-restart Floyd-Warshall algorithm. This is a general lesson: algorithms that are integrated in a software component are often employed in conditions that are more specific than those for which the algorithm was designed; this allows for specific optimizations.

A final lesson was perhaps best expressed by Italo Calvino in *Invisible Cities*: "Marco Polo describes a bridge, stone by stone. 'But which is the stone that supports the bridge?' Kublai Khan asks. 'The bridge is not supported by one stone or another,' Marco answers, 'but by the line of the arch that they form.' Kublai Khan remains silent, reflecting. Then

he adds: 'Why do you speak to me of the stones? It is only the arch that matters to me.' Polo answers: 'Without stones there is no arch.'"[12]

In the same way, without code there are no algorithms, and without practice, theory is of no use. But good algorithms and proper theories are often the essential building blocks of well-engineered, efficient software solutions.

Acknowledgments

This work largely benefited from the committed involvement of many people from bioMérieux Italia. In particular, we thank Andrea Carignano, who led the team with concreteness and an open mind, and Francesco Mancini, who helped to get all of this started.

References

1. R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.
2. D.L. Dill, "Timing Assumptions and Verification of Finite-State Concurrent Systems," *Automatic Verification Methods for Finite State Systems*, LNCS 407, Springer, 1990, pp. 197–212.
3. E. Vicario, "Static Analysis and Dynamic Steering of Time-Dependent Systems," *IEEE Trans. Software Eng.*, vol. 27, no. 8, 2001, pp. 728–748.
4. J. Bengtsson et al., "Uppaal—A Tool Suite for Automatic Verification of Real-Time Systems," *Hybrid Systems III*, LNCS 1066, Springer, 1996, pp. 232–243.
5. C. Kloukinas and S. Yovine, "Synthesis of Safe, QoS Extendible, Application Specific Schedulers for Heterogeneous Real-Time Systems," *Proc. 15th Euromicro Conf. Real-Time Systems* (ECRTS 03), IEEE CS Press, 2003, pp. 287–294.
6. G. Gardey et al., "Romeo: A Tool for Analyzing Time Petri Nets," *Computer Aided Verification*, LNCS 3576, Springer, 2005, pp. 261–272.
7. B. Berthomieu, P.-O. Ribet, and F. Vernadat, "The Tool TINA—Construction of Abstract State Spaces for Petri Nets and Time Petri Nets," *Int'l J. Production Research*, vol. 42, no. 14, 2004, pp. 2741–2756.
8. G. Bucci et al., "Oris: A Tool for Modeling, Verification and Evaluation of Real-Time Systems," *Int'l J. Software Tools for Technology Transfer*, vol. 12, no. 5, 2010, pp. 391–403.
9. A. Fehnker, "Scheduling a Steel Plant with Timed Automata," *Proc. 6th Int'l Conf. Real-Time Computing Systems and Applications* (RTCSA 99), IEEE CS Press, 1999, pp. 280–286.
10. K. Larsen, "Resource-Efficient Scheduling for Real Time Systems," *Embedded Software*, LNCS 2855, Springer, 2003, pp. 16–19.
11. I. AlAttili et al., "Adaptive Scheduling of Data Paths Using Uppaal Tiga," *Proc. 1st Workshop Quantitative Formal Methods: Theory and Applications* (QFM 09), Electronic Proceedings in Theoretical Computer Science, 2009, pp. 1–11.
12. I. Calvino, *Invisible Cities*, Harcourt Brace Jovanovich, 1978.

ABOUT THE AUTHORS

LORENZO RIDI is a member of the University of Florence's Software Technologies Laboratory. He's also pursuing a PhD in informatics, multimedia, and telecommunications engineering at the university. His research interests include formal techniques for the specification, qualitative verification, and quantitative validation of stochastic time-dependent systems and their integration in the development life cycle of real-time software. Ridi has an MS in computer engineering from the University of Florence. Contact him at lorenzo.ridi@unifi.it.

JACOPO TORRINI is a software engineering architect and consultant and collaborates with the University of Florence's Software Technologies Laboratory. His research interests include the experimentation of software technologies and the design and development of software solutions. Torrini has an MS in computer engineering from the University of Florence. Contact him at jacopo.torrini@unifi.it.

ENRICO VICARIO is a full professor at the University of Florence's School of Engineering. His research interests include formal methods for model-driven development, correctness verification of real-time systems, and quantitative evaluation of concurrent non-Markovian models. Vicario has a PhD in informatics and telecommunications engineering from the University of Florence. Contact him at enrico.vicario@unifi.it.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.