Solving Scheduling Problems by Untimed Model Checking

The Clinical Chemical Analyser Case Study

Anton Wijs
CWI, Department of Software Engineering
P.O. Box 94079
1090 GB Amsterdam
The Netherlands
A.J.Wijs@cwi.nl

Jaco van de Pol
CWI, Department of Software Engineering
P.O. Box 94079
1090 GB Amsterdam
The Netherlands
Jaco.van.de.Pol@cwi.nl

Elena Bortnik
Eindhoven University of Technology, Department of Mechanical Engineering
P.O.Box 513
5600 MB Eindhoven
The Netherlands
E.M.Bortnik@tue.nl

ABSTRACT

In this paper, we show how scheduling problems can be modelled in untimed process algebra, by using special *tick*-actions. As a result, we can use efficient, distributed state space generators to solve scheduling problems. Also, we can use more flexible data specifications than timed model checkers usually provide. We propose a variant on breadth-first search, which visits the states per time slice between ticks. We applied our approach to find optimal schedules for test batches of a realistic clinical chemical analyser, which performs several kinds of tests on patient samples.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*

General Terms

Measurement, Verification, Experimentation, Algorithms, Languages

Keywords

Process algebra, scheduling, search algorithms, untimed model checking

1. INTRODUCTION

The Clinical Chemical Analyser (CCA) is used to automatically analyse patient samples (blood, plasma or urine). TNO Industry, in cooperation with the Eindhoven University of Technology (TU/e), has been involved in the redesign of the CCA. The project charter was originally drawn up by Vital Scientific, a customer of TNO, to examine the possibility of a 100% throughput increase.

At TU/e several projects have been devoted to the CCA. First, the basic outline for the hardware was explored [22] while, in a parallel project, the scheduler was developed [20]. Then, the hardware for a CCA mock-up was designed [13]. Currently, a new scheduler is being designed [23]. The fact that a schedule providing optimal performance of the CCA still has not been found raised the idea to look at this problem using a modelling language.

Already a lot of research has been done in the field of timed automata to solve scheduling problems, translated to reachability problems. In a paper by Niebert, et al. [18], the problem of minimum-time reachability for timed automata is considered. It is shown that this problem can be solved by examining acyclic paths in a forward reachability graph generated on-the-fly from a timed automaton. Three algorithms are proposed to find a minimal-time path, all of which have a worst-case complexity which is worse than polynomial in the size of the simulation graph. (This result cannot be fairly compared to our algorithms presented in this paper, which are linear in the size of the state space, because their simulation graphs are symbolic in the representation of clock regions). In several papers by Behrmann, et al. [2, 3], linearly priced timed automata are introduced as an extension of timed automata with prices on both transitions and locations. Next they consider the minimum-cost reachability problem. An algorithmic solution is offered, based on a combination of branch-and-bound techniques, which can be used for limiting the search space and for quickly finding near-optimal solutions, and a new notion of priced regions. It is shown that using these techniques reduced the explored state space by 90% when compared to a straightforward breadth-first search.

Timed model checkers like Uppaal [14], a tool using timed automata to model systems, prove to be very well suited for handling scheduling problems. However, because of the usage of time, state space explosion can become a practical problem very quickly. Furthermore, it may be hard to work with a lot of data. That is why we considered the possibility to solve scheduling problems in a simpler, untimed setting. In other words, work with less theory and more brute force. In [19], the model checker SPIN is used for modelling and solving scheduling problems. The depth-first search algo-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FMICS'05, September 5–6, 2005, Lisbon, Portugal.
Copyright 2005 ACM 1-59593-148-1/05/0009 ...$5.00.

rithm of SPIN is enhanced with a branch-and-bound mechanism. The idea is that the LTL formula to be checked is modified during verification, to reflect the best solution found so far. The algorithm is implemented by linking C-code to the Promela model and therefore very specific to the architecture of the SPIN tool.

We wanted a more general approach. With this in mind, we looked at the modelling language μCRL [12]. Although not a lot of work has been done yet with μCRL in the field of scheduling, its powerful toolset [7] seems very able to attack this type of problems. Using μCRL, one can work with complex data structures, which was required for the CCA system. Besides that, recently the μCRL toolset was expanded with a distributed state space generator [5] and a distributed state space reduction tool [6], allowing very large state spaces to be generated within reasonable time. In this paper we present a general scheduling methodology for μCRL.

The paper is set up as follows: First we give an introduction to the CCA. Then we provide a short introduction to μCRL, followed by a description how to deal with scheduling problems in general using μCRL, and two methods to find a minimal-time trace in a state space. Then we give a small example of a scheduling problem and the μCRL model to solve it. After that we discuss the CCA models we used for the CCA case study, followed by the results obtained from these models. Finally, conclusions are given.

2. THE CHEMICAL ANALYSER

What follows is a description of the scaled-down CCA as we used it for the research described in this paper.

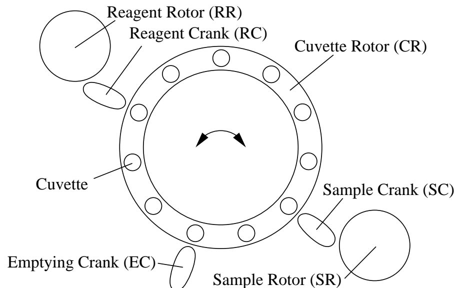

Figure 1: The scaled-down CCA

Figure 1 shows the setup of the CCA; There is a cuvette rotor containing 11 cuvettes, which are indexed from 0 to 10 counter-clockwise (this in contrast with both the CCA mock-up, which has 45 cuvettes, and the real CCA, which has 120 cuvettes). There are three cranks, which are able to perform actions on these cuvettes: The reagent crank can add a reagent from the reagent rotor to a cuvette, the sample crank can add a patient sample from the sample rotor to a cuvette, and the emptying crank can empty a cuvette. Besides that there is a mixing crank, but it is unimportant for the scheduling problem, which will become clear later on.

The use of the machine is to process test recipes. Each available patient sample should be processed according to one of three possible test recipes.

In Table 1 the three recipes are depicted. In recipe 1 first a reagent (R_1) and later a sample (S) is added to a cuvette. After that the cuvette is emptied (E). Recipe 2 is an extension of recipe 1 in the sense that after having added a sample to the cuvette a second reagent (R_2) must be added. Finally, recipe 3 requires even a third reagent (R_3) to be added to the cuvette. This adding of fluids cannot be done at any time however. The Δ occurrences in Table 1 represent delays of certain lengths (measured in time units). The values of $t_1, ..., t_7$ are limited to the following possibilities: $t_1 \geq 15, t_2 \leq 105, 3 \leq t_3 \leq 27, t_4 \leq 105 - t_3, 6 \leq t_5 \leq 21, 9 \leq t_6 \leq 42, t_7 \leq 105 - t_5 - t_6$.[1]

The CCA consists of a number of independently working parts (cranks and rotors) which have to be controlled using a set of low-level actions. In order to avoid problems, these actions are used as the building blocks for higher level instructions, so-called *operations*. Careful design of the operations has led to the property, that no errors occur within them. These are the operations available:

- $R_i(j)$: Reagent i of a test is added to cuvette j;
- $S(i)$: The sample for cuvette i is added;
- $E(i)$: Cuvette i is emptied.

Finally, a number of operations together form a *cycle*, which is the basic building block for a schedule. There are three types of cycles, the 12, 16 and 24-cycles, differing in the number of time units they require for execution. In the 12-cycles round 1 of operations occurs, in the 16-cycles rounds 1 and 2 occur, and in the 24-cycles all three rounds occur. The rounds being (in this order):

1. Given an empty cuvette i, the first reagent of a test can be added to this cuvette. At the same time, if possible, the sample for the test in cuvette $i-5$ can be added. Finally, also at the same time, if cuvette $i+3$ contains a finished test, the cuvette can be emptied.
2. If a cuvette j ($i \neq j$) is ready to receive a second or a third reagent, this reagent can be added.
3. If a cuvette k ($i \neq k, j \neq k$) is ready to receive a second or a third reagent, this reagent can be added.

The cycles can be named by listing the operations that occur in each round. We do not list the E operations though, since emptying is done whenever possible. For instance, in the 12-cycle $R_1(i)$ round 1 from the list above is carried out without adding a sample. When rounds 2 and 3 occur in a cycle, it will always be after having done round 1. Also for these rounds the necessary cuvette indexes are given. For instance, cycle $R_1SR_2(i,j)$ first performs round 1, with a first reagent being added to cuvette i and a sample being added at the same time to cuvette $i-5$, after which a second reagent is added to cuvette j in round 2. In the real machine it happens to be the case that there is no cycle which only empties a cuvette. This is important to know when looking at the results of the case study in section 7.

It was previously mentioned that there is a mixing crank. Mixing should happen every time an extra fluid is added to a cuvette. This, however, is not part of the scheduling problem, because mixing is done within the operations.

The scheduling problem is now the following: given a batch of tests to be processed, provide a sequence of cycles

[1]A time unit in the scaled-down CCA model corresponds with a duration of 4 seconds in the actual CCA.

Table 1: Recipes for the CCA

Description	Recipe
1-reagent	$R_1 \to \Delta t_1 \to S \to \Delta t_2 \to E$
2-reagent	$R_1 \to \Delta t_1 \to S \to \Delta t_3 \to R_2 \to \Delta t_4 \to E$
3-reagent	$R_1 \to \Delta t_1 \to S \to \Delta t_5 \to R_2 \to \Delta t_6 \to R_3 \to \Delta t_7 \to E$

that enable the CCA to process the tests in the minimum time possible.

3. THE LANGUAGE μCRL

Basically, μCRL is based on the process algebra ACP [4], extended with equational abstract data types [15]. In order to intertwine processes with data, actions and recursion variables can be parametrised with data types. Moreover, a conditional construct (if-then-else) can be used to have data elements influence the course of a process, and *alternative quantification* (also called *choice quantification*) is added to sum over possibly infinite data domains.

The language comes with a toolset [7] that can build a state space from a specification and store it in the `.aut` format, one of the input formats of the model checker CADP [11]. A large number of distributed systems have been verified in μCRL, for instance [9].

We will give a short overview of the language necessary for understanding this paper. For a complete reference, see [12].

A specification starts by defining the necessary data. These are specified as algebraic data types, consisting of sorts, function declarations, and equations. In fact, the Boolean sort is mandatory, since the conditional construct works with Boolean expressions. Algebraic data types yield flexibility, while keeping the language simple. In μCRL one can declare actions, which may have zero, one or several data parameters. Finally the process deadlock (δ), which cannot terminate successfully, and the internal action τ are predefined. There are eight operators in μCRL. We omit the parallel composition operator, the encapsulation operator, the renaming operator and the abstraction operator since we do not use them in this paper. We present the other four with an informal semantics.

1. The alternative composition operator (+). A process $p+q$ proceeds (non-deterministically) as p or q (if they can proceed).
2. The sum operator ($\sum_{d:D} X(d)$), with $X(d)$ a mapping from the data type D to processes, behaves as $X(d_1)+X(d_2)+\ldots$, i.e., as the possibly infinite choice between $X(d)$ for any data term d taken from D. This operator is used to describe a process that is reading some input over a data type [16].
3. The sequential composition operator (.). A process $p.q$ proceeds as p followed by q.
4. The process expression $p \lhd b \rhd q$ where p and q are processes, and b is a data term of data type *Bool*, behaves as p if b is equal to T (true) and behaves as q if b is equal to F (false). This operator is called the conditional operator.

The heart of a μCRL specification is the *proc* section, where the behaviour of the system is declared. This section consists of recursion equations of the following form, for $n \geq 0$:

$$\textit{proc} \quad X(x_1 : s_1, \ldots, x_n : s_n) = t$$

Here X is the process name, the x_i are variables and the s_i are sorts. Moreover, t is a process term possibly containing occurrences of expressions $Y(d_1, \ldots, d_m)$, where Y is a process name and the d_i are data terms that may contain occurrences of the variables $x_1, \ldots, x_n$. In this rule, $X(x_1, \ldots, x_n)$ is declared to have the same behaviour as the process expression t [10].

The initial state of the specification is declared in a separate initial declaration *init* section, which is of the form

$$\textit{init} \quad X(d_1, \ldots, d_n)$$

Here $d_1, \ldots, d_n$ represent the initial values of the parameters $x_1, \ldots, x_n$. In μCRL specifications the *init* section is used to instantiate the parameters of a process declaration, meaning that the d_i are data terms that do not contain variables.

4. TACKLING A SCHEDULING PROBLEM WITH μCRL

4.1 Modelling scheduling problems

Scheduling problems are about time; given a machine or combination of machines, which are able to perform tasks, the question in general is in which order these tasks should be performed in order to achieve the highest possible efficiency. Therefore, if we want to create a model of a system in order to solve a scheduling problem, at least we should be able to work with time.

The original process algebra μCRL has no built-in notion of time. A later addition, timed μCRL [12], adds absolute time stamps to all actions. However, these time stamps usually make state spaces infinite. Also, there are currently no tools for generating a state space for timed μCRL.

Instead, based on the work from [8, 21], we use a special *tick* action, which models time progression. This is comparable to relative discrete time [1]: A *tick* action indicates that the system moves to the next time slice. Using this technique, the duration of an execution equals the number of *tick* actions occurring in this trace. Now we can define the notion of a *minimal-time trace*:

Definition 1. Given an LTS and a transition label a, we say that there is a trace with execution time t ($t \in \mathbb{N}$) to a transition with label a iff there is a trace in the LTS starting from the starting state s_0 and reaching a transition with label a, such that the number of *tick* transitions occurring in this trace equals t. We define a trace from s_0 to a transition with label a to be *minimal-time* if there is no other trace in the LTS from s_0 to a with less *tick* transitions.

Using this definition, we can formulate a scheduling problem as a reachability problem: finding an optimal schedule to perform a batch of tasks successfully can also be seen as finding a minimal-time trace to a transition indicating successful termination in a state space containing all possible schedules as traces.

The question now is how to model scheduling problems in general using μCRL, and how to find a minimal-time trace in a state space generated from such a model. Modelling can be done by creating a scheduler process, which allows all valid executions.[2] By valid executions we mean all executions satisfying the available constraints within the system. So the scheduler can execute all available actions as long as the constraints are satisfied. The choices which valid actions to execute and when are non-deterministic; there are no built-in priorities.

It is possible though to create a scheduler process with a built-in strategy. By strategy we mean a plan saying when and how to execute the valid actions. This limits the number of possible executions (for more on strategy models, see section 4.4). The possible executions of this process can then be compared to the executions of the more general process, providing us with a way to check the effectiveness of the strategy in question.

Besides that we introduce a special action called *finished*. We use this action in such a way that it can be executed if and only if the scheduler process reaches the successful termination of an execution.

Having created a μCRL model, it is possible, using the μCRL toolset, to generate a state space from it. This state space incorporates all possible behaviour of the system described by the model. Somewhere in this state space there is at least one minimal-time trace to a successful finish. Given Definition 1, we use the *finished* action as transition a, in order to formulate a minimal-time trace to a succesful termination. Next we describe two methods to find such a trace. In the first method we use the tools as they originally exist. In the second the μCRL toolset is equiped with an optimised search algorithm.

4.2 Finding a minimal-time trace by full state space generation

One way is to build a counter in the model, which is used to keep track of the time spent since the start of the execution. If we also build in the action called *finished(t)* which is executed when a successful termination is reached, with current value t of the counter as a parameter, we can quickly find a minimal-time trace.[3] Using CADP it is possible to display all the action labels of a state space. Then we get an overview of all the occurrences of *finished(t)* with their different parameter values. We find the smallest parameter value and search for a trace leading to this *finished(t)* occurrence using a μ-calculus formula [17]. Note that in this case there is no real need anymore for *tick* actions; when a delay occurs the value of the counter is increased.

[2] One can decide to use other processes in parallel with a scheduler process. In that case it must be enforced that all *tick* actions are synchronised; if all processes can do a *tick* action, they perform a *tick* action together. If at least one of them cannot, no *tick* action occurs. How to enforce this behaviour can be found in [8, 24].

[3] Note that in the case of the *finished(t)* actions we use absolute timing [1].

Using the method described above, we need a complete state space before we can check anything. In a lot of cases though, the state space tends to be very big, in some cases even of infinite size. This possibility is even bigger when using this time counter, which never assumes the same value twice within an execution (there is no possibility for loops within the state space). It is possible though to limit the state space to the size necessary to find a minimal-time trace. To do this we can develop a strategy model; this technique is explained in section 4.4.

4.3 Finding a minimal-time trace using an optimised algorithm

There is an option in the μCRL toolset to search for a specific action while generating a state space. As soon as the action has been discovered, the toolset provides the trace to this occurrence of the action and can then stop generating. Because of the fact, that the generation is done breadth-first, as soon as an action has been found for the first time, the trace leading to this action will be the shortest one leading to it.

However, the shortest trace to an action is not always a minimal-time trace. For instance, let us say we have two traces leading to action *finished*, the first being $p_0 \xrightarrow{a} p_1 \xrightarrow{tick} p_2 \xrightarrow{b} p_3 \xrightarrow{tick} p_4 \xrightarrow{c} p_5 \xrightarrow{tick} p_6 \xrightarrow{finished} p_7$ and the second being $q_0 \xrightarrow{d} q_1 \xrightarrow{tick} q_2 \xrightarrow{tick} q_3 \xrightarrow{tick} q_4 \xrightarrow{tick} q_5 \xrightarrow{finished} q_6$. Even though trace q_0 to q_6 is the shortest trace, trace p_0 to p_7 is the fastest. What we need in order to find a minimal-time trace, is another search algorithm during generation, which deals with *tick* actions in a special way. Algorithm 1 is such an algorithm written in pseudo-code, where s_0 is the starting state of the state space and *finished* is the action we are looking for. Furthermore p and p' indicate states and a is an action. The algorithm processes a list of states, each time looking at the outgoing transitions of the chosen state. If a transition is a *tick* transition, the destination state is only checked once all states in the current time unit have been processed. In this way the generator checks states from time slice to time slice. The claim is that the algorithm searches in such a way, that when action *finished* is found for the first time, a minimal-time trace is found. Each time a new state is reached a pointer is kept to the parent state. This allows back-tracking to state s_0 once a *finished* transition is found.

Notice that we do not search traces with cycles. As pointed out in [18], we are allowed to do this. By using the set *Processed* we keep track of all the states already visited. If we visit a state for a second time, it will be at the same execution time or later than the first time we visited it. The time it took to go through the loop did not gain us anything, since we have arrived back at the same state.

It is clear that using this method it is not necessary in most cases to generate the complete state space, or generate the state space of a second (strategy) model. Worst-case the whole state space needs to be generated. A minimal-time trace is found in $O(l)$ time, with l being the number of transitions searched in the state space. Therefore this method is more efficient than the one described in section 4.2.

4.4 The use of a strategy model

As described earlier, scheduling problems can be modelled as a single process allowing all possible (valid) executions. This way we can be sure that a minimal-time trace in the

Algorithm 1 Pseudo-code algorithm for finding minimal-time trace on-the-fly

```
TimeSlice := ∅
Waiting := {s0}
Processed := ∅
while Waiting ≠ ∅ do
  TimeSlice := Waiting
  Waiting := ∅
  while TimeSlice ≠ ∅ do
    select p from TimeSlice
    for all p --a--> p' do
      if a = finished then
        return The path from s0 to p'
      else if p' not in Processed then
        if a = tick then
          add p' to Waiting
        else
          add p' to TimeSlice
        end if
      end if
    end for
    add p to Processed
  end while
end while
return No successful termination
```

resulting state space is really the fastest one possible. It can be useful however to create a model with a built-in strategy as well. Using such a model has several advantages.

First of all, a strategy model limits the amount of non-determinism, resulting in a smaller state space. For instance, we can assign different priorities to different actions, therefore eliminating non-deterministic choices between them. This means that larger problems can be solved by adding strategies. However, the found solutions may be suboptimal, because all minimal-time traces may have been pruned away.

Note that the solution found by a strategy model is an upperbound for the minimal time. So the second advantage of using a strategy model is that it can make the minimal-time trace detection method from section 4.2 more practical; using the strategy model we can get a minimal-time trace. Then we know how many time units this trace costs, say t. Following, by expanding the guards within the general model, we can force all executions of this model to stop after t time units have passed. The only expression that needs to be added to each guard is that the time counter has a value of at most t. Whether or not the strategy used is a good one, a minimal-time trace of the general model (with execution time t') can be found in the limited state space, since $t' \leq t$.

Of course it is also possible to pick a reasonable value for t, without using a strategy model. But the bigger the state space gets, the riskier this is; if the value chosen is too low, the time needed to generate the state space is probably still long and the final result will not contain a minimal-time trace. If the value is chosen too big, the generation will take too much time.

Finally, it can be checked, for small instances, whether or not strategy models provide optimal solutions: if a strategy model yields schedules of the same length as the fully non-deterministic model, this is an indication that the strategy is good. Note this is only an indication, because we can only check the strategy for problem instances. We cannot, at least using our methods, check a strategy in general.

5. EXAMPLE: 5 TASKS SCHEDULING PROBLEM

In order to facilitate comparison, we will look at the small static scheduling problem originally presented in [18] and adapted in [3]. A number of tasks (a_1, a_2, c, b_1 and b_2) need to be performed in a specific order. All tasks need to be performed precisely once, except task c, which can be performed zero or more times. The order is as follows: After task a_1 one should perform a_2, followed by (zero or more times) c. Then task b_1 needs to be executed, finishing with task b_2. The system is free to decide for itself how long it wants to delay after having performed a task. There are three timing constraints however:

1. The time between execution of a_1 and execution of b_1 should be at least 2 time units;
2. The time between execution of a_2 or the last execution of c and execution of b_1 should be no more than 1 time unit;
3. The time between execution of a_2 and execution of b_2 should be at least 3 time units.

What follows is the μCRL model of the system described above. We use three counters, x, y and z, to ensure timing constraints 1, 2 and 3 respectively. Standard sections defining the data types needed are omitted.

act $a_1, a_2, c, b_1, b_2,$ *tick, finished*

S(x:Nat, y:Nat, z:Nat, n:Nat) =

$$
\begin{aligned}
& a_1.S(x, y, z, n+1) \triangleleft n = 0 \triangleright \delta + \\
& tick.S(x+1, y, z, n) \triangleleft n = 1 \triangleright \delta + \\
& a_2.S(x, y, z, n+1) \triangleleft n = 1 \triangleright \delta + \\
& tick.S(x+1, y+1, z+1, n) \triangleleft n = 2 \triangleright \delta + \\
& c.S(x, 0, z, n) \triangleleft n = 2 \triangleright \delta + \\
& b_1.S(x, y, z, n+1) \triangleleft n = 2 \wedge x \geq 2 \wedge y \leq 1 \triangleright \delta + \\
& tick.S(x, y, z+1, n) \triangleleft n = 3 \triangleright \delta + \\
& b_2.S(x, y, z, n+1) \triangleleft n = 3 \wedge z \geq 3 \triangleright \delta + \\
& tick.S(x, y, z, n) \triangleleft n = 4 \triangleright \delta + \\
& finished \triangleleft n = 4 \triangleright \delta
\end{aligned}
$$

init S(0,0,0,0)

Using the μCRL toolset we can search for a minimal-time trace using the search method from section 4.3. This delivers the following trace, which takes three time units to execute: $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{tick} s_3 \xrightarrow{c} s_4 \xrightarrow{tick} s_5 \xrightarrow{b_1} s_6 \xrightarrow{tick} s_7 \xrightarrow{b_2} s_8 \xrightarrow{finished} s_9$. It took the state space generator less than three seconds to generate the necessary part of the state space and present a minimal-time trace.

The result is a different one from the one given in [3], but the execution times of the traces are the same. The only difference is due to the freedom to delay after a task is done. Because of this there are several minimal-time traces present in the state space.

6. CREATING THE MODEL FOR THE CCA

For the scheduling problem of the CCA it was not necessary to model all the parts of the machine in a very detailed level. It sufficed to concentrate on a scheduling process which allowed every valid sequence of cycle commands to happen. Invalid sequences would consist of cycles applied to inappropriate cuvettes or cycles applied too soon or too late.

When designing it was important to choose the parameters in a smart way. The more information you store, the bigger the resulting state space will be, therefore any unnecessary information must be avoided. We decided to not use test IDs; to solve the problem we do not need to link an individual sample with some particular reagents. We can assume that the reagent and sample rotors provide the right reagents and samples when required. Furthermore the number of samples and second and third reagents that still need to be added is not needed; it is clear what must be added when looking at the rotor and the number of unprocessed first reagents. That leaves us with the following:

- The cuvette list, consisting of 11 tuples. Each tuple stores which fluids are currently in the corresponding cuvette, which type of test is in the cuvette, and how much time is left before a new fluid may be added.
- How many 1-reagent tests should still be started.
- How many 2-reagent tests should still be started.
- How many 3-reagent tests should still be started.

When modelling it became clear how convenient the use of abstract data types was. The rotor could be modelled using a specially taylored list data type, and we could define functions to quickly check the status of the rotor (e.g. Are there any tests ready to receive a sample, is a certain test finished). This made working with complex data structures much easier than for instance in Uppaal.

We decided to build the model in an incremental way; first we built a model dealing only with 1-reagent tests and 12-cycles. It consists of a single process which has the 12-cycles as actions, together with the necessary guards and recursive calls, placed in alternative composition. The guards are there to check whether a chosen cuvette is indeed ready to receive a certain fluid. Note that it was not necessary to incorporate time in this model, as each action requires a delay of three time units; In such a case a minimal-time trace in a state space is also the shortest trace. Therefore we could do a breadth-first search for the *finished* action.

Using the model in practice though on a number of test batches we found that the freedom to place new tests anywhere on the rotor led to a state space explosion. We decided to build a second model allowing new tests to be placed only in the next empty cuvette, looking counter-clockwise. Since the cranks are placed in such a way that, rotating one cuvette at a time, a sample can be added to a cuvette the moment it reaches the sample crank, this restriction will not lead to a suboptimal solution. In fact, section 7 shows that this is indeed the case, for a test batch of five products.

Next we built a third model with a process using all possible cycles together with the necessary guards, placed in alternative composition. We also used this model to find schedules for different test batches. The results can be found in section 7. After that we created a fourth model, which was much more restricted in its possibilities; we put a strategy in it to cope with a batch of tests. We attached priorities to cycles, so that the model would always execute the enabled cycle with the highest priority. In short the strategy is to always perform as many operations in parallel as possible. Using the same batches of tests as input for this model we got the same results as we got using the strategy-free model (in cases where the complete state space of the latter model could be generated at least). This tells us that the strategy used in the strategy model is a good one for the test batches used.

Recently, the μCRL toolset was expanded with a distributed version of the state space generator. This makes it possible to generate state spaces using a cluster of computers. In this case study it became clear quite soon that an increase of the size of the test batch results in a big growth of the state spaces of most of the models. For some of the test batches a minimal-time trace could not have been found without distributed state space generation.

7. RESULTS

7.1 The scheduling results

Tables 2 and 3 show the results of our case study. A number of different batches were selected and used in the four different models. We used the method described in section 4.2 to get the results of Table 3 and a breadth-first search on the *finished* action in the case of Table 2. The tables should be read as follows: In every row a test batch is specified. In Table 2 the number of tests is displayed, in Table 3 the descriptions are of the form (a, b, c), where a, b and c indicate the number of 1-reagent, 2-reagent and 3-reagent tests, respectively. The results are in the following format: r/s, where r and s equal the number of time units and the number of cycles in the minimal-time trace, respectively. Results obtained by using distributed state space generation, instead of stand-alone state space generation, are marked by a star (*), and where results could not be obtained, due to technical reasons, a hyphen is written (-). Also, the number of states in the different state spaces is given. From the numbers it is clear that the state spaces grow rapidly in size when using bigger test batches. In the models without a strategy this is due to the fact that from every state the system can do any of the valid actions. In the strategy model this is due to the non-determinism concerning adding new tests (more precisely, deciding which test type should be added at which point). One could therefore decide to create another strategy model, which applies a fixed order of tests concerning their type (i.e. first adding 3-reagent tests).

Taking a closer look at the minimal-time traces found we conclude the following: Concerning the 12-cycles models, the minimal-time traces are straight-forward: The first five reagents need to be added without adding a sample, because of the incubation times. After that a reagent can be added together with a sample, until there are no reagents left to add and the final five samples can be added. Having a batch of i products will therefore lead to a minimal-time trace of $i + 5$ cycles, and (since every cycle takes three time units) will take $3.(i + 5)$ time units.

For the more general case, using 12, 16 and 24-cycles, it is more difficult to observe a pattern though. There does not seem to be any advantage gained by adding the reagents for

Table 2: 12-Cycles models search results

Model / # Tests	12-cycles	# States	12-cycles restr.	# States
5	30/10 *	416,352	30/10	447
10	-	-	45/15	9,878
15	-	-	60/20	528,699
20	-	-	75/25	8,403,885
30	-	-	105/35 *	222,613,811

Table 3: All cycles models search results

Model / # Tests	All cycles	# States	Strategy	# States
(3,1,1)	36/11	428	36/11	179
(1,3,1)	39/11	1,588	39/11	229
(1,1,3)	47/13	6,048	47/13	235
(6,2,2)	-	-	51/15	11,477
(3,5,2)	-	-	55/15	29,929
(1,2,7)	-	-	73/17	15,631
(7,4,4)	-	-	75/21 *	5,270,175
(4,8,3)	-	-	77/21 *	1,456,053
(2,5,8)	-	-	91/22 *	1,951,446

the different kinds of tests in a certain order (for instance first adding all the reagents for the 3-reagent tests). Besides that there does not have to be any pattern shared by the particular minimal-time traces found by CADP; it could very well be the case that there are several minimal-time traces coexisting in the same state space. We only get to see one though, which shows a possible solution, not necessarily a mandatory one.

Finally, we also used the optimised search algorithm to find minimal-time traces for the strategy model using five and ten products (in the varying type combinations). We found that the state spaces still needed to be generated almost completely in order to find the solutions. This may indicate the presence of a lot of minimal-time traces and only a few other traces.

7.2 Other findings

Looking at the $(4, 8, 3)$ batch within the strategy model produced some strange results; the state space turned out to be of infinite size. Since this was unexpected we looked at it in more detail and found a trace of infinite size showing that it would be wise to have a cycle which only empties a cuvette, if one wants to allow the scheduler to create any valid schedule. The trace in question will now be presented, where we always indicate the type of the test subjected to an operation, using a superscript i for an i-reagent test. Furthermore, ϵ is the 12-cycle in which no operation at all is executed; basically it is a delay. This is the trace: $[R_1^3(0), R_1^3(1), R_1^3(2), R_1^1(3), R_1^1(4), R_1^1S^3(5), R_1^1S^3(6),$ $R_1^2S^3R_2^3(7,0), R_1^2S^3R_2^1(8,1), R_1^2S^3R_2^1(9,2), R_1^2S^1R_3^3(10,0),$ $S^1R_3^3(0,1), R_1^2R_3^3(3,2), R_1^2(6), S^2(1), R_1^2S^2R_2^2(4,7),$ $S^2R_2^2(2,8), R_1^2R_2^2(5,8), S^2(8), S^2R_2^2(9,3), S^2R_2^2(0,4),$ $S^2R_2^2(10,6), S^2R_2^2(3,5), R_2^2(9), \epsilon, \epsilon, \epsilon, ...]$. In this trace all the cuvets get filled with tests in such a way that there is never a completed test at the emptying position. In the end the rotor is filled entirely with completed tests, but nothing can be removed, because there is no cycle in which only a removal operation is done.

8. CONCLUSIONS

The modelling language μCRL is well-suited for modelling scheduling problems. The data support it has is very convenient when working with complex data structures, as in the case of the CCA. In this regard no changes have to be made to the current μCRL toolset. Furthermore, it suffices to model a single process, which can generate all valid sequences of operations. This applies to scheduling problems in general, since the nature of this kind of problems is to find, within all possible sequences of commands, actions, etc. the minimal-time trace leading to a successful termination.

The number of possible execution sequences can grow very rapidly though. In case of the CCA, we already encountered technical problems concerning the size of the state space when working with 10 products in a test batch. It is possible however to limit the model in certain ways to make this state space smaller. In the case of the CCA, we restricted new tests to be added to the first empty cuvette on the rotor (counter-clock wise) available.

Another way is to build a model with a strategy. By introducing a strategy, the number of possible execution sequences can be brought down a lot, depending on the level of non-determinism still in the model. A strategy model can be used to compare a certain strategy to the general model, and it can serve to determine a practical limit for a state space generated from a general model.

As a side note, we showed an example of gaining results not related to the scheduling problem in question. When generating a state space you may notice some unexpected behaviour, which could lead to more insight into the system.

Finally, we extended the μCRL toolset with a new search algorithm to make on-the-fly searching for a minimal-time

trace in a state space possible. This discards the necessity to generate the complete state space of a system, before being able to search.

It would still be very interesting though, to see if it is possible to come up with other search algorithms. In particular a lot would be gained if it were possible to prune traces from the state space which are not promising. Important to note though, is that we are only interested in getting optimal solutions, not near-optimal ones.

9. FUTURE WORK

We plan to adapt the search algorithm for distributed state space generation. This may be based on the algorithm used in the distributed state space generator of μCRL to search for the shortest trace in a state space. Then we want to implement the distributed search algorithm in the μCRL toolset and see what kind of results we get using this.

We want to see whether or not it is possible in our own setting to develop more efficient search algorithms, as is done in [2], where branch-and-bound algorithms are used. Our breadth-first algorithm resembles branch-and-bound, in the sense that we don't generate solutions that are strictly longer than necessary. However, we have not yet pruned traces that will not lead to a valid solution. In the CCA case study, nearly all traces can be extended to a valid solution.

The results so far can be used to find the best schedule for a given batch of tests. It is an interesting (and much harder) research question, to automatically synthesize an optimal *on-line* scheduler. In that situation, the scheduler should optimally react on the arrival of new jobs.

10. ACKNOWLEDGEMENTS

We thank all members of the TIPSy project meetings and the referees for their constructive comments.

11. REFERENCES

[1] J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing*. EATCS Monograph. Springer, 2002.

[2] G. Behrmann, A. Fehnker, T. Hune, K.G. Larsen, P. Pettersson, and J.M.T. Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In *Proc. TACAS 2001*, volume 2031 of *LNCS*, pages 174–188, 2001.

[3] G. Behrmann, A. Fehnker, T. Hune, K.G. Larsen, P. Pettersson, J.M.T. Romijn, and F. Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In *Proc. HSCC'01*, volume 2034 of *LNCS*, pages 147–161, 2001.

[4] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[5] S. Blom, I. van Langevelde, and B. Lisser. Compressed and distributed file formats for labeled transition systems. In *Proc. PDMC 2003*, volume 89 of *ENTCS*. Elsevier, 2003.

[6] S. Blom and S. Orzan. A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. In *Proc. of PDMC 2002*, volume 68 (4) of *ENTCS*. Elsevier, 2002.

[7] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A Toolset for Analysing Algebraic Specifications. In *Proc. CAV 2001*, volume 2102 of *LNCS*, pages 250–254, 2001.

[8] S.C.C. Blom, N. Ioustinova, and N. Sidorova. Timed verification with μCRL. In *Proc. PSI 2003*, volume 2890 of *LNCS*, pages 178–192, 2003.

[9] W.J. Fokkink, J.F. Groote, J. Pang, B. Badban, and J.C. van de Pol. Verifying a Sliding Window Protocol in μCRL. In *Proc. AMAST 2004*, volume 3116 of *LNCS*, pages 148–163, 2004.

[10] W.J. Fokkink, J.F. Groote, and M. Reniers. Modelling Distributed Systems. Unpublished manuscript, 2002.

[11] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. In *European Association for Software Science and Technology (EASST) Newsletter*, volume 4, pages 13–24, 2002.

[12] J.F. Groote. The Syntax and Semantics of timed μCRL. Technical Report SEN-R9709, CWI, 1997.

[13] P.M.C. Hesen. Design of the Clinical Chemical Analyzer. Technical report, Stan Ackermans Institute, 2000.

[14] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[15] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley-Teubner, Chichester, Stuttgart, 1996.

[16] S.P. Luttik. *Choice Quantification in Process Algebra*. PhD thesis, University of Amsterdam, 2002.

[17] R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, 2003.

[18] P. Niebert, S. Tripakis, and S. Yovine. Minimum-time reachability for timed automata. In *Proc. MED 2000*. IEEE, 2000.

[19] T.C. Ruys. Optimal scheduling using Branch-and-Bound with SPIN 4.0. In *Proc. 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 1–17, 2003.

[20] W.P.C. Spronk. Throughput Analysis of a Clinical Chemical Analyzer. Technical report, TU/e, 1999.

[21] I. Ulidowski and S. Yuen. Extending Process Languages with Time. In *Proc. AMAST'97*, pages 524–538, 1997.

[22] J. Vervoort. Model of a Chemical Analyzer. Technical report, TU/e, 1999.

[23] S. Weber. *Design of Real-Time Supervisory Control Systems*. PhD thesis, TU/e, 2003.

[24] A.J. Wijs and W.J. Fokkink. From $\chi_{\mathtt{t}}$ to μCRL: Combining Performance and Functional Analysis. In *Proc. 10th Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 184–193. IEEE Computer Society Press, 2005.