Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Solving scheduling problems by untimed model checking

A.J. Wijs, J.C. van de Pol, E. Bortnik

REPORT SEN-R0608 MAY 2006

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Solving scheduling problems by untimed model checking

ABSTRACT
In this paper, we show how scheduling problems can be modelled in untimed process algebra, by using special *tick* actions. A minimal-time trace leading to a particular action, is one that minimizes the number of *tick* steps. As a result, we can use any (timed or untimed) model checking tool to find shortest schedules. Instantiating this scheme to μCRL, we profit from a richer specification language than timed model checkers usually offer. Also, we can profit from efficient distributed state space generators. We propose a variant of breadth-first search that visits all states between consecutive *tick* steps, before moving to the next time slice. We experimented with a sequential and a distributed implementation of this algorithm. We also experimented with *beam search*, which visits only parts of the search space, to find near-optimal solutions. Our approach is applied to find optimal schedules for test batches of a realistic *clinical chemical analyser*, which performs several kinds of tests on patient samples.

1998 ACM Computing Classification System: D.2.4 [Formal methods, Model checking], F.2.2 [Sequencing and scheduling]
Keywords and Phrases: Process algebra;scheduling;search algorithms;untimed model checking
Note: This work was carried out under the project SEN2 - TIPSy

Solving Scheduling Problems by Untimed Model Checking
The Clinical Chemical Analyser Case Study

Anton Wijs[1], Jaco van de Pol[1], Elena Bortnik[2]
[1] *CWI, Department of Software Engineering, P.O. Box 94079, 1090 GB Amsterdam*
[2] *Eindhoven University of Technology, Department of Mechanical Engineering, P.O. Box 513, 5600 MB Eindhoven*

ABSTRACT

In this paper, we show how scheduling problems can be modelled in untimed process algebra, by using special *tick* actions. A minimal-time trace leading to a particular action, is one that minimizes the number of *tick* steps. As a result, we can use any (timed or untimed) model checking tool to find shortest schedules. Instantiating this scheme to μCRL, we profit from a richer specification language than timed model checkers usually offer. Also, we can profit from efficient distributed state space generators. We propose a variant of breadth-first search that visits all states between consecutive *tick* steps, before moving to the next time slice. We experimented with a sequential and a distributed implementation of this algorithm. We also experimented with *beam search*, which visits only parts of the search space, to find near-optimal solutions. Our approach is applied to find optimal schedules for test batches of a realistic *clinical chemical analyser*, which performs several kinds of tests on patient samples.

1998 ACM Computing Classification System: D.2.4 [Formal methods, Model checking], F.2.2 [Sequencing and scheduling]
Keywords and Phrases: Process algebra, scheduling, search algorithms, untimed model checking.
Note: This work was carried out under the project SEN2 - TIPSy.

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 [32] while, in a parallel project, the scheduler was developed [29]. Then, the hardware for a CCA mock-up was designed [16]. Currently, a new scheduler is being designed [33]. 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.

Quite some research has been done in the field of timed automata to solve scheduling problems, translated to reachability problems (problems where the goal is to arrive at a certain transition or location). In a paper by Niebert, et al. [22], the problem of minimum-time reachability (i.e. arriving at a certain transition or location in the smallest amount of time possible) 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 straight-forward breadth-first search.

Timed model checkers like UPPAAL [17], a tool using timed automata to model systems, prove to be well suited for handling scheduling problems. We considered, though, the possibility to solve scheduling problems in a simpler, untimed setting. In other words, work with less theory and more brute force. In [27], the model checker SPIN is used for modelling and solving scheduling problems. The depth-first search algorithm 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 decided to use an untimed setting, and adding a notion of time to it using *tick* actions, as described in [8, 30]. With this approach, we were able to reuse existing tools for untimed process algebra. For instance, the modelling language μCRL [15] has a powerful toolset [7], which seemed very usable for our approach. 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. Not a lot of work has been done yet with μCRL in the field of scheduling, so our goal was to develop a general scheduling methodology for μCRL. In this paper we present this methodology.

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. Next, we discuss beam search, which is a method to focus the search on promising parts of the state space, and to prune other parts. 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 by applying the sequential and distributed implementation of our (modified) breadth-first search on these models. We conducted more experiments, applying the implementations of some (modified) beam search variants on the CCA models. Finally, we compare the experimental results and draw conclusions.

To the preliminary version, which appeared in [35], we have now added experiments with a new distributed implementation of the proposed on-the-fly search algorithm. Also, we report on our recent findings to use several variants of beam search, for quickly finding near-optimal solutions.
Comparing the results obtained using these different techniques is done in more detail, for instance we have now included information on the time it took to find solutions, when using sequential state space generation.

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. Note that this is based on the design as given to us by mechanical engineers. Improving the design is regarded outside the scope of this paper.

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 $t_1,\ldots,t_7$ are limited to the following possibilities: $t_1 \geq 15, t_2 \leq 105, 3 \leq t_3 \leq 27, t_4 \leq 105 - t_3, 6 \leq t_5 \leq 21, 9 \leq t_6 \leq 42$,

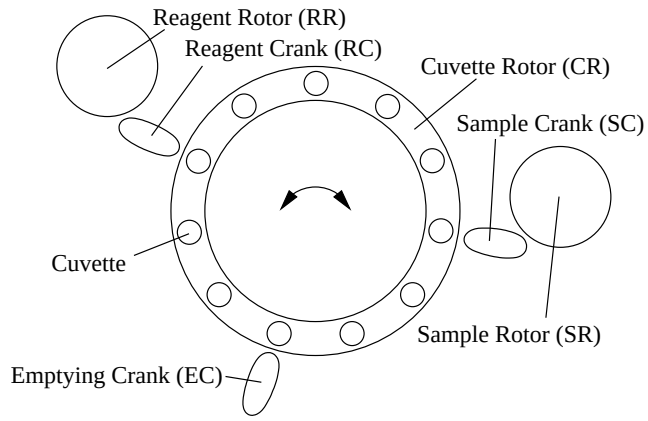

Figure 1: The scaled-down CCA

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	$R_1 \to \Delta t_1 \to S \to \Delta t_5 \to R_2 \to \Delta t_6 \to R_3 \to \Delta t_7 \to E$

$t_7 \leq 105 - t_5 - t_6$.[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 third reagent, this reagent can be added.

In Figure 2 the three types of cycles are visualised. All of them start with round 1, where the available operations (listed using hyphens) can be performed in parallel. After that, in the case of 16 and 24-cycles, a

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

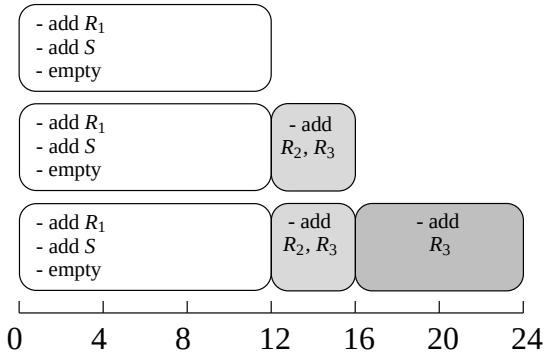

Figure 2: The 12, 16 and 24-cycles

second round is entered. In 24-cycles even a third round appears. This mandatory ordering in rounds means that even in a cycle, in which only a second and/or a third reagent is added, round 1 appears, even though no operation (or only an empty operation) is performed in this round.

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 presented in section 8.5.

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

3. Preliminaries

3.1 The language μCRL

Basically, μCRL is based on the process algebra ACP [4], extended with equational abstract data types [18]. 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[13]. A large number of distributed systems have been verified in μCRL, for instance [10].

We will give a short overview of the language necessary for understanding this paper. For a complete reference, see [15].

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 (the set of actions of a specification is referred to as the set *Act*). 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 [20].
3. The sequential composition operator (.). A process $p.q$ proceeds as p followed by q.
4. The process expression $p \triangleleft b \triangleright q$ where p and q are processes, and b is a data term of sort *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 $n \geq 0$:

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

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 [11].

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

$$init \quad X(d_1, \ldots, d_n)$$

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.

3.2 Labelled transition systems

Labelled transition systems (LTSs) capture the operational behaviour of concurrent systems. An LTS consists of transitions $s \xrightarrow{a} s'$, meaning that being in a state s, an action a can be executed, after which a state s' is reached. Each μCRL specification has a corresponding LTS, defined by the structural operational semantics for μCRL.

Definition 1 (Labelled transition system) *A* labelled transition system *is a tuple (S, Lab, $\longrightarrow$, s_0), where S is a set of states, Lab a set of transition labels, $\longrightarrow \subseteq S \times Lab \times S$ a transition relation, and s_0 the initial state. A transition (s,l,s′) is denoted by $s \xrightarrow{l} s'$.*

In our case, *S* consists of μCRL specifications, and *Lab* consists of actions from $Act \cup \{\tau\}$ parameterized by data.

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 (and on what machines) 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 [14], 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, 30], 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 2 (minimal-time trace) *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. This can be done by creating an abstract model containing one process, which allows all valid executions.[2] By valid executions we mean all executions satisfying the available constraints within the system. So the abstract model 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 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).

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 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 2, we use the *finished* action as transition *a*, in order to formulate a minimal-time trace to a successful 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 equipped 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 incorporate it in the *finished* action, we get *finished(t)*, where *t* is the current value of the counter. Now 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 [21]. Note that in this case there is no real need anymore for *tick* actions when using only one process in the model; after each performed action the counter is raised appropriately to keep track of the execution time (should there be multiple processes in the model, these *tick* actions may be necessary anyhow, for synchronisation purposes).

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

[2] One can decide to use multiple processes in parallel. 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, 34].

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

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 therefore important that, when using the technique described here, the modeller enforces that no infinite traces can occur. Another approach is 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 on-the-fly 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.

Algorithm 1 Pseudo-code algorithm for finding a 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
```

Notice that we do not search traces with cycles. As pointed out in [22], 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. Worst-case the whole state space needs to be generated (when a *finished* action can only be found at the end of 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 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.

Basically, 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. Using such a model has several advantages at the price of losing accuracy, since the answers obtained by using strategy models are near-optimal.

First of all, larger problems can be solved by adding strategies. The found solutions may be suboptimal, because all minimal-time traces may have been pruned away.

Second of all, using a strategy model can make the minimal-time trace detection method from section 4.2 more practical. Note that the solution found by a strategy model is an upperbound for the minimal time. Having such a solution we know how many time units this solution costs, say t. Next, 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. Keep in mind though, that if the value chosen is too small, 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.

4.5 Distributed implementations

As mentioned earlier, recently the μCRL toolset was expanded with a distributed state space generator [5] and a distributed state space reduction tool [6]. This allows the generation of very big state spaces. In order to be able to deal with bigger cases of the CCA scheduling problem, we implemented a distributed version of the on-the-fly search algorithm from section 4.3.

When compared to distributed full state space generation, using the distributed search algorithm allows us to deal with bigger scheduling problems. This is due not only to the fact that we do not need the complete state space anymore, but mainly because the method of section 4.2 has one big practical disadvantage, namely that in order to be able to search for a minimal-time trace, CADP needs one single state space, as opposed to the chunks of a state space obtained from a distributed state space generation. The merging of these chunks into one state space can become very impractical if these chunks together are several Gigabytes big. In other words, even when it is possible to generate a big state space for a given scheduling problem, it may turn out to be unfeasible to obtain a minimal-time trace from it.

We will not display the distributed algorithm here, but it suffices at the moment to mention that it is based on an algorithm which was already present in the distributed state space generator to find the smallest trace to a specific action.

5. PRUNING TECHNIQUES

5.1 Classic beam search

As can be seen later in section 8, the state space grows very rapidly relative to the number of tests. Of course we can deal with bigger state spaces if we move the state space generation from a standalone computer to a cluster of computers, but at some point we are again confronted with the limits of the setup.

Because of this problem, it was interesting to see if we could in some way prune traces which are not really promising from the state space while generating. In that way, we could limit the state space generation to the most interesting part, heavily increasing the speed of finding short solutions, at the expense of finding near-optimal (instead of minimal-time) solutions.

Looking for techniques which could be adapted to our setting, the beam search approach [28, 31] was found. Beam search is a heuristic method for solving combinatorial optimisation problems, which was originally used in the artificial intelligence community for speech recognition [19] and image understanding [26]. Later this technique has been applied to scheduling problems, for example in systems designed for complex job shop environments [12, 25] and for the job shop problem with both makespan and mean tardiness as performance measures [28]. Since then new variants of beam search have been introduced, such as filtered beam search [23, 24] and recovery beam search [9].

The beam search technique is an adaptation of branch-and-bound. Beam search is like breadth-first search, as it progresses level by level, but it does not process all the encountered nodes. At each level of a given search tree all the nodes are evaluated and at most a fixed number of them is selected for further examination. Because of the aggressive pruning the generation time is heavily decreased.

The classic beam search approach is a branch-and-bound technique where only the β most promising nodes at each level of the search tree are selected for further branching. This β is the so-called *beam width*, which is fixed to a value before searching. Other nodes are discarded, so searching can be done relatively quickly. Because of this, using the beam search technique does not guarantee finding an optimal solution, since wrong decisions can be made while pruning. To limit the possibility of wrong decisions one can increase the beam width, at the cost of an increase in computational effort.

Clearly the evaluation function used to select nodes is very important. In the past, two types of evaluation functions have been used: *priority evaluation functions* and *total cost evaluation functions*. A priority evaluation function calculates a priority for each action and selects based on those priorities, while a total cost evaluation function calculates an estimate of the total cost of the best schedule that can be found continuing from the partial schedule represented by the node. In cases where there are more than β actions or nodes, which receive the best evaluation value, a selection is made based on other criteria (depending on the implementation, for instance the order of encountering the actions or nodes). Priority evaluation functions have a local view of the problem, since they only consider the next job to be scheduled, while total cost evaluation functions have a more global view, taking the complete schedule into account. These two types of evaluation functions have led to two classic beam search variants, namely priority and detailed beam search, using a priority evaluation function and a total cost evaluation function, respectively.

Figure 3 shows the application of a beam search on a search tree. The grey nodes are selected using the evaluation function, while the obscured ones are nodes that would have been encountered, had their parents been selected. Typically, this is a detailed beam search as opposed to a priority beam search. In a priority beam search up to β transitions from the root of the tree are followed, after which in each subsequent level of the tree one outgoing transition with the highest priority is selected per examined node. In a detailed beam search however, at each level up to β nodes are selected to continue, regardless of what their parent nodes are, therefore it could be the case, as in level 4 of Figure 3, that some nodes have multiple selected children, while others have none. The reason for this is that one cannot simply compare priorities of actions which are connected to different executions, due to the fact that selection of an action depends on what came before in the execution. A total cost evaluation function does allow comparison of nodes from different executions though, since using such a function allows us to see the progress each execution is making.

The classic beam search approach proved to be very usable within our setting. The state spaces generated by our tool are search trees where the states are the nodes. We were able to incorporate both priority and total cost evaluation functions in the state space generator after having applied some minor changes to the original ideas

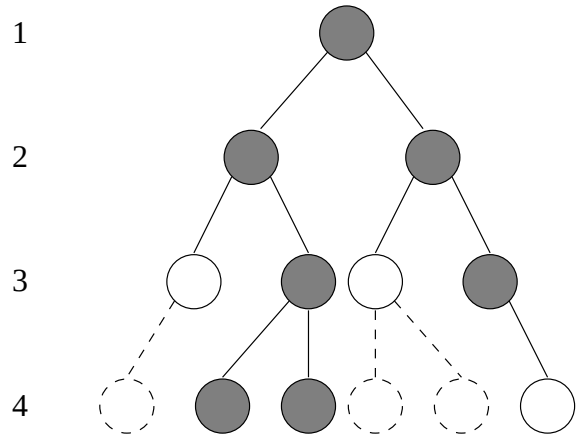

Figure 3: Example of applying a beam search with $\beta = 2$ to a search tree

as presented by, for instance, Valente and Alves [31], among others.

First of all, a user is allowed to supply a list of actions and their priorities. By default, when performing a priority beam search, all actions have priority zero. One can assign higher and lower priorities, which allows setting any order of actions without necessarily giving a complete list of actions. Instead of β, the user supplies the system with (α, l), where, effectively, $\alpha^l = \beta$. Then, when generating, in the first l levels of the state space, per state up to α transitions are selected for further exploration. In the following levels, only one outgoing transition with the highest priority at each state is selected, thereby limiting the total number of transitions at each subsequent level to β. Note the difference with the original notion of priority beam search, where the number of actions per level is fixed much sooner, namely after the evaluation of the actions originating from the root. The decision to change this in our setting is that with our scheduling models we get state spaces which typically have only a few outgoing transitions at the first level, but that number grows rapidly as one progresses into the state space. For this reason it does not make sense to limit the number of selected transitions in the beginning, as that number would be too small. One of the reasons for this rapid growth is that we build the system constraints into the model, allowing only valid traces to appear. This results in only a few outgoing transitions per state early in the state space, but, as we move further, more and more outgoing transitions appear per state. This in contrast with other settings in which beam search has been used, for instance by Valente and Alves [31]. They do not exclude traces, which violate timing constraints, from the search space. The decision to have the user supply (α, l) instead of β was made, to make it possible to postpone fixing the number of selected transitions per level, without the risk of exceeding the given width at some level.

Second of all, a user can perform a detailed beam search, i.e. a beam search using a total cost evaluation function. The user can specify a function at the command line, using constants and variables taken from the parameter list of the model, combining them with addition, subtraction and multiplication. Each encountered state will be evaluated using this function and at each level up to β states, the ones where the evaluation value is the smallest, will be selected for further generation.

5.2 Flexible beam search

While adapting the classic beam search techniques we found that our setting called for slight adaptations of the original notions. In section 5.1 we already mentioned a deviation from the original priority beam search. This deviation was mainly necessary because of the structure of the state spaces.

Another difference between our setting and the settings in which beam search is usually applied, is that our scheduling actions have several parameters. This means that the same action can appear multiple times as an outgoing transition of a given state, each time having different parameter values. This potentially leads to situations where, during selection, a large number of transitions or states have equal evaluations. A selection

then has to be made amongst these equally competent candidates if one of them happens to be the most promising transition or amongst the β-best states. Such cases require making decisions beyond the influence of the evaluation function, which is patently undesired.

Since this is unwanted, we developed a variant of priority beam search called *flexible* priority beam search, the word 'flexible' meaning that the beam width can change while generating, should that be necessary. In flexible priority beam search, in the first l levels (see section 5.1), at each state, up to α most promising outgoing transitions are selected plus any transition which has the same priority as the least competent member of these α transitions has. At level $l+1$ and onwards, at each state, all the transitions which have the same priority as the most promising transition of that particular state are selected (i.e. thinking $\alpha = 1$). In a similar fashion we developed flexible detailed beam search, in which, at each level, up to β most promising states are selected plus any other state which is as competent as the worst member of these β states. This achieves closure on the worst (i.e. highest) total-cost value being selected. Note that in flexible priority beam search, if the beam width is increased, it never returns to the intended β, while the beam width is readjusted to β in each level of flexible detailed beam search.

6. Example: 5 tasks scheduling problem

In order to facilitate comparison, we will look at the small static scheduling problem originally presented in [22] 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. The parameter n is used to encode which actions can be scheduled next. As can be seen, all parameters initially have the value 0.

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

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

$$init\ S(0,0,0,0)$$

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.

7. 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 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. It has to be stressed that we therefore incorporated explicitly the timing constraints as seen in section 2 in the model.

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 very easy.

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 and whether the timing constraints are met. Note that it was not necessary to keep track of the overall execution 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 normal 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 8 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 8. 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 and to get the first reagents of the tests as quickly as possible on the rotor. 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.

8. Experiments and results

8.1 The scheduling results using full state space generation

Table 2 and 3 summarize our findings when applying breadth-first search in a fully generated state space. We used the sequential implementation for the small cases, and switched to the distributed implementation for the larger cases (indicated with *). Table 2 considers the easier case where all test batches consist of a number of 1-reagent tests. In this setting only 12-cycles are needed. In Table 3 all cycles are incorporated. In both cases, we considered the model with and without a built-in strategy.

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. 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 Table 2, in case of the 12-cycles model, the size is increasing so rapidly, that already with 10 tests we had to conclude this would not be promising to continue. The restricted model was sufficient for us to find minimal-time traces for all configurations.

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

In Table 3 are the results we obtained when using models with the three types of tests. When using 10 tests, we were not able to get minimal-time traces anymore using the general model. Although generating the state spaces took a lot of time and effort, it was still possible. The problem was the fact that CADP, which was used to obtain minimal-time traces from the state spaces, needs the chunks of the state space, obtained from a distributed state space generation, to be merged into a single state space, since it only works sequentially at the moment. In the (6,2,2) test batch the resulting state space took about 30 Gigabytes of disk space, and was too big to handle afterwards. In the strategy model the size increase is mainly 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).

Table 3: All cycles models search results

Model / **# Tests**	All cycles	# States	Runtime	Strategy	# States	Runtime
(3,1,1)	36/11	1,148	7.408s	36/11	222	2.640s
(1,3,1)	39/11	5,352	27.498s	39/11	290	2.836s
(1,1,3)	45/12	16,380	1m16.985s	45/12	273	2.840s
(6,2,2)	-	-	-	51/15	11,477	44.919s
(3,5,2)	-	-	-	55/15	29,929	1m56.823s
(1,2,7)	-	-	-	73/17	23,895	1m34.842s
(7,4,4)	-	-	-	75/21 *	5,300,625	-
(4,8,3)	-	-	-	77/21 *	3,959,226	-
(2,5,8)	-	-	-	91/22 *	2,634,395	-

8.2 The scheduling results using on-the-fly searching

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). Table 4 contains the results of these tests. Please note that the number of states in this table cannot be straightforwardly compared to the numbers in Table 2 and 3. This is because for the on-the-fly searching we added the necessary *tick* actions to the model, resulting in more states in the state spaces.

In cases of five products we found that the state spaces still needed to be generated almost completely in order to find the solutions. When moving to bigger test configurations though, the payoff becomes considerate; in the (6,2,2) test batch a minimal-time trace was found halfway through the state space generation.

Table 4: All cycles models on-the-fly search results

Model / **# Tests**	All cycles	# States	Runtime
(3,1,1)	36/11	3,375 (of 4,001)	10.353s
(1,3,1)	39/11	13,194 (of 15,091)	30.482s
(1,1,3)	45/12	34,142 (of 39,132)	1m10.972s
(6,2,2)	51/15 *	341,704,322 (of 677,470,840)	-
(3,5,2)	-	-	-
(1,2,7)	-	-	-
(7,4,4)	-	-	-
(4,8,3)	-	-	-
(2,5,8)	-	-	-

The results of using this algorithm were twofold: on the one hand, we were able to find minimal-time traces with less effort; more specific, since we could find these traces on-the-fly, merging the state space chunks into a single state space and subsequently searching for a specific action using CADP could be avoided. This already saved us a lot of time. On the other hand, it still proved very difficult to get results for bigger test configurations as seen in Table 4. The state space for the (6,2,2) test batch was very big and took hours to generate. It has to be said that, although difficult, getting a minimal-time trace was only possible using on-the-fly searching, since converting the state space after generation and then searching the state space using a model checker would have provided technical difficulties (as mentioned earlier in section 8.1). For bigger test configurations we were unable to find minimal-time traces at the time, since we encountered technical bottlenecks, such as the speed of communication between the computers in the network we used. Other problems stemmed from this particular

case study and model, not from the search algorithm.

8.3 The scheduling results using beam search

Applying detailed and priority beam search to the chemical analyser case study proved very fruitful. It was possible to prune away traces which were not promising very effectively and it turned out to be very interesting to try and see how much could be pruned without removing all optimal solutions. Of course, one can only know if all optimal solutions are pruned if the execution time of these solutions is known. Using previous results (Tables 2, 3 and 4) the beam widths needed to get optimal solutions could be determined. These beam width values provide an indication of how big the beam widths would have to be for even bigger tests. Note that we used a combination here of a detailed beam search with a search from time slice to time slice (as decribed in section 4.3) so we could find minimal-time traces, as opposed to shortest traces.

In Table 5 the results are given of performing a detailed beam search through the state spaces. The evaluation function we used counted the number of fluids that still had to be added to the rotor. Worst case a given partial schedule can always be extended using n cycles, where n is the remaining number of fluids. Note that, in order to use this function, we had to add an extra parameter to the model described in section 7 to keep track of the total number of fluids left.

As can be seen, almost at all times were we able to deal with the test batches using a standalone computer. Notice that in the first number of configurations we were able to provide the number of states in the complete state space, thereby showing how much we could prune. As is shown with the (6,2,2) configuration, the number of pruned states can become considerate, in this particular case more than 99.9% of the state space. Looking at the results, we see that the needed beam width differs from test to test. This makes it hard to predict the needed beam width for bigger test configurations. The bigger you choose the beam width, the higher the probability that the solution found is a minimal-time trace, so when choosing a beam width value one should determine how much time and effort is reasonable to put into finding a solution.

The beam width is not growing in relation to the number of fluids in a test configuration. Probably this is due to the ordering of states while searching. Sometimes the generator is forced to make some selections which are not based on the evaluation values of the states, due to the hard limit of states per level set by the beam width. In those cases the order in which the states are encountered plays a role.

The execution times become very long already when dealing with 10 tests, no doubt because of the evaluation procedure. It seems interesting to try to optimise this procedure in the future, since a lot of time could be gained then.

Table 6 shows us results of performing a priority beam search in combination with the minimal-time trace search algorithm. Again, here we were able to find solutions for most of the test configurations using a standalone computer. As far as the needed beam widths for the different test configurations, similar arguments can be made as the ones for Table 5.

Finally, in Table 7 our experiences with flexible priority beam search are shown. The execution times of searches applied on batches up to 10 tests are very promising, but when dealing with bigger batches we were not able anymore to find a solution using a standalone computer. Since we could not use distributed state space generation together with flexible priority beam search, we were not able to get any numbers for batches containing 15 tests.

The major advantage of flexible beam search is that determining the beam width for each individual configuration is no longer an issue. In all the cases the beam width was initially set to 1, and was increased automatically where needed during exploration.

Note that we have not conducted any tests using flexible detailed beam search. Although we have implemented it in the toolset, we did not think that, in the CCA case study, it would show a much better performance than detailed beam search. More on this is mentioned in section 8.4.

8.4 Comparisons

Taking a closer look at the minimal-time traces found we conclude the following: Concerning the 12-cycles models, the minimal-time traces are straightforward: 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

Table 5: All cycles models detailed beam search results

Model / **# Tests**	All cycles	Needed beam width	# States	Runtime
(3,1,1)	36/11	25	1,461 (of 4,001)	3.428s
(1,3,1)	39/11	41	2,234 (of 15,091)	3.928s
(1,1,3)	45/12	19	1,598 (of 38,276)	3.464s
(6,2,2)	51/15	81	7,408 (of 677,470,840)	7.760s
(3,5,2)	55/15	765	67,470	49.447s
(1,2,7)	73/17	75,000	6,708,705	84m38.405s
(7,4,4)	75/21	35,000	3,801,607	41m1.804s
(4,8,3)	77/21	50,000	5,837,325	85m41.599s
(2,5,8)	-	-	-	-

Table 6: All cycles models priority beam search results

Model / **# Tests**	All cycles	Needed beam width	# States	Runtime
(3,1,1)	36/11	17	668 (of 4,001)	3.220s
(1,3,1)	39/11	29	843 (of 15,091)	3.344s
(1,1,3)	45/12	13	617 (of 38,276)	3.060s
(6,2,2)	51/15	215	9,963 (of 677,470,840)	17.697s
(3,5,2)	55/15	322	15,943	26.918s
(1,2,7)	73/17	44	3,058	5.342s
(7,4,4)	75/21	15,000	883,124	20m43.616s
(4,8,3)	t.b.d.	t.b.d.	t.b.d.	t.b.d.
(2,5,8)	-	-	-	-

Table 7: All cycles models flexible priority beam search results

Model / **# Tests**	All cycles	# States	Runtime
(3,1,1)	36/11	821 (of 4,001)	3.700s
(1,3,1)	39/11	1,133 (of 15,091)	4.060s
(1,1,3)	45/12	1,145 (of 38,276)	4.032s
(6,2,2)	51/15	45,402 (of 677,470,840)	2m33.654s
(3,5,2)	55/15	128,373	6m44.929s
(1,2,7)	73/17	122,449	4m2.939s
(7,4,4)	-	-	-
(4,8,3)	-	-	-
(2,5,8)	-	-	-

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 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 here; 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.

Next we compare the results of the different search techniques used. The first observation is, that when analysing the results of Table 3, the chosen strategy seems to be a good one, at least for the test configurations we used. So it seems to be a good approach to try to put the first reagents of tests as quickly as possible on the rotor and to try to do as much as possible in each cycle.

Table 4 tells us that for the smaller configurations (5 tests) the minimal-time traces present are not much shorter than the longest traces in the state spaces. We get this from the fact that not a big part of each state space was still unexplored when finding a minimal-time trace. An explanation for this may be the fact that with 5 tests, not a lot of freedom is given to the system to do actions, which lead to inefficient traces. When moving to the (6,2,2) configuration, a lot is gained though. Already halfway during state space generation did we encounter a minimal-time trace. This encourages us to believe that the on-the-fly searching method can help more and more with even bigger configurations.

The problem with the on-the-fly searching method of course is that still the amount of states that have to be explored grows rapidly when increasing the number of fluids in a configuration. At this moment we were not able to deal with configurations bigger than (6,2,2), but once the hardware gets improved and our generator gets optimised we will be able to in the future.

When performing a priority beam search, in the CCA case study, it turned out that the generation progresses much faster compared to using a detailed beam search. Furthermore, in most cases, we were able to get better results with smaller beam widths, when compared to using a detailed beam search. It may be that the evaluation function used for the detailed beam search could be improved. We have not investigated that yet. Another reason could be that this particular scheduling problem seems to be solvable by assigning priorities to actions. This was already noticable by the effectiveness of the strategy models. Based on these results we decided for the moment not to perform any tests using flexible detailed beam search. At least, the findings here are in contrast with experiences in other settings, for instance the results found by Valente and Alves [31]. One has to note, though, that due to these different settings, comparisons cannot be easily made.

Solutions are found quicker using beam search than using on-the-fly searching, but of course, when applied to bigger cases for which a minimal-time trace has not been found yet, this is at the expense of finding near-optimal solutions.

Using the flexible priority beam search we found that, with a beam width of 1 and the right priority assignments, the results obtained were the same as the ones using strategy models during the earlier testing. The flexible beam search technique therefore saves the modeller the effort of seperately specifying a strategy model, if such a model is only needed to assign priorities to actions. This is not only convenient, but also removes the possibility of errors or unwanted behaviour, which may appear when writing a strategy model. Besides that, it makes changing a strategy during testing very straightforward. Of course, this comes at a cost; finding a solution using flexible priority beam search took more time than finding the same solution using a strategy model, due to the evaluation procedure.

Compared with the other beam search variants used, we no longer have the problem of determining the beam width for each test batch when using flexible priority beam search.

8.5 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:

$[R_1^3(0), R_1^3(1), R_1^3(2), R_1^1(3), R_1^1(4), R_1^1S^3(5), R_1^1S^3(6), R_1^2S^3R_2^3(7,0), R_1^2S^3R_2^1(8,1), R_1^2S^3R_2^1(9,2), R_1^2S^1R_3^3(10,0),$
$S^1R_3^3(0,1), R_1^2R_3^3(3,2), R_1^2(6), S^2(1), R_1^2S^2R_2^2(4,7), S^2R_2^2(2,8), R_1^2R_2^2(5,8), S^2(8), S^2R_2^2(9,3), S^2R_2^2(0,4),$
$S^2R_2^2(10,6), S^2R_2^2(3,5), R_2^2(9), \varepsilon, \varepsilon, \varepsilon, ...]$.

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.

9. 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. Note that using strategy models does not guarantee finding minimal-time traces, depending on the effectiveness of the strategy chosen.

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, converting it and searching the state space using μ-calculus formulas. Then we wrote a distributed version of this algorithm and incorporated that into the distributed state space generator. In the distributed setting we were able to deal with bigger test configurations and we found that in those cases, bigger parts of the state spaces can be left alone. Several difficulties, some technical and some stemming from the CCA model used, meant that we found the limits of using this approach quite soon. In the future we will look further into removing these difficulties.

We presented several variants of beam search, which is a technique to prune traces from a state space. A lot is gained when it is possible to prune traces from a state space which are not promising. We used both detailed and priority beam search on the CCA models, the latter turning out to be more usable in this particular case study, meaning that using priority beam search smaller beam widths were needed to get similar solutions in shorter execution times. This could be due to the fact, that the CCA scheduling problem seems to be solvable by assigning priorities to actions, as could already be seen by the effectiveness of strategy models. Beam search allows one to make a trade-off between time and effort to spend and the quality of the solutions to find. Having both detailed and priority beam search to work with even increases the possibilities for such a trade-off. If one wants a certain level of quality, however, choosing the right beam width becomes a problem.

Because of this, we proposed an extension of priority beam search, called flexible priority beam search, in which the actual beam width can change while searching, in order to keep track of all actions with a sufficient priority in each level. This extension removes the necessity to create strategy models if they are only needed to assign priorities to actions. Flexible priority beam search combines the ease of use of beam search, meaning that no additional models have to be created to use it, with the flexibility of a strategy model, meaning that

there is no limit to the number of states or transitions explored per level.

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.

10. FUTURE WORK

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.

11. ACKNOWLEDGEMENTS

We thank all members of the TIPSy project meetings for their constructive comments, especially Nikola Trčka for his participation in creating the first design of the CCA μCRL model. Furthermore we thank Bert Lisser for the implementation of the minimal-time trace search algorithm and the help he provided for implementing the beam search algorithms.

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. F. Della Croce and V. T'kindt. A recovering beam search algorithm for the one-machine dynamic total completion time scheduling problem. *Journal of the Operational Research Society*, 53:1275–1280, 2002.
10. 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.
11. W.J. Fokkink, J.F. Groote, and M. Reniers. Modelling Distributed Systems. Unpublished manuscript, 2002.
12. M.S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*. PhD thesis, Carnegie Mellon University, 1983.
13. 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.

14. J.F. Groote. The Syntax and Semantics of timed μCRL. Technical Report SEN-R9709, CWI, 1997.
15. J.F. Groote and M.A. Reniers. *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier, 2001.
16. P.M.C. Hesen. Design of the Clinical Chemical Analyzer. Technical report, Stan Ackermans Institute, 2000.
17. 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.
18. J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley-Teubner, Chichester, Stuttgart, 1996.
19. B.T. Lowerre. *The HARPY speech recognition system*. PhD thesis, Carnegie Mellon University, 1976.
20. S.P. Luttik. *Choice Quantification in Process Algebra*. PhD thesis, University of Amsterdam, 2002.
21. 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.
22. P. Niebert, S. Tripakis, and S. Yovine. Minimum-time reachability for timed automata. In *Proc. MED 2000*. IEEE, 2000.
23. P.S. Ow and E.T. Morton. Filtered beam search in scheduling. *International Journal of Production Research*, 26:35–62, 1988.
24. P.S. Ow and E.T. Morton. The single machine early/tardy problem. *Management Science*, 35:177–191, 1989.
25. P.S. Ow and S.F. Smith. Viewing scheduling as an opportunistic problem-solving process. *Annals of Operations Research*, 12:85–108, 1988.
26. S. Rubin. *The ARGOS Image Understanding System*. PhD thesis, Carnegie Mellon University, 1978.
27. 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.
28. I. Sabuncuoglu and M. Bayiz. Job shop scheduling with beam search. *European Journal of Operational Research*, 118:390–412, 1999.
29. W.P.C. Spronk. Throughput Analysis of a Clinical Chemical Analyzer. Technical report, TU/e, 1999.
30. I. Ulidowski and S. Yuen. Extending Process Languages with Time. In *Proc. AMAST'97*, pages 524–538, 1997.
31. J.M.S. Valente and R.A.F.S. Alves. Beam search algorithms for the early/tardy scheduling problem with release dates. Working Paper 143, Faculdade de Economia do Porto, 2004.
32. J. Vervoort. Model of a Chemical Analyzer. Technical report, TU/e, 1999.
33. S. Weber. *Design of Real-Time Supervisory Control Systems*. PhD thesis, TU/e, 2003.
34. A.J. Wijs and W.J. Fokkink. From $\chi_{\mathbf{t}}$ 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.
35. A.J. Wijs, J.C. van de Pol, and E. Bortnik. Solving Scheduling Problems by Untimed Model Checking, The Clinical Chemical Analyser Case Study. In *Proc. 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'05)*, pages 54–61. ACM Press, 2005.