

Solving scheduling problems by untimed model checking

The clinical chemical analyser case study

Anton J. Wijs · Jaco C. van de Pol · Elena M. Bortnik

Published online: 19 March 2009
© Springer-Verlag 2009

Abstract In this article, we show how scheduling problems can be modelled in untimed process algebra, by using special *tick* actions. A minimal-cost trace leading to a particular action, is one that minimises 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 μ CRL, 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. In addition, we 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.

1 Introduction

In recent years, model checkers have been applied to solving combinatorial optimisation problems, i.e. problems where one of the best combinations of possible values for a given set of variables needs to be found. In particular, scheduling (or planning) problems have been considered, often using a range of available model checkers. Most notably, jobshop scheduling has been dealt with. The jobshop problem is the most classic scheduling problem in the literature. In its most basic form, we have a finite set M of resources, and a number of jobs J_1, \dots, J_n , which compete in using the resources in a specific order and for a finite number of time units. The problem is to allocate the resources such that the jobs are finished in minimal time.

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), e.g. [2, 8, 9, 36, 42]. Some of this work has led to the creation of a model checker focussed on solving this kind of problems, called UPPAAL CORA [9], which is based on UPPAAL [7]. Alternatively, one may use the model checker SPIN [32] to solve scheduling problems specified with the language PROMELA, as Ruys [49] describes, and the μ CRL model checker toolset [13], in combination with the μ CRL process algebra [65]. In this article, we briefly compare these three approaches, before explaining in more detail the latter of the three. Two of the major strengths of the μ CRL toolset are its ability to work with complex data structures, and the availability of powerful algorithms to search state spaces resulting from μ CRL specifications. Both these strengths prove to be critical for dealing with industrial scheduling problems, as is shown in this article by looking at a clinical chemical analyser (CCA), which is an industrial machine with a scheduling problem. Industrial scheduling problems

A. J. Wijs (✉)
INRIA/VASY, 655, Avenue de l'Europe, 38330 Montbonnot
St. Martin, France
e-mail: Anton.Wijs@inria.fr

J. C. van de Pol
Faculty EEMCS, University of Twente, P.O. Box 217,
7500 AE Enschede, The Netherlands
e-mail: vdpol@cs.utwente.nl

E. M. Bortnik
Takumi Technology B.V., Marinus van Meelweg 19a,
5657 EN Eindhoven, The Netherlands
e-mail: elena.m.bortnik@gmail.com

tend to involve a lot of data, something which is not considered in, more theoretical, jobshop problems. Because of this, industrial problems demand much more regarding both the expressiveness of the modelling language used, and the search efficiency of the model checker. Moreover, as it turns out, the CCA problem is unlike typical jobshop or, more general, task graph problems [3,46], in that it has no fixed set of tasks to perform, which implies that there are no fixed dependencies between them, and it incorporates concurrency which cannot be dealt with in an interleaved fashion.

The paper is set up as follows: First, we give an introduction to μCRL . Then, we discuss how scheduling problems can be modelled in general, such that model checkers can be used to solve them, and we explain how this can be done using μCRL . After that, we focus on finding (near-) optimal solutions to scheduling problems by searching state spaces of such problems in a number of ways using the μCRL toolset. Next, these search techniques are compared with techniques available in SPIN and UPPAAL CORA. Finally, we discuss the CCA models we used for the CCA case study, followed by the results obtained by applying the sequential and distributed implementations of our search methods on the resulting state spaces. Finally, we compare the experimental results and draw conclusions.

To the preliminary version, which appeared in [65], we have now added experiments with a new distributed implementation of the proposed on-the-fly search algorithm. Also, we report on our more recent findings to use several variants of beam search, for quickly finding near-optimal solutions. We explain the modelling approach in more detail, and place the work in comparison with techniques available for the model checkers SPIN and UPPAAL CORA.

2 Preliminaries

2.1 The language μCRL

The modelling language μCRL is based on the process algebra ACP [10], extended with so-called equational abstract data types [38]. In order to intertwine processes with data, actions and recursion variables can be parameterised 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* (or *choice*) *quantification* [40] is added to sum over possibly infinite data domains.

The language comes with a toolset [13] that can build a state space from a specification and store it in the .aut format, which can be read by the model checker CADP [26]. Next to that, in order to strive for precision in proofs, an important research area is the use of theorem provers such as PVS [43] to help in finding and checking derivations in μCRL . A large number of distributed systems have been verified in μCRL ,

often with the help of a proof checker or theorem prover, e.g. [5,27].

We will give an overview of the language necessary for understanding this paper. More elaborate explanations can be found e.g. in [28,29,62,66].

A specification starts by defining the necessary data as algebraic data types, consisting of sorts, function declarations, and equations. In fact, the Boolean sort \mathbb{B} 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. We denote actions a, b , etc. appearing in a specification \mathcal{M} as being elements of \mathcal{A} . The process deadlock (δ), which cannot execute itself, nor terminate successfully, and the internal action τ are predefined, with $\tau, \delta \notin \mathcal{A}$. Moreover, it is possible to define communication rules, which state which actions are able to communicate with each other, provided that they have exactly the same parameters.

Processes can be defined by means of *recursive equations*. A recursive equation is of the form $X(x_1 : \mathbb{D}_1, \dots, x_n : \mathbb{D}_n) = t$ for $n \geq 0$, where X is a process name, the x_i are variables and the \mathbb{D}_i are sorts. Moreover, t is a *process term* possibly containing occurrences of expressions $Y(d_1, \dots, d_m)$, where Y is a process name and the d_i are data terms that may contain occurrences of the variables x_1, \dots, x_n . In this rule, $X(x_1, \dots, x_n)$ is declared to have the same behaviour as the process term t . Besides the expressions, a process term may also contain actions. The expressions and actions can be combined using a number of operators. There are four basic operators for creating process terms in μCRL :

1. The alternative composition operator ($+$). A process term $P + Q$ proceeds (non-deterministically) as P or Q (if they can proceed).
2. The sum operator ($\sum_{d:\mathbb{D}} X(d)$), with $X(d)$ a mapping from sort \mathbb{D} to process terms, behaves as $X(d_1) + X(d_2) + \dots$, with $d_1, d_2, \dots \in \mathbb{D}$, i.e. as the possibly infinite choice between $X(d)$ for any data term d taken from \mathbb{D} . This operator is mostly used to describe a process that is reading some input over a data type [40].
3. The sequential composition operator (\cdot). A process term $P \cdot Q$ proceeds as P , which upon successful termination is followed by Q .
4. The process term $P \triangleleft b \triangleright Q$, where $b : \mathbb{B}$, 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 initial state of the specification is declared in a separate section, which is often of the form $X_1(\vec{d}_1) \parallel \dots \parallel X_k(\vec{d}_k)$, where the $X_i(\vec{d}_i)$ are process instantiations and the \vec{d}_i are vectors of data elements of the appropriate sorts.

Furthermore, the parallel composition operator (\parallel) is used here. A process term $P \parallel Q$ executes the actions of P and Q concurrently in an interleaved fashion, and allows the synchronisation of actions according to the provided communication rules. We conclude by noting that we have omitted the use of the renaming, abstraction, and encapsulation operators here, since we do not use these in this article. It suffices to say that the encapsulation operator is used to enforce the synchronisation of actions according to the communication rules.

2.2 Labelled transition systems

Labelled transition systems (LTSs) capture the operational behaviour of concurrent systems. An LTS consists of transitions $s \xrightarrow{\ell} s'$, meaning that being in a state s , an action ℓ 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 An LTS is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I})$, where \mathcal{S} is a set of states, \mathcal{A} a set of transition labels, \mathcal{T} a transition relation, and \mathcal{I} the set of initial states. A transition $(s, \ell, s') \in \mathcal{T}$ is denoted by $s \xrightarrow{\ell} s'$.

In our case, \mathcal{S} consists of μCRL specifications, \mathcal{A} consists of actions from $\mathcal{A} \cup \{\tau\}$ parameterised by data, and the single element of \mathcal{I} is provided by the initialisation section of a μCRL specification. The set of enabled transitions in state s of LTS \mathcal{M} is defined as $en_{\mathcal{M}}(s) = \{t \in \mathcal{T} \mid \exists s' \in \mathcal{S}, \ell \in \mathcal{A}. t = s \xrightarrow{\ell} s'\}$. Whenever $en_{\mathcal{M}}(s) = \emptyset$, we call s a *deadlock* state. We refer to the set of deadlock states as $\mathcal{B} = \{s \mid en_{\mathcal{M}}(s) = \emptyset\}$.

3 Modelling scheduling problems for model checkers

In this section, we discuss some techniques to solve scheduling problems using the μCRL toolset. While doing so, we compare these techniques with approaches for PROMELA (for SPIN) and priced timed automata (for UPPAAL CORA).

A scheduling problem, within the context of this paper, is about processing a certain number of entities (for instance, products or jobs, in the case of jobshop scheduling). The processing is done by a resource, or combination of resources, which can perform tasks¹ $t_1, \dots, t_m \in Ta$, provided that the accompanying sets of constraints C_1, \dots, C_m are met.² Furthermore, each task t_i has an execution time $d(t_i)$ associated with it, given by the function $d : Ta \rightarrow \mathbb{T}$, where \mathbb{T} is a time domain. In these problems, a certain goal should be

reached, usually having completely processed a finite batch of entities. The question asked in scheduling is not only *if* this goal can be reached, but *how efficiently* this can be done.

Over the years, many techniques have been developed to deal with this kind of scheduling problem, for instance by [19]. Certainly it has been shown that model checking can also be applied in this area, e.g. [4, 18, 36, 49, 65]. One could argue, however, whether model checking can compete here with other methods, the majority of which have been used much longer in this area and often specifically optimised to deal with this kind of problems. For instance, there are countless attempts to deal with jobshop scheduling, and when we apply model checking for this, the feared state space explosion problem arises very quickly.

However, a major strength of most model checkers is the expressiveness of their modelling languages. For instance, the language μCRL is a very expressive language and allows the use of abstract data types, by which most useful data structures can be defined. Model checkers are primarily designed to allow the modelling of complex industrial systems, which can then be functionally verified. This expressiveness justifies the use of model checkers for scheduling. In existing scheduling literature, the majority is either aimed at very specific types of scheduling problems, like jobshop scheduling, or an individual case to be scheduled, which usually means that an implemented algorithm to solve the case is directly built into the implementation of the problem. In other words, a general modelling technique is often lacking.

We want to achieve the possibility to model a system and use that one specification to do both functional analysis and scheduling, if so desired. We observe that in order to achieve this, we need to keep in mind that the techniques for scheduling should be applicable on arbitrary LTSs. In scheduling literature, the search space of a scheduling problem often resembles a highly structured tree, where the leaves represent the termination of a possible solution, and every node in level i of a tree with n levels has exactly $n - i$ outgoing edges. An example, where $n = 3$, is displayed in Fig. 1. In the figure, goal nodes are depicted as grey nodes. In an arbitrary LTS, however, there are cycles present, states can have multiple incoming transitions, and paths may end unsuccessfully (i.e. the system deadlocks). In this article, we deal with these more general search spaces.

In [42], 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. A number of algorithms to search these graphs are presented in e.g. [2]. Based on [42], Behrmann et al. [8] consider the model checker UPPAAL, describing how to deal with instances of jobshop scheduling. In [8], linearly priced timed automata are introduced as an extension of timed automata

¹ We denote task labels here as coming from a set Ta .

² To keep things general, we do not fix these constraints to a specific notation here. Suffice it so say that they can involve time and data.

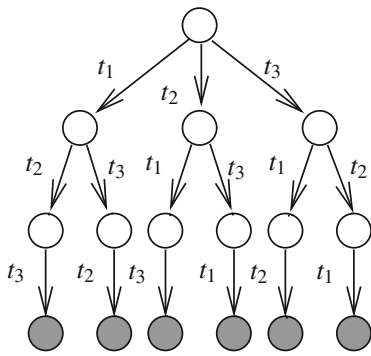


Fig. 1 Search tree for a scheduling problem with tasks t_1, t_2, t_3

with prices on both transitions and locations. They consider the minimum-cost reachability problem. An algorithmic solution is offered, based on a combination of Branch-and-Bound [35] 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 LTS by 90% when compared to a straight-forward breadth-first search. In [9], it is suggested for UPPAAL and UPPAAL CORA to model each job and resource as a timed automaton. Another technique is to model the problem as a single process, as [49] does with PROMELA. We return to these techniques later. The common approach here is to model the system at hand, such that the resulting LTS contains all possibilities to deal with the problem. In such an LTS, the problem is interpreted as a reachability problem, where the question is, in a system where costs are associated with transitions, what the minimal necessary cost is to reach a state $s \in \mathcal{G}$, where $\mathcal{G} \subseteq \mathcal{S}$ is a set of successful termination states (i.e. ‘goal’ states where a complete schedule for the given problem has been achieved). A trace providing this minimal cost then represents a schedule for the problem at hand.

As we perform scheduling using model checking tools, we are able to deal with complex industrial systems, the specifications of which tend to lead to very big, arbitrary LTSs. We model tasks as transitions, meaning that performing task t_i in an execution appears as $s_i \xrightarrow{t_i} s_{i+1}$ in an LTS \mathcal{M} , where s_i and s_{i+1} are two states in the trace corresponding with the execution. In LTSs where the traces represent schedules, we can observe the following.

A function $progress: \mathcal{S} \rightarrow \mathbb{K}$ can be constructed, where \mathbb{K} is some cost domain, which can access the state variables of a state s , using the underlying specification of \mathcal{M} , and quantifies the progress made to reaching some predetermined goal, for instance having completely processed a given batch of entities. In general, say we have $c_0, c_{end} \in \mathbb{K}, \forall s \in \mathcal{S}. c_0 \leq progress(s) \leq c_{end}$ and $\forall s \in \mathcal{S}. progress(s) = c_0$, in other words, c_0 is the initial (no) progress and c_{end} represents having reached the goal. We do not claim any monotonicity of

this function, as in general one can imagine tasks which provide negative progress, leading a schedule further from the goal.

Because of the presence of the *progress* function, we need to refine the description of deadlock states from Sect. 2.2. Now, we need to distinguish deadlock states and successful termination states. We can do this as described in Definition 2.

Definition 2 A state s is a *successful termination state* iff $en_{\mathcal{M}}(s) = \emptyset$ and $progress(s) = c_{end}$. A state s is an *unsuccessful termination state* iff $en_{\mathcal{M}}(s) = \emptyset$ and $progress(s) \neq c_{end}$.

Often, a scheduling problem is modelled such that each goal state is a successful termination state, although one can imagine goal states which are not termination states. In most cases, therefore, \mathcal{G} coincides with the set of successful termination states. In this context, we associate \mathcal{B} with the set of unsuccessful termination states, i.e. $\mathcal{B} = \{s \in \mathcal{S} \mid en_{\mathcal{M}}(s) = \emptyset\} \setminus \mathcal{G}$.

First of all, in order to model a scheduling problem, we need to model some notion of cost. One can create a specific variable for this and make sure that every time an action associated with a task t_i is fired, the value of this variable is raised by $d(t_i)$. This approach has been carried out using SPIN, μ CRL and UPPAAL CORA. Another approach in μ CRL is described next, based on the work by [14,33] and the extension described in [61–63]. Here, a special *tick* action is used, which models time progression. This is comparable with relative discrete time [6]: A *tick* action indicates that the system moves to the next time slice. The duration of an execution now equals the number of *tick* actions occurring in this trace. Of course, instead of time, one can also view *tick* more generally as the progression of cost. Note that this closely relates to delay transitions of timed automata, used in both UPPAAL and UPPAAL CORA, as described by e.g. [7]. Focussing on this latter approach, we can define a *minimal-cost trace* as presented by Definition 3, where, to keep things general, \mathbb{K} is a cost domain, possibly coinciding with \mathbb{T} .

Definition 3 Given an LTS \mathcal{M} and a set of goal states $\mathcal{G} \subseteq \mathcal{S}$, we say that there is a trace with total cost c ($c \in \mathbb{K}$) to \mathcal{G} iff there is a trace in \mathcal{M} starting from a starting state $s_0 \in \mathcal{S}$ and reaching a state $s \in \mathcal{G}$, such that the number of *tick* (or delay) transitions occurring in this trace equals c . We define a trace from \mathcal{S} to \mathcal{G} to be *minimal-cost* if there is no other trace in \mathcal{M} from \mathcal{S} to \mathcal{G} with fewer *tick* (or delay) 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-cost trace to a state in \mathcal{G} , in other words a

state representing success, in an LTS containing all possible schedules as traces.

The general structure of a specification of a scheduling problem in PROMELA, as described in [49], can be described as consisting of a process, which is an alternative composition of all tasks t_i , each followed sequentially by an update of the *cost* variable, in order to indicate the execution time (or cost) of each task. On top of that, the tasks t_i can only be executed if the accompanying conditions C_i are met, written in the specification as conditions for the actions representing the tasks, and, once executed, the task has an effect on the current state of the process (comparable with the function *progress*). Therefore, this model can execute all available tasks as long as the constraints are satisfied. The choices which tasks to execute and when are non-deterministic; there are no built-in priorities. In [49], however, the more general situation, in which unsuccessful termination states, i.e. bad states \mathcal{B} , are present in the LTS, is not considered. We note that it is possible to incorporate bad state detection and avoidance, as is demonstrated in [62]. For this purpose, on the modelling side, a flag *finished* should be raised whenever successful termination is reached.

In UPPAAL CORA, priced timed automata are used to specify a scheduling problem. Here, in general, multiple processes, which synchronise with each other using channels, together express the problem. Recall that a scheduling problem often consists of a set of resources and a set of jobs [9]. A resource process is usually a two-location cyclic process with one local clock. The locations indicate that the resource is either waiting or operating. The resource starts operating whenever a job synchronises over a **start** channel, resetting the clock. The moment a certain use time is reached, the resource moves back to the waiting location and initiates synchronisation over a channel **done**.

A job process is an acyclic sequence of locations, where the initial state represents the start of the job, and the final location, which we call **Finished** here for comparison reasons, indicates that the job is complete. The locations in between represent the acquisition and release of resources. A resource is acquired by achieving synchronisation over the correct **start** channel and setting the use time. It remains in the same location until synchronisation is performed over the **done** channel. The reachability problem is formulated in UPPAAL CORA as the question whether a state can be reached in which all the jobs are in the location **Finished**.

Moving our attention to μCRL , we can create a specification of a scheduling problem as described in this section, in ways very similar to both the PROMELA and the timed automata approach. Like [49], we can often model a scheduling problem in just one process. In μCRL , we model $d(t_i)$ in an action-based manner, using, as mentioned earlier, the special action label *tick*. We present the general form of a μCRL scheduling process in Definition 4.

Definition 4 A μCRL scheduling process equation is a recursive equation of the following form:

$$\begin{aligned} X(d : \mathbb{D}) = & \sum_{i \in I} \sum_{e_i \in \mathbb{D}_i} a_i(f_i(d, e_i)) \cdot \text{tick}^{w_i(d, e_i)} \cdot X_i(g_i(d, e_i)) \\ & \triangleleft h_i(d, e_i) \triangleright \delta + \\ & \text{finished} \cdot \surd \triangleleft \text{progress}(d) = c_{\text{end}} \triangleright \delta \end{aligned}$$

where I is a finite index set, $\mathbb{D}, \mathbb{D}_i, \mathbb{D}_{a_i}, \mathbb{K}$ are sorts, $a_i \in \mathcal{A}$, $a_i : \mathbb{D}_{a_i}$, $\text{tick} : \mathbb{K}$, $f_i : \mathbb{D} \times \mathbb{D}_i \rightarrow \mathbb{D}_{a_i}$, $w_i : \mathbb{D} \times \mathbb{D}_i \rightarrow \mathbb{K}$, $g_i : \mathbb{D} \times \mathbb{D}_i \rightarrow \mathbb{D}$, $h_i : \mathbb{D} \times \mathbb{D}_i \rightarrow \mathbb{B}$, and \surd represents successful termination.

Here, the different states of the process, described by the equation, are represented by the data parameter $d : \mathbb{D}$. Types $\mathbb{D}, \mathbb{D}_i, \mathbb{D}_{a_i}$ may be Cartesian products of data types. The data parameter e_i can influence the parameter of action a_i , its cost w_i , the condition h_i , and the resulting state g_i . Parameter e_i is typically used to let a read action range over a data domain (i.e. choice quantification). Of course, in this equation, actions $a_i(f_i(d, e_i))$ correspond with tasks t_i , conditions $h_i(d, e_i)$ relate to the scheduling conditions C_i , and costs $w_i(d, e_i)$ represent the costs of the tasks. In relation to LTSs with costs, $w_i(d, e_i) = c$ iff transitions with label a_i have cost c . Note the special notation for the *tick* actions, where tick^n denotes a sequence of n *tick* actions.³ Furthermore, we use a special action called *finished* to indicate successful termination (i.e. in $\mathcal{M}, \forall s \in \mathcal{S}. (\exists s' \in \mathcal{S}. s \xrightarrow{\text{finished}} s' \iff s \in \mathcal{G})$). This is mainly necessary to express reachability using the μ -calculus [34, 41] later on. The condition for the successful termination alternative is a direct translation of the *progress* check as explained earlier.

With μCRL , it is moreover possible to specify a scheduling problem in a way very similar to the technique described by e.g. [9] for timed automata. When, for instance, applied on jobshop problem instances, as described earlier in this section, the technique involves mapping each resource and job to an individual process. The feasibility of this technique first of all hinges on synchronisation over the channels **start** and **done**, which can be specified with μCRL using appropriate communication rules and the encapsulation operator. Second of all, synchronisation of timing is essential, i.e. all processes in the specification must agree on the progression of time. This is achievable with μCRL by using e.g. the special operator $|\{\text{tick}\}|$, which is a parallel composition operator which enforces the synchronisation of *tick*-actions of all the processes running in parallel in a system [14, 33]. Because of this, we can directly adopt the same recipe to construct the resource and job processes.

Having created a specification, it is possible, using the appropriate toolset, to generate an LTS from it. This LTS

³ An alternative is to use parameterised *tick* actions [61].

incorporates all possible behaviour of the system described by the specification. Given that there exist successful traces in the LTS, i.e. at least one successful termination state is reachable, somewhere in this LTS there is a minimal-cost trace to a successful finish. Given Definition 3, we use the *finished* action to detect states $s \in \mathcal{G}$, in order to be able to capture in the μ -calculus a minimal-cost trace to a successful termination. In UPPAAL CORA, as previously mentioned, a state $s \in \mathcal{G}$ is identified as a state where all the job processes are in the **Finished** location. When using (state-based) LTL [45] formulas in practice with SPIN, however, it appears we are not able to incorporate the detection of successful termination in the formulas themselves. When following the approach of [49], where the formula is used to bound the search through each trace, incorporating this detection will result in less efficient bounding behaviour, or even the removal of it. The detection can sometimes, however, be performed by other means, while in other cases it can be avoided altogether, at the cost of an increase of the LTS size. For this we refer the reader to [62].

4 Finding optimal schedules

In this and subsequent sections, we describe the search algorithms used for scheduling in the μ CRL toolset and the model checker CADP, and how these relate to techniques available for UPPAAL CORA and SPIN. Here, we consider μ CRL as the input language of CADP, although LOTOS [17] can also be used.

4.1 Iterative searching

The most straightforward technique to search for solutions to a scheduling problem is to iteratively search the LTS using a set of formulas, written in a temporal logic, such as LTL or μ -calculus.

Using the specification of a scheduling problem and the matching toolset, the complete LTS needs to be generated. Next, one needs to formulate, using a temporal logic, the property ϕ that every trace in \mathcal{M} has a cost greater than or equal to $U \in \mathbb{K}$ before reaching successful termination. Here, U is chosen as an upper-bound to the actual minimal cost of reaching successful termination. Given that U is an upper-bound, the model checker will be able to find a counter-example to the property and provide a new, smaller, possibly minimal cost $U' \in \mathbb{K} < U$. Again, now with U' , the property is checked, possibly leading to another counter-example and a new value $U'' \in \mathbb{K} < U'$. This process is repeated until the model checker finds that the property holds, at which point the currently minimal cost is the minimal cost we are looking for and the counter-example given in the previous iteration is one of the minimal-cost traces.

The practical application of this technique differs from toolset to toolset. In CADP, one can use regular, alternation-free μ -calculus to express properties. We need to count the *tick* labels in each trace, in order to determine its cost. In a μ -calculus formula, we are able to differentiate between successful and unsuccessful termination by referring to the action *finished*:⁴

$$\phi = [\neg \text{tick}^*.((T \mid \epsilon).(\neg \text{tick}^*))^{U-1}.\text{finished}] F$$

Since CADP searches LTSs in a breadth-first manner, on average it has to explore a lot more states, compared to SPIN, which searches in a depth-first manner, before it is potentially able to find a counter-example, since it will consider all possible traces at the same time, therefore only reaching \mathcal{G} at a later stage.

This technique works, but is highly inefficient, and therefore quickly becomes unusable for bigger problem instances. The main reason for this is that the entire LTS needs to be generated and searched multiple times, both when property checking can be performed on-the-fly and when it needs to be done after generation. The searching takes up a number of iterations, each time worst-case going over all the states in the LTS. In general, one can say that a depth-first search works more efficiently here than a breadth-first search.

4.2 *g*-Synchronised or minimal-cost search

One way of improving the iterative searching method is the use of Branch-and-Bound (BnB). This, however, is not always applicable, since it requires the possibility of updating the temporal logic formula while searching, as is done in [49]. Another approach is to manipulate the search order in such a way, that the intermediate cumulated costs of all traces can be compared on-the-fly. Approaches like this, however, require that the model checker is extended with new techniques.

The μ CRL toolset has been extended with new generation algorithms. One of these is called *minimal-cost search*, also referred to as *g-synchronised search* [55], as the function $g : \mathcal{S} \rightarrow \mathbb{K}$ is typically used to indicate the (so-called *cumulated*) cost to reach a state s from \mathcal{S} . This function is typically *monotonic*, meaning that it is non-decreasing along a trace, i.e. $\forall s, s' \in \mathcal{S}. s \xrightarrow{\ell} s' \implies g(s') \geq g(s)$. Here, *tick* transitions are used to represent the progress of cost, and other transitions are in fact without cost. Basically, *g-synchronised search* equals *uniform-cost search* [22], where the cost is modelled using additional actions.

⁴ Here it is checked that all traces leading to *finished* do not contain $U - 1$ or fewer *tick* transitions. The $(T \mid \epsilon)$ expression accepts at most one action (including *tick*). Finally, the A^n notation is not a valid μ -calculus expression, but a shorthand for A written n times in sequence.

Algorithm 1 presents this technique, where the LTS is generated as a list of LTS levels \mathcal{L}_i , if $\text{select}(\mathcal{L}_i) = \mathcal{L}_i$ and $\text{selprio}(\text{en}_{\mathcal{M}}(s)) = \text{en}_{\mathcal{M}}(s)$. The functions $\text{select} : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$ and $\text{selprio} : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$ can be used, and will be later on, to select a subset of states from \mathcal{L}_i , and a subset of transitions from $\text{en}_{\mathcal{M}}(s)$, respectively.

Whether a state s is in \mathcal{G} is deduced here by determining whether it is reached via a *finished* transition or not. Besides the levels \mathcal{L}_i , there is a set \mathcal{W} . For all s in the current \mathcal{L}_i to be expanded, a successor s' ends up in \mathcal{W} if $s \xrightarrow{\text{tick}} s'$, and in \mathcal{L}_{i+1} otherwise. When all states in \mathcal{L}_i have been expanded, the search moves to \mathcal{L}_{i+1} . If this level is empty at the start, all states in \mathcal{W} are moved to \mathcal{L}_{i+1} and the searching continues, in other words, the algorithm starts considering states with a greater cumulated cost. The last lines of the algorithm take care of *duplicate detection*. There, it is checked whether a state has been visited before, and if so, it will be ignored.

Algorithm 1 Minimal-cost search with *tick*-encoded costs

Require: $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I})$

Ensure: If exists, a minimal-cost trace to a goal state is returned

```

 $\mathcal{W} \leftarrow \emptyset$ 
 $i \leftarrow 0$ 
 $\mathcal{L}_i \leftarrow \mathcal{I}$ 
 $\mathcal{L}_{i+1} \leftarrow \emptyset$ 
while  $\mathcal{W} \neq \emptyset \vee \mathcal{L}_i \neq \emptyset$  do
   $\mathcal{L}_i \leftarrow \text{select}(\mathcal{L}_i)$ 
  if  $\mathcal{L}_i = \emptyset$  then
     $\mathcal{L}_i \leftarrow \mathcal{W}$ 
     $\mathcal{W} \leftarrow \emptyset$ 
  end if
  for all  $s \in \mathcal{L}_i$  do
    for all  $s \xrightarrow{\ell} s' \in \text{selprio}(\text{en}_{\mathcal{M}}(s))$  do
      if  $\ell = \text{finished}$  then
        return  $\text{GeneratePath}(\{s'\})$ 
      else if  $\ell = \text{tick}$  then
         $\mathcal{W} \leftarrow \mathcal{W} \cup \{s'\}$ 
      else
         $\mathcal{L}_{i+1} \leftarrow \mathcal{L}_{i+1} \cup \{s'\}$ 
      end if
    end for
  end for
   $i \leftarrow i + 1$ 
   $\mathcal{L}_i \leftarrow \mathcal{L}_i \setminus \bigcup_{j=0}^{i-1} \mathcal{L}_j$ 
   $\mathcal{W} \leftarrow \mathcal{W} \setminus \bigcup_{j=0}^{i-1} \mathcal{L}_j$ 
end while
return false

```

Searching with this ordering principle means we know that we find a minimal-cost solution to the problem the first time we find a solution, and can therefore stop immediately. Say there is a state $s \in \mathcal{G}$ in \mathcal{L}_j and another goal state $s' \in \mathcal{L}_k$, with $j \leq k$. If $j = k$, then trivially, $g(s) = g(s')$. If $j < k$, then there are two possible cases for the search from \mathcal{L}_{j+1} to \mathcal{L}_k : Either the states in \mathcal{W} were never moved to any \mathcal{L}_i , with $j < i \leq k$, while searching, in which case $g(s) = g(s')$, or the

states in \mathcal{W} were moved to at least one \mathcal{L}_i , with $j < i \leq k$, while searching, in which case $g(s) < g(s')$. As is shown later in this article, this technique pays off; the bigger the problem instance, the higher the percentage of the LTS that can be skipped entirely.

5 Finding near-optimal schedules

Up to now, we described techniques which guarantee finding an optimal solution. To be able to guarantee this, bounding the search through an LTS needs to be limited to situations where a cost upper-bound has been reached, or, when searching in a smart way, the search must be exhaustive up to the point of reaching a goal state. In practice, however, \mathcal{M} can be very large. One could consider not keeping the expanded states in memory and writing them directly to disk, in cases where the LTS of a scheduling problem resembles a tree.⁵ But even then, although memory is not an issue anymore, searching can take a very long time. In cases where a near-optimal solution practically suffices, one can prevent exhaustive searching.

As remarked in [24], regular breadth-first and depth-first search can be used to return solutions to a problem with costs, but they rarely return an optimal solution. There, it is mentioned that in UPPAAL, breadth-first search quickly runs out of memory, and depth-first search actually returned the worst possible solution when analysing the Sidmar Steel Plant case study. The problem here lies in the fact that both breadth-first and depth-first search do not take cumulated costs into account.

For some problems, e.g. the Traveling Salesman Problem (TSP) [37], the so-called *nearest neighbour heuristic*, or *Gradient Descent*, can provide acceptable solutions [49]. This search selects for every state, which in the case of TSP represents a city, the nearest successor state for further exploration. Since the other successors are discarded, it can only promise to find near-optimal solutions. Although the concept is promising, the search only appears useful for problems where a local view on states, i.e. for each state only considering the next transition to take, suffices. It is our experience that the search seems to be particularly ineffective if the LTS contains unsuccessful traces which initially appear promising.

Another technique, called *beam search*, e.g. [11,44,51], can be seen as an extension of the nearest neighbour heuristic. Here, firstly, the local view can be “broadened” by

⁵ It should be noted that there are techniques known which allow writing states directly to disk even when the LTS does not resemble a tree, e.g. [30] describes a technique where duplicate detection is performed using a so-called Bloom filter. This filter is inquired whenever it needs to be determined whether a state has already been written to disk earlier in the search, or not.

increasing the selection parameter β , and secondly, by using a so-called *estimation* function, the search tries to determine the remaining cost to reach a goal state from the current state, and incorporates this into the selection procedure. For the μ CRL toolset, we extended the main concept of beam search, and a closely related search working with priorities, to work with arbitrary LTSs instead of highly structured search trees. Next, we explain these techniques.

5.1 g-Synchronised beam searches

Beam search is a heuristic search algorithm for combinatorial optimisation problems, which was originally used in the artificial intelligence community [39] for speech recognition, and in [47] for image understanding. Later on, this technique has been applied to scheduling problems, e.g. in [25,50,53], in systems designed for jobshop environments. Since then, new variants of beam search, such as filtered beam search [44,51,52] and recovery beam search [20,57] have been introduced.

In [55], two basic versions of beam search, called *detailed* and *priority* beam search [56], have been extended to work with arbitrary LTSs. These extensions have been implemented in the μ CRL toolset. Here, we briefly explain so-called *g-synchronised detailed beam search* (*g-SDBS*), and *g-synchronised priority beam search* (*g-SPBS*), which are both connected to *g-synchronised* search in Algorithm 1. After that, we describe a technique which again extends these two searches, leading to so-called *flexible* versions.

We describe the concept behind *g-synchronised* beam search inductively. Let \mathcal{L}_{i-1} be the set of states, to be expanded after round $i-1$.⁶ We partition this set into equivalence classes c_0, \dots, c_n , where $n \in \mathbb{N}$, such that $\mathcal{L}_{i-1} = c_0 \cup \dots \cup c_n$ and $\forall s \in \mathcal{L}_{i-1}. s \in c_j \iff g(s) = j$. Essentially, this is what constitutes the *g-synchronisation*. Subsequently, c_k , where $c_k \neq \emptyset \wedge \forall j < k. c_j = \emptyset$, becomes the set of states \mathcal{L}_i after duplicate detection. On \mathcal{L}_i , pruning will be applied next in round i . We differentiate two possibilities for pruning here, one leading to *g-SDBS*, the other to *g-SPBS*.

Algorithm 1 describes *g-SDBS* if we let the *select* function select up to β states, where β , called the *beam width*, is some predetermined element of \mathbb{N} . This selection is typically done using a state-based estimation function $h : \mathcal{S} \rightarrow \mathbb{K}$, which expresses the expected remaining cost to reach a goal state from the current state.⁷

Alternatively, Algorithm 1 describes *g-SPBS* if we let the *selprio* function select up to α outgoing transitions per state in the first l iterations of the search, where both α , called the *sta-*

bilisation level [55], and l , called the *widening factor* [55], are predetermined elements of \mathbb{N} . Note that after l iterations, due to branching in the LTS, approximately α^l states are being expanded. In each subsequent iteration, $\alpha = 1$, thereby avoiding a further increase of the number of selected transitions. This selection, which is action-based, is typically done using a priority function $prio : \mathcal{A} \rightarrow \mathbb{Z}$, which assigns priorities to actions.

Returning to the basic search, according to the pruning, some of the successors of \mathcal{L}_i are visited, constituting the set \mathcal{L}' . The search then continues with $\mathcal{L}_i' = \mathcal{L}' \cup (\mathcal{L}_{i-1} \setminus c_k)$, hence still unexpanded states in c_k are pruned away.

Please note that in *g-SDBS* and *g-SPBS*, once a goal state is found, searching can safely terminate. This is because at a goal state s , $h(s) = 0$ (there is no remaining cost), and since the algorithm always follows traces with minimal g (see Sect. 4.2, and remember that g is monotonic), state s is reached before another state s' iff $g(s) \leq g(s')$.

5.2 Flexible beam searches

In [55,62], a further extension is described to beam search. Note that the searches in Sect. 5.1 are strictly limited to select no more than a fixed upper-bound of states in each round. This can be problematic in situations where e.g. for *g-SDBS* more than β states are promising enough to be selected. Say we have already selected β_1 states in a round, and wish to select another β_2 states, where $\beta = \beta_1 + \beta_2$. Furthermore, say we have n states with minimal h -value in the remaining set of states to select from, with $n > \beta_2$. Now, how should we select no more than β_2 states? This problem is referred to in the literature as *tie-breaking*; a selection needs to be made here based on other criteria, for instance by using a “first-in-first-out” policy, selecting the first of these n states considered. However, these other criteria are generally undesired, since they remove influence from the constructed estimation function. To avoid this, flexible beam searches avoid tie-breaking altogether by selecting, in our example, all n most-promising states. This means that more than β states can be selected in, what is called, *g-synchronised flexible detailed beam search* (*g-SFDBS*), if the selection criterion cannot strictly determine β best states. A similar approach applies on *g-SFPBS*, the flexible priority beam search, where more than α transitions in the first l rounds, and more than 1 transition in all subsequent rounds can be selected, if the *prio* function cannot be used to select no more than α (or 1) transitions.

In practice, we see that this avoidance of tie-breaking can lead to good results, as seen later on for the CCA. One of the reasons for this is that scheduling actions can have several parameters, which often leads to the same action appearing 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

⁶ “Round” i corresponds to a logical (i.e. not necessarily horizontal) level in the LTS, which is processed in the i th iteration of the search.

⁷ More traditional versions of beam search use an evaluation function $f(s) = g(s) + h(s)$ and do not synchronise g -values [55].

transitions or states have equal evaluations. A non-flexible search then needs to make a (often unfortunate) selection from these equally competent candidates if one of them happens to be a transition with sufficient priority or amongst the β -best states.

6 Distributed searches

As mentioned earlier, recently the μ CRL toolset was expanded with a distributed state space generator [12, 15] and a distributed state space reduction tool [16]. These tools allow several workstations to collaborate on generating and analysing LTSs, hence very big LTSs can be processed. We implemented distributed versions of minimal-cost search and the beam search variants covered in Sect. 5 [62, 64].

When compared to distributed full state space generation, using the distributed search algorithms allows us to deal with bigger scheduling problems. This is due not only to the fact that we do not need the complete LTS anymore, but mainly because the method of Sect. 4.1 has one big practical disadvantage, namely that in order to be able to search for a minimal-cost trace, CADP needs one single LTS, as opposed to the chunks of an LTS obtained from a distributed generation. The merging of these chunks into one LTS can become very impractical if these chunks together are several Giga-bytes big. In other words, even when it is possible to generate an LTS for a given scheduling problem, it may turn out to be unfeasible to obtain a minimal-cost trace from it.

We do not display the distributed algorithms here, but it suffices to mention that they are based on an algorithm which was already present in the distributed generator to find the smallest trace to a specific action. The interested reader is referred to [62, 64] for more details. Section 8 presents results obtained by employing distributed minimal-cost search.

7 Connections with SPIN and UPPAAL CORA

Concerning iterative searching, in [49], a depth-first version is explained for SPIN. There, the depth-first search is also extended to include a BnB mechanism, by modifying the LTL formula to be checked 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. In [62], this method is extended to deal with unsuccessful termination. Finally, [49] describes how to use the nearest neighbour heuristic.

UPPAAL CORA has a number of searches, which can help in solving scheduling problems. Uniform-cost search, identified in UPPAAL CORA as *best-first search*, is available to find cost-optimal schedules. In [24], an algorithm to perform BnB on priced timed automata is described, comparable with

depth-first BnB in SPIN, setting a time upper-bound and using the global clock for comparison. For near-optimal searching, the nearest neighbour heuristic is present in UPPAAL CORA as *best depth-first search*, as is a random depth-first search. A search using estimations but no cumulated costs is also present, called *smallest heuristic-first search*, which is in fact *greedy search* [48]. Finally, if an estimation function is provided, UPPAAL CORA automatically incorporates it into its uniform-cost and nearest-neighbour searches, making them comparable with beam search with $\beta = \infty$ and $\beta = 1$, respectively.

8 A clinical chemical analyser

8.1 Introduction

In this section, we describe and analyse an industrial case study of a CCA. The 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 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 in [59], while, in a parallel project, the scheduler was developed [54]. Then, the hardware for a CCA mock-up was designed in [31]. Currently, a new scheduler is being designed [60]. 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.

8.2 Description of the problem

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

Figure 2 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.

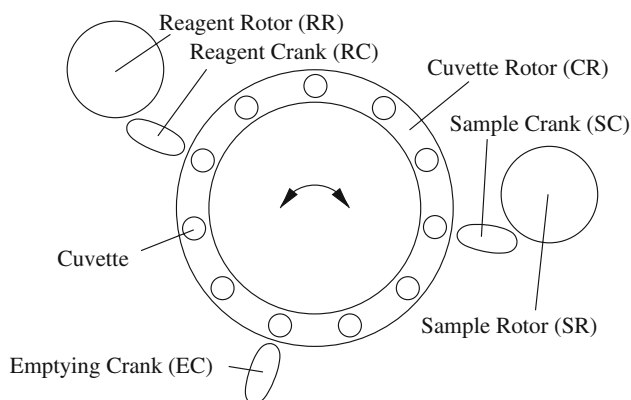


Fig. 2 The scaled-down CCA

Table 1 Recipes for the CCA

Type	Recipe
1-reagent	$R_1 \rightarrow_{\Delta t_1} S \rightarrow_{\Delta t_2} E$
2-reagent	$R_1 \rightarrow_{\Delta t_1} S \rightarrow_{\Delta t_3} R_2 \rightarrow_{\Delta t_4} E$
3-reagent	$R_1 \rightarrow_{\Delta t_1} S \rightarrow_{\Delta t_5} R_2 \rightarrow_{\Delta t_6} R_3 \rightarrow_{\Delta t_7} E$

The use of the machine is to process test recipes. Each available patient sample should be subjected to a test according to one of three possible test recipes. Which recipe to follow is predetermined for each sample.

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, \dots, t_7 are limited to the following possibilities: $t_1 \geq 15, t_2 \leq 105, 3 \leq t_3 \leq 27, t_4 \leq 105 - t_3, 6 \leq t_5 \leq 21, 9 \leq t_6 \leq 42, t_7 \leq 105 - t_5 - t_6$.⁸

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:

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

⁸ A time unit in the scaled-down CCA specification corresponds with a duration of 4 s in the actual CCA.

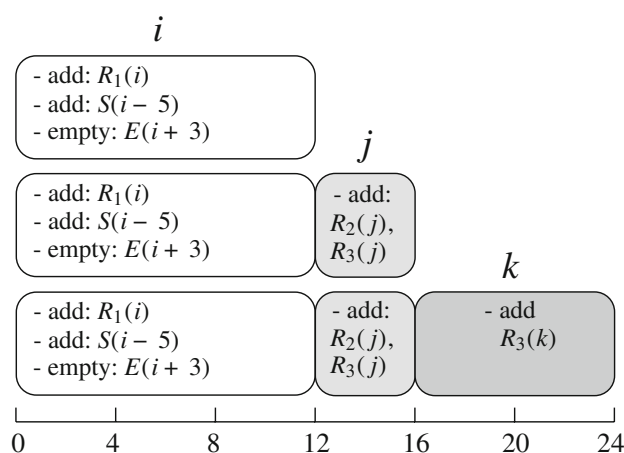


Fig. 3 The 12, 16, and 24-cycles for the CCA

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. Concurrently, if possible, the sample for the test in cuvette $i - 5$ can be added. Finally, also concurrently, 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 Fig. 3, 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, relative to a cuvette i . After that, in the case of 16 and 24-cycles, a second round is entered, adding a second or a third reagent to cuvette j . In 24-cycles, even a third round appears, adding a third reagent to cuvette k . 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 is performed in this round.

The cycles can be named by listing the operations that occur in each round, and the cuvette indices on which they are applied. We do not list the E operations, though, since emptying cuvettes 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. In the 12-cycle $R_1 S(i)$, round 1 involves adding an R_1 reagent to cuvette i , and

concurrently adding a sample to cuvette $i - 5$. When rounds 2 and 3 occur in a cycle, it will always be after having done round 1. For instance, cycle $R_1SR_2(i, j)$ first performs round 1, with an R_1 reagent being added to cuvette i and a sample being added 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 particular Sect. 8.8.

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.

8.3 Creating the specification of the CCA

For the scheduling problem of the CCA, it is not necessary to specify all the parts of the machine at a very detailed level. It suffices to concentrate on a specification incorporating all valid sequences of tasks, as explained in Sect. 3. Invalid sequences would consist of tasks applied to inappropriate cuvettes or tasks applied too soon or too late. It has to be stressed that we therefore incorporate explicitly the timing constraints, as seen in Sect. 8.2, in the specification.

What remains is to identify the tasks here. At first, the operations of the CCA seem to correspond with the tasks of a general scheduling problem; a given batch of tests with their corresponding recipes dictates a fixed number of operations to be executed. However, note that the CCA is a case which incorporates ‘parallel’ behaviour, i.e. in round 1 of a cycle, several components may perform actions concurrently. The CCA is a system for which its schedules of operations cannot be simply represented in an interleaved fashion. This connects to research in planning literature, e.g. [23,58], where such situations are also considered. The main complication here is that the time needed to perform two operations in the same cycle is not the same as performing them in consecutive cycles, while the latter situation is sometimes unavoidable. If concurrent executions of actions are represented in an interleaved fashion in the LTS, then they are indistinguishable from sequential executions. By constructing a specification which allows sequences of *cycles* to be executed, though, we can introduce true concurrency of operations. This, however, is something which cannot so evidently be achieved when modelling each component of the system as a single process. For this reason we choose here the modelling approach shared with the one for SPIN (as in Definition 4), and not for

the one shared with UPPAAL CORA.⁹ Note that if we identify the cycles as the tasks of the problem, we no longer have a fixed number of tasks to execute, nor are there fixed dependencies between them, making the CCA problem unlike typical scheduling problems.

When designing, it is important to choose the parameters in a smart way. The more information you store, the larger the resulting LTS will be, therefore any unnecessary information must be avoided. We decide not to 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 specifying, it becomes clear how convenient the use of abstract data types is. The rotor is specified using a specially tailored list data type, whose elements, representing cuvettes, are again of a special type, which includes a description of its current state (which fluids are present and what the type of the test is) and a timer to indicate the incubation time left. Furthermore, there are functions to quickly check the status of the rotor (e.g. whether there are any tests ready to receive a sample, or whether a certain test is finished). This makes working with complex data structures easier.¹⁰

We decided to build the specification in an incremental way; first, we built a specification 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, conform Definition 4. 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 is not necessary to keep track of the overall execution time in this

⁹ Dierks et al. [21] also deal with true concurrency. Also there, the approach is to model it by means of additional actions, e.g. one may have actions a and b , and the concurrent execution of the two, called ab .

¹⁰ In this article, we avoid the technical details of abstract data types. The interested reader is referred to [28,29].

specification, as each action requires a delay of three time units; in such a case a minimal-cost trace in an LTS is also the shortest trace. Therefore, we can do a normal breadth-first search for the *finished* action.

Using the specification in practice, though, on a number of test batches, we found that the freedom to place new tests anywhere on the rotor leads to a state space explosion. Therefore, we decided to build a second specification 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, Sect. 8.4 shows that this is indeed the case, for a test batch of five products.

Next, we built a third specification with a process using all possible cycles together with the necessary guards, placed in alternative composition. An example of an alternative in this specification is the following, where L is the specially tailored list mentioned earlier, L' is the same list after cycle R_1SR_2 has been fired, and i and j are rotor positions:

$$\sum_{i=0}^{10} \sum_{j=0}^{10} R_1SR_2(i, j) \cdot X(L', R1_{left} - 1, R2_{left}, R3_{left}) \\ \triangleleft readyforR_1(L, i) \wedge readyforS(L, i - 5) \\ \wedge readyforR_2(L, j) \wedge i \neq j \triangleright \delta +$$

We used this specification to find schedules for different test batches. The results can be found in the following section. After that, we created a fourth specification, which is much more restricted in its possibilities; we put a strategy in it to cope with a batch of tests. We attached priorities to cycles, such that the specification will 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 specification, we got the same results as we got using the strategy-free specification (in cases where the latter provided results at least). This tells us that the strategy used in the strategy specification is a good one for the test batches used.

The distributed generator of the μ CRL toolset makes it possible to generate LTSs 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 LTSs of most of the specifications. For some of the test batches a minimal-cost trace cannot be found without distributed state space generation.

Table 2 Exhaustive search results for the CCA with only 12-cycles

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

* Distributed search

8.4 Results using exhaustive state space search

Tables 2 and 3 show our findings when applying exhaustive breadth-first search. All sequential experiments in this article have been performed on a machine with a 64 bit Athlon 2.2 GHz CPU and 1 GB RAM, running SUSE Linux 9.2, using the μ CRL toolset version 2.17.13, while the distributed experiments have been carried out on 16 of these machines. We used the sequential implementation for the small cases, and the distributed implementation for the bigger cases (indicated with an asterisk). Table 2 considers the simpler 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 consider the specification 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-cost trace, respectively. Searches not performed due to technical issues, such as out of memory, are marked with hyphens. Also, the number of states in the different LTSs is given. Finally, the time needed to find the results is given in the format ‘minutes:seconds’.

From the numbers, it is clear that the LTSs grow rapidly in size when using bigger test batches. In the specifications 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 specification, the size is increasing so rapidly, that already with ten tests we had to conclude this would not be promising to continue. The restricted specification was sufficient for us to find minimal-cost traces for all configurations.

Table 3 contains the results we obtained when using specifications with the three types of tests. When using ten tests, we are not able to get minimal-cost traces anymore using the general specification. Although generating the LTSs takes a lot of time and effort, it is still possible. The problem is the fact that CADP, which is used to obtain minimal-cost traces

Table 3 Exhaustive search results for the CCA

Case	Result	All cycles		Strategy all cycles	
		#States	Runtime	#States	Runtime
(3,1,1)	36/11	1,148	00:07.41	222	00:02.64
(1,3,1)	39/11	5,352	00:27.50	290	00:02.84
(1,1,3)	45/12	16,380	01:16.99	273	00:02.84
(6,2,2)	51/15	-	-	11,477	00:44.92
(3,5,2)	55/15	-	-	29,929	01:56.82
(1,2,7)	73/17	-	-	23,895	01:34.84
(7,4,4)	75/21	-	-	5,300,625*	83:48.21
(4,8,3)	77/21	-	-	3,959,283*	63:31.45
(2,5,8)	91/22	-	-	1,951,446*	1897:53

* Distributed search

Table 4 Minimal-cost search results for the CCA

Case	Result	Full LTS	Minimal-cost search	
		#States	#States	Runtime
(3,1,1)	36/11	4,001	3,375	00:10.35
(1,3,1)	39/11	15,091	13,194	00:30.48
(1,1,3)	45/12	39,132	34,142	01:10.97
(6,2,2)	51/15	677,470,840*	341,704,322*	1524:56.00

* Distributed search

from the LTSs, needs the chunks of the LTS, obtained from a distributed state space generation, to be merged into a single LTS, since it only works sequentially at the moment. In the (6,2,2) test batch, the resulting LTS takes about 30 GB of disk space, and is too big to handle afterwards. In the strategy specification, the size increase is mainly due to the non-determinism of adding new tests (more precisely, deciding which test type should be added at which point). One can therefore decide to create another strategy specification, which applies a fixed order of tests concerning their type (i.e. first adding 3-reagent tests).

8.5 Results using on-the-fly searching

We also employed minimal-cost search to find minimal-cost traces for the strategy specification, using five and ten products (in the varying type combinations). Table 4 contains the results of these tests. For comparison reasons, the sizes of the complete LTSs are also displayed. Please note that the number of states in this table cannot be straightforwardly compared to the numbers in Tables 2 and 3. This is because for on-the-fly searching, we added the necessary *tick* actions to the specification, resulting in more states in the LTSs.

In the cases of five products, we find that the LTSs still need to be generated almost completely in order to find the solutions. When moving to bigger test batches though, the payoff becomes considerable; in the (6,2,2) test batch, a solution can be found halfway through the generation.

The results of using minimal-cost search are twofold: on the one hand, we are able to find minimal-cost traces with less effort; more specifically, since we can find these traces on-the-fly, merging the LTS chunks into a single LTS and searching the LTS using CADP can be avoided. On the other hand, it still proves very difficult to get results for bigger test batches, as seen in Table 4. The LTS for the (6,2,2) test batch is very big and takes hours to generate. It has to be said that, although difficult, getting a minimal-cost trace is only possible using on-the-fly searching, due to the difficulties involving CADP mentioned earlier. For bigger test batches, we are currently unable to find minimal-cost traces, since we encounter technical bottlenecks, such as the speed of communication between the computers in the cluster we use. Other problems stem from this particular case study and specification, not from the search algorithm.

8.6 Results using beam search

Applying *g*-SDBS, *g*-SPBS, and flexible variants to the CCA case study proved to be very fruitful. It was possible to prune away traces, which are not promising, very effectively, and it turned out to be very interesting to try and see how much can be pruned without removing all optimal solutions. Of course, one can only know if all optimal solutions are pruned if the total cost of these solutions is known. Using previous results (Tables 2, 3, 4), the beam widths needed to get optimal solutions could be determined for those particular problem instances. These beam width values provide an indication of how big the beam widths will have to be for even bigger instances.

In Table 5, the results are given which are obtained using *g*-SDBS through the LTSs. The estimation function *h* we use counts the number of fluids that still have 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 have to add

Table 5 *g*-SDBS results for the CCA

Case	Result	<i>g</i> -SDBS		
		β	#States	Runtime
(3,1,1)	36/11	25	1,461	00:03.43
(1,3,1)	39/11	41	2,234	00:03.93
(1,1,3)	45/12	19	1,598	00:03.46
(6,2,2)	51/15	81	7,408	00:07.76
(3,5,2)	55/15	765	67,470	00:49.45
(1,2,7)	73/17	75,000	6,708,705	84:38.41
(7,4,4)	75/21	35,000	3,801,607	41:01.80
(4,8,3)	77/21	50,000	5,837,325	85:41.60

Table 6 *g*-S(F)PBS results for the CCA

Case	Result	<i>g</i> -SPBS			<i>g</i> -SFPBS	
		(α, l)	#States	Runtime	#States	Runtime
(3,1,1)	37/12	(1,1)	48	00:03.03	n.a.	n.a.
(3,1,1)	36/11	(2,5)	179	00:03.52	821	00:03.70
(1,3,1)	39/11	(1,1)	50	00:03.08	1,133	00:04.06
(1,1,3)	45/12	(1,1)	57	00:03.08	1,145	00:04.03
(6,2,2)	52/16	(1,1)	67	00:02.63	n.a.	n.a.
(6,2,2)	51/15	(2,9)	479	00:03.06	45,402	02:33.65
(3,5,2)	58/18	(1,1)	74	00:02.65	n.a.	n.a.
(3,5,2)	55/15	(3,13)	4,125	00:13.47	128,373	06:44.93
(1,2,7)	73/17	(1,1)	90	00:02.99	122,449	04:02.94
(7,4,4)	84/30	(1,1)	107	00:03.14	n.a.	n.a.
(7,4,4)	75/21	(3,25)	151,379	08:14.66	20,666,509	872:55.71
(4,8,3)	88/30	(1,1)	112	00:03.14	-	-
(4,8,3)	77/21	(3,25)	148,015	08:28.38	-	-
(2,5,8)	106/32	(1,1)	132	00:05.55	-	-
(2,5,8)	94/25	(3,25)	150,088	09:40.77	-	-

n.a. not applicable

an extra parameter to the specification described in Sect. 8.3, to be able to keep track of the total number of fluids left.

As can be seen, we were always able to deal with the listed test batches using a standalone computer. Notice that these numbers can be compared with the ones in Table 4, therefore in some cases we can see how many states have been pruned. As is shown with the (6,2,2) batch, the number of pruned states can become considerable, in this particular case more than 99.9% of the LTS. 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 larger test batches. The larger you choose the beam width, the higher the probability that the solution found is a minimal-cost 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 batch. Probably this is due to the ordering of

states while searching. Sometimes the generator is forced to perform tie-breaking, 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 runtimes became very long already when dealing with ten 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 the results obtained by performing *g*-SPBS and *g*-SFPBS. Again, here we were able to find solutions for the test batches using a standalone computer. The *prio* function stimulates to perform as many operations in parallel as possible. To facilitate comparison, with *g*-SPBS we searched for solutions for all the test batches with $\alpha, l = 1$, a search which could in fact be called *g*-synchronised heuristic breadth-first search, which has much in common with nearest neighbour heuristic, and, in most cases, with $\alpha, l > 1$. This shows the effect of raising the widening factor and

choosing the stabilisation level further down the LTS. The runtimes of g -SFPBS applied on batches up to ten tests are very promising. The major advantage of g -SFPBS is that determining the beam width for each individual batch is no longer an issue. In all the cases, initially $\alpha^l = 1$, and α is increased automatically where needed during exploration. When dealing with batches bigger than ten tests, we see that the runtime and the number of states rapidly increase. This expresses the drawback of a flexible search: it avoids tie-breaking, as mentioned already several times, but the result of this is that the space and computation time requirements are no longer linear to the maximum search depth.

Note that we did not conduct any tests using g -SFDBS. Although we have implemented it in the toolset, we did not think that, in the CCA case study, it will show a much better performance than g -SDBS. More on this is mentioned in Sect. 8.7.

8.7 Comparisons

Taking a closer look at the minimal-cost traces found, we conclude the following: Concerning the 12-cycles specifications, the minimal-cost 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 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-cost trace of $i + 5$ cycles, which will take $3 \times (i + 5)$ time units, since every cycle takes three 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-cost traces found here; it could very well be the case that there are several minimal-cost traces coexisting in the same LTS. 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. Therefore, 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-cost traces present are not much shorter than the longest traces in the LTSs. We get this from the fact that only a small part of each LTS is left unexplored when finding a minimal-cost trace. An explanation for this may be the fact that with five 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 through the LTS search do we encounter a minimal-cost 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 batch. At this moment, we are not able to deal with batches 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 using g -SPBS, it turns out that the search progresses much faster compared to using g -SDBS. Furthermore, in all cases, we are able to find the optimal solutions with smaller beam widths. It shows that the evaluation function used for g -SDBS can be improved. We have not tried to improve the total-cost evaluation function yet. It turns out that this particular scheduling problem is well solvable by assigning priorities to actions. This is already noticeable by the effectiveness of the strategy specifications. Based on these results, we decided not to perform any tests using g -SFDBS, but in other cases, this search has been very successful [55,64].

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-cost trace has not been found yet, this is at the expense of finding near-optimal solutions.

Using g -SFPBS, we find that, with $\alpha^l = 1$ and the right priority assignments, the obtained results are the same as the ones obtained from the strategy specification during the earlier testing. The flexible beam search technique, therefore, saves the user the effort of separately specifying a specification with a built-in strategy, if such a specification is only needed to place an ordering on actions. This is not only convenient, but also removes the possibility of errors or unwanted behaviour, which may appear when writing a specification with a strategy. Besides that, it makes changing a strategy during testing very straightforward. Of course, this comes at a cost; finding a solution using g -SFPBS takes more time than finding the same solution using a specification with a built-in strategy, due to the evaluation procedure. Compared to the other beam search variants used, we no longer have the problem of determining the beam width for each test batch when using flexible beam search.

8.8 Other findings

Looking at the (4,8,3) batch within the strategy specification initially produced some strange results; the LTS turned out to be of infinite size. Since this is 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 exclude the possibility for the scheduler to create an invalid schedule. The trace in question will now be presented, where we 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

$$\begin{aligned} &[R_1^3(0), R_1^3(1), R_1^3(2), R_1^1(3), R_1^1(4), R_1^1 S^3(5), R_1^1 S^3(6), \\ &R_1^2 S^3 R_2^3(7, 0), R_1^2 S^3 R_2^1(8, 1), R_1^2 S^3 R_2^1(9, 2), \\ &R_1^2 S^1 R_3^3(10, 0), S^1 R_3^3(0, 1), R_1^2 R_3^3(3, 2), R_1^2(6), \\ &S^2(1), R_1^2 S^2 R_2^2(4, 7), \\ &S^2 R_2^2(2, 8), R_1^2 R_2^2(5, 8), S^2(8), S^2 R_2^2(9, 3), S^2 R_2^2(0, 4), \\ &S^2 R_2^2(10, 6), S^2 R_2^2(3, 5), \\ &R_2^2(9), \epsilon, \epsilon, \epsilon, \dots] \end{aligned}$$

In this trace, all the cuvettes 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, both as is common in UPPAAL CORA and SPIN. The data support it has is very convenient when working with complex data structures. In this regard, the μCRL toolset did not have to be changed. In other regards, it had to be extended with searches other than breadth-first search.

The number of possibilities in an LTS can grow very rapidly though, when increasing the size of the problem instance. We already encountered technical problems concerning the size of the LTS when working with ten tests in a test batch. It is possible, however, to limit the specification in certain ways to make this LTS smaller. We restricted new tests to be added to the first available empty cuvette on the rotor (counter-clock wise).

Another way is to build a specification with a built-in 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 specification. A specification with a built-in strategy can be used to compare a certain strategy with the general specification. Besides that, it can serve to determine an upper-bound for a bounded search through an LTS of a general specification. Note that using a specification with a built-in strategy does not guarantee finding minimal-cost traces, depending on the effectiveness of the strategy chosen.

In a distributed setting, we are able to deal with bigger problem instances. When performing minimal-cost search

in this setting, we found that the larger the LTS, the higher the percentage of states which can be avoided in the search.

We used both $g\text{-SDBS}$ and $g\text{-SPBS}$, the latter turning out to be more effective in this particular case study, meaning that smaller beam widths are needed to get similar solutions in shorter runtimes. This can be due to the fact that the CCA scheduling problem seems to be well solvable by assigning priorities to actions, as can already be seen by the effectiveness of specifications with a built-in strategy. Beam search allows one to make a trade-off between computation time 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 introduced flexible beam search, in which the actual beam width can change while searching, in order to keep track of all actions and states which appear promising enough (i.e. avoiding tie-breaking). The experiments suggest that from case to case, the beam widths of flexible beam searches do not have to be increased. The experimental results suggest that $g\text{-SFPBS}$ removes the necessity to create additional specifications with built-in strategies, if they are only needed to assign priorities to actions. $g\text{-SFPBS}$ combines the ease of use of beam search, meaning that no additional specifications have to be created to use it, with the flexibility of a specification with a built-in strategy, meaning that there is no limit to the number of states or transitions expanded per level. The major benefits of flexible beam searches are the relative ‘stability’ of the beam widths (i.e. when increasing the size of the test batch, the beam width can be left unchanged) and the avoidance of tie-breaking, but this comes at a price, namely that the space and computation time requirements of these searches are not linear to the maximum search depth.

As a side note, in Sect. 8.8, we showed an example of gaining results not related to the scheduling problem in question. When generating an LTS you may notice some unexpected behaviour, which could lead to more insight into the functional quality of the system.

10 Future work

- Up to now, we have considered fixed batches of tests for the CCA. A next step would be to try to synthesise an online scheduler, which provides a strategy for the CCA in general, independent of the test input. This is related to the work in e.g. [1], where uncertainty due to the actions of the environment is incorporated into the problem.
- Another generalisation is to consider problems with multiple cost variables, such as money, time, etc., as is e.g. expressed in [9]. We believe we can deal with such

problems by 'stacking' synchronisations, e.g. with two cost variables, we would, in each round of the search, first synchronise states on g , then on some other cost function, and finally make a selection of states to expand.

Acknowledgments 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 μ CRL model. We thank Bert Lissner for the implementation of the (distributed) minimal-cost search algorithm and the help he provided for implementing the beam search algorithms. Finally, we thank Mohammad Torabi Dashti and the anonymous reviewers for their constructive comments.

References

- Abdeddaïm, Y., Asarin, E., Maler, O.: On optimal scheduling under uncertainty. In: Proceedings of TACAS 2003. LNCS, vol. 2619, pp. 240–253. Springer, Berlin (2003)
- Abdeddaïm, Y., Asarin, E., Maler, O.: scheduling with timed automata. *Theor. Comput. Sci.* **354**(2), 272–300 (2006)
- Abdeddaïm, Y., Kerbaa, A., Maler, O.: Task graph scheduling using timed automata. In: Proceedings of FMPPTA 2003 (2003)
- Abdeddaïm, Y., Maler, O.: Preemptive job-shop scheduling using stopwatch automata. In: Proceedings of TACAS 2002. LNCS, vol. 2280, pp. 113–126. Springer, Berlin (2002)
- Badban, B., Fokkink, W.J., Groote, J.F., Pang, J., van de Pol, J.C.: Verifying a sliding window protocol in μ CRL and PVS. *Formal Aspects Comput.* **17**(3), 342–388 (2005)
- Baeten, J.C.M., Middelburg, C.A.: *Process Algebra with Timing*. EATCS Monograph. Springer, Berlin (2002)
- Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Proceedings of SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Berlin (2004)
- Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J.M.T.: Efficient guiding towards cost-optimality in Uppaal. In: Proceedings of TACAS 2001. LNCS, vol. 2031, pp. 174–188. Springer, Berlin (2001)
- Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.* **32**(4), 34–40 (2005)
- Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. *Inform. Control* **60**(1–3), 109–137 (1984)
- Bisiani, R.: Beam search. In: *Encyclopedia of Artificial Intelligence*, pp. 1467–1568. Wiley, New York (1992)
- Blom, S.C.C., Calamé, J.R., Lissner, B., Orzan, S., Pang, J., van de Pol, J.C., Torabi Dashti, M., Wijs, A.J.: Distributed analysis with μ CRL: a compendium of case studies. In: Proceedings of TACAS 2007. LNCS, vol. 4424, pp. 683–689. Springer, Berlin (2007)
- Blom, S.C.C., Fokkink, W.J., Groote, J.F., van Langevelde, I., Lissner, B., van de Pol, J.C.: μ CRL: A toolset for analysing algebraic specifications. In: Proceedings of CAV 2001. LNCS, vol. 2102, pp. 250–254. Springer, Berlin (2001)
- Blom, S.C.C., Ioustinova, N., Sidorova, N.: Timed verification with μ CRL. In: Proceedings of PSI 2003. LNCS, vol. 2890, pp. 178–192. Springer, Berlin (2003)
- Blom, S.C.C., van Langevelde, I., Lissner, B.: Compressed and distributed file formats for labeled transition systems. In: Proceedings of PDMC 2003. ENTCS, vol. 89. Elsevier, Amsterdam (2003)
- Blom, S.C.C., Orzan, S.: Distributed state space minimization. *Int. J. Softw. Tools. Technol. Transf.* **7**(3), 280–291 (2005)
- Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. *Comput. Netw.* **14**(1), 25–59 (1987)
- Bozga, M., Kerbaa, A., Maler, O.: Scheduling acyclic branching programs on parallel machines. In: Proceedings of RTSS 2004, pp. 208–215. IEEE Computer Society Press, Washington, DC (2004)
- Brucker, P., Jurisch, B., Sievers, B.: A branch and bound algorithm for the job-shop scheduling problem. *Discrete Appl. Math.* **49**(6), 107–127 (1994)
- Della Croce, F., T'kindt, V.: A recovering beam search algorithm for the one-machine dynamic total completion time scheduling problem. *J. Oper. Res. Soc.* **53**(11), 1275–1280 (2002)
- Dierks, H., Behrmann, G., Larsen, K.G.: Solving planning problems using real-time model checking (translating PDDL3 into timed automata). In: Proceedings of Workshop on Planning via Model Checking 2002, pp. 30–39 (2002)
- Dijkstra, E.W.: A note on two problems in connection with graphs. *Numer. Math.* **1**, 269–271 (1959)
- Edelkamp, S.: Taming numbers and duration in the model checking integrated planning system. *J. Artif. Intell. Res.* **20**, 195–238 (2003)
- Fehnker, A.: Citius, vilius, melius: guiding and cost-optimality in model checking. PhD thesis, Radboud University of Nijmegen (2002)
- Fox, M.S.: *Constraint-Directed Search: A Case Study Of Job-Shop Scheduling*. PhD thesis, Carnegie-Mellon University, Pittsburgh (1983)
- Garavel, H., Lang, F., Mateescu, R.: An overview of CADP 2001. In: *European Association for Software Science and Technology (EASST) Newsletter*, vol. 4, pp. 13–24 (2002)
- Groote, J.F., Monin, F., van de Pol, J.C.: Checking verifications of protocols and distributed systems by computer. In: Proceedings of CONCUR 1998. LNCS, vol. 1466, pp. 629–655. Springer, Berlin (1998)
- Groote, J.F., Ponse, A.: The Syntax and Semantics of μ CRL. In: Proceedings of ACP 1994, Workshops in Computing Series, pp. 26–62. Springer, Berlin (1995)
- Groote, J.F., Reniers, M.A.: Algebraic Process Verification. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) *Handbook of Process Algebra*, chap. 17, pp. 1151–1208. Elsevier, Amsterdam (2001)
- Hammer, M., Weber, M.: "to Store or Not to Store" reloaded: reclaiming memory on demand. In: Proceedings of FMICS 2006. LNCS, vol. 4346, pp. 51–66. Springer, Berlin (2006)
- Hesen, P.M.C.: Design of the Clinical Chemical Analyzer. Technical report. Stan Ackermans Institute, Eindhoven (2000)
- Holzmann, G.J.: *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Reading (2004)
- Ioustinova, N.: Abstractions and Static Analysis for Verifying Reactive Systems. PhD thesis, Vrije Universiteit Amsterdam (2004)
- Kozen, D.: Results on the propositional μ -calculus. *Theor. Comput. Sci.* **27**(3), 333–354 (1983)
- Kumar, V.: Branch-and-bound search. In: Shapiro, S. (ed.) *Encyclopedia of Artificial Intelligence*, pp. 1468–1472. Wiley, New York (1992)
- Larsen, K.G.: Resource-efficient scheduling for real time systems. In: Proceedings of EMSOFT 2003. LNCS, vol. 2855, pp. 16–19 (2003)
- Lawler, E., Lenstra, J.K., Rinnooy Kan, A., Shmoys, D. (eds.): *The Traveling Salesman Problem*. Wiley, New York (1985)
- Loeckx, J., Ehrich, H.-D., Wolf, M.: *Specification of Abstract Data Types*. Wiley, New York (1996)
- Lowerre, B.T.: *The HARPY Speech Recognition System*. PhD thesis, Carnegie-Mellon University, Pittsburgh (1976)
- Luttik, S.P.: *Choice Quantification in Process Algebra*. PhD thesis, University of Amsterdam (2002)

41. Mateescu, R., Sighireanu, M.: Efficient On-the-fly model-checking for regular alternation-free Mu-calculus. *Sci. Comput. Program.* **46**(3), 255–281 (2003)
42. Niebert, P., Tripakis, S., Yovine, S.: Minimum-time reachability for timed automata. In: *Proceedings of MED 2000*. IEEE Computer Society Press, Washington, DC (2000)
43. Owre, S., Rushby, J.M., Shankar, N.: PVS: a Prototype Verification System. In: *Proceedings of CADE 1992*. LNCS, vol. 607, pp. 748–752. Springer, Berlin (1992)
44. Pinedo, M.: *Scheduling: theory, algorithms, and systems*. Prentice-Hall, Englewood Cliffs (1995)
45. Pnueli, A.: The temporal semantics of concurrent programs. *Theor. Comput. Sci.* **13**(1), 45–60 (1981)
46. Rasmussen, J.I., Larsen, K.G., Subramani, K.: Resource-optimal scheduling using priced timed automata. In: *Proceedings of TACAS 2004*. LNCS, vol. 2988, pp. 220–235. Springer, Berlin (2004)
47. Rubin, S.: The ARGOS image understanding system. PhD thesis, Carnegie-Mellon University, Pittsburgh (1978)
48. Russell, S., Norvig, P.: *Artificial intelligence: A modern approach*. Prentice-Hall, Englewood Cliffs (1995)
49. Ruys, T.C.: Optimal scheduling using Branch-and-Bound with SPIN 4.0. In: *Proceedings of SPIN 2003*. LNCS, vol. 2648, pp. 1–17. Springer, Berlin (2003)
50. Sabuncuoglu, I., Bayiz, M.: Job shop scheduling with beam search. *Eur. J. Oper. Res.* **118**(2), 390–412 (1999)
51. Si Ow, P., Morton, E.T.: Filtered beam search in scheduling. *Int. J. Prod. Res.* **26**(1), 35–62 (1988)
52. Si Ow, P., Morton, E.T.: The single machine early/tardy problem. *Manage. Sci.* **35**(2), 177–191 (1989)
53. Si Ow, P., Smith, S.F.: Viewing scheduling as an opportunistic problem-solving process. *Ann. Oper. Res.* **12**(1–4), 85–108 (1988)
54. Spronk, W.P.C.: Throughput analysis of a clinical chemical analyzer. Master's thesis, Technical University of Eindhoven (1999)
55. Torabi Dashti, M., Wijs, A.J.: Pruning state spaces with extended beam search. In: *Proceedings of ATVA 2007*. LNCS, vol. 4762, pp. 543–552. Springer, Berlin (2007)
56. Valente, J.M.S., Alves, R.A.F.S.: Beam search algorithms for the single machine total weighted tardiness scheduling problem with sequence-dependent setups. Working Paper 186, Faculdade de Economia do Porto (2005)
57. Valente, J.M.S., Alves, R.A.F.S.: Filtered and recovering beam search algorithms for the early/tardy scheduling problem with no idle time. *Comput. Ind. Eng.* **48**(2), 363–375 (2005)
58. Veloso, M.M., Pérez, M.A., Carbonell, J.G.: Nonlinear planning with parallel resource allocation. In: *Proceedings Innovative Approaches to Planning, Scheduling and Control*, pp. 207–212 (1991)
59. Vervoort, J.: Model of a Chemical Analyzer. WPA Report 420216, Technical University of Eindhoven, Eindhoven (1999)
60. Weber, S.: Design of Real-Time Supervisory Control Systems. PhD thesis, Technical University of Eindhoven, Eindhoven (2003)
61. Wijs, A.J.: Achieving discrete relative timing with untimed process algebra. In: *Proceedings of ICECCS 2007*, pp. 35–44. IEEE Computer Society Press, Washington, DC (2007)
62. Wijs, A.J.: What to do next? Analysing and optimising system behaviour in time. PhD thesis, Vrije Universiteit Amsterdam (2007)
63. Wijs, A.J., Fokkink, W.J.: From χ_t to μ CRL: combining performance and functional analysis. In: *Proceedings of ICECCS 2005*, pp. 184–193. IEEE Computer Society Press, Washington, DC (2005)
64. Wijs, A.J., Lissers, B.: Distributed extended beam search for quantitative model checking. In: *Proceedings of MoChArt 2006*. LNAI, vol. 4428, pp. 165–182. Springer, Berlin (2007)
65. Wijs, A.J., van de Pol, J.C., Bortnik, E.: Solving scheduling problems by untimed model checking—the clinical chemical analyser case study. In: *Proceedings of FMICS 2005*, pp. 54–61. ACM Press, New York (2005)
66. Wouters, A.G.: Manual for the μ CRL tool set. Technical Report SEN-R0130, CWI (2001)