A DSL for the extended Job-Shop Scheduling Problem

Technical University of Denmark
Department of Informatics and Mathematical Modelling

Jakob Jakobsen Boysen
s112344

Sune Mølgaard Laursen
s082583

ABSTRACT

Many solvers and Domain Specific Languages focuses on very simple Job-Shop Scheduling problems.

When more advanced scheduling functionality is needed one often have to resort to expensive software suites.

We propose a Domain Specific Language with rich functionality that translates scheduling problems into constraints and uses the state-of-the-art SMT-solver Z3 for solving.

1. INTRODUCTION

In this paper we describe the widely known NP-hard Job-Shop Scheduling Problem (JSSP) and propose a domain specific language (DSL) that can be used to specify instances of JSSP in a precise and readable manner.

We provide a parser for the DSL and a problem solver based on the SMT-solver Z3 by Microsoft Research. One of the goals of the DSL is to abstract away from the rather advanced constraint generation to a simpler more intuitive level. The DSL can easily be used for smaller scheduling problems or act as a mid-end for more advanced problem generation.

The problem solver and the parser for the DSL are implemented in the functional programming language F# using the tools `FsLex` as lexer and `FsYacc` as parser generator. The parser generation phase is out of the scope of this paper, which is why we focus on the language design of the DSL in Section 2, how constraints are generated in Section 7.7 and how problems are solved using Z3 in Section 3.

2. THE JOB-SHOP SCHEDULING PROBLEM

In the JSSP we have a number of *jobs* each consisting of a number of *operations*, also called *tasks*, with a specified *processing time*. An operation is done on a specific *machine*. In general literature engaged in the JSSP, e.g. [8][7][10], are most often occupied with the simple version of the problem with the following constraints:

- Only one job on one machine at a time, i.e. if we consider a machine a *resource*, only *unary* resources are supported.
- No pre-emption.
- Each job can only have one operation processed at a time.
- Operations on jobs are scheduled in predetermined given order.

The time required for all jobs to complete all of their operations is called the *makespan*. In the simple version of the problem an *objective* typically is to minimize the makespan.

In this paper we say that a JSSP is a set of resources, jobs and objective, we thereby never user the term *task* in the following.

2.1 Problem complexity

The general Job-Shop problem has been proven $\mathcal{NP}$-hard[12] so no efficient algorithm solving this scheduling problem exist. Therefore we practically have to solve it using brute-force where there for a $n \times m$ problem exists $(n!)^m$ possible solutions [7], where n denotes jobs and m machines.

By using a commercial available solver, which makes use of a lot of clever and sophisticated heuristics, we can significantly cut down the state-space that needs to be explored hence in general solve these problems faster.

2.2 Constraint Satisfaction Problem

The constraint satisfaction problem (CSP) can be specified as variables $x_i, i \in \{1,...,n\}$, domains $D_i, i \in \{1,...,n\}$ and constraints $c_j(x_1,...,x_n), j \in \{1,...,m\}$ and finding a solution $(x_1,...,x_n) = (v_1,...,v_n)$ where $v_i \in D_i, i \in \{1,...,n\} \wedge c_j(v_1,...,v_n), j \in \{1,...,m\}$, that is a solution is an assignment $(v_1,...,v_n)$ not violating any constraints. This can be extended to use objective functions by introducing $h(v_1,...,v_n)$ and adding one more conditional: minimize $h(v_1,...,v_n)$.

JSSP can of course be expressed as CSP by letting variables be start times of jobs, the domains are the possible start times of jobs and we have some natural constraints on the start times of jobs depending on which resources each job uses. The objective function in the classical JSSP would be to minimize the makespan.

In this paper we are primarily concerned about expanding the JSSP, but the transition from JSSP to CSP is important in that the Satisfiable Modulo Theories (SMT) problem can be thought of as a certain form of CSP. The problem of SMT is whether some logical formula is satisfiable in the context of some background theory - in the JSSP the background theory naturally will be integer arithmetic. The job of SMT-solver is to find an interpretation of a SMT-formula that makes it true, e.g. the SMT-formula $x + y > 5 \wedge x < 0 \wedge y > 0$ is satisfiable in the context of the theory of integer arithmetic, because $x \mapsto -2, y \mapsto 8$ makes the formula true.

2.3 Related work

Much of the available literature [7][8] is primarily concerned about techniques and implementation strategies of the solver and not so much about the expressiveness of problems. As a result a majority of the available benchmark suites for JSSP are limited to simple examples with unary resources.

To the best of our knowledge only IBM's CPLEX-suite is concerned about allowing more advanced problem descriptions, hence much of our inspiration is drawn from here. Since CPLEX lack any real competitors, no benchmarks supporting the use of CPLEX on advanced problems are public available.

Concerning the task of proposing a domain specific language [3] is great inspiration of what the language could contain and how.

2.4 Performance evaluation

Many of the used problems uses functionality inspired from CPLEX [6], so the performance evaluation will be done on basis of the performance achieved benchmarking the free 90-day trial version of CPLEX on a collection of their own examples.

Furthermore we will also evaluate and discuss the different approaches of finding the best solution according to a given objective:

- Satisfying the objective function using bi-section or a lower-bounding iteratively method.
- A comparison of using Z3 via the API where constraints are built programmatically or by parsing a problem in the SMT-LIB format.
- Several different ways of setting and manipulating constraints using the Z3 API.

Notice we do not consider the use of other solvers as Z3 currently[1] are considered the leading in terms of precision and performance[11].

3. THE Z3 SOLVER

The state-of-the-art SMT-solver Z3 is developed by Microsoft Research and is free to use for non-commercial purposes[2]. Z3 is used in different areas, e.g. software verification (SPEC#) and test case generation (PEX). Recently (November 2012) the source code of Z3 has been released to the public, and is now available for further investigation as supplement to the *many* papers written mainly by Leonardo de Moula and Nikolaj Bjørner about Z3. Originally Z3 was written for internal use in Microsoft which explains the lack of documentation. The missing documentation naturally makes it hard to know how to do things right when using Z3, which is why the performance optimisations of the tool developed to solve the extended JSSP are mainly made by our own empirical experiences with Z3.

[1]Z3 did not enter the *SMT-COMP* competition this year (2012) due to their SMT-LIB2 format not being finished, although the results from 2011 are still better than the competitors results from 2012.

[2]Licensed under MSR-LA. This license allows users to redistribute, copy, modify and experiment with Z3.

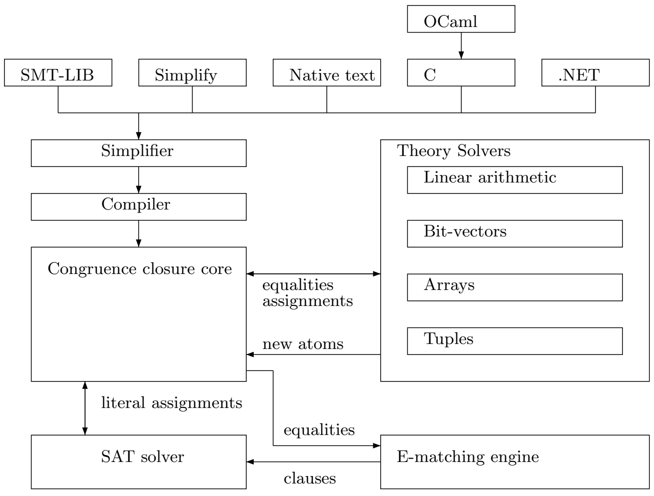

Figure 1: Architecture of Z3 from [5]

In the following we briefly describe the inner workings of Z3, mainly by focusing on the architecture illustrated in Figure 3. Z3 accepts the SMT-LIB format but also has a variety of APIs for different programming languages, among these the .NET API, which is what we are using to build our solving tool for the extended JSSP. Z3 will simplify the input and compile it into a data-structure consisting of a set of clauses. Now Z3 works by combining several solvers; a state-of-the-art SAT solver assigns truth values to atoms, a so-called E-graph is build of the atoms and equalities asserted by the SAT-solver. This E-graph has nodes that can point to theory solvers, that can assign truth values to atoms as well as producing fresh atoms that in turn is given to the SAT-solver again. E-matching is used to instantiate quantified variables from the E-graph - the E-matching in Z3 is done by new efficient algorithms, which is the main reason for the significant better performance of Z3 compared to other SMT-solvers. [5] has a more detailed description of how Z3 works.

Z3 is heavily configurable. A look at the output of `z3.exe /ini?` reveals that more than 250 parameters can be set on the solver. Z3 will try to configure the solver in the best way based on the input given, which is why we in this paper have not spent much time on investigating these parameters further.

4. EXTENDING THE PROBLEM

In the following we extend the presented version of the JSSP. In Section 6 we propose a domain specific language to describe the extended JSSP. In particular we want to extend the simple specification of the JSSP to include following features:

- **Precedence rules:** Specify in which order tasks are being processed.
- **Discrete resources:** Allow capacities above 1.
- **Reservoir resources:** Introduce consumable resources with the availability of adding more capacity to resources as time goes by.
- **More objectives:** Introduce new objectives, e.g. *minimize the use of a specific resource.*
- **Weighted objectives:** Give weights to different objectives.
- **Multiple resource use:** Allow tasks to use several resources both by allowing the use of choosing between resources but also by allowing a task to require the use of all resources specified.
- **Resource amount use:** Specify the amount of a resource used.
- **Soft precedence constraints:** Specify precedence constraints that are preferable but not necessary.

The introduction of discrete resources and specifying the resource amount used by a job, effectively makes it possible to share a resource between several jobs as long as the total use does not exceed the capacity of the resource.

Furthermore we also make a slight simplification by removing the jobs-level, so that a problem consists of jobs where each job has the same properties as a task.

With the features listed above we will be able to express a broader class of problems, though we have some natural bounds on which problem we will be able to specify and solve. The solution to the original JSSP is a static schedule, which means that the factors influencing a problem *must* be specified in the problem. Naturally this means that problems influenced by uncontrolled factors, e.g. the weather, sudden breakdowns, etc. cannot be specified. As we give static schedules as solutions, we require that everything influencing the solution have to be explicitly specified in the problem.

5. EXAMPLES

The previous section leaves us with a problem specification that has the features to solve a broad class of scheduling problems. In the following we list two example problems of very different nature, that should give rise to the domain specific language. The evaluation in Section 8 contains further examples to investigate.

5.1 Tiramisu recipe

This example is a subset of the tiramisu recipe described in [13]. We have extended the example with more advanced functionality in order to better model reality and emphasize the expressiveness of our language. For example in the original example some tasks needed a bowl, sometimes a regular bowl and sometimes a heatproof, in this example we allow steps that previously requested a regular bowl to also accept a heatproof-bowl, this can possibly help shorten the overall makespan.

Furthermore [13] is only concerned about the order of carrying out the recipe, and see what can be carried out concurrently, so they have no notion of time, meaning that all jobs take equal amount of time. We have here augmented each task with a duration we believe to be realistic.

Finally we have added two precedence relations, a soft and a hard.

```
Resources {
  consumable mascarpone 225
  consumable milk 430
  consumable espresso 360
  consumable sugar 175
  semaphore woodenSpoon
  semaphore saucepan
  semaphore bowl
  semaphore heatProofBowl
}
Jobs {
  MakeCreamTopping {
    duration 3
    use saucepan & heatProofBowl
    consume 430 milk
    consume 100 sugar
    produce creamTopping
  }
  AddMascarpone {
    duration 2
    use woodenSpoon & (bowl |
        heatProofBowl)
    consume 225 mascarpone
    consume creamTopping
    produce custard
  }
  MixCoffeeSyrup {
    duration 4
    use bowl | heatProofBowl
    consume 360 espresso
    consume 75 sugar
    produce coffeeSyrup
  }
  MakeLayers {
    duration 3
    consume coffeeSyrup
    consume custard
    produce tiramisu
  }
  MakeCreamTopping > addMascarpone
  MakeCreamTopping << MakeLayers
}
Objectives {
  minimize makespan
}
```

Running this example gives rise to the solution at Figure 2. At this figure we can see which sub-tasks can be carried out in parallel to shorten the overall cooking time. We could easily get a more realistic plan taking the number of available chefs into account by modelling this number of chefs as an additional resource and let each sub-task that requires active participation from a chef use such a resource.

5.2 Holiday planning

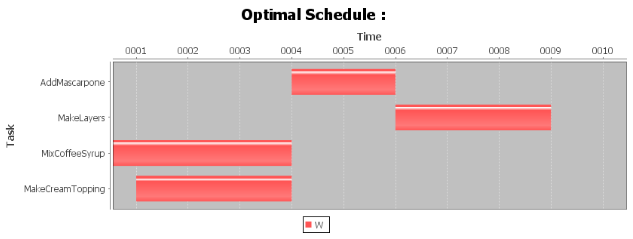

Figure 2: The most effecient way to carry out the tiramisu recipe by disregarding the currently unsupported OR-operator. Notice that the schedule cannot be carried out satisfying the soft-precedence constraint, so that is disregarded in order to make the schedule satisfiable.

A sub problem of the Nurse Scheduling Problem described in [1] is the holiday planning problem of nurses at a hospital. A hospital have a minimum need of nurses at all times, and nurses have some preferred weeks where they want to plan their holidays. The optimal holiday plan is where as many nurses as possible have as many of the preferred weeks as possible assigned as their holiday.

This problem can be modelled with the proposed features in the previous section: A discrete resource can represent the number of free holiday weeks at the hospital at any time. For each nurse we need a discrete resource with capacity 1 indicating whether a nurse has holiday or not. w jobs are needed for each nurse, where w is the number of holiday weeks per nurse. Each of these jobs will use the resource representing the number of free holiday weeks. Finally soft constraints can be used to express the preferred weeks a nurse want to have hers/his holiday. The soft constraint is expressed as a real time constraint on the tasks for each nurse. This means that each time unit in this example represents a week.

```
Resources {
  semaphore holiday 3
  semaphore nurse1
  ...
  semaphore nurseN
}
Jobs {
  Nurse1Holiday1 {
    use holiday & nurse1
  }
  Nurse1Holiday2 {
    use holiday & nurse1
  }
  ...
  NurseNHoliday1 {
    ...
  }
  Nurse1Holiday1 > 3
  Nurse1Holiday1 < 5
  Nurse1Holiday2 > 3
  Nurse1Holiday2 < 5
  ...
  NurseNHoliday1 > 4
}
Objectives {
  maximize satisfied soft constraints
}
```

In the specification above we have an example with N nurses where at most 3 nurses can have their holidays in the same week. As can be seen the number of tasks is the number of nurses multiplied by the number of holidays weeks per nurse. The soft constraints for Nurse1 state that he/she prefer holiday in weeks 3, 4 and 5. We see the number of soft constraints can vary between zero to the number of weeks each nurse has preferences multiplied by two because we need to express the preferred week as between week x and y.

The specification above is obviously not that simple and it is clear that for just a few nurses we will see a quite large problem description, but nonetheless we are still able to express the problem with the proposed features.

This example also shows that the DSL and the tool attached to the DSL should be targeted towards mid-end use, i.e. a front-end used to enter data and abbreviate it into the form of the above should be build. The developer building this front-end should only be concerned about representing the data, our tool will in turn create the constraints of the problem and solve it by using Z3. In this way the developer still have the ability to express a complex problem, but need not to care about how the rather complex constraints of a problem should be formulated.

6. LANGUAGE DESIGN

The examples in the previous section give rise to the Job-Shop Scheduling Problem Language, abbreviated JSPL. In designing the language it is important for us that the resulting language is easy to read, in the sense that one should be able to quickly understand what a specification of a problem is trying to solve. With this said we realize that the problems that can be modelled with this language is often much more complex than e.g. the tiramisu example, which is why we see the language and tool as a mid-end tool targeted towards organizations and people that need to solve problems but do not have the resources to formulate the advanced constraints themselves.

One can of course specify problems directly in the language, but we would encourage to build a front-end on top of the tool. This is acceptable because in most cases we would expect that the tool will used to solve problems of similar kind by the same people, e.g. room assignments at universities. Because of this our focus has turned a bit away from making it very easy to understand at a glance and over to focus more on the actual language features instead.

6.1 EBNF Specification

In the following we present the language formulated in the EBNF style derived from regular expressions. This means that the usual regular expression operators: trailing * for *0 or more*, trailing + for *1 or more* and trailing ? for *optional* applies. Keywords are highlighted with the `type writer` font.

⟨*problem*⟩ ::= ⟨*resources*⟩? ⟨*jobs-precedences*⟩ ⟨*objectives*⟩

⟨*resources*⟩ ::= '`Resources` {' ⟨*resource*⟩+ '}'

⟨*resource*⟩ ::= '`semaphore`' ⟨*ID*⟩ ⟨*INT*⟩?
  | '`consumable`' ⟨*ID*⟩ ⟨*INT*⟩? ('/' ⟨*INT*⟩)?

$\langle$*jobs-precedences*$\rangle$ ::= 'Jobs {' $\langle$*job*$\rangle$+ $\langle$*precedence*$\rangle$* '}'

$\langle$*job*$\rangle$ ::= $\langle$*ID*$\rangle$ '{ duration' $\langle$*INT*$\rangle$ ('use' $\langle$*resource-use*$\rangle$)? $\langle$*prod-con*$\rangle$* '}'

$\langle$*resource-use*$\rangle$ ::= $\langle$*INT*$\rangle$? $\langle$*ID*$\rangle$
| $\langle$*resource-use*$\rangle$ ('|' | '&') $\langle$*resource-use*$\rangle$
| '(' $\langle$*resource-use*$\rangle$ ')'

$\langle$*prod-con*$\rangle$ ::= ('produce' | 'consume') $\langle$*INT*$\rangle$? $\langle$*ID*$\rangle$

$\langle$*precedence*$\rangle$ ::= $\langle$*ID*$\rangle$$\langle$*pre-op*$\rangle$($\langle$*ID*$\rangle$ | $\langle$*INT*$\rangle$)

$\langle$*pre-op*$\rangle$ ::= '<' | '«' | '>' | '»'

$\langle$*objectives*$\rangle$ ::= 'Objectives {' $\langle$*objective*$\rangle$+ '}'

$\langle$*objective*$\rangle$ ::= $\langle$*INT*$\rangle$? 'minimize makespan'

Where $\langle INT \rangle$ is a positive integer and $\langle ID \rangle$ is an identifier. A problem of the form above has to adhere to a set of rules formally described in the following sections.

6.1.1 Problem P

A problem consists of a set of resources R, jobs J, precedences C and objectives O:

$$P = \langle R, J, C, O \rangle$$

6.1.2 Resources R

Resources are something that can be used, consumed or produced by jobs. We have two kinds of resources:

Semaphore This is either a unary or discrete resource. This kind of resource can only be *used* by a job. E.g. if a job states that it uses 5 of a semaphore resource with the capacity 7, another job can use the same resource at the same time, as long as it only requires $\leq 7 - 5$.

Consumable This is a reservoir resource; more precisely it is a resource where the capacity is either lowered or raised permanently when jobs interacts with it. We say that jobs either *produces* or *consumes* a consumable resource.

A resource R_r is described as:

$$R_r = \langle id_r, c_r, m_r, t_r \rangle \in R$$

id_r is the name of the resource, $c_r \in \mathbb{N}^0$ is the initial capacity of the resource, $m_r \in \mathbb{N}^+$ is the maximum capacity for the resource and $t_r \in \{semaphore, comsumable\}$ is the type of the resource.

From the grammar we can see that if a resource is of type *semaphore* a maximum capacity cannot be set. If no capacity is set it is implied that the resource is a unary resource, and the capacity c_r is 1. Semaphore resources with capacity $c_r > 1$ is called discrete resources, and the domain of c_r is restricted to $\mathbb{N}^+$ (T.1). If the resource is of type *consumable* a maximum capacity can be set and $c_r \leq m_r$ (T.2) should hold. If no maximum capacity is set it is implied to be ∞.

6.1.3 Jobs J

Jobs are the entities being scheduled. A job is described by:

$$J_i = \langle id_i, s_i, p_i, r_i, pc_i \rangle \in J$$

id_i is the name of the job, $s_i \in \mathbb{N}^0$ is the start time of the job, $p_i \in \mathbb{N}^0$ is the duration or processing time of the job, r_i is a set of all the resources used by the job and pc_i is a set of all the resources produced or consumed by the job. The grammar shows that all of this except the start time is initial described by the user. The start time is what the solver should find based on the constraints generated.

r_i is not just a set of resources but a set of pairs: $(R_r, u_{ir}) \in r_i \mid R_r \in R$ (T.3), where R_r is a resource and $u_{ir} \in \mathbb{N}^+$ is the amount used by job J_i of this resource R_r. The grammar allows to leave out u_{ir}, in these cases the use is implied to be just 1. The following must be fulfilled:

$$\forall (R_r, u_{ir}) \in r_i : t_r = semaphore \land u_{ir} \leq c_r \quad \text{(T.4)}$$

pc_i is also a set of pairs $(R_r, a_{ir}) \in pc_i \mid R_r \in R$ (T.5), where R_r is a resource and $a_{ir} \in \mathbb{Z} \setminus \{0\}$ is the amount job J_i produces or consumes of resource R_r. a_{ir} is positive if the resource is produced and negative if it is consumed. Here we see that the input specification by the user is transformed when a problem is created in the program - users never enter a negative number, but only state whether the job consumes or produces a resource, it is the job of the parser to generate the correct set of produce and consume resources. The grammar allows to leave out the amount, in these cases it is implied that the amount is 1. The following must be fulfilled:

$$\forall (R_r, a_{ir}) \in pc_i : \\ t_r = consumable \land (a_{ir} < 0 \lor (a_{ir} > 0 \land a_{ir} \leq m_r)) \quad \text{(T.6)}$$

The amount consumed can at no point exceed the current capacity c_r of a resource R_r, but this cannot be checked statically because the initial capacity of a consumable resource is allowed to be lower as a job consuming it, as another job can produce it before.

If a job J_i produces the amount a_{ir} of a resource R_r it means that *after* $s_i + p_i$ the capacity is now $c_r = c_r + a_{ir}$. If a job J_i consumes the amount a_{ir} of a resource R_r the capacity of the resource is $c_r = c_r + a_{ir}$ (because a_{ir} in this case is negative we also add a_{ir} to c_r) immediately *after* s_i. This gives the two obvious constraints that at any time the capacity a consumable resource cannot exceed its maximum capacity and it cannot be below 0.

When a job can be executed it is naturally constrained by the availability of a resource; when it is not used by another job, when the capacity is large enough for the job to consume or when it can be produced without exceeding the maximum capacity.

6.1.4 Precedences C

In JSPL precedences is the closest you get to setting constraints directly. We have two kinds of precedences; soft and hard, the main difference between soft and hard precedences is that soft precedences can be removed completely if no satisfiable schedule can be found, strategies for doing this is described later in the paper. Besides that a precedence can be described as a before-after relation between jobs or before-after relation between jobs and real times. Formally a precedence is described by:

$$C_i = \langle e_1, rel, e_2 \rangle \in C$$

Where $e_1 \in J$ and $e_2 \in J \cup \mathbb{N}^0$ (T.7) and the relation rel can be either one $<$ or $>$ meaning it is a soft precedence or it can be two succeeding $<$ or $>$ meaning it is a hard precedence. The meaning of rel is described below:

$$\begin{aligned} > : e_1 \in J \wedge e_2 \in J : s_1 \geq s_2 + p_2 \\ \vee\, e_1 \in J \wedge e_2 \in \mathbb{N}^0 : s_1 \geq e_2 \end{aligned}$$

In other words if e_1 and e_2 are jobs, e_1 can start as soon e_2 has finished, if only e_1 is a job it can start after the time e_2 has passed.

$$\begin{aligned} < : e_1 \in J \wedge e_2 \in J : s_1 + p_1 \leq p_2 \\ \vee\, e_1 \in J \wedge e_2 \in \mathbb{N}^0 : s_1 + p_2 \leq e_2 \end{aligned} \quad \text{(T.8)}$$

Again, in other words if e_1 and e_2 are jobs, e_2 can start as soon e_1 has finished, if only e_1 is a job it has to end no later than the time e_2.

From the above we an only deduce one requirement that we can check statically; namely that the last case (T.8).

All of the rules marked with (T.x) in the above is implemented as a static type checker before the actual solving of the problem takes place. The file `TypeChecking.fs` contains references to this section.

7. CONSTRAINTS

This section describes which constraints are required from our problem description and the motivation behind our choices. In Section 7.7 we briefy describe how the constraints are generated using Z3.

7.1 Hard Precedences

The hard-precedence rules are quite simple as they just define the order on how the jobs should be executed. The precedences constraint are described formally in the previous section.

7.2 Soft precedences

The soft-precedence rules denotes rules that do not have to be enforced but are preferable to do. There are countless ways to interpret this, but we currently chose to define it as; If the model is not satisfiable treating the soft-precedences as hard-precedences, all soft-precedences are removed.

There are some reasoning and limitations to this rather rough approach, namely:

- These soft-precedence are not taken into consideration when trying to optimize the objective function. In some cases we may be able to achieve a much better target value by removing soft-precedence(s).
- In some cases it might be preferable to just alter one or more of the soft-precedences than entirely remove them. The alteration would be to push the precedence-value by some pre-determined fraction or value.
- When we remove the soft-precedence, we could try to fulfil as many as them of possible. This would require solving the problem $n!$ times where n is the number of soft precedences, and as these problem generally are computational intractable this could lead to extreme long running times when solving. We could bring down the required number of tries by just removing the constraints one by one, and then stop once the model gets satisfiable. The removal of precedences should probably not happen by random, so some sort of removal-order should be imposed; but this introduces the problem when the only un-satisfiable precedence is in the bottom, so that all other precedences needs to be removed in order to make it satisfiable.

We think that a linear-weighting scheme can overcome many of these problems, but this requires careful consideration and technique limiting the greatly increased state-space that needs to be considered.

As stated before a JSSP is a constraint satisfaction problem, these kinds of problems can be augmented with soft constraints as well. [9] describes how the general approach to supporting soft constraints is to associate costs to soft constraints and finding the minimum aggregated cost. As we stated in Section 2.2 JSSP can be expressed as a CSP, when adding soft constraints we have a special kind of CSP namely the weighted CSP, or WCSP. If we see a WCSP as a tree of solutions, there are a variety of different approaches to solving WCSP of which [9] describes how branch-and-bound can be used to find a partial solutions and create sub problems of the original problem by keeping track of the cost of the current solution and comparing it to removing a constraint. The branch-and-bound algorithm can be used to decide whether a sub-tree should be pruned or not.

Future work includes researching on how to efficiently support this increased expressiveness of problems by the use of soft-precedences, including how much work should be done before giving a problem to Z3 and how much work Z3 itself should take care of.

7.3 Initialization of the upper-bound

As the computational complexity of calculating the resource use is greatly determined[3] by the length of the initial upper-bound UB_{init}, we try to bound this interval as much as possible before use by setting:

$$UB_{init} = UB_{seq} + \Sigma_{J_i \in J} p_i$$

Where UB_{seq} is the upper-bound achieved by disregarding all resources, only constrained by the precedence rules and by the requirement that the tasks should be placed sequentially, although *absolute less than* precedences needs to be ignored for this computation as they might lead to a false unsatisfiable conclusion. As the unary-resource rule is used to ensure this, the computational complexity are just determined by the amount of jobs and does not require any specified initial upper-bound.

[3]See Section 8 for the size of impact.

We need to add all jobs duration times, as the producible and consumable resources in conjunction with the absolute precedence constraints can exceed the bound found by UB_{seq} as they can implicitly model additional precedence rules[4].

If the given schedule can be solved, the tightest upper-bound UB_{best} is hence constrained by $UB_{best} \leq UB_{init}$ as forcing all jobs to start at UB_{seq} all real-times precedence constraints, which are the only thing that can give rise to gaps in a schedule, are practically disregarded and we can just add the sum of the duration of each task in order to get the longest possible makespan of any given schedule.

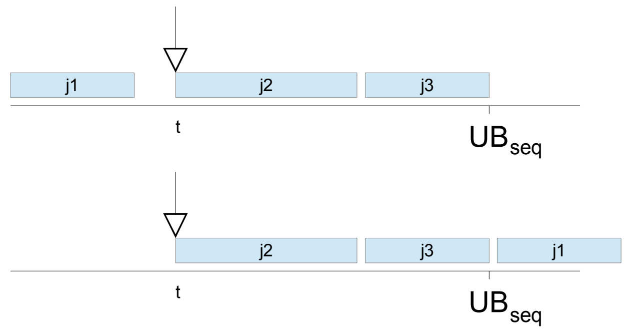

Figure 3: (u) Calculating UB_{seq} from the precedence rules.(l) UB_{init} can exceed that of UB_{seq} when accounting for producible and consumable

For clarification consider the following example:

> Regard Figure 3 where j_2 is not allowed to start before time t and j_3 should follow j_2. j_1 consumes a resource which is only generated by j_3 hence implicitly models the constraint that j_1 should come after j_3. As calculating UB_{seq} cannot account for the later implicit precedence, j_1 is just placed before j_2 as this gives rise to the shortest makespan. The calculation of UB_{seq} is therefore potentially exceeded once trying to actually place the jobs, as a result we need to use $UB_{seq} + \Sigma_{J_i \in J} p_i$ to be safe.

This function also serves the purpose of checking whether the model are even satisfiable at all although it cannot catch all unsatisfiable problems, some will first be catched during the actual solving. Naturally if a problem is shown to be unsatisfiable we abort here before the actual solving.

Further work involves creating a more precise, hence tight, estimation of UB_{init}.

7.4 Semaphore resources

We define the set $JU_{R_r} \mid R_r \in R$ as jobs using the semaphore resource R_r. For unary resources the following constraints expressing that two jobs using the same unary resource cannot be executed at the same time are needed:

$$\begin{gathered}\forall R_r \in R \mid c_r = 1 \wedge t_r = \mathit{semaphore} : \\ \forall J_i \in JU_{R_r} : \\ s_i + p_i \leq s_j \vee s_j + p_j \leq s_i \\ \text{where } J_j \in JU_r \setminus J_i\end{gathered}$$

For discrete resources we need a more sophisticated way of describing constraints, as a resource should be available to more than one job at a time. [2] describes how we can decide whether a discrete resource can be used by more jobs or if the current jobs using it makes it impossible for other to use it as well. The basic idea is that for every time t we need to check that the resource usage of resource $R_i \in R \mid t_i = \mathit{semaphore}$ does not exceed c_i. We let the variable $x_{it} \in \{0,1\}$ denote whether a job J_i is executed at time t:

$$\begin{gathered}\forall J_i \in J, \forall t \in \{0,...,UB\} : \\ \texttt{if } ((s_i \leq t) \wedge (s_i > t - p_i)) \\ \texttt{then } x_{it} = 1 \\ \texttt{else } x_{it} = 0\end{gathered} \tag{7.4.1}$$

Note that we assume the lower bound is always 0. The above can now be used to express the following constraint for each discrete resource:

$$\begin{gathered}\forall R_r \in R \mid c_r > 1 \wedge t_r = \mathit{semaphore} : \\ t \in \{0,...,UB\} : \\ \sum_{J_i \in JU_{R_r}} u_i \cdot x_{it} \leq c_r\end{gathered}$$

Where u_i is the amount of a resource a job uses. This constraint basically just sums all the use of a resource R_r at time t and makes sure it does not exceed the capacity c_r of that resource. [2] proposes some optimizations specifically when encoding constraints to SMT-solvers, namely to introduce the following:

$$y_i \leftrightarrow (s_i \leq t) \wedge (s_i > t - p_i)$$

And replacing (7.4.1) by y_i. Their findings are that on smaller instances (problems with less than 50 job descriptions) this gives better performance results, but as we are encoding the value x_{it} as a function in Z3 by using the for all quantifier we do not need to and cannot make this optimization, because Z3 already do some simplifications itself, see Section 3.

The discrete resource notation is more expressive, and can also be used to model the unary resources, but as the use of unary resources are considered quite common and the unary resource formula are less computational demanding we choose to differentiate between these two.

7.5 Consumable resources

In the following section we define $JPC_{R_r} \mid R_r \in R$ as the set of jobs producing or consuming the consumable resource R_r.

In the previous section we saw that we need the following constraint on each consumable resource R_r:

$$\begin{gathered}\forall t \in \{0,...,UB\} : \\ c_r \geq 0 \wedge c_r \leq m_r\end{gathered}$$

[4]As in the tiramisu example

If m_r is ∞ we only need the first inequality in the formula above. A way to express this is by letting f_{it} denote whether a job J_i has been executed at time t:

$$\begin{gathered}\forall J_i \in J, \forall t \in \{0,...,UB\}: \\ \texttt{if}\ \ (t \geq s_i + p_i) \\ \texttt{then}\ \ f_{it} = 1 \\ \texttt{else}\ \ f_{it} = 0\end{gathered}$$

We can now define the capacity c_{rt} of each consumable resource R_r on each time t:

$$\begin{gathered}\forall R_r \in R \mid \wedge t_r = \mathit{consumable}: \\ t \in \{0,...,UB\}: \\ c_{rt} = c_r + \sum_{J_i \in JPC_{R_r}} a_{ir} \cdot f_{it} + \sum_{J_i \in JPC_{R_r} \mid a_{ir} < 0} a_{ir} \cdot x_{it}\end{gathered}$$

The first of the above summations will make sure that the produced or consumed amount a_{ir} of jobs already executed at time t counts toward the new capacity c_{rt}, the second will make sure that the consumed amount by jobs consuming a resource is subtracted immediately after the job has started execution. c_r is the initial capacity of the resource which of course as well should be taken into account when calculating the current capacity.

Finally we are able to state the constraints needed to make sure consumable resources are never exceeding their maximum or going below 0:

$$\begin{gathered}\forall R_r \in R \mid t_r = \mathit{consumable}: \\ t \in \{0,...,UB\}: \\ c_{rt} \geq 0 \wedge c_{rt} \leq m_r\end{gathered}$$

Where the last inequality is only necessary if m_r is not ∞.

7.6 Objective Functions

The support of different objectives is as of now not implemented and is thereby unfortunately not something the user can specify. We currently only have support for this single objective functions:

- `minimize makespan` the general objective used throughout the paper, which instructs Z3 to find the shortest possible schedule.

The calculation of this proceeds by initially setting $UB = UB_{init}$, i.e. that were only interested in schedules less or equal to this value. Once such a schedule is found, we proceed by bi-sectioning until the best possible schedule is found. The simplified bisection algorithm is shown below:

```
rec bisection min max =
  if min = max then
    min
  else
    let x = (min + max) / 2
    let b = Z3.(b => (x < UB))
    let status = Z3.Check(b)
    match status with
      |   SATISFIABLE -> bisection min x
      | UNSATISFIABLE -> bisection x max
      | UNDECIDABLE   -> abort
```

This gives rise to precisely $log_2 UB_{init}$ checks that needs to be performed. As here shown the alteration is implemented using Z3s support for retraction of statements, i.e. when performing `Z3.check(b)` we solve the context with the additional requirement that `b = true`.

We found that, somehow, Z3 uses much less time on concluding something being unsatisifiable rather than satisfiable. So we tried a similar approach iteratively bounding it from below, where we set an initial stride to $log_2(UB_{init})$ and low to 0. As long as it is unsatisfiable we continue with the same stride, when we get a satisfying assignment we go back to the last unsatisfiable assignment and iteratively increases the bound with a stride of 1.

```
rec iterative low stride =
  let b = Z3.(b => (x < UB))
  let status = Z3.Check(b)
  match status with
    | SATISFIABLE when stride = 1 -> low
    | SATISFIABLE    -> iterative  (low -
      stride) 1
    | UNSATISFIABLE  -> iterative  (low +
      stride) stride
    | UNDECIDABLE    -> abort
```

This gives rise to at most $2log_2 UB_{init}$ iterations.

A comparison in terms of performance can be seen in Section 8.

Future work includes support of

- `minimize usage <resource>` Finds a schedule minimizing the usage of a given resource.
- `minimize output <consumable>` Finds a schedule that executes all tasks so that the value of a given consumable resource is as low as possible.

Naturally these should also be implemented with `maximize`.

As well as allowing for solutions composed of several objective functions. This would need a weighting scheme, similar to that of soft-precedences, to specify the importance of each objective which would lead to an explosion in the state-space that needs explored. Naturally techniques limiting this would need to be researched.

7.7 Constraint Generation in Z3

In the previous sections we saw which constraint are needed for a problem. In the following we will explain how constraints are described in Z3.

When we encode JSSP into a SMT-formula we only have one kind of variable of interest: the start time of each job. For each job J_i in J a IntConsts variable $\texttt{j\_}id_i$ that represents the start time is created. These variables are the unknown in the SMT-formulation of the JSSP, hence these are the variables Z3 should find an satisfiable assignment to. As described in the previous section, the bisection and iterative method focuses on narrowing the upper bound on the start times because the only objective supported is minimizing the makespan. If we had another objective to minimize the usage of a resource, we would still be interested in the start times of jobs, but an additional constraint on the total use of resources would be introduced and the bisection method would focus on narrowing this total use rather than narrowing the upper bound on the total makespan.

The variables x_{it} and f_{it} are modelled with quantifiers in Z3. The following shows how it is encoded in SMT-LIB format:

```
(declare-fun x_value (Int Int Int) Int)
(assert (forall (task Int) (time Int) (dur
      Int)
          (= (x_value task time dur)
          (ite (and (<= task time)
                    (> task (- time dur)))
                1
                0))))
```

The Z3 API for .NET is used in an imperative fashion, which is why the code is a mixture of functional programming and imperative programming. Variables are created as integer constants on the Z3 context. Constraints are also created on the context. When a constraint is build it has to be asserted on the Z3 solver, which is created from the context. How x_{it} is implemented and how constraints are generated in general in Z3 can be inspected further in the file `Solver.fs`.

8. EVALUATION

In this section we evaluate the performance of our implementation as briefly described in Section 2.4.

As said before the source code of Z3 got released to the public during the period of writing this paper, and no literature or documentation explaining the best practices were available to us, as result some of our design choices are based on these measurements treating Z3 in a black-box manner.

For evaluation we use the following four scheduling problems found in the example folder shipped with CPLEX, naturally these are translated into our representation:

`sched_intro` This is a basic problem that involves building a house; the masonry, roofing, painting, etc. must be scheduled. Some tasks must necessarily take place before others, and these requirements are expressed through precedence constraints.

This is a simply problem consisting of 10 jobs and 14 precedence constraints. `sched_intro_mod` is a modification of this problem, making it harder to solve.

`sched_production` A chemical manufacturer produces batches of specialty chemicals to order. An order consists of a set of jobs. Each job has an optional precedence requirement, arrival week of the job, duration of the job in weeks, the week that the job is due, the number of reactors required, distillation columns required, and centrifuges required. The objective is to minimize the completion time of all orders.

This is modelled using 8 different jobs, using 18 precedence constraints and a 3 semaphore resources.

`sched_shipload` The problem consists of scheduling the loading of a ship.

This is modelled using 34 different jobs, using 43 precedence constraints and a single semaphore resource with capacity of 8.

`sched_cumul` This is a problem of building five houses in different locations; the masonry, roofing, painting, etc. must be scheduled. Some tasks must necessarily take place before others and these requirements are expressed through precedence constraints. There are three workers, and each task requires a worker. There is also a cash budget which starts with a given balance. Each task costs a given amount of cash per day which must be available at the start of the task and Payments are received on at pre-determined dates.

This is a fairly complex problem modelled using 50 jobs, 130 precedence constraints, a semaphore resource and a consumable resource.

All tests has been carried out using an `Intel Core i5 3317U` processor and are averaged over 10 runs.

8.1 Comparison of the Z3-API and SMT-LIB

It is common for tool-chains to use some sort of standardized file-representation as input for the convenience of the user. Using the recognized SMT-LIB format the objective functions can only be implemented by the use of iteratively computing solutions with the use of constraints on the makespan, this means that we continuously needs to edit the SMT-LIB file used for solving, and that Z3 needs to parse this input repetitively, not to mention destroying its entire context and learned lemmas on each iteration.

We have previously made such an prototype [4], which is here used as basis for comparison, we use the same *bi-section* algorithm and constraint generating algorithms on both, so that any performance difference can be ascribed to the difference in interaction with the solver.

The old prototype did not ensure a safe initial calculation of the upper-bound as described in Section 7.3, it just used $\Sigma_{J_i \in J} p_i$ as UB_{init}, we call this `SMT-LIB unsafe`. So during the comparison we tested this implementation at the same upperbound setting as the old, called `API unsafe`, as well as with a safe estimate `API safe`.

The difference in performance can be seen at figure 4 from which we can conclude that our new fine-tuned implementation is almost twice as fast as the old prototype. Even when comparing the safe `API` against the unsafe `SMT-LIB` prototype we still complete slightly faster.

These results also tells us the impact of not choosing the initial UB_{init} as tight as possible, i.e. the $unsafe$ benchmarks worked for these instances having an UB_{init} just half of the safe size.

8.2 Comparison of bi-sectioning and lower-bounded objective functions

We noticed that in general it was more costly in terms of execution speed calculating satisfiable schedules than unsatisfiable ones, so we benchmarked the bi-section algorithm

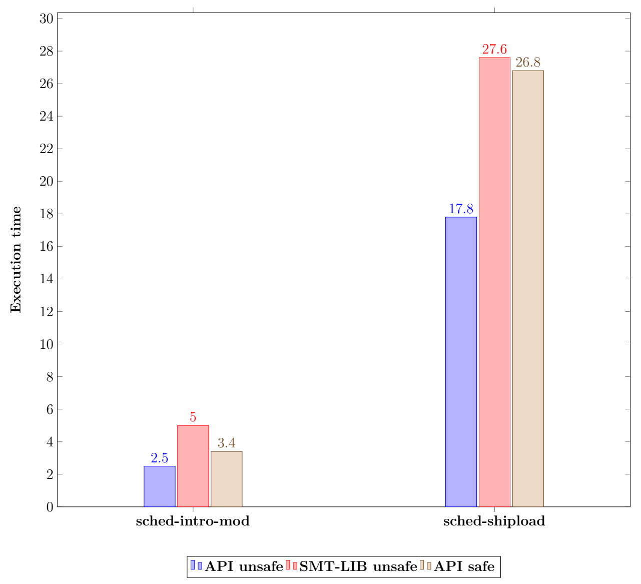

Figure 4: Comparison using SMT-lib presentation or API. *API safe* has an UB_{init} twice as large as the other two.

against the iteratively lower-bounding method. The results can be inspected at figure 5.

One thing to note is that bisection uses **exactly** $log_2 UB_{init}$ iterations, whereas the iterative uses **at most** $2log_2 UB_{init}$: The iteratively lower-bounding method is in general much

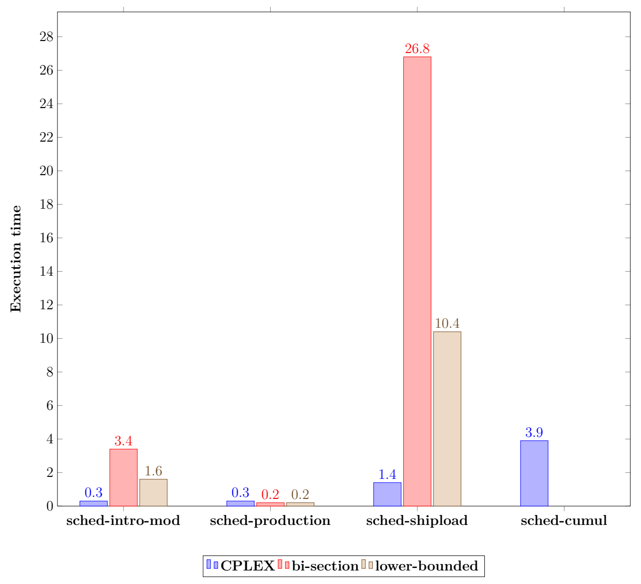

Figure 5: Comparison of objective strategies and CPLEX.

faster than doing bisection due to satisfiable conclusions in general being harder for Z3 to compute than unsatisfiable ones. Although when testing upperbounds close to what is achievable the execution time for at satisfiable / un-satisfiable conclusion seems to be equal.

Comparing the performance against the commercial off-the-shelve scheduling suite in CPLEX from IBM, we see that our solution lacks way behind. The only details about CPLEX's engine available is that they also model their problems as CSP and uses a tailor-made solver on these constraints.

Most likely CPLEX use another representation or algorithm to model their discrete and consumable resources as their execution time in many cases seems unaffected of the capacity and use of these whereas ours execution time is very much dependent on this.

Unfortunately proved `sched-cumul` to be hard for our solver, using the iteratively we found a solution within 5 hours. Bi-sectioning were not even close given the same timeframe.

The issue in this schedule is the length of the optimal schedule being quite high, which greatly increases the computational complexity as previously discussed.

8.3 Empirical findings of constraint-generation

We have tried a lot of different techniques to improve the performance. Some of them are mentioned here:

- We have tried implementing the solver using stacks to represent the assertions, via. the `push` and `pop` statements as well as assumptions that allows retraction of assertions. The former instructs Z3 to use an incremental internal solver and requires recomputing the lemmas upon `pop()`. The approach using assumptions lets Z3 automatically configure which solvers to use and allows for reuse of learned lemmas. Overall, to our great surprise, the push/pop approach were slightly faster.
- We have also tried regenerating all the discrete-resource constraints on each iterations, instead of just hoping Z3 would disregard those unreachable by the UB variable. The overall execution time were often much lower than by reuse, but varied very much with the problem in question.
- When we limit our problems to only consist of unary-resources, very large problems becomes easy solvable with execution times not that far from CPLEX's.

9. CONCLUSION

We have here proposed a domain specific language for advanced Job Shop Scheduling Problems with support of discrete resources, consumable resources, several resource use, soft constraints and several objective functions. A mid-end tool parsing and solving these kinds of problems has been developed - although several resource use (only partly), soft constraints and several objective functions has not been implemented.

By regarding the scheduling as a constraint satisfaction problem we can solve it using a third-party solver. We have here used Z3 as it, as the time of writing, is the best performing solver. The source code of Z3 has only recently been released to the public so there are not yet much available information about its inner workings and best practices.

Hence have we fine-tuned the constraint generation using an semi-empirical approach and achieved a general speed-up of a couple of factors over the old prototype.

We also compared the performance against CPLEX which also can solve advanced scheduling problems using a tailor-made CSP-solver. Unfortunately proved our implementation to be several magnitudes slower than the CPLEX-suite due to our modelling of discrete and consumable resources. Using only unary resources, corresponding to the classical job-shop problem, the performance seems to be more or less on par.

We conclude that Z3, as of now, is not a good match for solving these advanced scheduling problems, although there is a chance that future development will increase the performance and that literature about programming-practices with Z3 will be released, which will aid us in achieving a better performance, i.e. many techniques that intuitively were expected to increase the performance actually decreased it, e.g. bi-section, assumptions, regeneration of constraints, etc..

10. FURTHER WORK

- Syntactic sugar in the grammar, e.g. for declaring multiple similar task.
- Implement the use of the OR-operator on resources.
- In [2] it is argued that redundant constraints, that is constraints expressing the same but modelled in slightly different ways, in some cases can help the solver in finding solutions faster. This should be investigated if also holds for Z3.
- Use a weighting scheme to support multiple objectives.
- Use a weighting scheme to better support the use of soft-constraints.
- Further performance tuning.

References

[1] A. Ohuchi A. Jan, M. Yamamoto. Evolutionary algorithms for nurse scheduling problem. *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on In Evolutionary Computation, 2000.*, 1:196–203, 2000.

[2] Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem. In *SARA*, 2011.

[3] Howard Beck, Ken Currie, and Austin Tate. A domain description language for job-shop scheduling, 1993.

[4] Jakob Jakobsen Boysen, Niklas Quarfot Nielsen, and Sune Mølgaard Laursen. A scheduling specific dsl. Unpublished project, 2011.

[5] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceed*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. ISBN 978-3-540-78799-0. doi: http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[6] IBM. Ibm ilog scheduler v6.7 user's manual. 2009. URL `http://lia.deis.unibo.it/Courses/AI/applicationsAI2009-2010/materiale/cp15doc/pdf/usrscheduler.pdf`.

[7] A. S. Jain and S. Meeran. Deterministic job-shop scheduling: Past, present and future. *European Journal of Operational Research*, 113:390–434, 1998.

[8] Mikkel T. Jensen and Tage Kiilsholm Hansen. Robust solutions to job shop problems, 1999.

[9] Javier Larrosa and Thomas Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159(1–2):1 – 26, 2004. ISSN 0004-3702. doi: 10.1016/j.artint.2004.05.004. URL `http://www.sciencedirect.com/science/article/pii/S0004370204000815`.

[10] Ferdinando Pezzella and Emanuela Merelli. A tabu search method guided by shifting bottleneck for the job shop scheduling problem, 2000.

[11] SMT-EXEC. Summary view of the official smt-comp'11 competition run. 2011. URL `http://www.smtexec.org/exec/?jobs=856f`.

[12] Yu. N. Sotskov and N. V. Shakhlevich. Np-hardness of shop-scheduling problems with three jobs. *Discrete Appl. Math.*, 59(3):237–266, May 1995. ISSN 0166-218X. doi: 10.1016/0166-218X(93)E0169-Y. URL `http://dx.doi.org/10.1016/0166-218X(93)E0169-Y`.

[13] Frank P. M. Stappers, Sven Weber, Michel A. Reniers, Suzana Andova, and Istvan Nagy. Formalizing a domain specific language using sos: An industrial case study. In Anthony M. Sloane and Uwe Aßmann, editors, *SLE*, volume 6940 of *Lecture Notes in Computer Science*, pages 223–242. Springer, 2011. ISBN 978-3-642-28829-6. URL `http://dblp.uni-trier.de/db/conf/sle/sle2011.html#StappersWRAN11`.