Solution Checking with CPMpy

Hendrik ‘Henk’ Bierlee and Tias Guns

KU Leuven, Belgium

ModRef 2026

Paper Code Slides Course

Motivation

Auto-grading constraint models

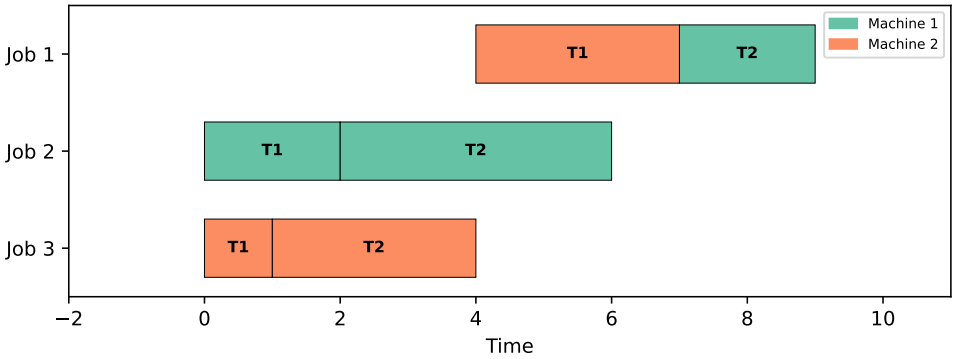
Specification → Student model → Candidate solution → Checker → Grade + feedback

- Auto-grading is a **great** way to give personalized and objective feedback to students
- However, checker inconsistencies can be costly:
  - Confusing and demotivating for students
  - Computational load in re-grading past submissions
  - Checker can leak unspecified information
- Prior work at ModRef'17 explains the MiniZinc approach [1]

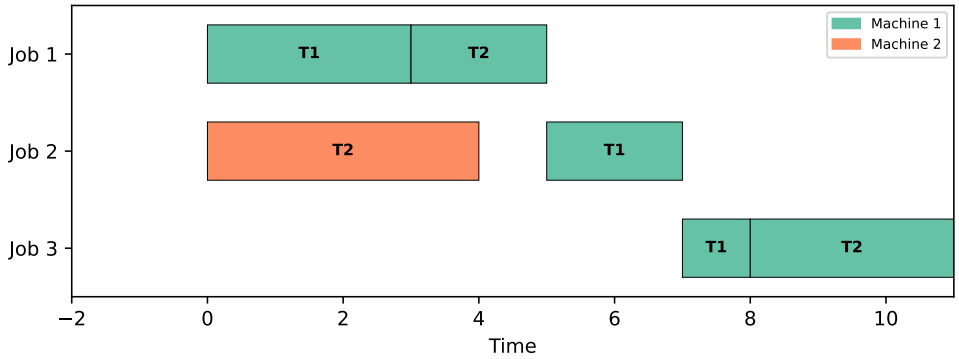
Flexible Jobshop Scheduling (FJSS) specification

Given an instance of `n` jobs, `m` tasks per job, `k` machines, and an `n`-by-`m` duration matrix (e.g. `{"n":3, "m":2, "k":2, "duration":[[3,2],[2,4],[1,3]]}`), the model contains variables `X` where `X[i,j]` denotes the start time for job `i`, task `j`.

Constraints: tasks within a job must be executed in order; start times are non-negative; tasks on the same machine cannot overlap. **Objective:** minimize total flowtime (i.e. sum of end times).



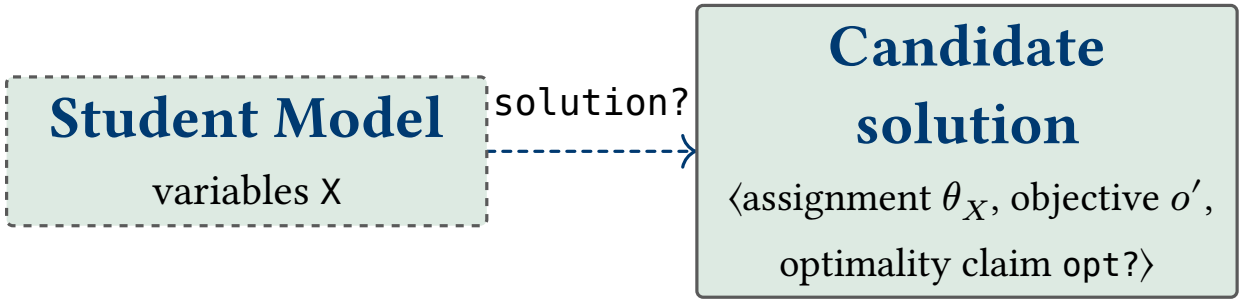
Correct solution



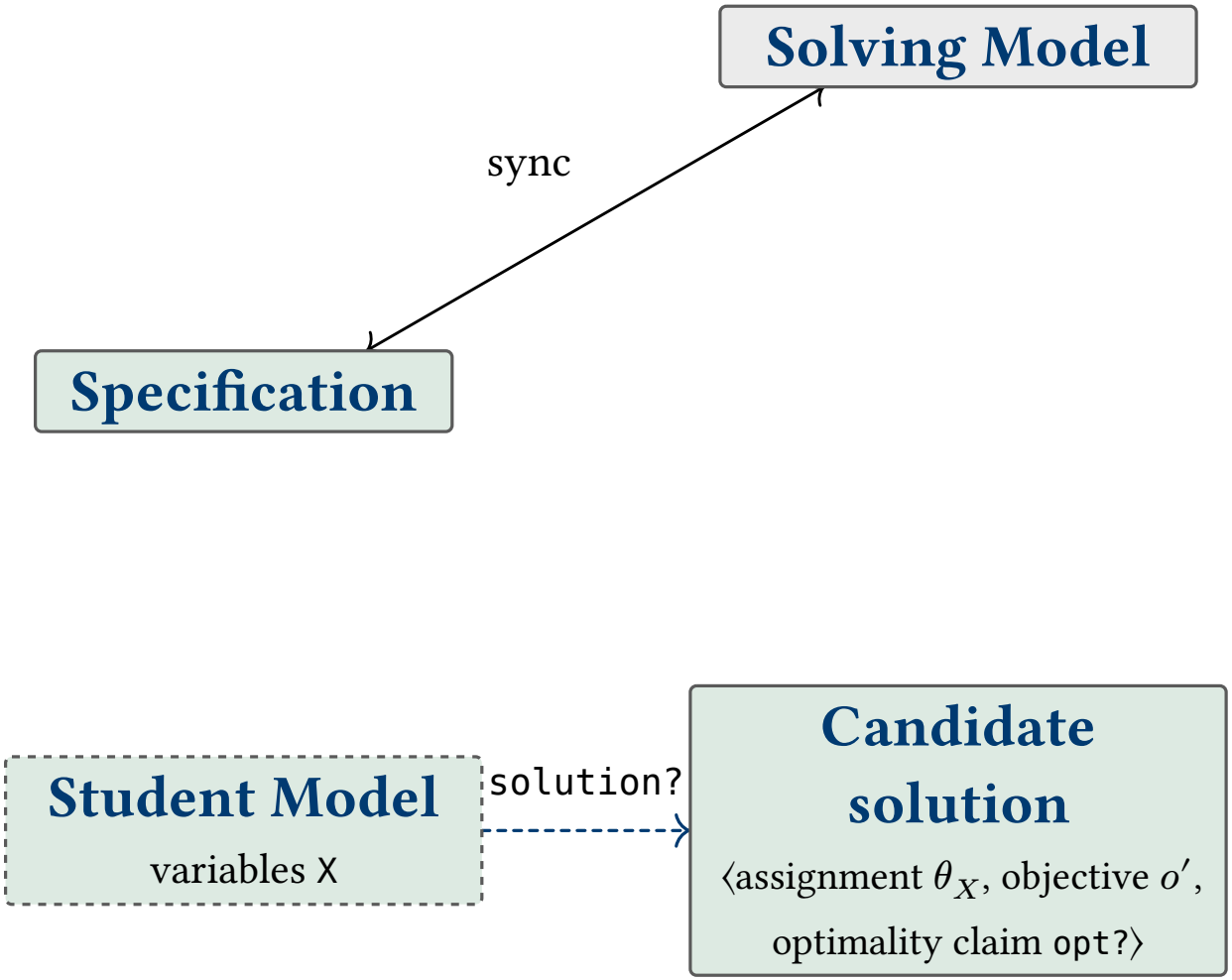
Incorrect solution

Flexible Jobshop Scheduling (FJSS) specification

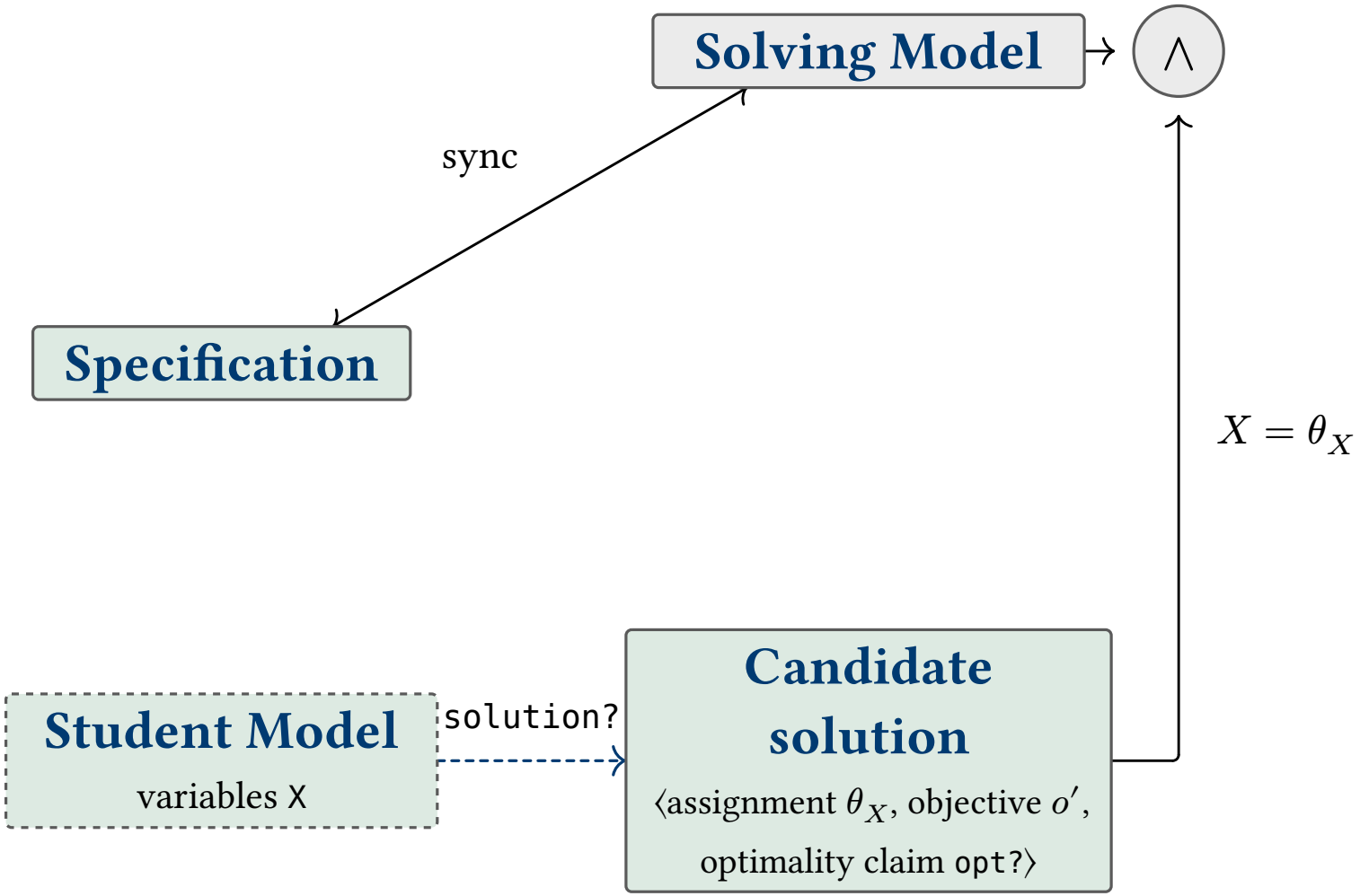
Specification



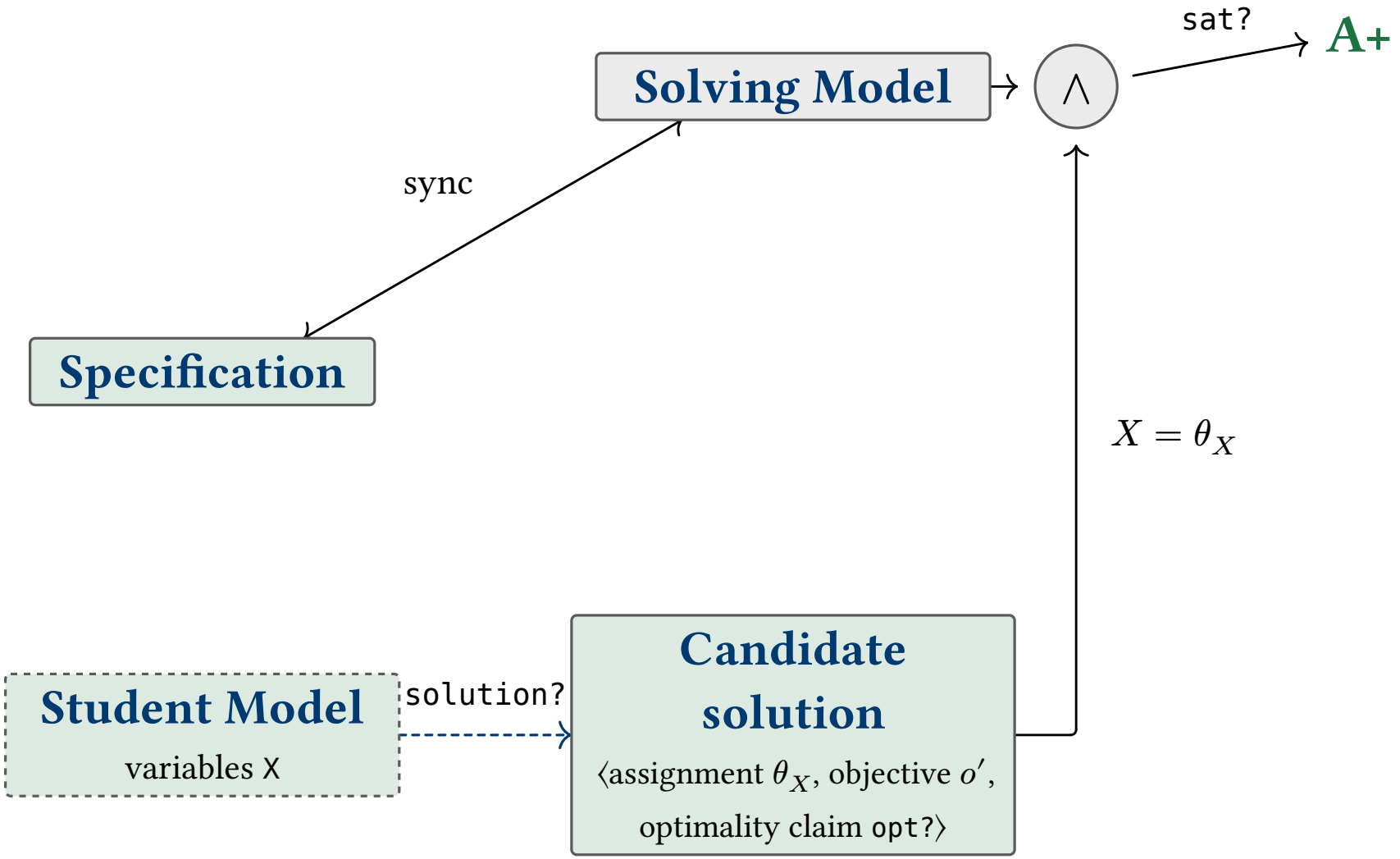
Flexible Jobshop Scheduling (FJSS) specification



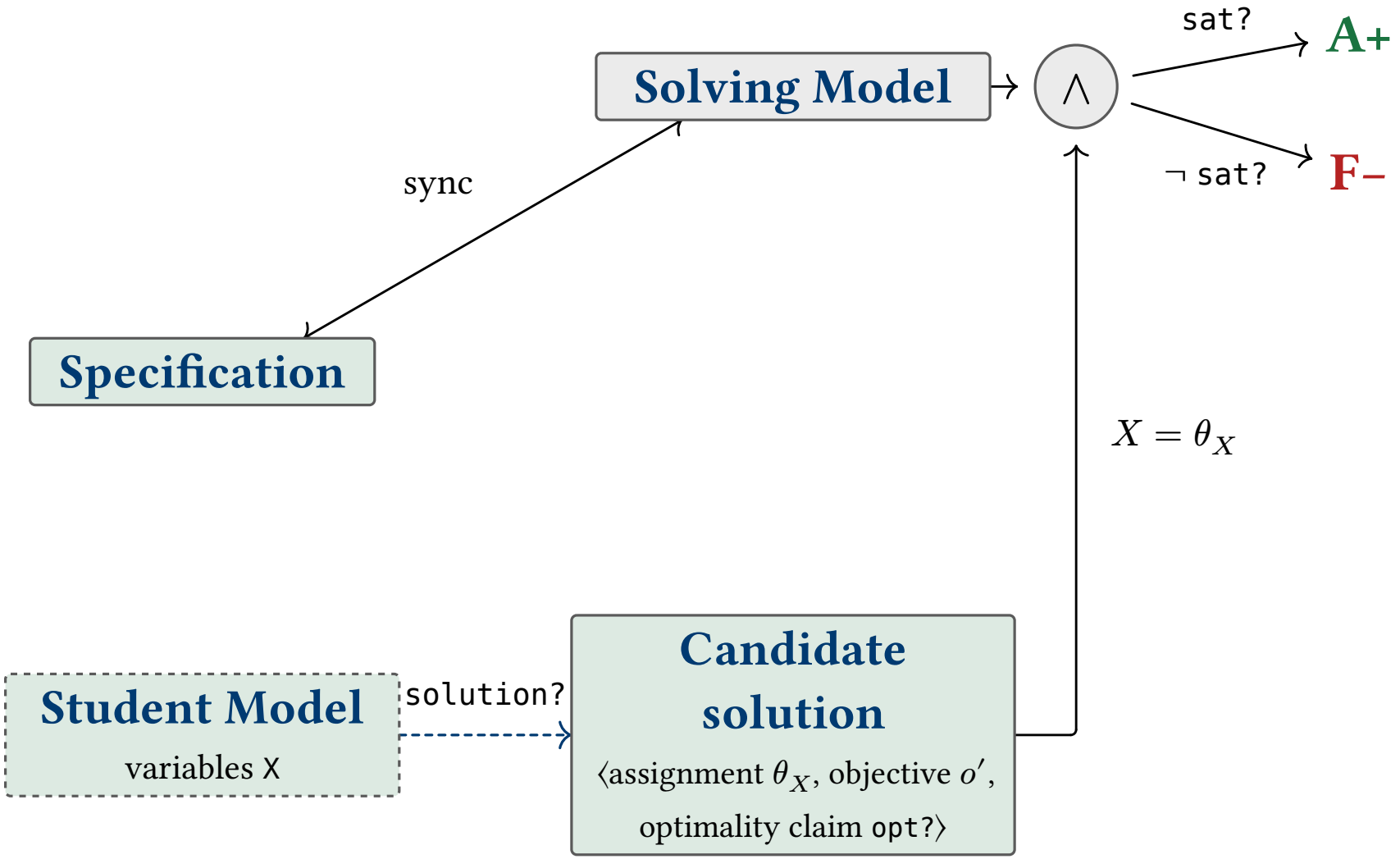
Flexible Jobshop Scheduling (FJSS) specification



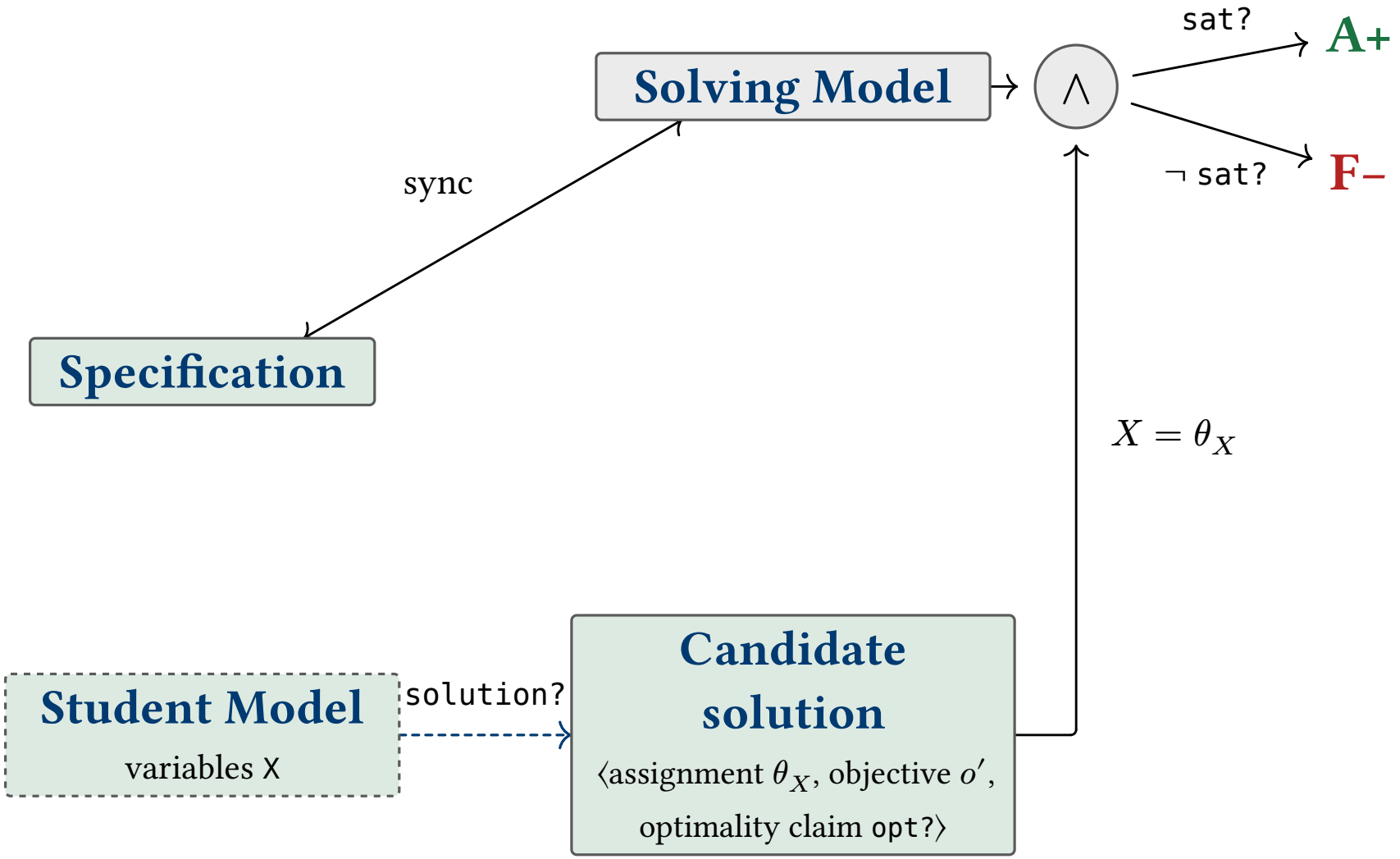
Flexible Jobshop Scheduling (FJSS) specification



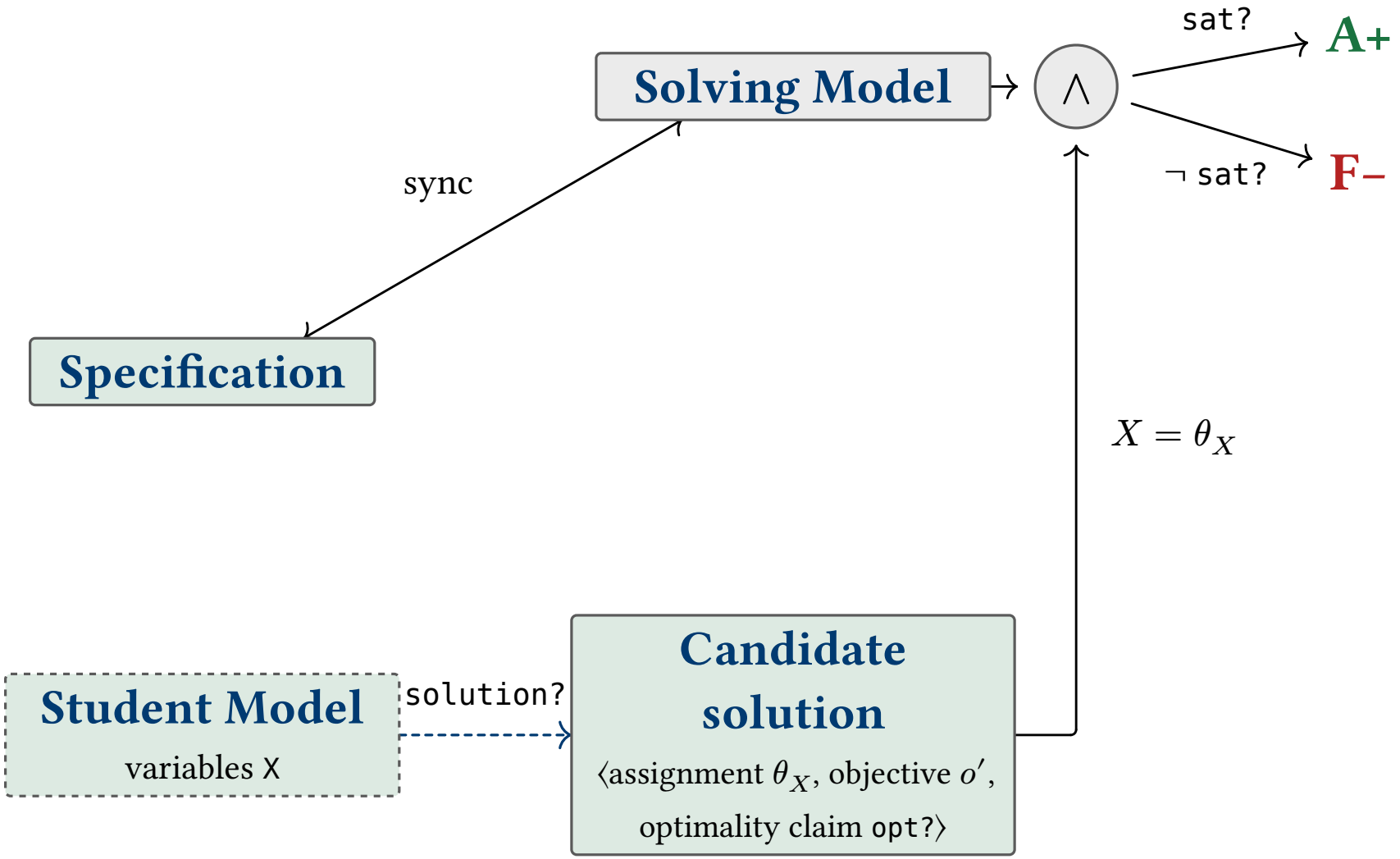
Flexible Jobshop Scheduling (FJSS) specification



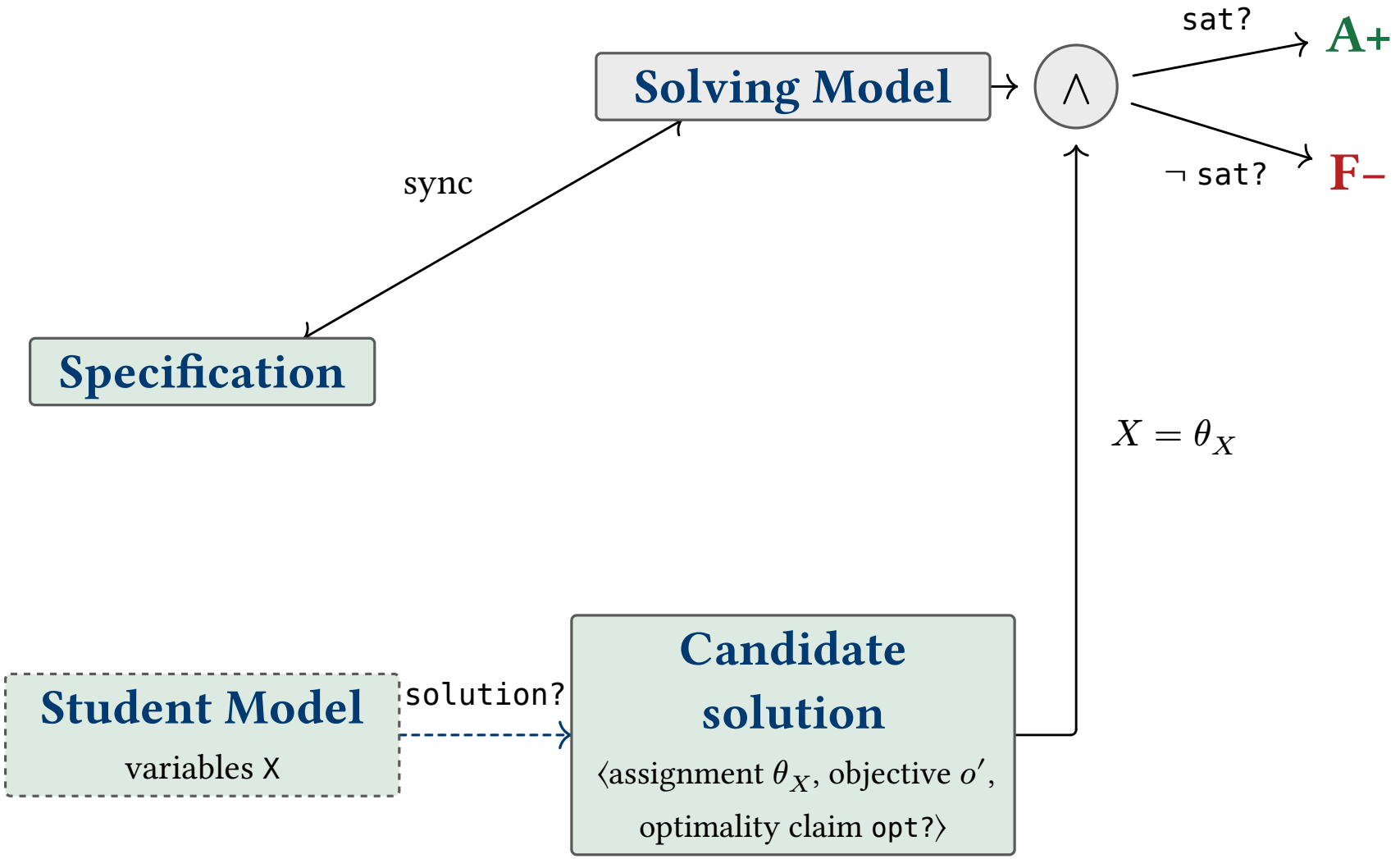
Flexible Jobshop Scheduling (FJSS) specification



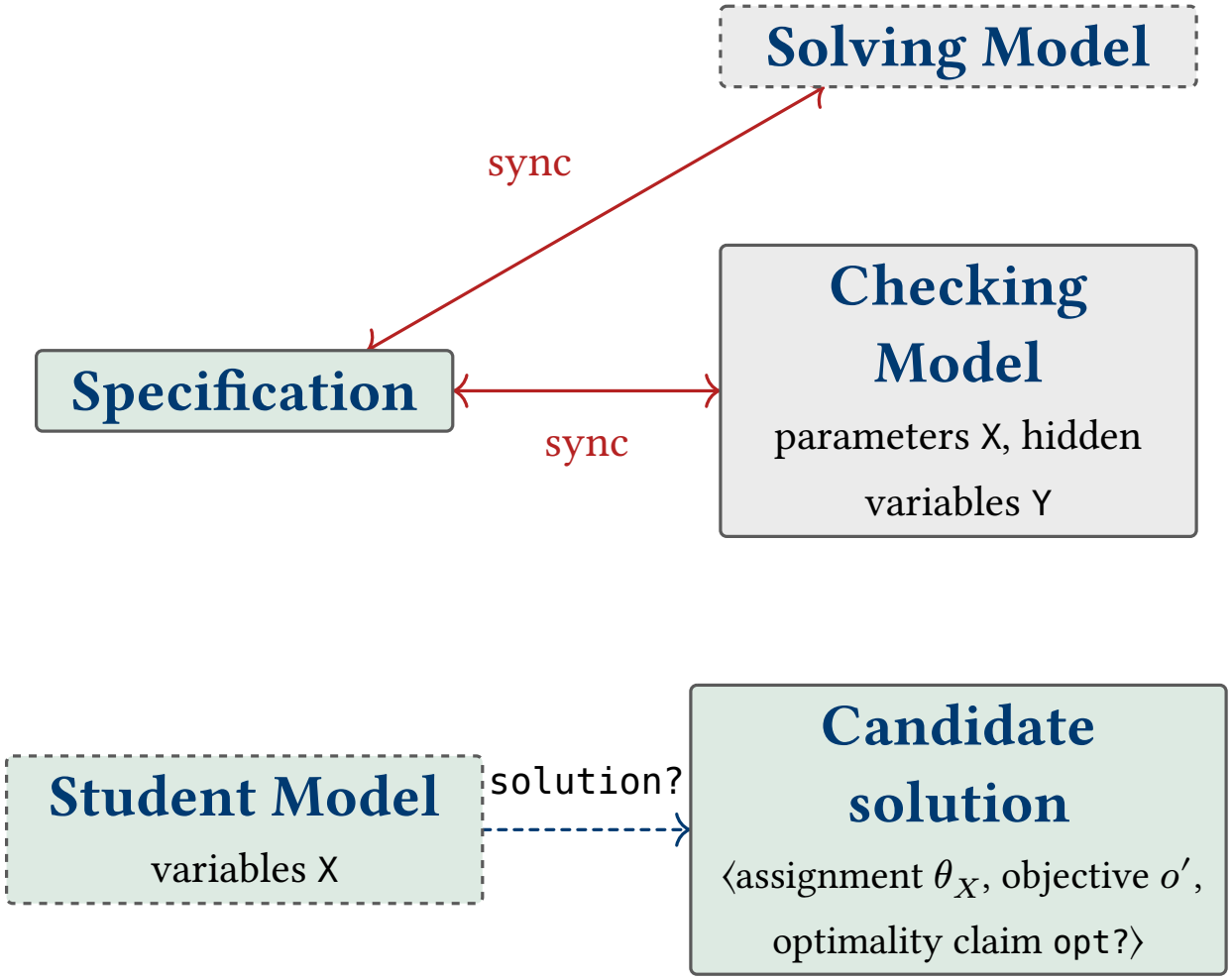
Flexible Jobshop Scheduling (FJSS) specification



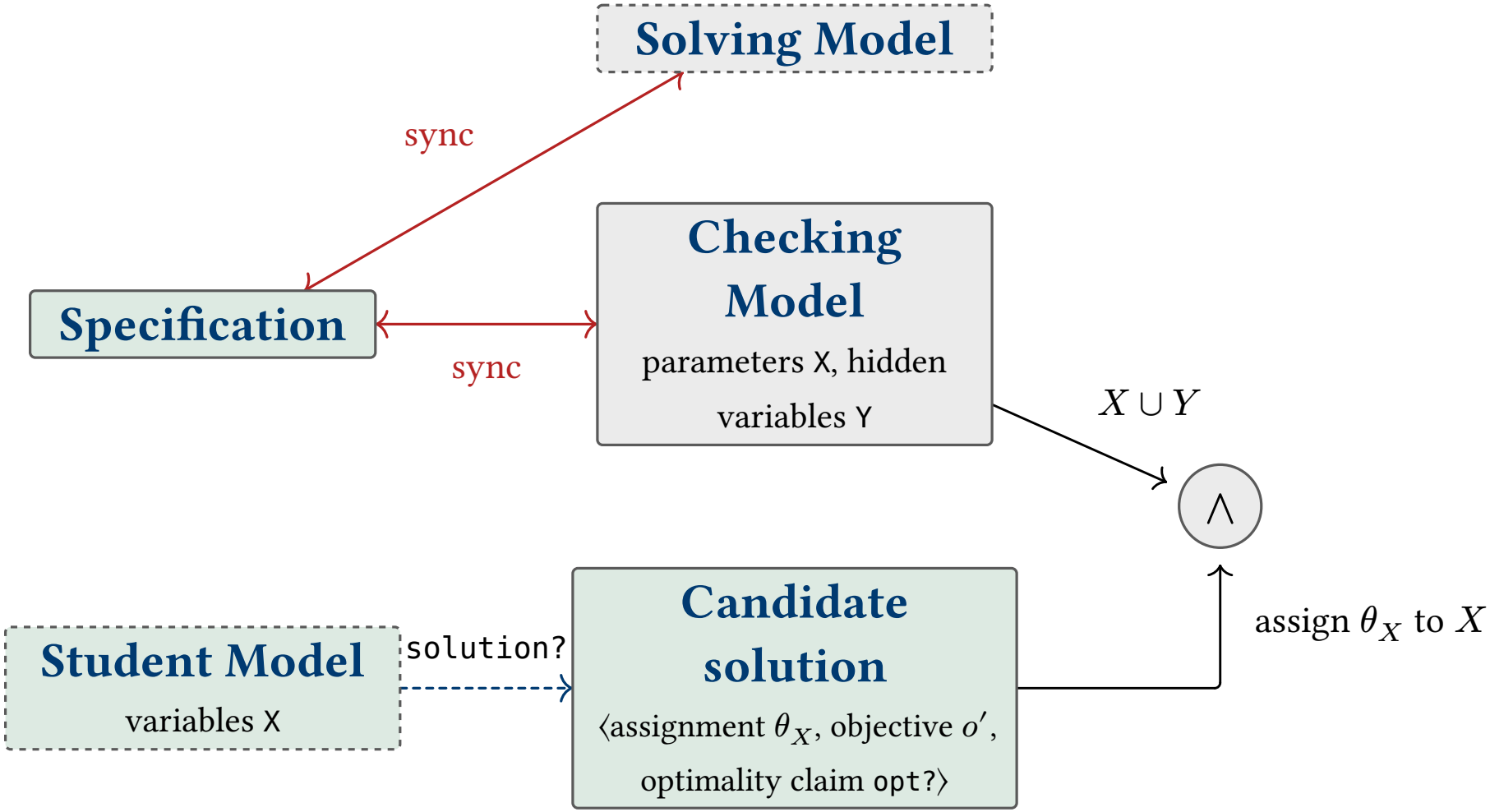
Flexible Jobshop Scheduling (FJSS) specification



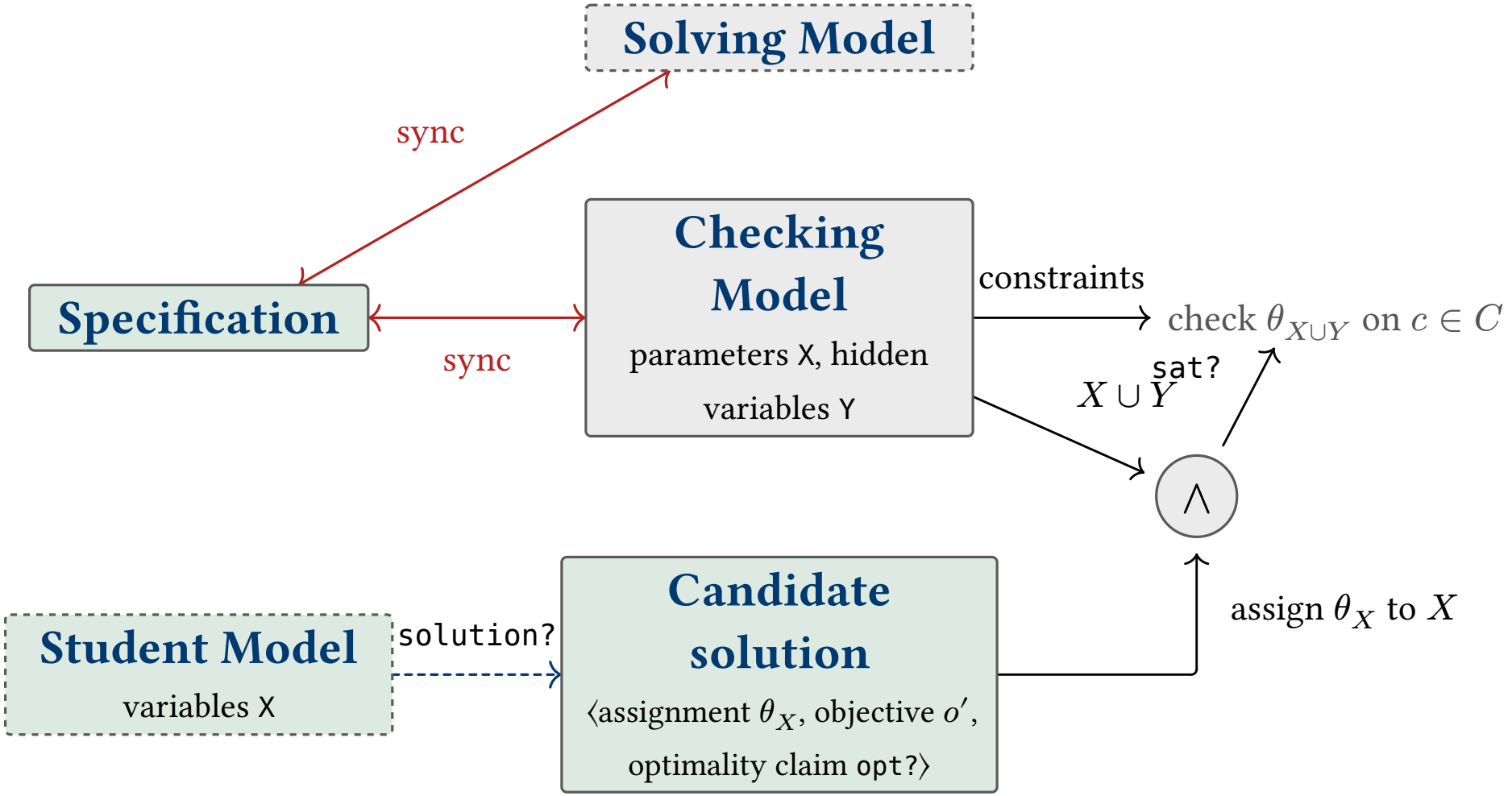
Flexible Jobshop Scheduling (FJSS) specification



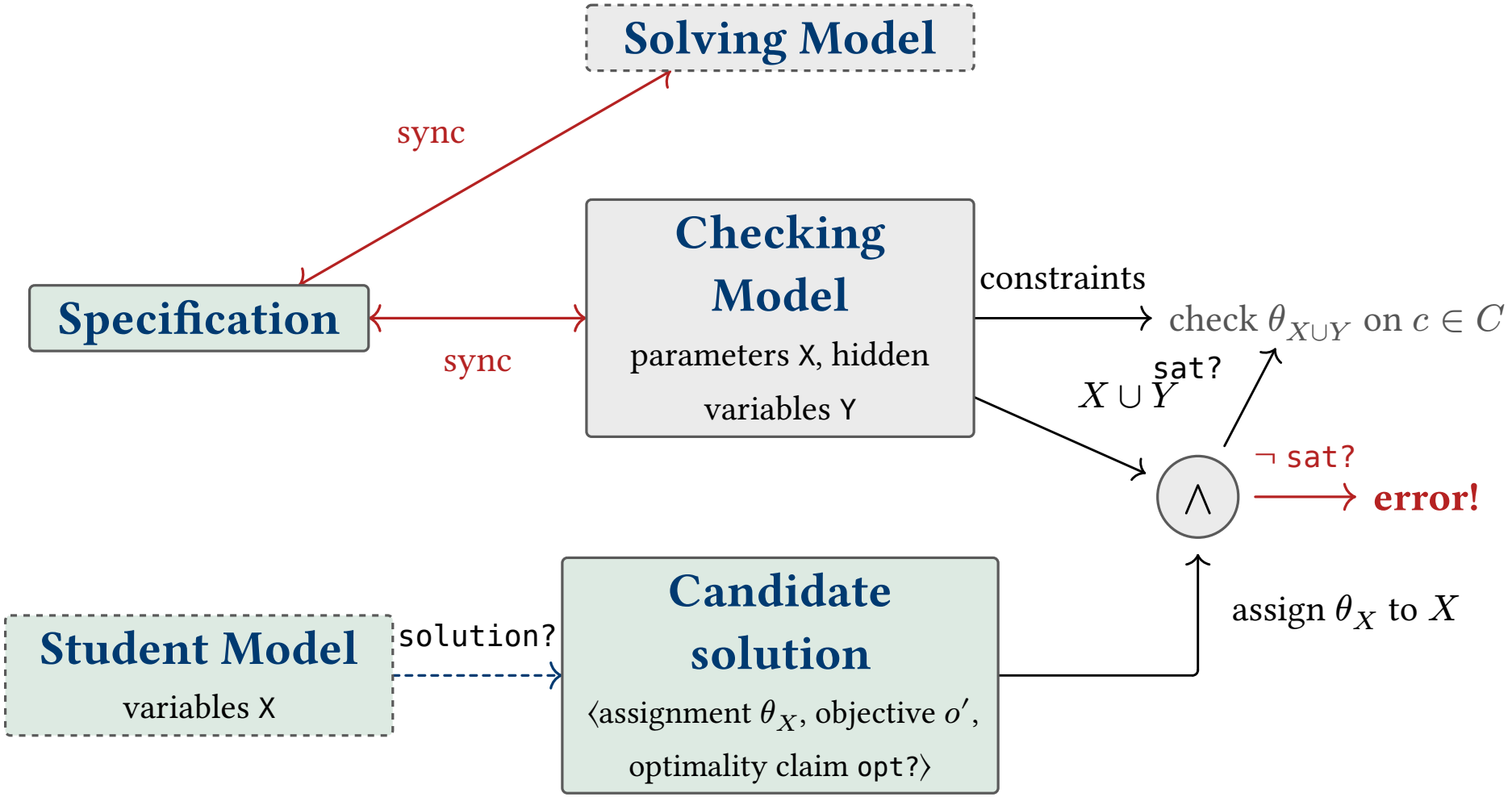
Flexible Jobshop Scheduling (FJSS) specification



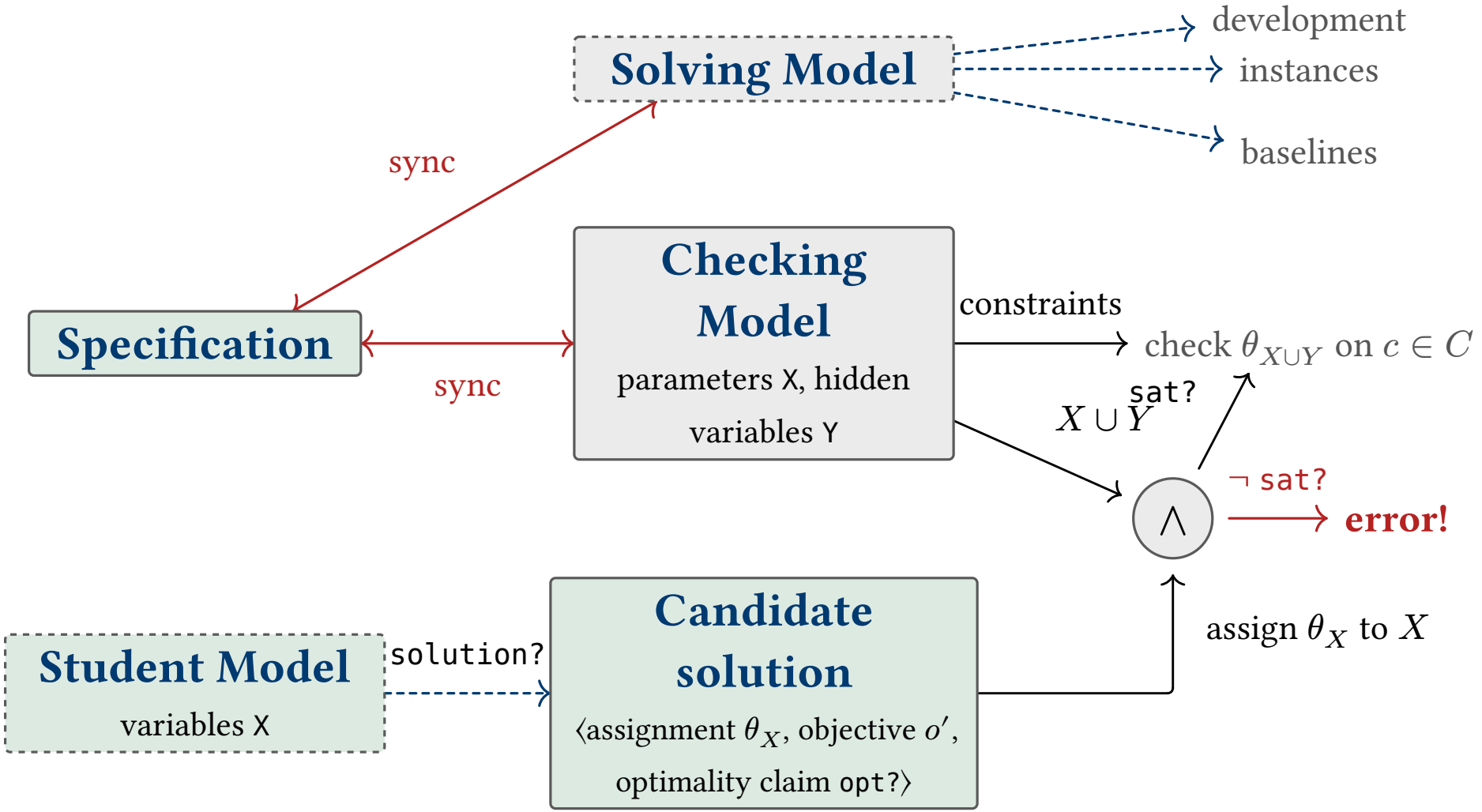
Flexible Jobshop Scheduling (FJSS) specification



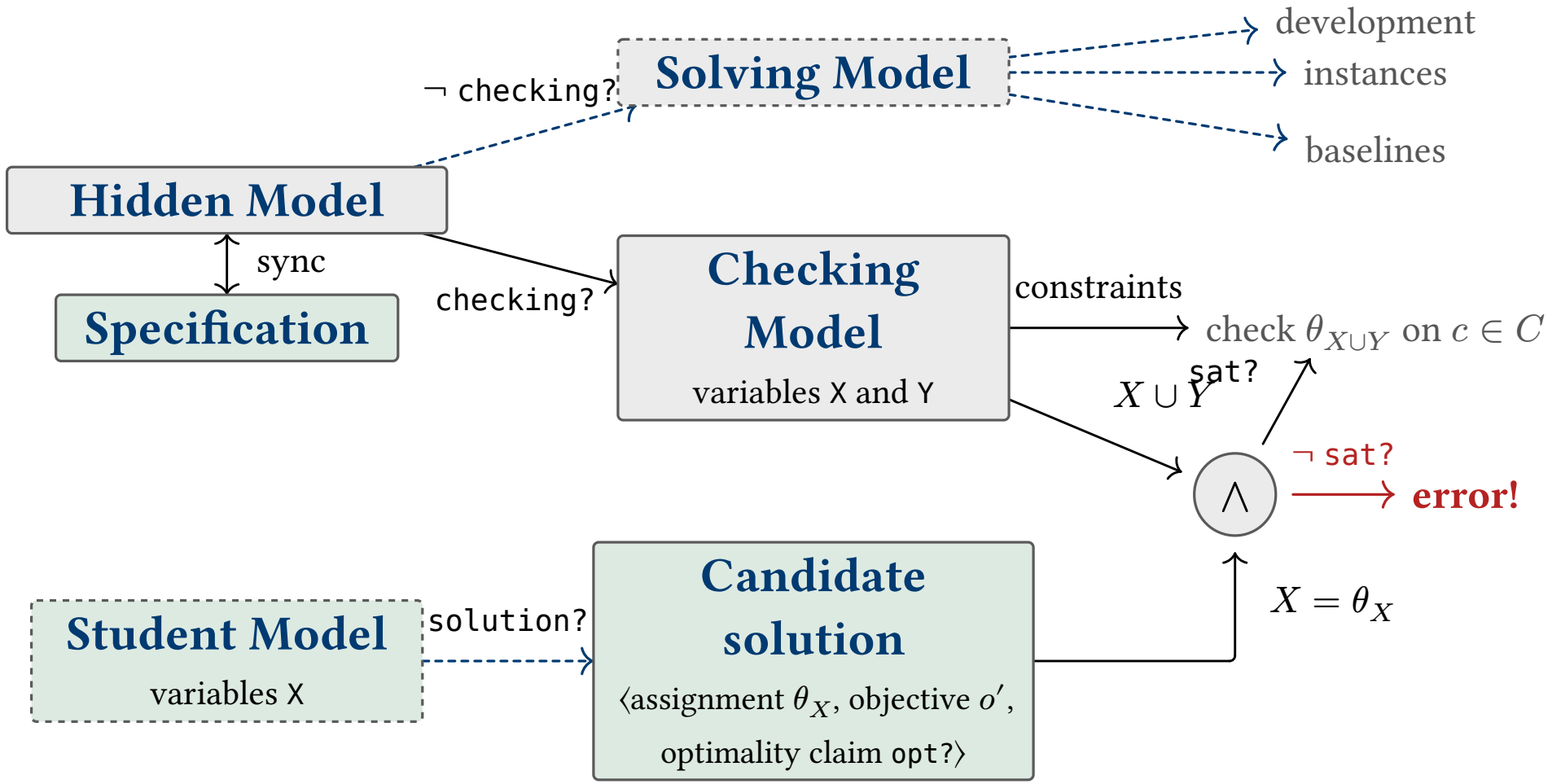
Flexible Jobshop Scheduling (FJSS) specification



Flexible Jobshop Scheduling (FJSS) specification



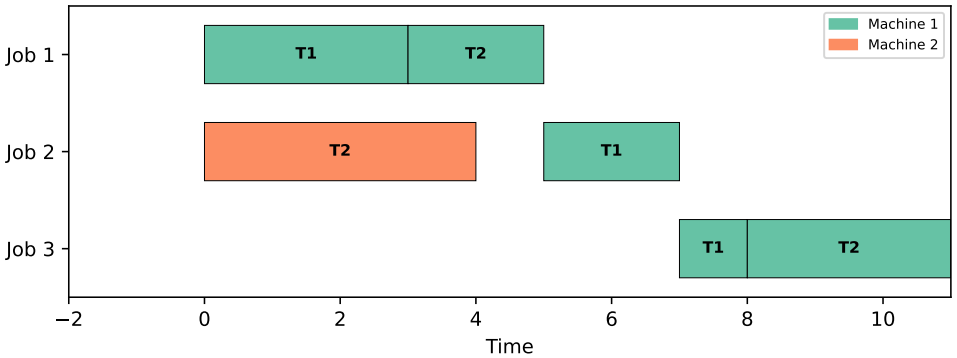
Flexible Jobshop Scheduling (FJSS) specification



"tasks within a job must be executed in order"

Feedback template uses assignment variable names

```
for i in range(n):
    for j in range(m - 1):
        self.add(X[i, j] + dur[i, j] <= X[i, j + 1], descr=f"Precedence: task
({i + 1},{j + 2}) starts at {{{X[i, j + 1].name}}} before task ({i + 1},{j +
1}) ends at {{{X[i, j].name}}}+{dur[i, j]}")
```

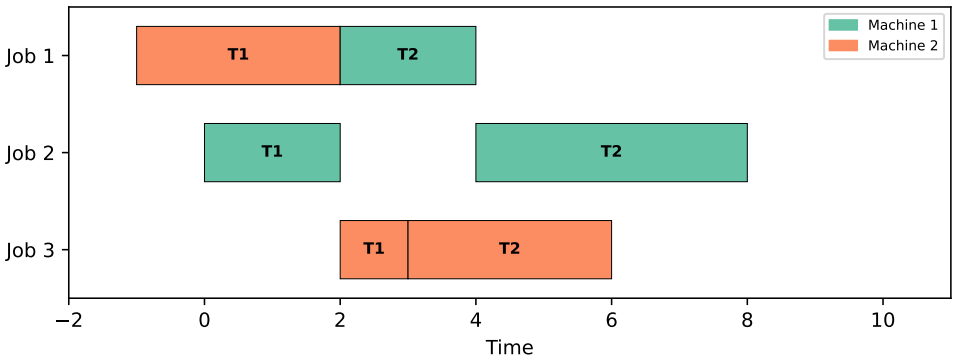


" Precedence: task (2,2) starts at 0 before task (2,1) ends at 5+2 "

"start times are non-negative"

Lower bounds are specified; upper bounds are not

```
if checking:
    for i in range(n):
        for j in range(m):
            self.add(X[i, j] >= 0, descr=f"Negative start time for task ({i +
1},{j + 1})")
```



" Negative start time for task (1,1) "

"tasks on the same machine cannot overlap"

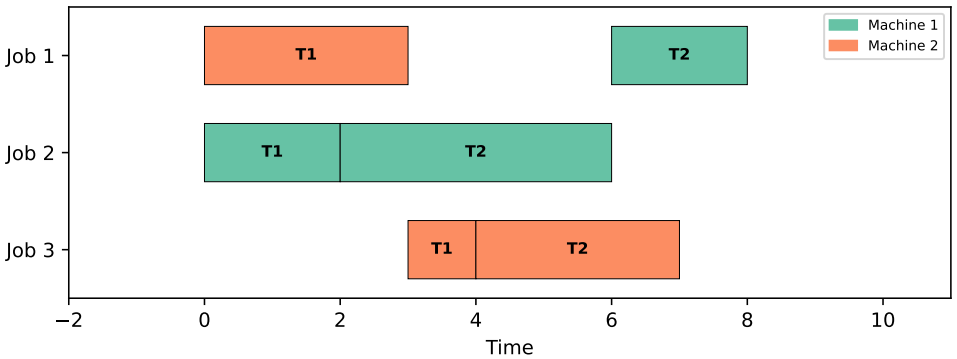
```
if not checking:
    for r in range(k):
        self.add(  # No two tasks on the same machine may overlap
            cp.NoOverlapOptional(X.flatten(), dur.flatten(), is_present=Y[:, :,
r].flatten())
        )
else:
    all_tasks = [(i, j) for i in range(n) for j in range(m)]
    for r in range(k):
        for (i1, j1), (i2, j2) in combinations(all_tasks, 2):
            self.add(
                (Y[i1, j1, r] & Y[i2, j2, r]).implies(
                    (X[i1, j1] + dur[i1, j1] <= X[i2, j2])
                    | (X[i2, j2] + dur[i2, j2] <= X[i1, j1])
                ),
                descr=f"Tasks ({i1 + 1},{j1 + 1}) and ({i2 + 1},{j2 + 1}) overlap on
machine {r + 1}",
            )
```

"minimize total flowtime (i.e. sum of end times)"

An *implicit* constraint is automatically added to check whether the candidate objective value o' matches the real objective f:

$$f = o'$$

This solution calculates makespan rather than total flowtime:



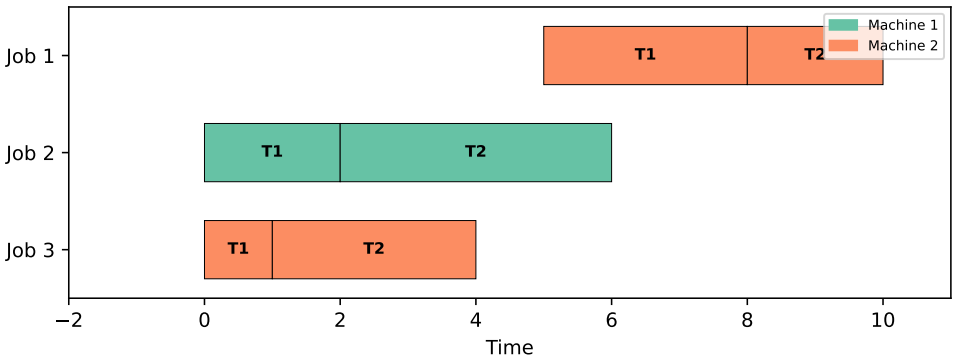
" Student objective value is 8, but hidden model computes a different value for the same assignment "

"minimize total flowtime (i.e. sum of end times)"

An implicit constraint is added when the student claims optimality:

$$f(\theta_X) \leq o$$

This solution finds a schedule with flowtime 20, but the baseline is 19:



" Student claims optimal objective 20, but baseline objective is 19 "

Hidden Variables

MiniZinc `photo.mzc`

```
int: n;
array[1..n] of int: pos;
array[1..n] of var 1..n: who;  % hidden variables
% photo.mzn: constraint inverse(pos, who);

test alldifferent(array[int] of int: x) =
  forall(i,j in index_set(x) where i < j)(x[i] != x[j]);
constraint if alldifferent(pos) then inverse(pos, who)
           else forall(i in 1..n)(who[i] = 1) endif;
```

- The specified variables `pos` connect to hidden variables `who` via an `inverse`
- Not enforcing `inverse` means `who` is unconstrained and assigned randomly
  - E.g. `pos = [3,1,2]; who = [1,1,1]` instead of `who = [2,3,1]`
- But enforcing `inverse` can lead to UNSAT
  - E.g. `pos = [2,1,2]; constraint inverse(pos, who)`
- **Solution**: enforce `inverse` but with an `alldifferent` *guard*

Hidden Variables

CPMpy `photo/project.py`

```
pos = cp.intvar(0, n - 1, shape=n, name="pos")
who = cp.intvar(0, n - 1, shape=n, name="who")  # Hidden variables
self.add(cp.AllDifferent(pos).implies(cp.Inverse(pos, who))
```

- *Hard* constraints are added without `descr`
- Guards can be added as constraints
- Or can be avoided by relying on the **sequential** execution of the checker:

Hidden Variables

CPMpy `photo/project.py`

```python
pos = cp.intvar(0, n - 1, shape=n, name="pos")
who = cp.intvar(0, n - 1, shape=n, name="who")  # Hidden variables
self.add(cp.Inverse(pos, who))
```

- *Hard* constraints are added without `descr`
- Guards can be added as constraints
- Or can be avoided by relying on the **sequential** execution of the checker:

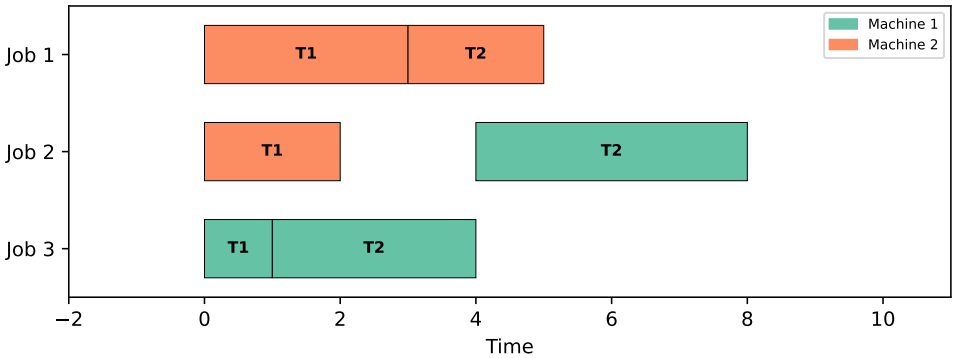
$X = \theta_X \longrightarrow C_{\text{soft},1} \longrightarrow C_{\text{hard},1}$ (e.g. `Inverse`) $\longrightarrow C_{\text{soft},2} \longrightarrow \cdots \longrightarrow (f = o') \wedge (f(\theta_X) \leq o)$

"tasks on the same machine cannot overlap"

No guard is possible for the task-machine variables, `Y`. Instead, we compute a **Maximally Satisfiable Subset** to find the *least* violating assignment

- zero violations if θ_X was correct
- bonus: a favourable assignment of `Y` if θ_X was incorrect

```
Y = cp.boolvar(shape=(n, m, k), name="Y")
for i in range(n):
    for j in range(m):
        self.add(sum(Y[i, j, :]) == 1)
```



" Tasks (1,1) and (2,1) overlap on machine 2 "

Obfuscation

Sharing the checking model allows *local* checking but reveals the solution.

Obfuscation through encryption is **reversible**: the key is shared with the lock

However, **equivalence-preserving mutations** (also used in the CPMpy fuzz-tester [2]) are irreversible

- Shuffle constraint order
- Shuffle commutative arguments
- Flip comparisons ($a \leq b \equiv b \geq a$)
- Swap (non-)strict ($a \leq b \equiv a < b + 1$)
- Add constant ($a \leq b \equiv a + k \leq b + k$)

Obfuscation

Sharing the checking model allows *local* checking but reveals the solution.

Obfuscation through encryption is **reversible**: the key is shared with the lock

However, **equivalence-preserving mutations** (also used in the CPMpy fuzz-tester [2]) are irreversible

- Shuffle constraint order
- Shuffle commutative arguments
- Flip comparisons ($a \leq b \equiv b \geq a$)
- Swap (non-)strict ($a \leq b \equiv a < b + 1$)
- Add constant ($a \leq b \equiv a + k \leq b + k$)

```
Constraints (45):
  (sum(np.int64(3), 40, X[0,0])) < (sum(1, X[0,1], 40))
    description: Precedence: task (1,2) starts at
{X[0,1]} before task (1,1) ends at {X[0,0]}+3
  76 <= (X[1,0]) + 76
    description: Negative start time for task (2,1)
  ...
  sum(Y[2,0,0], Y[2,0,1], -3) == -2
  sum(Y[0,1,0], 40, Y[0,1,1]) == 41
  ...
  ((Y[2,0,0]) and (Y[0,1,0])) -> ((((X[0,1]) + 2) < (1 +
(X[2,0]))) or ((1 + (X[2,0])) < (1 + (X[0,1]))))
    description: Tasks (1,2) and (3,1) overlap on machine
1
  ...
```

Checker Performance using CP-SAT [3]

n, m, k, min_dur, max_dur	Solve (s)	Check (s)
3, 2, 2, 1, 5	0.01	0.02
4, 3, 3, 2, 6	0.04	0.06
5, 3, 3, 3, 8	12.3	0.18
5, 4, 3, 3, 8	60.0†	0.32
6, 4, 3, 3, 8	60.0†	0.48

- Checking FJSS completes fast enough for *interactive* use
- But to grade performance for *easier* problems, we require *large* instances
  - The checker binary becomes *large* and *slow*
  - E.g. non-trivial instances of JSS with a makespan objective are too large

Qualitative feedback: 20/29 positive comments mention the projects

Conclusion and Future work

Conclusion

- One hidden CPMpy model serves as both **solver** (`checking=False`) and **checker** (`checking=True`)
- New strategies to resolve hidden variables
- Less thorough but more **irreversible** obfuscation for local checking
- Basic random instance generation, specification through doc-strings, student boilerplate, checker CLI, public project utilities (e.g. `visualize`), weighted grading, student leaderboard, …

Future work

- CNF-level obfuscation
- AutoIG for instance generation
- LLM-proof exercises..?

Questions?

Paper Code Slides Course hbierlee.one

Bibliography

[1] C. Coffrin, P. J. Stuckey, S. Liu, and G. Tack, "Solution checking with MiniZinc," in *Proceedings of the 16th workshop on Constraint Modelling and Reformulation at CP (ModRef 2017)*, 2017.

[2] W. Vanroose, I. Bleukx, J. Devriendt, D. Tsouros, H. Verhaeghe, and T. Guns, "Mutational Fuzz Testing for Constraint Modeling Systems," in *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, Girona, Spain, September 2-6, 2024*, P. Shaw, Ed., in LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, p. 29:1–29:25. doi: 10.4230/LIPICS.CP.2024.29.

[3] L. Perron and F. Didier, "CP-SAT." [Online]. Available: https://developers.google.com/optimization/cp/cp_solver/