An Answer Set Programming Approach for Academic Event Allocation: A Case Study at the Federal University of Ceará

Thiago Pereira Valente
Russas Campus
Universidade Federal do Ceará
Russas, Ceará, Brazil
thiagop00@alu.ufc.br

Pedro Henrique Ferreira
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, São Paulo, Brazil
pedrohferreira@ime.usp.br

Rildian Beserra Maia
Russas Campus
Federal University of Ceará
Russas, Ceará, Brazil
rildian@alu.ufc.br

Dmontier Aragão
Russas Campus
Federal University of Ceará
Russas, Ceará, Brazil
dmontier.aragao@ufc.br

Alexandre Matos Arruda
Quixadá Campus
Federal University of Ceará
Quixadá, Ceará, Brazil
alexandre.arruda@ufc.br

Abstract

Managing the allocation of oral presentations in large academic events is an inherently combinatorial and computationally complex task that involves satisfying numerous hard constraints, including room availability, time slots, and evaluator expertise. This paper proposes a declarative approach using Answer Set Programming (ASP) to model and solve the scheduling problem for the University Meeting at the Federal University of Ceará (UFC). The proposed workflow introduces a clear separation between instance data and allocation rules by utilizing a pre-processing stage that translates JSON-formatted event specifications into ASP facts. By explicitly encoding institutional rules and logistical requirements as logical integrity and choice constraints, the system systematically generates valid allocations that respect authorship conflicts, balance evaluator workloads, and accommodate advisor participation. The model was evaluated on both synthetic benchmarks and a real-world instance comprising 183 presentations, successfully producing a conflict-free schedule in approximately 30 minutes. Preliminary results demonstrate that this ASP-based approach not only significantly reduces the manual effort traditionally required by organizing committees, but also scales effectively, ensuring the feasibility, modularity, and adaptability of the final scheduling system for future editions.

Keywords

Answer Set Programming, Timetabling, Academic Event Allocation, Declarative Programming, Constraint Satisfaction, Clingo

1 Introduction

Academic conferences and university events serve as important environments for scientific dissemination, academic integration, and community engagement. However, the logistical organization behind such events, especially the allocation of hundreds of presentations into limited rooms, time slots, and evaluator groups, presents a significant scheduling challenge. This task is inherently combinatorial, since each allocation decision may interact with several others through conflicts involving rooms, authors, advisors, evaluators, availability, and workload distribution. When performed manually, the process is time-consuming, difficult to verify, and prone to errors, such as assigning an evaluator to simultaneous presentations or scheduling two presentations by the same author at the same time.

Known by different names across institutions, annual university meetings are held at universities throughout Brazil. Their primary objective is to provide students with a formal venue to present the projects, research activities, and academic work they have developed throughout the year to both the university community and the general public. These gatherings are fundamental for fostering academic collaboration, showcasing student research, and connecting the campus with the broader society. They are particularly oriented toward scholarship holders, who are institutionally required to present the results of their funded projects. Nevertheless, these events are open to all students, which fosters a collaborative environment in which diverse initiatives can be shared, discussed, and evaluated. Although these events encompass multiple categories, this paper focuses on the allocation of oral presentations because of the greater scheduling complexity arising from the larger number of presentations, rooms, and time slots.

The allocation of oral presentations can be viewed as a form of academic event timetabling. Each presentation must be assigned to exactly one time slot and one room, while evaluators must be distributed according to institutional rules. In addition to basic room and time-slot availability, the schedule must prevent authorship conflicts, avoid assigning advisors to evaluate their own advisees, balance evaluator workloads, and preserve logistical feasibility during the event. These requirements make the problem difficult to address using ad hoc manual procedures or rigid imperative algorithms, especially because the rules may change between different editions of the event.

To address this challenge, we propose an automated solution based on Answer Set Programming (ASP), a declarative programming paradigm grounded in the stable model semantics of logic programming [10], which is more thoroughly explained in Subsection 2.2. Beyond being a solver-oriented technology, ASP offers a high-level modeling language for expressing complex constraints in a compact and readable form. This makes it particularly suitable for institutional scheduling problems, where the main difficulty lies not in enumerating candidate solutions, but in accurately capturing the rules that define valid schedules. ASP has been applied successfully

to different academic scheduling scenarios, including curriculum-based course timetabling, departmental course scheduling, personalized course schedule planning, and real university timetabling deployments [2, 6, 8, 11]. These studies demonstrate that ASP can provide compact encodings, support institution-specific constraints, and generate valid schedules through solver execution.

This work is related to prior course timetabling approaches [1–3, 8, 11], but its focus is fundamentally different. Rather than allocating recurring weekly lectures for student cohorts, it addresses the single-occurrence allocation of oral presentations in an academic event. The model considers unique structural characteristics, such as strict authorship concurrency limits and highly localized conflict-of-interest rules involving advisors and evaluators. The main contribution is therefore the adaptation of an ASP-based timetabling approach to a real event-allocation scenario, providing novel modeling insights such as (i) a dynamic workload balancing formulation that adapts to varying evaluator-to-presentation ratios using flexible bounds, (ii) specific encoding patterns for institutional requirements, including impartiality and advisor-advisee networks, and (iii) a preliminary evaluation of the ASP-based model.

Beyond these specific contributions, a key advantage of the proposed methodology is the clear separation between event data and allocation rules, which makes the system easier to inspect, modify, and reuse in future editions. In this sense, the paper highlights not only a practical scheduling application, but also the suitability of ASP as a declarative language for modeling complex real-world allocation rules.

The remainder of this paper is organized as follows. Section 2 presents the theoretical background on event allocation, Answer Set Programming, and related work in academic timetabling. Section 3 describes the proposed methodology, including data pre-processing, the ASP encoding, constraints, and solver execution. Section 4 presents the preliminary results obtained from the UFC case study. Finally, Section 5 concludes the paper and discusses directions for future work.

2 Literature Review

2.1 Event Allocation

Event allocation can be understood as a timetabling problem in which a set of activities must be assigned to available resources and time periods while satisfying a collection of constraints. In academic contexts, this problem appears in course timetabling, exam timetabling, curriculum-based course timetabling, and event scheduling. Although the concrete entities may vary, the underlying structure is similar: lectures, exams, or presentations must be placed into rooms and time slots, and additional constraints involving teachers, students, evaluators, or institutional rules must be respected.

Timetabling problems are combinatorial in nature and are commonly described as computationally difficult. Practical systems for course scheduling emphasize that timetable generation is NP-hard or NP-complete, which helps explain why manual construction becomes costly and error-prone as the number of activities, resources, and restrictions increases [6, 13]. In universities, the difficulty is not limited to finding an available room and time slot. A valid schedule must also avoid resource conflicts, respect participant availability, distribute workloads, and remain adaptable when data change close to the event. Recent advances in the field have explored a wide range of algorithmic strategies to address the inherent complexity of university course timetabling. These approaches include metaheuristics such as Genetic Algorithms [5], Simulated Annealing [4], and Tabu Search [7], graph-based and event-grouping approaches [12], as well as declarative paradigms such as Answer Set Programming [2], which is the strategy adopted in this paper. While metaheuristics scale efficiently, they often rely on penalty functions that yield approximate schedules. ASP was chosen because it strictly enforces all hard constraints, guaranteeing a perfectly conflict-free solution quality. Although ASP's worst-case runtime is higher, its declarative expressiveness offers a decisive advantage for managing complex academic rules.

In the specific case of academic presentation events, each presentation must be scheduled exactly once, usually in one room and one time slot. Two presentations cannot occupy the same room at the same time, an evaluator cannot be assigned to simultaneous presentations, an advisor should not evaluate the work of their own advisee, and presentations by the same author should not occur simultaneously. These constraints are analogous to hard constraints studied in course and curriculum timetabling, where infeasible combinations must be excluded from the final schedule [1, 2].

2.2 Answer Set Programming

Answer Set Programming (ASP) is a declarative programming paradigm based on the stable model semantics of logic programming, originally introduced by Gelfond and Lifschitz [10]. In ASP, a program describes a problem through logical statements, and the task of finding a solution is delegated to a solver. The solutions computed by the solver are called answer sets, or stable models. Each answer set represents a possible interpretation of the program that satisfies its facts, rules, and constraints.

Unlike imperative programming, ASP does not require the programmer to specify a sequence of instructions for constructing a solution. Instead, the programmer describes what properties a valid solution must have. This modeling style is especially useful for combinatorial search and constraint satisfaction problems, where the number of possible combinations is large and the main difficulty lies in eliminating invalid configurations [14, 15]. For this reason, ASP has been widely applied to scheduling, planning, resource allocation, configuration, and timetabling problems.

An ASP program is commonly composed of facts, rules, choice rules, and constraints. Facts represent information that is unconditionally true in a given problem instance. For example, in an academic event allocation problem, presentations, rooms, time slots, and evaluators can be represented as facts:

Listing 1: Examples of ASP facts

```
presentation(1).
room(1).
timeslot(1).
evaluator(e1).
```

The predicate `presentation(1)` states that there is a presentation identified by the constant 1. Similarly, `room(1)`, `timeslot(1)`, and `evaluator(e1)` declare the existence of a room, a time slot,

and an evaluator. In larger instances, ASP also allows compact interval notation, such as `presentation(1..148)`, to generate several facts at once.

Rules are used to derive new atoms from existing ones. A rule has a head and a body, separated by the symbol `:-`. The head is derived whenever the body is satisfied. For example:

Listing 2: Example of an ASP rule

```
available_room(R) :-
    room(R),
    not blocked(R).
```

This rule states that a room `R` is available if `R` is a room and there is no evidence that it is blocked. The expression `not` denotes default negation, also known as negation as failure. It does not mean classical logical negation; rather, it means that the atom cannot be derived from the available information. This closed-world style of reasoning is useful in allocation problems, where the input data define the known entities and relationships of the instance.

Choice rules are one of the most important ASP constructs for scheduling problems. They allow the solver to choose atoms from a set of alternatives, possibly under cardinality bounds. For instance:

Listing 3: Example of an ASP choice rule

```
{ assign(P,T,R) :
    timeslot(T),
    room(R)
} ← 1 :- presentation(P).
```

This rule states that, for each presentation `P`, the solver must choose exactly one atom of the form `assign(P,T,R)`, where `T` is a time slot and `R` is a room. In other words, each presentation must be assigned to exactly one time slot–room pair. The rule does not prescribe how the pair should be selected; it only defines the space of possible assignments and the required cardinality. The solver is responsible for searching among these possibilities while respecting the remaining constraints of the program.

Hard constraints, also called integrity constraints, are rules without a head. They are used to eliminate invalid answer sets. For example:

Listing 4: Example of an ASP integrity constraint

```
:- assign(P1,T,R),
   assign(P2,T,R),
   P1 ≠ P2.
```

This constraint forbids any answer set in which two different presentations `P1` and `P2` are assigned to the same time slot `T` and the same room `R`. Since the rule has no head, it does not derive a new atom. Instead, it rejects any candidate solution satisfying its body. This mechanism is central to ASP modeling: valid schedules are obtained not by constructing them procedurally, but by defining possible choices and excluding the combinations that violate the problem requirements.

ASP also supports cardinality constraints and aggregates, which are useful for expressing workload and counting requirements. For example:

Listing 5: Example of a cardinality constraint

```
2 { evaluator_presentation(E,P) :
    evaluator(E)
} 2 :- presentation(P).
```

This rule states that each presentation `P` must be assigned exactly two evaluators. The expression between braces defines the set of candidate evaluator assignments, and the bounds indicate that exactly two of them must be selected. Similar constructs can be used to ensure that each evaluator receives a balanced number of presentations or that coordinators and committee members receive a specific workload.

This modeling style is particularly suitable for timetabling problems because the domain can be represented as facts, while scheduling decisions and feasibility conditions can be encoded separately. Facts may describe presentations, rooms, time slots, evaluators, advisors, and authorship relations. Choice rules define possible assignments, while integrity constraints eliminate invalid schedules. Applied ASP timetabling works use this same structure to encode hard constraints, aggregates, and, when needed, soft constraints or penalty atoms for optimization [1, 3]. Furthermore, the use of ASP allows for a "design space approach," where the interaction between different constraints can be systematically analyzed and refined to meet complex institutional requirements [16].

The ASP workflow also helps separate instance data from the general allocation model. Systems such as *teaspoon* convert curriculum-based course timetabling instances into ASP facts and combine them with reusable encodings before calling a solver [2]. This separation is important for academic event allocation because the rules of the model may remain stable across event editions, while the input data, such as the list of presentations, rooms, evaluators, and time slots, change each year.

2.3 Clingo

The ASP model proposed in this work was implemented and executed using *clingo*, one of the main systems of the Potassco (Potsdam Answer Set Solving Collection) project. Clingo is an integrated ASP environment that combines the functionalities of the *gringo* grounder and the *clasp* solver into a unified system [9]. Its primary purpose is to compute answer sets for logic programs that model combinatorial search and optimization problems.

The execution pipeline of clingo is divided into two main stages: grounding and solving. In the grounding stage, the input program, which may contain variables and high-level first-order rules, is transformed into an equivalent propositional representation through the *gringo* grounder. This transformation is necessary because ASP solvers operate on variable-free logic programs [17]. The resulting grounded program is then processed by the *clasp* solver, which searches for stable models (answer sets) satisfying all rules and integrity constraints.

One of the main advantages of clingo lies in its tight integration between grounding and solving. Unlike traditional approaches in which the grounding and solving phases are executed independently, clingo allows both processes to interact more closely, enabling advanced reasoning strategies such as incremental and multi-shot solving [9]. In this paradigm, the logical program can evolve during execution through the addition or modification of facts and

constraints without requiring the entire solving process to restart from scratch.

Another relevant feature of clingo is its support for embedded scripting, particularly through Python and Lua interfaces. These interfaces allow ASP programs to be integrated with external applications and facilitate tasks such as instance generation, result processing, and dynamic control of the solving workflow [18]. In the context of this work, clingo was used as the core reasoning engine responsible for generating feasible allocations that satisfy all scheduling constraints defined in the ASP model.

Furthermore, clingo has been successfully applied in several domains involving complex combinatorial reasoning, including university timetabling, configuration, robotics, planning, and resource allocation [2]. Its expressive declarative language, combined with efficient solving capabilities, makes it particularly suitable for academic event allocation problems such as the one addressed in this paper.

3 Methodology

3.1 Overview of the Approach

The proposed approach treats the event allocation problem as a constraint satisfaction task. The real-world problem is transformed into an ASP model, where the event data are represented as facts and the logistical requirements as integrity constraints. The general workflow, illustrated in Figure 1, consists of five main stages: (i) data input, which receives a JSON file containing the instance parameters necessary for the correct execution of the algorithm, (ii) pre processing where all the data from the JSON, through a python auxiliary script, is converted into an ASP program, that is used to build the model, with all its predicates, constraints and particularities, (iii) model building which is completely done by the ASP program built in the last step, (iv) solver execution where the Clingo solver, using the model built from all the rules defined in the ASP program, searches for a valid allocation and (v) output, the result of the execution of the solver, containing a valid allocation, if possible.

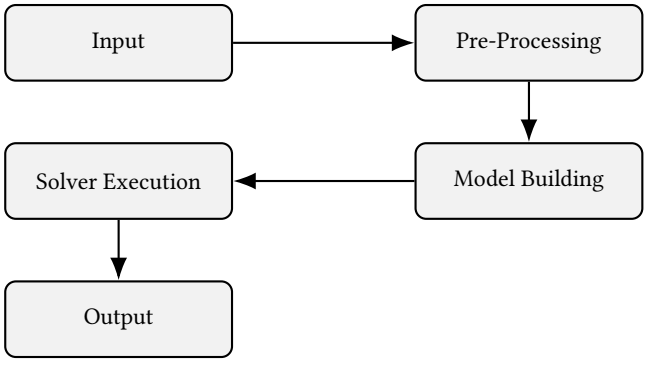

Figure 1: Overview of the proposed ASP-based allocation workflow.

By separating the specific event data from the general allocation rules, the model ensures high flexibility. The core of this proposal lies in the declarative modeling of the problem, allowing the solver to explore the combinatorial search space efficiently rather than relying on a fixed procedural algorithm.

3.2 Input Data

The input data is provided in JavaScript Object Notation (JSON) format, containing all essential information required for the construction of the ASP program, namely:

- The number of presentations;
- The number of available rooms;
- The number of timeslots;
- The set of shifts, represented as a mapping
$$\mathcal{S} = \{(s_i, p_i)\}_{i=1}^{n},$$
where each pair associates a shift identifier s_i with a value p_i, corresponding to the number of timeslots belonging to that shift. Thus, if
$$\mathcal{S} = \{(1, 4), (2, 6), (3, 5)\},$$
then the event has three shifts, the first shift containing 4 timeslots, the second shift containing 6 timeslots and the third shift containing 5 timeslots;
- The number of evaluators required for each presentation;
- The set of advisor–presentation relations, represented as a mapping
$$\mathcal{A} = \{(a_i, P_i)\}_{i=1}^{m},$$
where each pair associates an advisor a_i with a set of presentations P_i supervised by that advisor. Thus, if
$$\mathcal{A} = \{(a_1, \{1, 2\}), (a_2, \{3, 4, 5\})\},$$
then there's two advisors and advisor a_1 supervises presentations 1 and 2, while advisor a_2 supervises presentations 3, 4, and 5.
- A list containing all evaluators.
- A list containing all postgraduate evaluators.
- A list containing all evaluators who are also committee members.
- A list containing all evaluators who are also coordinators.
- A list containing all pairs of presentations that are authored by the same person.

All this information is necessary for the correct construction of the ASP program and will be thoroughly explained later in subsections 3.4 and 3.5.

3.3 Pre-Processing

The pre-processing stage connects the institutional instance data to the ASP encoding used by the solver. Its role is to transform the event description, provided in JSON format, into a set of ASP facts that can be directly grounded and solved by *clingo*. By isolating data preparation from the logical model, this stage improves modularity and allows the same allocation rules to be reused across different editions of the event with minimal changes.

The auxiliary script reads the JSON file containing the instance description and generates a new ASP file from scratch. It first writes the facts that characterize the specific event, including the number of presentations, rooms, time slots and shifts, the advisor–presentation relations, the evaluator categories, committee membership, coordination roles and pairs of presentations authored

by the same student. These facts constitute the input instance on which the general allocation rules operate.

A notable aspect of this step is the automatic construction of the shift predicates. Rather than hardcoding shift boundaries in the ASP program, the script derives them from the JSON structure and emits the corresponding facts programmatically. This makes the model more adaptable to event editions with different shift configurations. The script also computes the workload parameters required by the evaluator-assignment rules, ensuring that the generated ASP instance remains consistent with the size and composition of the current event.

After generating all the facts, the script appends the general timetabling rules and any additional optional constraints, and finally includes the directive specifying the output predicate to be displayed by the solver. The resulting file therefore contains both the event data and the allocation model in a single executable ASP program.

This pre-processing stage reduces manual intervention, avoids repetitive editing of ASP files, and supports the reuse of the same logical encoding across multiple instances. In this way, the proposed workflow separates data management from constraint modeling while preserving the declarative structure of the allocation problem. It is also important to emphasize that all reasoning is performed entirely by Clingo, as the Python script acts solely as a translator from raw input data into an ASP model.

3.4 Predicates Definition

At the outset, it is necessary to generate all predicates that hold prior to the execution of the solver and the application of the logical rules. These initial facts describe the concrete elements of the event being scheduled and serve as the foundational knowledge upon which the reasoning process is built. Because these predicates depend directly on the characteristics and requirements of each specific event, this phase cannot be fully automated. Instead, it must be adapted to the unique configuration of the event in question, ensuring that the solver receives an accurate and complete representation of the problem domain.

To fully represent a specific event, the model requires information about the available resources and participants involved in the scheduling process. This includes the number of presentations, rooms, and time slots, the shifts in which the event takes place, as well as the identification of evaluators, advisors, authors, coordinators, and committee members. These elements are encoded as ASP predicates and constitute the instance data of the problem. The input data are initially provided in JavaScript Object Notation (JSON) format. Before the execution of the solver, a pre-processing script converts this structured input into ASP predicates that can be interpreted by the model. This conversion step is responsible only for translating and organizing the event data; the allocation itself is performed exclusively by the ASP solver through the application of the rules and constraints defined in the program.

In code 6, where P is the number of presentations that must be assigned to a room and a time slot, R is the number of rooms available for scheduling and T is the number of time slots during which the event takes place, each line assigns truth to a set of fundamental predicates. For example, the first line states that

$$\forall x \mid 1 \leq x \leq P,\ \text{presentation}(x)$$

and, since no truth values are assigned to presentations outside this range, it also follows that

$$\forall x \mid x < 1 \ \vee \ x > P,\ \neg\text{presentation}(x)$$

The same reasoning applies to `room(x)` and `timeslot(x)`.

Listing 6: Fundamental predicates

```
1 presentation(1..P).
2 room(1..R).
3 timeslot(1..T).
```

In code 7, the set of predicates provides a partition of the base predicate timeslot(x) into disjoint shifts. In other words, if both timeslot(t) and timeslot_shift$_s$(t) are true, it means that the timeslot t belongs to the shift s. It is also important to note that p_i denotes the last time slot that belongs to shift i, while $p_0 = 0$ by definition.

For the University Meeting, the schedule consisted of four shifts: Wednesday afternoon, Thursday morning, Thursday afternoon, and Friday morning. However, different events and even different editions of the same event may involve different sets of shifts, so introducing a general formulation is appropriate.

This set of predicates is not required for the algorithm to function, but is useful to optimize the logistics of the event. The underlying idea is that if an evaluator is assigned to a room during a given shift, they remain in that same room for the entire shift. This constraint prevents evaluators from having to move between rooms, thereby reducing delays and improving the overall efficiency of the event.

Listing 7: Shifts definition

```
1 timeslot_shift_i(p_{i-1} + 1..p_i).
```

In code 8, the predicates describe additional relationships and classifications required by the allocation model. The predicate advisor_presentation(A, P) states that advisor A supervises presentation P, while same_author(P_1, P_2) states that presentations P_1 and P_2, with $P_1 \neq P_2$, were authored by the same student.

The remaining predicates classify the people who may participate in the evaluation process. The predicate evaluator(E) declares that E is eligible to evaluate presentations. The predicate postgraduate(Pg) identifies postgraduate evaluators, which is necessary when postgraduate presentations must be evaluated only by postgraduate evaluators. Finally, committee_member(Cm) and coordinator(C) identify evaluators with specific organizational roles, allowing the model to assign them different workloads when necessary.

Listing 8: Instance-specific predicates

```
1 advisor_presentation(A, P).
2 postgraduate(Pg).
3 evaluator(E).
4 committee_member(Cm).
5 coordinator(C).
6 same_author(P_1, P_2).
```

3.5 Constraints

This subsection represents the core of the proposed algorithm, as it formally specifies all logical constraints that govern the behavior of the allocator. These constraints capture the fundamental rules and requirements of the event timetabling problem, ensuring that every solution produced is valid and consistent.

In particular, the constraints define which combinations of entities are admissible and explicitly forbid assignments that lead to conflicts or violations of the problem specifications. They are responsible for enforcing essential properties such as uniqueness, exclusivity, and dependency relations among elements such as presentations, rooms, time slots and evaluators.

By constraining the solution space, the allocator is guided toward feasible and coherent allocations, while invalid or inconsistent configurations are systematically eliminated. Consequently, the set of constraints plays a crucial role not only in guaranteeing the correctness of the solutions, but also in influencing the efficiency of the search process and the overall quality of the resulting allocations.

In code 9, it is first enforced that each presentation P is assigned to exactly one time slot–room pair, thus preventing a presentation from being allocated to multiple time slots or rooms. Furthermore, it defines the predicate `presentation_timeslot_room(`P`, `T`, `R`)`, which represents the association between a presentation P and its assigned time slot T and room R. This predicate is essential for the correct functioning of the algorithm, as it is used in subsequent constraints.

After that, it is also eliminated all cases in which different presentations are scheduled in the same time slot and room, which would result in a conflict between presentations. By enforcing this constraint, it is guaranteed that each time slot–room pair is assigned to at most one presentation.

With this, it is established a one-to-one correspondence between presentations and time slot–room pairs. In code 9, the first line guarantees that every presentation is assigned exactly one available time slot and room, ensuring completeness of the allocation and the second line enforces exclusivity by preventing multiple presentations from being scheduled in the same time slot and room combination. As a result, the allocation process produces conflict-free schedules in which each presentation occupies a unique time slot–room pair, and each such pair is associated with at most one presentation. This combination of constraints is fundamental for maintaining the consistency, correctness, and feasibility of the overall scheduling solution.

Listing 9: Presentation allocation constraints

```
1 { presentation_timeslot_room(P, T, R) : timeslot(T
      ↪ ), room(R)} ← 1 :- presentation(P).
2 :- presentation_timeslot_room(P1, T, R),
      ↪ presentation_timeslot_room(P2, T, R),
      ↪ P1 ≠ P2.
```

In code 10, it is defined the evaluator assignment policy. In the first line, *Ne* denotes the number of evaluators that must be assigned to each presentation. This rule requires that, for each presentation P, exactly *Ne* evaluators are assigned to P. In the specific case of the UFC University Meetings, each presentation was evaluated by two evaluators. However, a generalized parameter was adopted since this number may vary depending on the event specifications.

In the second line, *Np_E* denotes the minimum number of presentations that each regular evaluator is required to evaluate. This rule ensures that, for every evaluator E who is neither a committee member nor a coordinator, the number of presentations assigned to E is bounded between *Np_E* and *Np_E* + 1. These bounds are introduced to ensure a balanced workload among evaluators and to avoid both under-utilization and overload of the evaluators. It is also important to note that the most overloaded evaluator has at most one more presentation than the most under-utilized one. The value of *Np_E* is calculated from the total number of presentations and the number of eligible evaluators, as follows:

$$Np\_E = \left\lfloor \frac{N_p}{N_e} \right\rfloor \tag{1}$$

In the remaining lines, *Np_Cm* and *Np_C* denote, respectively, the exact number of presentations that a committee member and a coordinator are required to evaluate. Since the number of committee members and coordinators is typically smaller than the number of regular evaluators, it is reasonable to assign them a fixed and exact workload.

In contrast, regular evaluators are assigned workload bounds rather than an exact number of presentations. This design choice is necessary because the total number of presentations may not be exactly equal to the sum of the workloads imposed by fixed assignments, namely

$$N_E \cdot Np_E + N_{Cm} \cdot Np_{Cm} + N_C \cdot Np_C,$$

where N_E, N_{Cm}, and N_C denote, respectively, the number of regular evaluators, committee members, and coordinators. Therefore, at least one category of evaluators must be subject to flexible bounds to guarantee the existence of a feasible allocation.

Regular evaluators were chosen to have bounded workloads because they usually constitute the largest group of evaluators, making them the most suitable candidates to absorb variations in the total number of presentations while maintaining a balanced and feasible assignment.

Listing 10: Workload constraints

```
1 { evaluator_presentation(E, P) : evaluator(E)} ←
      ↪ Ne :- presentation(P).
2 Np_E {evaluator_presentation(E, P) : presentation(
      ↪ P)} Np_E + 1 :- evaluator(E), not
      ↪ committee_member(E), not coordinator(E).
3 evaluator_presentation(Cm, P) : presentation(P) ←
      ↪  Np_Cm :- evaluator(Cm), committee_member(
      ↪ Cm).
4 evaluator_presentation(C, P) : presentation(P) ←
      ↪ Np_C :- evaluator(C), coordinator(C).
```

In code 11, the first line prevents an evaluator from being assigned to more than one presentation occurring at the same time slot and in the same room. By eliminating such assignments, it avoids situations in which an evaluator would be required to evaluate multiple presentations simultaneously at the same location,

thus ensuring temporal and spatial consistency in the evaluation schedule.

Moreover, the second line entails that an evaluator can be assigned to at most one room within a given time slot. It explicitly forbids assignments in which the same evaluator is associated with multiple rooms at the same time, which would be physically infeasible.

Finally, the third line guarantees that two presentations authored by the same student are not scheduled in the same time slot and room, thus avoiding simultaneous presentations. Together, these three constraints are essential in ensuring a valid, conflict-free schedule.

Listing 11: Spatio-temporal allocation constraints

```
1 :- evaluator_presentation_timeslot_room(E,P1,T,R),
      ↪ evaluator_presentation_timeslot_room(E, P2
      ↪ , T, R), P1 ≠ P2.
2 :- evaluator_timeslot_room(E, T, R1),
      ↪ evaluator_timeslot_room(E, T, R2), R1 ≠ R2.
3 :- same_author(P1, P2), presentation_timeslot_room
      ↪ (P1, T, R), presentation_timeslot_room(P2,
      ↪ T, R).
```

In code 12, it is enforced that an advisor cannot be assigned as an evaluator of their own advisee's presentation. This constraint prevents conflicts of interest and helps ensure fairness and impartiality throughout the evaluation process, which is a standard requirement in academic assessment settings.

Listing 12: Impartiality constraints

```
1 :- advisor_presentation(A, P),
      ↪ evaluator_presentation(E, P), A = E.
```

In code 13, the first line precludes the simultaneous scheduling of multiple presentations overseen by the same advisor. This rule ensures the physical feasibility of the advisors' presence in every presentation they supervise. Analogously, the second line precludes an advisor from serving as an evaluator in a timeslot that coincides with an advisee's presentation.

While these constraints are not strictly mandatory, as the original event specifications did not necessitate advisor attendance, they were incorporated to align the schedule with institutional preferences that encourage supervisor participation. From a modeling perspective, these rules function as optional integrity constraints, as they enhance the quality of the generated schedule by ensuring compatibility with advisor availability. Furthermore, the modular nature of this approach allows these constraints to be omitted in future iterations without compromising the system's core logic or structural integrity.

Listing 13: Advisor constraints

```
1 :- advisor_presentation(A, P1),
      ↪ advisor_presentation(A, P2),
      ↪ presentation_timeslot(P1, T),
      ↪ presentation_timeslot(P2, T), P1 ≠ P2.
2 :- advisor_advises_presentation(A, P),
      ↪ evaluator_timeslot(A, T),
      ↪ presentation_timeslot(P, T).
```

3.6 Execution and Output

The model was implemented using the *clingo* solver, part of the Potassco suite. The execution follows the standard ASP grounded-and-solve pipeline, where the grounder (*gringo*) transforms the first-order encoding into a propositional format, and the solver (*clasp*) searches for valid answer sets. Each returned answer set represents a conflict-free allocation that satisfies all defined hard constraints.

The raw output of the solver, consisting of atom sets, is converted back into a structured format through a post-processing script. This script groups the allocated presentations by room and time slot and produces a finalized table containing the presentation title, schedule, room number, and assigned evaluators. Currently, this conversion generates a formatted text output, and the automatic generation of Excel spreadsheets or PDF reports is planned for future versions of the system.

4 Results and Discussions

The model was evaluated through a benchmarking procedure implemented in an auxiliary Python script. The results are presented in Figure 2. The evaluation aimed to analyze the scalability of the ASP encoding under increasingly larger problem instances while preserving the structural characteristics of the real-world dataset.

Starting from instances containing 16 presentations and progressively increasing up to 90 presentations, synthetic datasets were generated by maintaining the same proportions observed in the real event. More specifically, the ratio between presentations and each participant category was preserved throughout the entire process. For example, the original dataset contained 183 presentations and 73 evaluators, corresponding to an evaluator-to-presentation ratio of approximately 0.3989. Consequently, an instance with 100 presentations would contain 40 evaluators, obtained by rounding to the nearest integer. The same proportional scaling strategy was applied to postgraduate evaluators, committee members, coordinators, and presentations sharing the same author.

The number of rooms, timeslots, and shifts was kept constant throughout the benchmarking process. Specifically, all generated instances considered 8 rooms, 32 timeslots, and 4 shifts, each shift comprising 8 timeslots. These values closely reflect the characteristics of the real-world dataset and were intentionally fixed to isolate the impact of increasing the number of presentations on solver performance.

To preserve the structure of authorship conflicts observed in the original instance, the proportion of presentations sharing the same author was also maintained. Whenever the scaling procedure required an increase in this category, two presentations that were previously associated with distinct authors were randomly selected and reassigned to the same author. For simplicity, each author was allowed to be associated with at most two presentations. Although this assumption does not cover all possible real-world scenarios, it was sufficient to reproduce the conflict patterns observed in the case study while keeping the instance generation process straightforward and controlled.

Finally, the number of evaluators required for each presentation was fixed at two, matching the evaluation policy adopted in the real event.

For each presentation count, five independent random instances were generated. Each instance was processed by the data preprocessing pipeline, translated into an ASP model, and solved using Clingo. The solver runtime was recorded for every execution. Subsequently, the mean runtime, as well as the minimum and maximum observed runtimes, were computed for each problem size. The resulting benchmark indicates a clear superlinear growth in runtime over the evaluated range. Nevertheless, all tested instances were solved within practical execution times, suggesting that the proposed ASP encoding remains suitable for instances of comparable size to the real-world case study.

All benchmark experiments were conducted on a machine equipped with an 11th Generation Intel Core i5-11400H processor running at 2.70 GHz, featuring 6 physical cores and 12 hardware threads, and 24 GB of RAM. No specialized hardware acceleration was employed. All ASP instances were solved using Clingo under the same hardware configuration to ensure consistency across the reported measurements.

The real-world instance considered in this study consisted of 183 presentations, 73 evaluators, 45 postgraduate evaluators, 10 committee members, 3 coordinators, and 6 pairs of presentations sharing the same author. The event infrastructure comprised 6 rooms and 32 time slots distributed across 4 shifts, containing 10, 9, 10, and 3 time slots, respectively.

The allocation of this instance required approximately 30 minutes to be produced. Although this runtime exceeds a direct extrapolation of the benchmark, which was limited to 90 presentations with fixed resources, this discrepancy arises from the real dataset's non-uniform density of interlocking constraints (e.g., highly concentrated advisor networks) compared to the uniform synthetic instances. Nevertheless, this represents a substantial improvement over the manual scheduling process traditionally employed by the organizing committee. According to the organizers, manual scheduling used to take several days of work and often resulted in undetected conflicts, making this 30-minute automated solution a massive reduction in administrative effort and, while potentially slow for interactive use, a perfectly acceptable runtime for an offline scheduling task. The ASP model successfully generated a valid schedule satisfying all hard constraints, including both the general institutional requirements and the additional instance-specific constraints introduced to capture particular availability restrictions and organizational preferences. No conflicts involving rooms, evaluators, advisors, or authors were detected in the resulting allocation.

5 Conclusions and Future Work

This paper presented an ASP-based approach to the academic event allocation problem, with a case study on the UFC event. The proposed model demonstrates that ASP is well suited for this type of problem, as it enables a compact and declarative representation of complex institutional rules, including room and timeslot assignment, workload balancing, authorship conflicts, and advisor-related restrictions. The separation between instance data and allocation rules also contributes to the modularity and adaptability of the approach, making it easier to reuse across different editions of the event, or even adapt to other institutions and different types of academic conferences by simply modifying the input data and specific constraint rules.

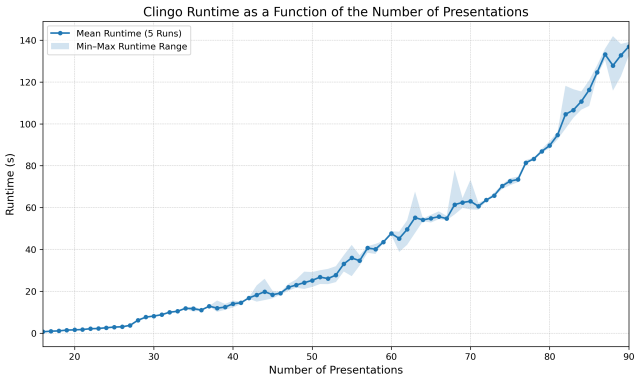

Figure 2: Clingo runtime as a function of the number of presentations. Each point represents the mean runtime over five randomly generated instances, and the shaded region denotes the observed minimum–maximum range.

The preliminary results indicate that the proposed solution is effective in producing valid schedules with low computational overhead. In addition, the benchmark analysis suggests that the ASP encoding remains tractable for the evaluated instances, which supports its practical use in real allocation scenarios.

Future work will focus on three main directions. First, we plan to develop a Python-based post-processing module capable of converting Clingo output into a structured JSON representation containing all generated allocations. This extension would make the system easier to integrate with external applications and would provide a natural path toward exposing the allocator as an API, in which a JSON instance is received as input and a JSON schedule is returned as output. Second, we intend to design a web-based frontend to transform the current prototype into a complete user-facing application, allowing event organizers to submit instances, inspect solver results, and visualize schedules interactively. Third, we plan to conduct a more detailed scalability analysis by varying one instance parameter at a time, such as the number of presentations, evaluators, or time slots, in order to better understand the behavior of the model under controlled growth conditions. Although this type of experiment was not performed in the present work due to its broader scope, it would provide a more fine-grained characterization of the performance of the proposed ASP encoding.

Furthermore, to address the current limitation of relying solely on hard constraints, the model will be extended to incorporate soft constraints using ASP's optimization capabilities, balancing room, time, and evaluator preferences to improve overall fairness. Additionally, to handle unsatisfiable scenarios where tight constraints prevent a valid allocation, future research will investigate mitigation strategies, such as the systematic relaxation of non-critical requirements.

Artifact Availability

The authors declare that the research artifacts supporting the findings of this study are accessible at https://doi.org/10.5281/zenodo.22149511.

References

[1] Irvi Firqotul Aini, Ari Saptawijaya, and Siti Aminah. 2017. Bringing Answer Set Programming to the Next Level: A Real Case on Modeling Course Timetabling. In *2017 International Conference on Advanced Computer Science and Information Systems*. IEEE, 471–476. doi:10.1109/ICACSIS.2017.8355076

[2] Mutsunori Banbara, Katsumi Inoue, Benjamin Kaufmann, Tenda Okimoto, Torsten Schaub, Takehide Soh, Naoyuki Tamura, and Philipp Wanko. 2019. teaspoon: Solving the Curriculum-Based Course Timetabling Problems with Answer Set Programming. *Annals of Operations Research* 275, 1 (2019), 3–37. doi:10.1007/s10479-018-2757-7

[3] Mutsunori Banbara, Takehide Soh, Naoyuki Tamura, Katsumi Inoue, and Torsten Schaub. 2013. Answer Set Programming as a Modeling Language for Course Timetabling. *Theory and Practice of Logic Programming* 13, 4–5 (2013), 783–798. doi:10.1017/S1471068413000495

[4] Edmund K Burke and James P Newall. 1995. A search methodology for examination timetabling. *Practice and Theory of Automated Timetabling* (1995), 279–293.

[5] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. 1998. A genetic algorithm to solve the timetable problem. *Journal of Scheduling* 1, 1 (1998), 39–53.

[6] Ana Y. F. de Lima, Briza M. D. de Sousa, Daniel P. Cardeal, Jessica Y. N. Sato, Lorenzo B. Salvador, Renato L. Geh, and Bruna Bazaluk. 2023. LUNCH: An Answer Set Programming System for Course Scheduling. In *Anais do XX Encontro Nacional de Inteligência Artificial e Computacional*. Sociedade Brasileira de Computação, Porto Alegre, 954–968. doi:10.5753/eniac.2023.234540

[7] Luca Di Gaspero and Andrea Schaerf. 2000. Tabu search techniques for examination timetabling. In *International Conference on the Practice and Theory of Automated Timetabling*. Springer, 104–117.

[8] Christian Dohrmann, Ulrike Lucke, Torsten Schaub, and Sebastian Schellhorn. 2025. AI-Based Tool for Curriculum-Based Course Timetabling at the University of Potsdam. In *Proceedings of EUNIS 2024 Annual Congress in Athens (EPiC Series in Computing, Vol. 105)*. EasyChair, 188–199. doi:10.29007/t32z

[9] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2019. Multi-shot ASP Solving with clingo. *Theory and Practice of Logic Programming* 19, 1 (2019), 27–82. doi:10.1017/S1471068418000054

[10] Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Robert A. Kowalski and Kenneth A. Bowen (Eds.). MIT Press, 1070–1080.

[11] Muhammed Kerem Kahraman and Esra Erdem. 2019. Personalized Course Schedule Planning Using Answer Set Programming. In *Practical Aspects of Declarative Languages*. Lecture Notes in Computer Science, Vol. 11372. Springer, 37–45. doi:10.1007/978-3-030-05998-9_3

[12] Velin Kralev, Radoslava Kraleva, and Borislav Yurukov. 2016. An Event Grouping Based Algorithm for University Course Timetabling Problem. *International Journal of Computer Science and Information Security* 14, 6 (2016), 394–401. arXiv:1607.05601.

[13] João Leite, José Júlio Alferes, and Belopeta Mito. 2009. Resource Allocation with Answer-Set Programming. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems*, Vol. 1. International Foundation for Autonomous Agents and Multiagent Systems, 649–656.

[14] Vladimir Lifschitz. 2008. What Is Answer Set Programming?. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 1594–1597.

[15] Ilkka Niemelä. 1999. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence* 25, 3–4 (1999), 241–273. doi:10.1023/A:1018930122475

[16] Adam M. Smith and Michael Mateas. 2011. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 187–200. doi:10.1109/TCIAIG.2011.2158545

[17] Potassco Team. 2025. clingo and gringo. https://potassco.org/clingo/

[18] Potassco Team. 2025. clingo Python API Documentation. https://potassco.org/clingo/python-api/current/clingo/