On the adaptive delegation and sequencing of actions.

Audun Stolpe*
Norwegian Computing Center,
Department of Applied Research in
Information Technology
audun.stolpe@nr.no

Jo Erskine Hannay
Norwegian Computing Center,
Department of Applied Research in
Information Technology
jo.hannay@nr.no

ABSTRACT

Information systems support to crisis response and management relies on presenting actionable information in a manner that supports cognitive processes. We outline how AI Planning can be used viably to support the *delegation and sequencing* of tasks, using standard operating procedures as initial specifications of plans. When expressed in the AI planning language *Answer set Programming* (ASP), machine reasoning can be used in a *pre-incident review* to display the cooperative structure inherent in a plan. The purpose of this is to uncover weaknesses and to optimize the plan. Further, adaptive planning can be supported in *during-incident reviews* by updating the current status and recomputing the consequences. At this point, initial goals may no longer be viable and the explicit suggestion of prior sub-optimal goals now worth pursuing can be a game-changer under stress.

Keywords

Decision support, AI Planning, Delegation, Sequencing, Adaptivity, Cognitive processes.

INTRODUCTION

One reason why effective emergency management is such a challenging problem, most responders would agree, is because it is not strictly speaking *a* problem. Rather it is an open-textured large conglomerate of interdependent cases often held together by ad hoc or very situation specific connections. A typical emergency situation involves adaptation to sudden and unexpected events (Christian et al. 2017; Chen et al. 2008), management of precarious time pressure and urgency, allocation and reallocation of insufficient resources and mitigation of infrastructure disruptions (Janssen et al. 2010).

Whereas some of these tasks will be loosely interwoven and incidental to the emergency at hand, others core tasks recur frequently and link together to form stable patterns. These task are interesting insofar as they can be isolated, studied and sometimes uniformly implemented across information systems. There is arguably many benefits to be expected from carving emergency management at such "natural" joints, as opposed to at its institutional joints only. It is a known problem that information systems for emergency management usually reflect departmental borders a bit too closely. This results in fragmented and unrelated computerized applications that, although they overlap in function and content, cause major interoperability problems (Magnusson et al. 2018; Janssen et al. 2010).

In the present work-in-progress paper we isolate and make a *first foray* into the study of one such recurring task that we shall call the *delegation-and-sequencing* problem. In brief outline this is the problem of allocating manpower to collaborative tasks under constraints on the proper ordering of these tasks. That is, we are presented with a set of tasks, a set of teams or companies of responders, and a set of constraints on the temporal ordering of these tasks. The problem is to allocate manpower in a way that respects the temporal constraints *including implicit ones* that may arise from the interplay between the collaborative structure of the tasks and the actual size of the teams to which these tasks can be delegated at a given moment.

We adopt a simple, yet real, running example of an operation procedure taken from a small fire department studied in (Simpson 2008). We identify its key sources of complexity and describe a simple approach for computing a

*corresponding author

deployment of it, understood as a detailed task assignment over a given set of teams that conforms to the constraints expressed in the procedure. The implementation is a purely declarative *Answer Set* Program (Gelfond 2008) which is at this point best regarded as a proof of concept. The take-away message is not the implementation as such. Rather it is the methodology,a s it offers a way of regimenting a standard operating procedure in a computable knowledge representation language in a manner that facilitates automated adaptive planning.

In (Chen et al. 2008), the activities involved in emergency response management are broken up into three commonly perceived phases: the *pre-incident phase*, the *during-incident phase* and the *recovery phase*. Each in turn iterates five activities associated with *task flow*, *resource allocation*, *information dissemination*, *decision making* and *response* respectively, cf. Fig. 1. The kind of application proposed in this paper fits particularly well into the task flow related activities in the pre-incident phase. Here, the *coordination issues*, according to the said framework, concern operating procedure deployment, and the *coordination goals* are to give an exact mapping from objectives to tasks and sub-tasks distributed over actors. We argue further that our basic encoding can easily be extended, using a programming paradigm called *multishot answer set solving*, to cater for the *evolution and adaptation* of the task flow also in the during-incident phase. Here the coordination issues concern rescheduling of tasks in response to contingencies that make the initial deployment of an operating procedure unrealizable for some reason or other.

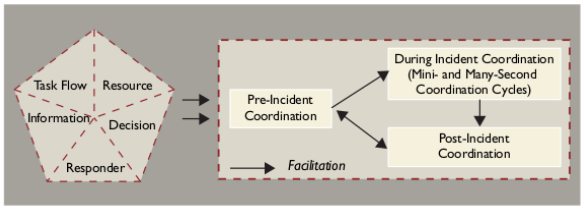

Figure 1. The coordination life-cycle (Chen et al. 2008)

The need for decision support for collaborative coordination in emergency management is emphasized in several recent studies. To mention but a few, (Fogli et al. 2017) suggest an approach to system design based on design patterns, the more conceptually oriented work (Treurniet and Wolbers 2021) studies the relationship between information sharing and distributed decision making, (N. Power 2018) analyses team decision making from the point of view of sociotechnical networks and procedural guidelines and (O'Brien et al. 2020) studies collaborative systems in terms of distributed situation awareness. A full survey is out of scope for the present paper, for obvious reasons.

In comparison our ASP encoding of the delegation-and-sequencing problem is focused exclusively on the interaction between temporal constraints and the required sizes of collaborating teams. This is a deliberate choice intended to make the combinatorial essence of the problem stand out in clear relief. However, many other types of constraint can easily be fit into this basic pattern to make for a more realistic support system. We discuss some possibilities towards the end of the paper.

AN EXAMPLE

The East Aurora Fire Department, or EAFD, is an unpaid department within the village government of East Aurora. The Village owns and maintains six fire trucks, and is staffed by an all-volunteer organization of residents (Simpson 2008). Table 1 gives the EAFD operating procedure for responding to confirmed residential structure fire. The EAFD is small for an emergency department, of course, and the procedure minute compared to how complex emergency operations can get. However, it displays the delegation-and-sequencing problem clearly, the present claim being that this particular problem is an invariant of emergency operations as such.

Table 1 exemplifies several typical constraints on the delegation of responsibilities: there are temporal constraints on the execution of actions, there are requirements on the number of agents needed to expedite tasks, and there are stipulated capacities or roles that agents must fill.

The last two rows give alternative courses of action to be pursued in the event of a *flashover*. A flashover is an abrupt combustion that occurs when a free-burning fire within an enclosure heats all materials in its immediate surroundings to their respective ignition temperatures. Firefighters cut vent holes in the roof (vertical ventilation) and/or removes windows (horizontal ventilation) to ventilate the smoke and superheated gases that result. he following points are to be be noted:

Actors and companies. The operation procedure requires coordination between different *companies*: the first attack engine company, the second attack engine team and the ladder tower team. It is mostly, though not always explicit which company is responsible for which task (a counterexample being ventilation), and sometimes, but not always, explicit how many (e.g. in the case of **h**) or who (the attack engine driver in the case of **f**) among the members of a company that are supposed to expedite it.

Table 1. Task description for response to confirmed residential structure fire

Task description	Abbreviation	Immediate predecessor(s)
Department turn-out	**a**	
Travel of attack engine to 911 address	**b**	**a**
Travel of second engine to 911 address	**c**	**a**
Travel of ladder tower to 911 address	**d**	**b**
Attack engine crew advance 1.75” hose to seat of fire	**e**	**b**
Attack engine driver prepare to pump water	**f**	**b**
Attack fire with tank water aboard attack engine	**g**	**e** and **f**
Second engine crew member prepare nearest hydrant for hook-up	**h**	**c**
Second engine drop 5” hose between attack engine and nearest hydrant	**i**	**c**
Second engine hook to hydrant and to attack engine and prepare to pump water	**j**	**h** and **i**
Attack fire with hydrant water	**k**	**g** and **j**
Set up ladder tower	**l**	**d**
In the event of a flashover		
- horizontal ventilation, or	**m**	**e** and **f**
- vertical ventilation	**n**	**e**, **f** and **l**

Task structure. Tasks are of three different kinds: *primitive actions*, such as hooking a hose to a hydrant or removing a window, can be executed directly by a single agent. *Cooperative actions* require joint effort, obvious examples being department turn-out and transportation of the companies. Less obviously, and not explicitly stated in the table, this category also includes tasks such as **e**, since hoseline advancement typically takes several firefighters on the attack line, see e.g. (Kerber et al. 2019). Finally, *abstract tasks* are goals that can be decomposed into a disjunction of subgoals, each of which provides a *method*[1] for executing that abstract task. The aforementioned task of providing ventilation is a case in point. Ventilation is also an example of a *conditional task* that should only be performed in the event of a flashover.

Execution order. Ordering constraints are given in the third column of the table. Most constraints reflect physical or logical necessity. For instance, the first attack engine crew could not advance a hose to the seat of fire (**e**) if the first attack engine were not at the scene (**b**). Other constraints express procedural proprieties that reflect best practices for fire fighters. For instance, that ventilation (**m** and **n**) be preceded by hoseline preparedness (**e** and **f**). These *explicit ordering constraints* form the partial order given in Figure 2.

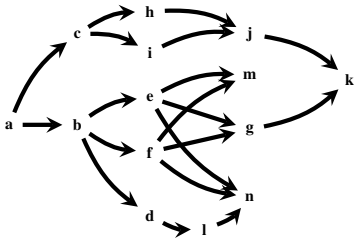

Figure 2. The partial ordering of tasks

Implicit ordering constraints. In general, there will also be *implicit ordering constraints* that are *deducible* from the task list and the staffing of companies. Suppose for the sake of argument that the tasks **h** and **i** both require two members from the second engine company and that the second engine company consists of three people. By the pigeonhole principle, it follows that **h** and **i** cannot be performed simultaneously. Implicit constraints are computationally significant, and we shall come back to them later.

FROM GUIDELINES TO DEPLOYMENT: SOURCES OF COMPLEXITY

Whereas the EAFD operating procedure gives *guidelines* for coordination and cooperation, it does not, by necessity, say exactly who is to do what when and under what conditions, since that depends on the size of companies and the evolution of the actual emergency. There is, in other words, a necessary gap between the rules of thumb expressed

[1] adhering to established terminology from the computational study of *hierarchical task networks*

by a standard operating procedure, on the one hand, and the specific assignment of actions to companies and their members that a successful fire fighting operation relies on, on the other.

Let **ae**, **se** and **lt** stand for the attack engine company, the second engine company and the ladder tower company respectively. Suppose, for the sake of argument that each company has three members, each with a designated driver. One plausible distribution of responsibilities is hypothesized in Table 2. Here, tasks **a**-**d** are analyzed, quite naturally, as group actions that involve the whole respective company. Task **f** on the other hand is delegated specifically to the driver of the attack engine. Task **g** refers to the application of water from the exterior by the first-arriving engine company before making an interior attack. It typically requires more than one firefighter; one person to control the nozzle, and one of or more of his team members to advance the hose. Similar considerations apply to task **e**. The corresponding entries in Table 2 both claim two firefighters, therefore.

From a combinatorial point of view, selecting firefighters from companies is an instance of the general problem of selecting a subset of size k from a set of size n—it is a *combination*:

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

On the assumption that there are three firefighters in each company, there are $C(3,2) = 3$ ways of delegating each of tasks **g** and **e** to two of them.

For fixed, k the growth of $C(n,k)$ is $O(n^{min\{k,n-k\}})$, so the number of possible delegations grows polynomially as the size of the company increases. This effect makes itself felt if one is not as specific as one can be when responsibilities are distributed. Consider the entry for task **l** in Table 2. It is left open which particular company the two firefighters are to be taken from. As there are 3×3 firefighters in total, there are $C(9,2) = 36$ ways of delegating it to two agents. Assuming that horizontal and vertical ventilation are not both necessary, the operation procedure has two branches. For each branch there is a total of 36×3^6 ways to delegate responsibilities to agents, putting the total number of ways of deploying Table 1 in accordance with the distribution scheme of Table 2 up to 157464.

Table 2. A distribution scheme for companies ae, se, lt each of size 3

Task ID	Company	Number
a	**se**	all
b	**ae**	all
c	**se**	all
d	**lt**	all
e	**ae**	2
f	**ae**	driver
g	**ae**	2
h	**se**	1
i	**se**	2
j	**se**	2
k	**se**	2
l	any	2
m	**lt**	2
n	**lt**	1

In this particular example, experienced fire fighters would simply ignore the combinatorial complexity and come up with a viable assignment based on what is most available at any particular moment. From a computational point of view, however, it points to a source of complexity that is inherent in the delegation-and-sequencing problem and that needs to be tackled algorithmically if the application is to scale to larger, more feature-rich examples. Indeed, it is the task of the information system to handle this complexity, and by doing so, to abstract complexity away for the benefit of the decision maker. The human ability to ignore complexity and make good enough decisions in line with fast and frugal processes (Gigerenzer and Todd 1999) and naturalistic decision making (Lipshitz et al. 2001; Klein et al. 1989) is what we are looking to support. Such decision making usually succeeds in calm conditions. However, the adverse effects of situational stress on cognition and skilled performance has been studied both functionally (Dismukes et al. 2015) and medically (Robinson et al. 2013). Even seemingly trivial tasks regulated by checklists (a particularly rigid form of standard operating procedure used, for example, by aircraft personnel) are prone to failure and faulty sequencing and delegation during acute stress. An information system that can display viable options in real time as a crisis situation evolves is therefore worth exploring.

Sequencing

Sequencing is the problem of assigning an execution order to actions in accordance the ordering constraints stipulated in the operation procedure. It involves defining a mapping from a sequence of discrete timesteps to the set of delegated cooperative actions in a way that is compatible with the partial order in Fig. 2.

The EAFD operation procedure does not express an opinion on the temporal ordering of *every* pair of actions. This is as it should be. There is little reason to, say, instruct the attack engine company to wait until the the second engine company has prepared the hydrant for hookup (**h**) before advancing the 1.75” hose to the seat of the fire (**e**), since the 1.75” hose draws water from the attack engine and not the hydrant.

This suggests that incomparable actions should be considered *prima facie simultaneously executable*, which in turn entails that the task sequence can be compressed into a linear sequence of possibly concurrent actions.

It is a standard result of lattice theory that there is exactly one such compression or *layering* provided that every maximal chain in the given partial order has the same length. This unique layering is represented by a function ρ from the natural numbers to elements of the poset satisfying

1. $x \leq y$ implies $\rho(x) \leq \rho(y)$, and
2. if $x \leq y$ and $x \leq z \leq y$ implies $x = z$ or $y = z$ then $\rho(y) = \rho(x) + 1$

The EAFD operation procedure is not layered in this sense, since the maximal chain of temporal constraints from task **a** to task **m** is strictly shorter than the maximal chain from task **a** to **n**. It follows that there is more than one way to compress Table 1 wrt. time. In fact there are two, cf. Figure 3.

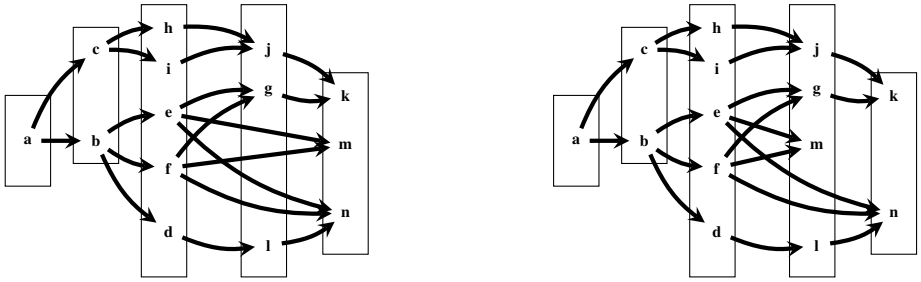

Figure 3. Two possible sequencings of the EAFD procedure

As an illustration of the interplay between the cooperative structure of tasks and temporal constraints, we pause to note the effect of implicit time constraints on the admissibility of layerings. Suppose for the sake of argument that the operation procedure in Table 1 is changed in accordance with the maxim 'be as specific as you can' such that task **l** (setting up the ladder tower) is allocated specifically to the ladder tower company instead of to any available agent. Then since there are only three firefighters in **lt**, one cannot select two of them to remove windows simultaneously with **l**. It follows that **l** and **m** form a *mutex group*, i.e. a set of tasks all of which cannot both be performed at the same time. Applied to Fig. 3, this rules out the right-hand side as inadmissible.

In general *mutex groups* can have more than two elements. There may be multiple ways to resolve them, the resolution of one may create another, and the length of the execution sequence may have to be extended. Keeping tabs on these ripples, by reallocating and re-sequencing, is a core functionality of the application we are proposing.

A FLY-BY OF ANSWER SET PROGRAMMING.

The present section gives a brief overview of *Answer Set Programming*, a relatively young offshoot of the logic programming family of languages (Lifschitz 2008). We keep the presentation on an informal level and try to steer a course between the Scylla of technical nomenclature and the Charybdis of imprecision.

Language basics. We consider an ASP program a collection of rules of one of two forms:

1. Normal rules:
$$L \text{ :- } body. \tag{1}$$
2. Choice rules :
$$m\{L : body_0\}n \text{ :- } body_1. \tag{2}$$

The symbol :- denotes implication from right to left. L is a positive *literal* in the sense of classical logic, that is, an atomic formula formed from *variables*, *constants*, *functions* and *predicates* with the usual inductive definition of termhood for the arguments of functions and predicates. A literal can be prefixed by the *negation as failure operator* '*not*', in which case it is called a *naf*-literal.

In the schematic rules 1 and 2, $body_n$ is a placeholder for a set of positive and negative (*naf*-)literals. Together they state the conditions of application for the given rule instance. This condition is satisfied if all of the positive literals

are provably true and none of the negative ones are. A rule of form 1 may have an empty body, in which case it is a *fact*, or an empty head, in which case it is an *integrity constraint* (to be explained). A rule of form 2 is a *choice rule* with *lower and upper bounds m* and *n* respectively. The bounds apply to the set expression $\{L : body_0\}$, which denotes the set of all literals L satisfying $body_0$. Thus, a choice rule asks the inference engine to select at least m and at least n literals qualified by $body_0$ for every rule instance that satisfies $body_1$ The utility of choice rules for expressing combinations, in the sense of the preceding section, will be amply illustrated.

ASP has certain other programming constructs that we shall have occasion to use, notably *aggregates* and *optimizations statements*. These are expressive modeling constructs used for respectively counting (among other things) and model selection (choosing the *best* model according to stipulated criteria). They will be explained as we go.

Answer set solvers. The basic idea of ASP is to describe a problem by means of a nonmonotonic logic program. Solutions correspond to the models of that program, the so-called *answer sets* aka. *stable models*. The rules and constraints of the program demarcate the space of all possible solutions. ASP programs are purely declarative, the express the problem but not the logic of an algorithm for computing answers.

That task is deferred to a dedicated inference engine called an *answer set solver*. An ASP solver attacks the evaluation of a program in a two-stage process: it starts with *grounding* the ASP program, instantiating its variables by constants and, more generally, by variable-free terms. The resulting program has the same answer sets as the original one but is propositional. Next, the answer sets of the grounded program are computed using a suitably repurposed satisfiability checking algorithm modified to exclude the classical models that are not stable in the requisite sense. Thus, given an ASP program, an answer set solver grounds the program and generates models in the form of sets of facts that satisfy all rules of the program and that violate none of its integrity constraints (Gelfond 2008).

Modeling methodology. The different language constructs of ASP are usually played off against each other in a matrix called the *generate-and-test* methodology. This methodology is inspired by, and closely resembles, the definition of the class of NP problems to which many classical AI problems belong: a solution or model is chosen at random and it is checked whether or not it satisfies all the required properties. Analogously, an ASP program is often organized into a *generate part* consisting of choice rules and normal rules, and a *test* part consisting of integrity constraints. Integrity constraints are normal rules with empty heads, for example

$$:\!- \, p(X), \neg q(X). \tag{3}$$

They have the function of *excluding models*, namely the models that satisfy the conjunction of all of the literals in the body. Rule (3) excludes models in which $p(X)$ is true and $q(X)$ false for some value of X.

The generate-and-test methodology is a helpful template to keep in mind in the following.

AN ASP REPRESENTATION

In the rest of this paper we sketch a general strategy for representing and reasoning about operation procedures conceived as delegation-and-sequencing problems; how they are to be deployed and how they are to be revised and adapted as an emergency evolves. Centering the discussion on our running example, we explain in a few easy steps how to leverage the generate-and-test methodology to obtain a computable representation that is split conceptually into a *delegation stage* and a *sequencing stage*. We shall refer to this encoding as the *basic representation*.

Preliminaries about the actor taxonomy. An essential component of the delegation-and-sequencing problem is selecting agents for collaborative tasks from a set of companies. Note though, that a coordinating entity, typically a remote emergency operations center, may not always need maximum specificity. Rather, if a task requires no specialist skills or competencies, it might be advisable to leave the allocation of responsibilities up to onsite personnel, as exemplified in Table 2 by **l**.

In combinatoric terms this is equivalent to selecting agents from the superclass that is the union of all companies. In general, there is of course the entire range of options represented by the boolean algebra of sets of agents to choose from, and complex selection criteria can be expressed by unions, intersections and complements of agent groups. To support such set-based reasoning, the representation of companies forms an inheritance hierarchy.

Reasoning about inheritance hierarchies is one of the oldest topics in symbolic AI, and would be covered by any introductory textbook; e.g. (Gelfond and Kahl 2014, chap. 4). Our representation adds nothing new, so we will forego the particulars. One detail worth mentioning, though, is that we standardize the nomenclature for ascribing unary properties to agents, asserting for instance that John is a fire engine driver:

$$property(john, driver). \tag{4}$$

Unary properties provide a simple but general mechanism for associating capacities, roles and competencies with agents or sets of agents. Applied to our running example, the expression (4) together with a membership expression

$$member(john, ae). \tag{5}$$

allows us e.g. to refer to John as the driver of the attack engine. More generally, we conjecture that the combination of membership assertion and property ascription will allow us to represent the concept of a role, which is a fundamental one for the description of any multiagent system. A role defines the capacities in which an agent may act, and/or the institutional powers and privileges he or she is invested with. By associating conditions with roles, it is possible to describe the flow and transference of authority and responsibility between institutions during collaborative operations. This is a planned extension of the current work which must be left at that for now.

The delegation stage. As previously mentioned, the EAFD procedure can be analyzed in terms of three different sorts of task; primitive, cooperative and abstract. Corresponding to each kind of task, our representation contains a *deployment description*, e. g.:

$$\begin{array}{l} primitive(f). \\ responsible(f, Ag) \text{ :- } \quad deploy(f), \\ \qquad\qquad\qquad\quad property(Ag, driver), \\ \qquad\qquad\qquad\quad member(Ag, ae). \end{array} \tag{6}$$

$$\begin{array}{l} collaborative(e). \\ delegate(e, 2, ae) \text{ :- } \quad deploy(e). \end{array} \tag{7}$$

$$\begin{array}{l} abstract(v). \\ 1\{deploy(m); deploy(n)\}1 \text{ :- } \quad deploy(vent), holds(flashover). \end{array} \tag{8}$$

On this encoding, only the deployment of primitive actions issue direct responsibilities. In contrast, descriptions of collaborative tasks issue constraints on the distribution of primitive actions, whereas the deployment of abstract tasks chooses one among a specified set of alternative *methods* to substitute for that abstract task.

Complying distributions of responsibilities are computed by adding a *choice rule* exemplifying the generation phase of the generate-and-test methodology:

$$N\{responsible(Ac, Ag) : member(Ag, Group)\}N \text{ :- } \quad delegate(Ac, N, Group). \tag{9}$$

The rule (9) converts a delegation statement into a set of individual responsibilities as per the associated delegation rule (the affinity between (9) and combinations $C(x, y)$ should be apparent). Since delegation statements are detached only from delegation rules, (9) applies only after a task has been chosen for deployment. Presupposing a rule $objective(Ac)$:- $primitive(Ac)$, and similar ones for abstract and collaborative tasks, a concrete deployment is selected by a second choice rule with no condition of application, i.e. no body:

$$\{deploy(Ac) : objective(Ac)\}. \tag{10}$$

There is more than one way to satisfy (10) since there are as many models of it as there are ways to choose tasks. Not all these deployments are intuitively valid. For instance models in which collaborative and/or primitive tasks are omitted are inadmissible, as are models that deploy more of the methods of an abstract task than stipulated by the associated delegation rule. Such models are removed by integrity constraints in what amounts to the *test* phase of the *generate-and-test* methodology.

The sequencing stage. The sequencing stage is similar in structure to the delegation stage in the sense that it employs the generate-and-test methodology. This time, however, we are choosing time points for *executing* the delegated responsibilities:

$$1\{executes(Ac, Ag, T) : time(T)\}1 \text{ :- } responsible(Ac, Ag). \tag{11}$$

Rule (11) chooses exactly one time point for the execution of each responsibility that has been output from the delegation stage.

Time points must be explicitly represented in ASP. There is no such thing as a for-loop, and a point in time is just one discrete entity among others. The delegation-and-sequencing program should be generic, i.e. applicable to different operating procedures. Since we cannot in general assume that tasks are executable concurrently, we need as many time points as there are individual responsibilities.

This can be implemented with a programming construct called an *aggregate expression*. Aggregates are expressive modeling constructs that allow for forming numeric values from sets of items (Gebser et al. 2011). Together with the equality predicate they can be used to declare *ranges* of values. The rule

$$time(1..Y) \text{ :- } Y = \#count\{Ac, Ag : responsible(Ac, Ag)\}. \tag{12}$$

states that the interval between 1 and the cardinality of the set of individual responsibilities are time steps, thus securing a large enough interval to work with.

Of course, there is nothing so far that forces time points to self-organize into a gapless linear execution sequence starting at 1. This needs to be ensured by integrity constraints in the test phase of the program. Here is a sample of constraints in an inefficient but reader-friendly representation:

$$T_1 = T_2 \text{ :- } \quad executes(Ac, Ag, T_1), executes(Ac, Ag, T_2). \tag{13}$$

$$preceded(Ac_1) \text{ :- } \quad executes(Ac_1, Ag_1, T), executes(Ac_2, Ag_2, T\text{-}1), time(T). \tag{14}$$

$$\text{:- } executes(Ac, Ag, T), T > 1, not\ preceded(Ac). \tag{15}$$

$$\begin{aligned} start \text{ :- } \quad & executes(Ac, Ag, 1). \\ \text{:- } & not\ start \end{aligned} \tag{16}$$

Rule (13) is equivalent to the constraint that a task only be executed once, the rule (14) defines a predecessor relation for time points that is is used in the constraint (15) to shape the set of time points into a cohesive sequence. Finally the pair (16), guarantees that this cohesive sequence starts at 1.

It remains to ensure that the generated execution sequence respects the ordering constraints stipulated by in the third column of Table 1. Presupposing an encoding of that column by expressions of form $before(d, b)$, one last constraint will do:

$$\text{:- } executes(Ac_1, Ag_1, T_1), executes(Ac_2, Ag_2 T_2), not\ \ (T_1 < T_2), before(Ac_2, Ac_1). \tag{17}$$

Before rounding off this section, we should quickly revisit the question of *time compression* or *layering*. It is natural to assume that these are the deployments one would normally be interested in when assessing the merits of a plan.

The ASP language offers *optimization statements* for such purposes. They are modeling constructs that extend the basic question of whether a set of atoms is an answer set to whether it is an *optimal* answer set. Thus, the statement

$$\#minimize\{T : executes(Ac, Ag, T)\} \tag{18}$$

instructs the answer set solver to generate an execution sequence within a minimal time interval bounded by an integer T. This has two interesting consequences. First, and as expected, it compresses the execution sequence or conversely maximizes parallelism, generating the layerings in Fig. 3. Secondly, it makes the answer set solver

perform a branch-and-bound search for an optimal model, aborting the search once the first such model has been found. This built-in branch-and-bound algorithm provides a satisfactory algorithmic solution to the inherent complexity of delegation mentioned above. Adding (18) to the running example, makes the answer set solver choose *one* of the 157464 ways to distribute responsibilities. Since we have no basis for preferring any of these over another, nothing is lost. Moreover, even though combinations grow in the order $O(n^{min\{k,n-k\}})$ in the size n of teams, the application is likely to scale well in this respect, since larger teams do not make the computation of an optimally time-efficient execution sequence significantly harder.

ADAPTING TO CONTINGENCIES

It is somewhat of a *sui generis* paradox of emergency response management that zealous planning often leads to response inflexibility (Chen et al. 2008). Mendonça (2007) argues that information systems for managing standard operating procedures are not likely to be perceived as useful unless they can support decision makers in modifying these procedures to make them feasible and relevant to the goals of the response. Janssen et al. (2010) calls for flexible coordination mechanisms that can be easily customized for the specific situation and provide better support for improvised responses. There seems to be consensus that this is a core problem that is nearly *always* a problem and something belonging to the essence of emergency response (Maynard et al. 2015).

An example in point is the London Grenfell Tower incident (Moore-Bick et al. 2019). Besides catastrophic structural failure of the building itself, the London Fire Brigade experienced problems in sequencing and delegating crucial tasks. Of particular consequence was the sequencing in time, with other tasks, of deciding the change of status from "stay put" to "get out" and the delegation of who was responsible to take that decision and who was responsible to relay that status change from the initial observation from boots on the ground via incident command to the control room operators who were handling communications with residents in the building.

Plan revision with multishot ASP

From an abstract point of view, continually adapting a plan to an evolving scenario is a matter of revising an initial plan upon learning of a deviation. For instance, if a particular agent is incapacitated or an emergency vehicle is stuck in traffic, it is going to have ramifications for the allocation of responsibilities. Adapting plans and goals as events unfold is the modern way of handling the uncertainty inherent in forecasting the future. Agile management and self-organizing teams are modes of operation that are designed for gathering information continuously, learning quickly and taking timely decisions (making plans) when there is information of sufficient quantity and quality to do so (Hannay et al. 2015). That timeliness relies on the delegation of responsibility to where information is at hand. One important proviso that seems natural, very much studied in the field of theory revision, is that the evolution of the plan over time should heed the *maxim of minimal change*. That is, the plan before and after the revision should be as similar as possible.

In the present section we sketch, with rather broad strokes, how adaptive planning can be implemented with this proviso. What we propose is to animate, so to speak, the basic representation of delegation and sequencing outlined in preceding sections in a procedural extension to the basic ASP idiom known as *multishot solving*.

The concept of multishot solving can be described as *iterative, stateful grounding and solving*. The idea is to model evolving processes by repeatedly grounding and solving a program in steps, accumulating internal state in the form of known facts as one goes (Kaminski et al. 2020). Multishot solving refers specifically to the functionality of Clingo, its support for incremental ASP constructs, and its built-in Python API for manipulating logic programs. Multishot solving has three key features listed below.

Subprograms. A multishot ASP program is split into *subprograms* by means of a #*program*-directive. A *subprogram* has a name and an optional list of parameters and extends down to the next such directive or to the end of the file. The program name *base* is a reserved name for a special subprogram with an empty parameter list. In addition to the rules in its immediate scope, it collects all rules not preceded by a #*program*-directive.

Externally defined atoms. An external atom is an ASP atom whose truth value is not deduced from the rules of the ASP program itself, but set in a control script outside of the solver (hence the name *external*). Such atoms are declared in ASP by #*external*-directives appended to a subprogram declaration.

```
1  #program base.
2
3  /*
4  ... task declarations and actor taxonomy
5  */
6
7  #program step(n).
8  #external remove(Ac, Ag, n).
9  #external add(Ac, Ag, n).
10
11 :- responsible(Ac, Ag, n),  remove(Ac, Ag, n).
12 responsible(Ac, Ag, n):- add(Ac, Ag, n).
13
14 ....
15
16 N{responsible(Ac, Ag, n): member(Ag, Group)}N :-  delegate(Ac, N, Group, n).
17 {deploy(Ac, n): objective(Ac)}.
18
19 #minimize{Ac, Ag: responsible(Ac, Ag, n-1), not responsible(Ac, Ag, n)}.
20 #minimize{Ac, Ag: responsible(Ac, Ag, n), not responsible(Ac, Ag, n-1)}.
21
22 #script (python)
23 from clingo import Function
24 import clingo
25
26 def main(prg):
27     i = 0
28     revision = ""
29     prg.ground([("base", [])])
30     while True:
31         revision = input(">> ")
32         prg.ground([("step", [i])])
33         term = clingo.parse_term(revision)
34         predicate = term.name
35         action, agent = term.arguments[0], term.arguments[1]
36         prg.assign_external(Function(predicate, [action, agent, i]), True)
37         i+=1
38         prg.solve()
39 #end.
```

Listing 1. Elementary structure of adaptive program

Embedded Python control. A Python control script can be enclosed in a #*script*-environment in the ASP code itself. This script has access to the grounder and solver through the Clingo Python API, enabling procedural control over the grounding and solving of subprograms, as well as over the assignment of truth values to external atoms.

The idea is to utilize this apparatus to implement adaptive deployment in four steps: first we make responsibilities time-relative in a subprogram *step*(n) that takes an integer input, intuitively a time point. Second, we define a Python control loop that sets the truth value of external atoms *add* and *remove* for each increment of n. These external atoms take tasks, agents, and time points as arguments and interpolates new responsibilities or negations of responsibilities into the solving process. The ASP solver will consequently output a new allocation of responsibilities and a sequencing of tasks taking these new facts (possibly none) into consideration. Finally, as a way of adhering to the maxim of minimal change, we add optimization statements to make the solver prefer models in which the allocated responsibilities differ as little as they can from the deployment computed in the previous iteration.

A skeletal and partial implementation of these ideas is given in Listing 1. The program has two subprograms *base* and *step*. The former contains all static information that remains unchanged over time, such as the actor taxonomy and the static part of task declarations (lines 1-6). The delegation rules go into the *step* program where they are made relative to the current value of the parameter n. This is exemplified by the temporal recasting of the generating rules (9) and (10) in lines 16 and 17 (the particular delegation rules for the different objectives are hidden behind the ellipsis in line 14 but conform to the same ternary format). The distribution of responsibilities to agents is constrained in each increment by external input *add* and *remove* declared in lines 8-9. The incremental process is controlled by the *main* routine in lines 26-38, which takes a Clingo control object *prg* as argument. In *main* the base program is first grounded in line 29, before the flow of control steps into the infinite loop in lines 30-38. In each iteration, the program waits for an input from the user, here exemplified by command line input on line 31. Once

received, the input is parsed and the corresponding external atom is set to true for the current iteration of the control loop. If the atom in question is *add* then the rule in line 12 ensures that there is a corresponding responsibility. If, on the other hand, the atom is *remove*, then the integrity constraint in line 11 *eliminates* models containing the corresponding responsibility. Finally, the models of each increment are prioritized by the optimization statements in lines 19 and 20, instructing the solver to select a model that differs minimally from the optimal model computed in the previous iteration.[2]

SUMMARY AND OUTLOOK

The machine reasoning support for delegation and sequencing that we have outlined in this discussion is the nucleus of a decision support system that is *model-driven* in terms of (D. J. Power 2002) by virtue of ASP specifications and the answer set models generated. Further, it is *passive* in terms of (Hättenschwiler 1999), since the system would not make decisions, but rather augment (in fact, by simplifying) the information cues that decision makers will use both before and during acute stress.

The conceptual power and expressive economy of AI planning languages such as ASP entails that our small "delegation and sequencing machine reasoning module" can easily be expanded with further planning features; including the logical representation of resource requirements, institutional representation, role transfer, geographical proximity, to name but a few.

In any case, the off-the-shelf answer set solver (Clingo in our discussion) performs all the computations and renders any systems development unnecessary; apart from embedding this reasoning power into existing command & control tools and operations support tools. The extreme computational economy of the AI planning paradigm means that one can easily run this functionality on handheld devices in the field.

By integrating an ASP-based scoring system into the scenario modelling language, the ASP-planner can also be used for simulation and training purposes. Low -hanging fruits would be to evaluate the relative merit of alternative courses of actions or to compute the optimal reallocation of responsibilities given a preference ordering over temas and agents. At the time of writing this is work in progress.

REFERENCES

Chen, R., Sharman, R., Rao, R., and Upadhyaya, S. (May 2008). "Coordination in Emergency Response Management". In: *Commun. ACM* 51, pp. 66–73.

Christian, J., Christian, M., Pearsall, M., and Long, E. (2017). "Team adaptation in context: An integrated conceptual model and meta-analytic review". In: *Organizational Behavior and Human Decision Processes* 140, pp. 62–89.

Dismukes, R. K., Goldsmith, T. E., and Kochan, J. A. (2015). *Effects of Acute Stress on Aircrew Performance: Literature Review and Analysis of Operational Aspects*. Technical Report NASA/TM—2015–218930. National Aeronautics and Space Administration, Ames Research Center.

Fogli, D., Greppi, C., and Guida, G. (2017). "Design patterns for emergency management: An exercise in reflective practice". In: *Information & Management* 54.7, pp. 971–986.

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., and Schneider, M. (2011). "Potassco: The Potsdam answer set solving collection". In: *Ai Communications* 24.2, pp. 107–124.

Gelfond, M. (2008). "Answer Sets". In: *Handbook of Knowledge Representation*. Ed. by F. van Harmelen, V. Lifschitz, and B. W. Porter. Vol. 3. Foundations of Artificial Intelligence. Elsevier, pp. 285–316.

Gelfond, M. and Kahl, Y. (2014). *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press.

Gigerenzer, G. and Todd, P. M., eds. (1999). *Simple Heuristics that Make Us Smart*. Oxford University Press. Chap. 4.

Hannay, J. E., Brathen, K., and Hyndøy, J. I. (2015). "On How Simulations Can Support Adaptive Thinking in Operations Planning". In: *Proc. NATO Modelling and Simulation Group Symp. on M&S Support to Operational Tasks Including War Gaming, Logistics, Cyber Defence (STO-MP-MSG-133)*.

Hättenschwiler, P. (1999). "Neues anwenderfreundliches Konzept der Entscheidungsunterstützung". In: *Gutes Entscheiden in Wirtschaft, Politik und Gesellschaft*. vdf Hochschulverlag AG an der ETH Zürich, pp. 189–208.

[2]A proof-of-concept implementation is available for download at https://gitlab.com/Audunsto/delegation-and-sequencing/

Janssen, M., Lee, J., Bharosa, N., and Cresswell, A. (Mar. 2010). "Advances in multi-agency disaster management: Key elements in disaster research". In: *Information Systems Frontiers* 12, pp. 1–7.

Kaminski, R., Romero, J., Schaub, T., and Wanko, P. (2020). *How to build your own ASP-based system?!* arXiv: 2008.06692.

Kerber, S., Regan, J. W., Horn, G. P., Fent, K. W., and Smith, D. L. (2019). "Effect of firefighting intervention on occupant tenability during a residential fire". In: *Fire technology* 55.6, pp. 2289–2316.

Klein, G., Calderwood, R., and Macgregor, D. (1989). "Critical decision method for eliciting knowledge". In: *IEEE Transactions on Systems, Man and Cybernetics* 19, pp. 462–472.

Lifschitz, V. (2008). "What is Answer Set Programming?" In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*. AAAI'08. Chicago, Illinois: AAAI Press, pp. 1594–1597.

Lipshitz, R., Klein, G., Orasanu, J., and Salas, E. (2001). "Taking Stock of Naturalistic Decision Making". In: *J. Behavioral Decision Making* 14.5. Ed. by J. F. Yates, pp. 331–352.

Magnusson, M., Nyberg, L., and Wik, M. (May 2018). "Information Systems for Disaster Management Training Information Systems for Disaster Management Training: Investigating User Needs with a Design Science Research Approach". In:

Maynard, M., Kennedy, D., and Sommer, S. (Jan. 2015). "Team adaptation: A fifteen-year synthesis (1998–2013) and framework for how this literature needs to "adapt" going forward". In: *European Journal of Work and Organizational Psychology* 24, pp. 1–26.

Mendonça, D. (2007). "Decision support for improvisation in response to extreme events: Learning from the response to the 2001 World Trade Center attack". In: *Decision Support Systems* 43.3, pp. 952–967.

Moore-Bick, M., Istephan, T., and Akbor, A. (2019). *Grenfell Tower Inquiry: Phase 1 Report. Report of the Public Inquiry into the Fire at Grenfell Tower on 14 June 2017*. Report HC 49-I. The Grenfell Tower Inquiry.

O'Brien, A., Read, G. J., and Salmon, P. M. (2020). "Situation Awareness in multi-agency emergency response: Models, methods and applications". In: *International Journal of Disaster Risk Reduction* 48, p. 101634.

Power, D. J. (2002). *Decision Support Systems: Concepts and Resources for Managers*. Greenwood Publishing Group.

Power, N. (2018). "Extreme teams: Toward a greater understanding of multiagency teamwork during major emergencies and disasters." In: *American Psychologist* 73.4, p. 478.

Robinson, S. J., Leach, J., Owen-Lynch, P. J., and Sünram-Lea, S. I. (2013). "Stress Reactivity and Cognitive Performance in a Simulated Firefighting Emergency". In: *Aviation, Space, and Environmental Medicine* 84.6, pp. 592–599.

Simpson, N. (Jan. 2008). "East Aurora Fire Department". In: *SSRN Electronic Journal*.

Treurniet, W. and Wolbers, J. (2021). "Codifying a crisis: Progressing from information sharing to distributed decision-making". In: *Journal of Contingencies and Crisis Management* 29.1, pp. 23–35.