Action Languages and COVID-19: Lessons Learned*

Marcello Balduccini, Michael Barborak, and David Ferrucci

Elemental Cognition
{marcellob,mikeb,davef}@elementalcognition.com

Abstract. We recently conducted an exercise in which we evaluated the use of RAC and action languages to formalize policies related to COVID-19. In this paper, we summarize the most salient lessons we learned from this exercise. We believe our findings are relevant not only to this specific domain, but also to policy formalization in general and possibly even to tasks beyond policy formalization.

In this short paper, we summarize the lessons we learned from an exercise in which we evaluated the use of techniques based on Reasoning about Actions and Change (RAC) and action languages [6] for formalizing policies [1] aimed at describing individuals' readiness to perform given activities (work, school, sports, etc.) during the COVID-19 pandemic.

The task of easing COVID-19 lock-down restrictions is made complicated by a variety of factors, including the uncertainty about the way in which the virus spreads and the challenges in adapting pre-existing environments that were not meant to limit the spread of such an aggressive airborne virus. A practice commonly used with the aim of reducing risks consists in adopting policies that determine an individual's fitness to access an environment or perform an activity. For instance, for an employee to be granted access the office environment on a given day, the person may be required to have been symptom-free for the past 24 hours and have not been in contact with sick individuals for the past 7 days. Policies of this kind often become very complex, especially when they are used to predict the evolution of a user's state over time.

Due to how quickly conditions and recommendations change, and to the many ramifications of resuming regular activities or not, it is paramount to have tools that simplify the task of developing and enforcing policies. Particularly, it is essential to enable (1) quick development of policies, (2) trust that the implementation of the policies reflects the original intent, (3) transparency of the decisions made by such implementations.

At first glance, RAC appears to be a suitable candidate for accomplishing this. In many cases, decisions depend on the state of the individual, a state that frequently evolves over time – e.g., an employee that tests positive for COVID-19 may be considered unfit to work until the employee tests negative

or receives a positive antibody test. Action language statements are typically terse and are stated at a high level of abstraction, often close to that of the original specification. Reasoning mechanisms can be used to draw a variety of conclusions.

In this exercise, we used a variant of action language $\mathcal{AL}$ [3] for modeling various realistic policies reflecting those in use by companies and institutions. Reasoning mechanisms were used to make decisions about a user's fitness and to explain such decisions to the user. The goal of the exercise was to determine if RAC, and specifically action languages, are suitable for the task. We believe that the exercise is relevant not only to the specific domain of COVID-19 policies, but also to policy formalization in general and possibly to other domains. Next, we discuss our findings.

Lesson 1: Another Case for the Use of RAC in Practical Applications. The outcome of the exercise was largely positive. The use of RAC substantially simplified and sped up the task of implementing policies. We found that all policies of interest – some of which having considerable complexity – can be implemented solely using state constraints. Observations were used to model the results of tests and users' answers to questionnaires. To evaluate development speed, we compared the time it took to implement the same policy using RAC vs using an imperative paradigm. The implementors had comparably extensive experience in the use of RAC and of imperative languages, respectively. Implementing policies using RAC was consistently 7 times faster than implementing them imperatively. In further experiments, other subjects, with extensive imperative programming experience and little or no prior knowledge of declarative programming, were asked to implement policies of similar complexity to the previous ones. In a preliminary 2-hour session, they were introduced to transition diagrams, fluents and the informal semantics of RAC statements. The programmers were able to produce satisfactory RAC policy implementations immediately. The ability to easily and clearly define sophisticated reasoning mechanisms made it possible to automatically generate explanations for the decisions produced by the policies and to make them accessible even to non-expert users.

Lesson 2: Traditional Action Language Constructs Had to Be Extended. Similarly to [10], observations were assumed correct and were the source of state change.[1] As one might expect, policy implementation required the use of both inertial and defined fluents, e.g. being fit-for-access is often conveniently modeled by a positively defined fluent that defaults to false unless explicitly set by a law. However, traditional action language constructs[2] are insufficient to conveniently capture statements whose validity has a certain duration, e.g. "after international travel, one is not allowed access to the office for 14 days." Formalizing such statements requires "wall-clock" time and changing the value of fluents

[1] Details on the approach are beyond the scope of this short report.

[2] If one were to look outside of the context of action languages, then a promising path is that of [8], which is based on Event Calculus.

without intervening observations. Action languages such as $\mathcal{H}$ [4], while suitable, have a complex semantics and can reduce performance considerably due to the complexity of the underlying implementation. In all policies we studied, time and change are simpler than in $\mathcal{H}$ and do not justify its use. Additive fluents [9] offer a potential solution, but it was not clear to us how to conveniently cause value changes, even in combination with, e.g., triggers. For instance, the statement "one is not allowed access to the office for 14 days" requires the ability to count down the given amount of time and to cause a fluent, say, has_access to be false for that duration. While additive fluents could be used to represent the amount of time left, changing that value over time in a convenient way seems less straightforward. An additional challenge is that one will also want somehow to ensure that has_access is allowed to revert to true at the end of that period, unless other conditions intervene. Thus, we (1) associated a constant duration with state transitions[3] and (2) introduced *timed fluents*, i.e. numerical fluents whose value decreases by a constant amount at every state transition until they reach 0. The above statement can thus be encoded via a timed fluent $days\_left\_from(intl\_travel)$, which causes has_access to be false if its value is greater than 0, e.g.:

$$\begin{cases} fluent(days\_left\_from(intl\_travel), timed). \\ fluent(has\_access, negatively\_defined). \\ \\ \neg has\_access \text{ if } days\_left\_from(intl\_travel) > 0. \end{cases}$$

Lesson 3: Observations at Imprecise Times Require Dedicated Handling. Another lesson learned was that the assumption frequently made in RAC that the time associated with an observation is known is too strong. Some policy questionnaires ask questions such as "were you in close contact with a sick individual in the past 7 days?" Suppose an individual answers "yes" one day and "no" on the next day (or two days later). Through commonsense, a human would be able to pinpoint with fair precision when the contact occurred. Timed fluents could then be used to track isolation periods. However, observations as they are normally introduced in action languages do not support this. We had to extend the observation mechanism to support observations over a time frame and we defined their semantics to allow for drawing conclusions such as the above one. It should be noted that the semantics can be defined entirely in terms of truth of fluents in given states, which makes the extension fully modular w.r.t. the semantics of action theories. Below, we give an example showing the encoding of the above observations, expressed by means of newly defined keywords *sometime_in_past* and *never_in_past*. Note how the expressions of the form $obs(f, v, t)$, introduced in [7], extend elegantly to capture more sophisticated types of observations. The sample encoding also includes a law stating that the subject has 14 days of isolation left whenever the subject has been in close contact with a sick individual. It is interesting to note the separation of concerns between observations and

[3] Remarkably, this duration can be easily changed in due course thanks to the elaboration tolerance afforded by action languages.

laws, which makes it possible for a knowledge engineer to write the law without concerning himself/herself with the potential uncertainty in the observations.

$$\begin{cases} obs(close\_contact, \text{sometime\_in\_past}(7), 10). \\ obs(close\_contact, \text{never\_in\_past}(7), 11). \\ \\ days\_left\_from(close\_contact) = 14 \text{ if } close\_contact. \end{cases}$$

Lesson 4: Explaining To Non-Experts Is Non-Trivial. Producing explanations for a given conclusion is a heavily researched task [5] (see also [2] for an interesting and related approach, whose relation with ours is yet to be investigated). As we observed, producing explanations that can be understood by a non-expert user (often in an adversarial posture – "why *can't* I play the match?!") adds another layer of complexity. In our approach, an explanation for a given conclusion is obtained by recursively extracting the relevant dependencies from the dependency graph of the answer set program that implements the theory. This yields what amounts to a "proof tree" for the conclusion. The information is then graphically presented to the user, allowing the user to expand branches of the tree as desired. However, parts of the tree may be meaningless to a non-expert. Consider a (heavily simplified) situation in which inertial fluent *ai* ("assumed infectious") is first made true via state constraint "*ai* if *covid_positive*" and persists in the following time steps due to inertia. A naïve rendition of the proof tree for *ai* would include a number of claims "*ai* holds because it held in the previous step," which provide little insight to a non-expert user. Leveraging the fact that users will look at the explanation through the lens of commonsense, we found it more effective to omit the parts of the explanation related to inertia (as well as policy fragments) and provide the final part of the explanation, e.g. "because you tested positive to COVID." Users will often infer the missing links, and can ask HR for clarifications if needed. In many cases, the natural language associated with the explanations must also be tuned to convey enough/appropriate information, e.g. "because you tested positive to COVID-19 on Tuesday." We found that all of the above can be conveniently achieved by introducing rules defining special predicates *nlg* (for associating natural language with an explanation), *hide* (drop a given component of the explanation) and *self_evident* (the current component is self-evident and should not be expanded further). The rules' bodies check the truth value of fluents in a corresponding state to determine how to best treat a component of the explanation. The user interface can then use the information to tailor the presentation of the proof tree. In fact, the approach can be further refined to customize the presentation for a class of users, e.g. non-expert, HR, or developer.

Acknowledgments. We would like to thank the anonymous reviewers for their valuable suggestions and comments.

References

1. https://www.cdc.gov/coronavirus/2019-ncov/community/guidance-business-response.html
2. Aguado, F., Cabalar, P., Fandinno, J., Muniz, B., Perez, G., Suarez, F.: A Rule-Based System for Explainable Donor-Patient Matching in Liver Transplantation. In: 35th International Conference on Logic Programming (ICLP19) (2019)
3. Baral, C., Gelfond, M.: Reasoning Agents In Dynamic Domains. In: Workshop on Logic-Based Artificial Intelligence. pp. 257–279. Kluwer Academic Publishers (Jun 2000)
4. Chintabathina, S., Watson, R.: Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday, chap. A New Incarnation of Action Language H, pp. 560–575. Lecture Notes in Artificial Intelligence (LNCS), Springer Verlag, Berlin (2011)
5. Fandinno, J., Schulz, C.: Answering the âĂIJWhyâĂİ in Answer Set Programming – A Survey of Explanation Approaches. Journal of Theory and Practice of Logic Programming (TPLP) **19**(2), 114–203 (Mar 2019)
6. Gelfond, M., Lifschitz, V.: Action Languages. Electronic Transactions on AI **3**(16) (1998)
7. Inclezan, D., Gelfond, M.: Modular action language. Journal of Theory and Practice of Logic Programming (TPLP) **16**(2) (2016)
8. Kowalski, R.A., Sadri, F.: Reactive Computing as Model Generation. New Generation Computing **33**(1), 33–67 (2015)
9. Lee, J., Lifschitz, V.: Additive Fluents. In: Provetti, A., Son, T.C. (eds.) Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning. AAAI 2001 Spring Symposium Series (Mar 2001)
10. Son, T.C., Pontelli, E., Gelfond, M., Balduccini, M.: An Answer Set Programming Framework for Reasoning about Truthfulness of Statements by Agents. In: 32nd International Conference on Logic Programming (ICLP16) (2016)