# Logic Programs as Executable Experimental Task Specifications

**Can Serif Mekik (mekikc@rpi.edu)**
Department of Cognitive Science, 110 8th Street
Troy, NY 12180 USA

## Abstract

This paper proposes a formalized approach to the specification of experimental tasks in cognitive science. Put briefly, the proposal is to represent the structure of a task by a logic program that accepts only valid experimental event logs for the chosen paradigm. It is argued that the proposed approach stands to benefit the research process at various stages as it involves the creation of executable documentation for experimental tasks, which may facilitate the communication, validation, implementation, and analysis of experimental tasks. A worked example is presented in detail and some potential new directions of research at the intersection of psychology and computer science are discussed.

**Keywords:** logic programming; formal methods; experimental task; task analysis; reproducibility

## Introduction

In cognitive science, the structure and design of experimental tasks is an essential aspect of empirical research. The interpretation of experimental results depends on the exact sequencing, timing, and content of experimental events. Moreover, experimental work often progresses by refining or extending existing experimental tasks through the introduction of new conditions, task demands, and measurements.

Even the simplest of experimental tasks exhibit intricacies that require careful reporting and analysis. Such reporting and analysis is currently carried out, by and large, in natural language. Given the importance and intricate nature of experimental tasks, formal representations of task structure and design could prove highly useful. At the very least, such formal representations may help improve communication, classification, and documentation of experimental tasks by providing a structured medium in which assumptions could be made explicit and ambiguities reduced.

This paper proposes an approach to formalizing the structure of experimental tasks using *executable experimental task specifications*. The proposal is motivated by a simple premise. The specification of an experimental task involves defining the set of acceptable experimental events, timings, and data. All such requirements bear on operational aspects of an experiment and are consequently amenable to being checked against experimental event logs. Therefore, complete and accurate algorithms for validating experimental logs should naturally contain all essential operational information about the structure of experimental tasks. When written as logic programs, such algorithms may serve as executable specifications against which implementations, replications, and analyses of an experimental task can be evaluated.

The proposal goes beyond the specification of *experimental designs*, which involve specification, at an abstract level, of how experimental measures are to be sampled. What is being proposed here is related and much more granular. The goal is to capture, in a formal language, the procedural details of experimental tasks down to individual trial-level events such that another researcher could, in theory, reimplement the same experimental task based on these specifications alone (assuming experimental assets, such as stimuli, are available).

The proposed approach is similar to task analysis techniques ubiquitous in human-computer interaction (Diaper, 2004). Among these techniques, it is closest in spirit to GOMS. Nevertheless, the proposal distinguishes itself in the following respects. First, it targets purely normative representations. Second, the proposal is designed to leverage the expressiveness and flexibility of logic-based knowledge representations. Third, the proposal establishes an explicit link between the structure of experimental tasks and theoretial computer science by casting experimental task specifications as decision procedures.

This paper aims to develop and motivate the proposal through detailed study of a worked example. The target task is the simple reaction time task, which is arguably one of the simplest tasks in cognitive psychology. The paper starts out with a conceptual overview, then proceeds to develop a specification for the target task. Example code is presented in SWI-Prolog (Wielemaker, Schrijvers, Triska, & Lager, 2012). A brief primer on Prolog is included to help readers navigate the worked example.

## Conceptual Overview

An *experimental event log* (XEL) is a database that associates experimental event types with timestamps and contains other important data such as subject characteristics, equipment characteristics, experimental parameters and so on. An *executable experimental task specification* (XTS) is, then, a logic program that takes an XEL as input and outputs an answer to the question "Is this data compliant with the experimental protocol?". In other words, an XTS is a decision procedure for validating experimental event logs against a specific experimental protocol. The present paper proposes

XTSs as executable documentation for experimental tasks and demonstrates how the proposal may be implemented using readily available tools and techinques, notably through the use of the Prolog programming language and the event calculus formalism.

The main technical challenge for writing an XTS is to adequately express and validate requirements related to the flow and effects of experimental events. The difficulty is that, when reasoning about events and their effects, we inevitably want to talk about facts that hold at certain times and do not hold at others. Such facts are called *fluents* and care is needed when formulating logical theories about them. Of the various available approaches to reasoning about fluents, event calculi are a natural choice for reasoning about experimental events. In event calculi, reasoning is based on a narrative of events (see Shanahan, 1999, for an introduction) i.e., a record of what happens and when. Events determine the values of fluents based on a collection of *domain-general* and *domain-specific* axioms. The domain-general axioms establish general rules for reasoning about when fluents hold. Domain-specific axioms encode information about the effects of events in so-called *effect axioms* and about narratives in the form of *event axioms*.

## Prolog Primer

This brief overview of Prolog is designed to help the reader navigate the code presented in the remainder of this paper. The interested reader is referred to resources such as Bartko (1986) for further information.

In Prolog, a program is expressed as a collection of *clauses* against which queries may be run. To run a query, one provides a *goal* to the Prolog interpreter. The interpreter then attempts to prove the goal from the given clauses. If a proof is found, the query *succeeds*, otherwise it *fails*. Additionally, if the goal contains variables, the interpreter returns any instantiations of the variables for which the goal succeeds. In other words, the interpreter answers the following questions relative to the set of given clauses "Is the goal proposition true?" and "For what value(s) of the variable(s) is the goal proposition true?". If a queried proposition cannot be proved from given clauses, it is assumed to be false. This way of expressing negation is called *negation as failure*, and it differs subtly from classical negation. Generally, Prolog programs have two readings, a semantic reading, which pertains to the logical content expressed by the program, and a procedural reading, which pertains to how the interpreter processes queries.

Prolog clauses consist of *terms*, of which there are several subtypes: *atoms*, *numbers*, *variables*, and *structures*. Typically, atoms occur as strings of letters, digits, and underscores that start with a lowercase letter (e.g., `my_atom1` and are used to represent individual objects, relations, or propositions. Variables are syntactically similar to atoms, but start with an initial uppercase character or an underscore. If a variable starts with an uppercase character and appears multiple times in a term or clause, each instance of the variable is bound to the same value. If a variable starts with an underscore, it is an *anonymous variable*, and repeated instances of the variable within a term or clause may be bound to different values. A structure is a compound term consisting of an initial atom, which is called the *functor*, and a sequence of terms, which are called *arguments*. Anonymous variables are typically used when the specifc value of a variable is not of interest but a variable must be provided to properly form a structure (i.e., with the appropriate number and placement of arguments).

There are two types of Prolog clause: *rules* and *facts*. Rules are clauses of the following form.

```
HEAD :- BODY.
```

In a rule, the `HEAD` is an atom or a structure. The atom or structure at the head of a rule is called a *predicate*. The `BODY` of a rule consists of a set of predicates, called the *goals* of the rule, which may be joined by a set of special logical operators. The operators '`,`' and '`;`' denote conjunction and disjunction respectively. The operator `\+` denotes the *not* operator, which provides negation as failure, and may be read as "it is not provable that ...". Operator precedence may be explicitly controlled using parentheses. There may be multiple rules with the same head, and the head may reappear in the rule body, allowing for recursion. Semantically, a rule can be read as an assertion that "`HEAD` (is true) if `BODY` (is true)". From a procedural point of view, a rule can be viewed as an instruction stating "To prove `HEAD`, prove `BODY`". Finally, a fact is a clause of the following form, where `FACT` is an atom or structure.

```
FACT.
```

Semantically, facts are interpreted as assertions that the corresponding predicate is true. Procedurally, a querying a fact always succeeds.

## Setup

To illustrate the proposed approach, we will analyze a basic version of the simple reaction time task (SRT; see, e.g., Deary, Liewald, & Nissan, 2011; Niemi & Näätänen, 1981, for analyses). For the present paper, we assume XELs are Prolog programs primarily listing facts of the following form.

```
happens(Event, Time).
```

The binary predicate `happens/2` represents an event log entry, the `Event` variable is a symbol designating the event type and the variable `Time` is a timestamp. In terms of the event calculus, facts about the `happens/2` predicate are event axioms. The nullary predicate `invalid/0` will be our running example of an XTS program.

An SRT has one stimulus and one designated response. Subjects are tasked with issuing the response as quickly as possible on presentation of the stimulus. On each trial, the stimulus is presented with a randomized delay, called the stimulus onset asynchrony (SOA) and a timeout occurs if the

```
% Metadata

machine(e61f75505a65).
refresh_rate(e61f75505a65, 60).

subject(s1).
machine(s1, e61f75505a65).

% Event Log

happens(session_start(s1), 0).
happens(stimulus_onset(s1), 672).
happens(press(s1, space), 846).
happens(stimulus_offset(s1), 855).
happens(stimulus_onset(s1), 1705).
happens(timeout(s1), 2205).
happens(stimulus_offset(s1), 2222).
happens(session_end(s1), 2231).
```

Figure 1: Sample event log for a simple reaction time task.

subject does not respond quickly enough. The basic experimental measures are durations of SOA and response latency periods.

Given the above, an XEL for an SRT may look like Figure 1. The header of this database contains metadata about experimental equipment and subjects. For instance, it states that subject `s1` ran the experiment on machine `e61f75505a65`. Subsequent lines record experimental events. Through structured event types like `press(Subject, Key)`, the format can provide detailed information about experimental events.

Even in a paradigm as simple as this one, there are many subtle design decisions and manipulations. For instance, our version of the task does not include a warning signal to mark the start of the SOA period. Other versions of the task include such signals and consequently may have larger gaps between trials. This kind of fine-grained procedural detail, which may be lost in all but the most careful verbal descriptions, is what the proposed approach aims to capture.

## The Anatomy of an XTS

It is easy enough to validate static data. For instance we may write the following checks to make sure we are not missing metadata in our example task.

```
invalid :-
    machine(M),
    \+ refresh_rate(M, _Rate).
invalid :-
    subject(S),
    \+ (machine(S, M), machine(M)).
```

The first clause indicates that an experimental record is considered invalid by the program if there is a machine for which no screen refresh rate data is available. The second clause follows in the same vein, but checks that each subject is assigned to a machine. To be clear, these checks are illustrative and certainly not complete. Just as an example, there almost certainly should be an additional clause declaring a record invalid if it associates multiple machines with a single subject.

To complete implementing the XTS, we simply need to continue adding clauses for the predicate `invalid/0` until we have complete coverage of experimental requirements. In the SRT, we are concerned about the sequencing and duration of SOA and latency periods. The correctness of an implementation of the SRT hinges primarily on whether, in every trial, there is first an SOA followed by a stimulus presentation and latency period and whether the durations of these periods are within the bounds set by the experimental design. The timing and order of experimental events matter in as much as they affect the occurrence and duration of these periods of interest. Such constraints may be expressed in the event calculus.

### Event Calculus Interlude

This paper uses a slight variant of the simplified event calculus (SEC; see Mueller, 2008, for a survey of different variants). The main difference from the ordinary formulation is that, in the present variant, all fluents are initially negative. One important caveat about the SEC is that it may arrive at incorrect conclusions with incomplete event information. For this paper, we will assume that we are working with complete experimental records.

The core (domain-general) axioms of the calculus are presented in Figure 2. In particular, the Prolog predicates `holds_at/2` and `clipped/3` render the main axioms of the classical SEC. The predicate `holds_at/2` says that a fluent holds at some chosen point in time if an event that initiates the fluent happens at some prior point in time and it cannot be proven that the fluent is clipped between the time the initiating event happens and the chosen point in time. The predicate `clipped/3` says that a fluent is clipped in the interval between two time points if an event that terminates the fluent happens at some time point within the interval. Domain-specific axioms provide rules for evaluating the predicates `initiates/3`, `terminates/3`, and `happens/2` which appear in these definitions. To the core axioms, we add an additional predicate, for retrieving periods in which fluents hold, called `holds_in/3`. Note that this predicate does not pick out *maximal* periods in which fluents hold.

Having established the basic axioms of SEC, we can define some general validity constraints for XELs. The clauses below check that only explicitly declared events and fluents are used in XELs and that all fluents are eventually terminated. These checks help guard against subtle coding errors.

```
invalid :-
    happens(E, _Time),
    \+ event(E).
invalid :-
    happens(Event, Time),
    initiates(Event, F, Time),
```

```
initiation(Fluent, Time) :-
    happens(Event, Time),
    initiates(Event, Fluent, Time).

termination(Fluent, Time) :-
    happens(Event, Time),
    terminates(Event, Fluent, Time).

holds_at(Fluent, Time) :-
    initiation(Fluent, Time0),
    Time0 < Time,
    \+ clipped(Time0, Fluent, Time).

clipped(Time0, Fluent, Time1) :-
    termination(Fluent, Time),
    Time0 < Time,
    Time < Time1.

holds_in(Fluent, Time0, Time1) :-
    initiation(Fluent, Time0),
    termination(Fluent, Time1),
    \+ clipped(Time0, Fluent, Time1).
holds_in(Fluent, Time0, inf) :-
    initiation(Fluent, Time0),
    \+ (termination(Fluent, Time1),
        Time0 < Time1).
```

Figure 2: Basic definitions for a variant of the SEC in SWI-Prolog. The first two clauses serve to associate fluents with time points in which they are actually initiated or terminated.

```
    \+ fluent(F).
invalid :-
    happens(Event, Time),
    terminates(Event, F, Time),
    \+ fluent(F).
invalid :- holds_in(_Fluent, _Time, inf).
```

## Characterizing Experimental Narratives

We can now start characterizing the expected structure of experimental narratives in SRTs. This process minimally involves formulating definitions for the following constructs:

- Experimental events,
- Experimental fluents,
- Effect axioms,
- Constraints on the experimental narrative.

Through the first three of these activities, we establish the basic structure of the task and give ourselves the means to express vailidity constraints. Finally, in the last step, we explicitly express essential requirements and rule out edge cases. In the following, we work through each step.

**Defining Events and Fluents** For our basic SRT, it is sufficient to provide the following experimental event definitions.

```
event(session_start(_Subject)).
event(session_end(_Subject)).
event(stimulus_onset(_Subject)).
event(stimulus_offset(_Subject)).
event(press(_Subject, _Key)).
event(timeout(_Subject)).
```

We use experimental fluents to represent SOA and latency periods. We also add fluents for tracking experimental sessions and stimulus presentations. Trials are not explicitly represented in the interest of simplifying the presentation.

```
fluent(session(_Subject)).
fluent(stimulus(_Subject)).
fluent(soa(_Subject)).
fluent(latency(_Subject)).
```

Note that, like our experimental events, our experimental fluents are indexed by subject ID.

**Defining Effect Axioms** Now, we must provide effect axioms relating experimental events to each experimental fluent. For the `session/1` fluent, these axioms are straightforward.

```
initiates(session_start(S), session(S), _).
terminates(session_end(S), session(S), _).
```

The first axiom states that the session fluent for some subject `S` is initiated by a session start event for that subject. Likewise, the second axiom states that the session fluent for a subject `S` is terminated by a session end event for that subject. There are a number of possible events that may constitute the session start or end, these are abstracted away as they are not essential to the experiment.

We provide similarly simple effect axioms for the fluent `stimulus/1`.

```
initiates(stimulus_onset(S), stimulus(S), _).
terminates(stimulus_offset(S), stimulus(S), _).
```

These axioms state that stimulus periods are initiated by stimulus onset events and terminated by stimulus offset events.

Next, we analyze the `soa/1` fluent. The effect axioms for this fluent are slightly more complex.

```
initiates(session_start(S), soa(S), _).
initiates(stimulus_offset(S), soa(S), T) :-
    initiation(stimulus(S), T1),
    T < T1.
terminates(stimulus_onset(S), soa(S), _).
```

The first effect axiom here states that an SOA begins as soon as the experimental session starts. This means that any preliminary information screens, consent forms, and other questionnaires are not considered part of the SRT session. There isn't any hard-and-fast rule to include or exclude these elements. Here, we are focusing only on the essential features of the SRT and thus glossing over some of the procedural details. Before treating the second axiom, let us first look at the `terminates/3` predicate. This predicate tells us that, at any

time, an SOA is terminated for a subject if a stimulus onset event occurs for that subject at that time. To correctly deal with this setup, the second effect axiom exhibits a *precondition*. This axiom says that a stimulus offset event initiates a new SOA for the subject at time `T` if it can be proved that a stimulus presentation occurs for that subject at a later time `T1`. The precondition prevents the period after the last stimulus offset from being considered an SOA.[1]

The final set of effect axioms characterize latency periods.

```
initiates(stimulus_onset(S), latency(S), _).
terminates(press(S, space), latency(S), T) :-
    holds_at(latency(S), T).
terminates(timeout(S), latency(S), _).
```

These axioms simply state that a latency period begins simultaneously with stimulus onset and that it ends on response or timeout. We can also see from these axioms that the designated response for our SRT is a press on the space bar. Note that the press event is only held to terminate the latency fluent if, at the time the press occurs, the latency fluent holds.

**Defining Validity Constraints** We are done specifying the basic form of the task. We now move on to declare new clauses for `invalid/0` to enforce our validity constraints.

We start off with some basic checks. First, we stipulate that we do not expect fluent initiating (terminating) events to occur at times where it can (cannot) be proved that the relevant fluents hold.

```
invalid :-
    fluent(F),
    initiation(F, Time),
    holds_at(F, Time).
invalid :-
    fluent(F),
    termination(F, Time),
    \+ holds_at(F, Time).
```

These checks eliminate various problem cases including, for example, situations in which multiple successive stimulus onsets appear in the record. That said, these constraints are rather strong as they effectively disallow any event where even just one of the fluents that may be affected remains unchanged. For instance, stray key presses would be caught by the second clause defined above had we not placed a precondition allowing keypress effects only when the latency fluents hold.

Next, we define constraints concerning the `session/1` fluent. The first clause ensures that, for each subject, the `soa/1` and `latency/1` fluents are both bounded by the `session/1` fluent.

```
invalid :-
    holds_in(session(S), SessStart, SessEnd),
    (Fluent = soa(S); Fluent = latency(S)),
    holds_in(Fluent, Time0, Time1),
    (Time0 < SessStart; SessEnd < Time1).
```

The second and third clauses together ensure that each subject is associated with exactly one session.

```
invalid :-
    subject(S),
    \+ holds_in(session(S), _T0, _T1).
invalid :-
    holds_in(session(Subject), Start1, End1),
    holds_in(session(Subject), Start2, End2),
    (Start1 =\= Start2; End1 =\= End2).
```

Next, we introduce some constraints on the relative timings of the `stimulus/1`, `soa/1` and `latency/1` fluents. The stipulations, enforced by the following three clauses are as follows. Firstly, the `stimulus/1` and `soa/1` fluents for the same subject may not overlap. This does allow for the fluents to overlap if they are associated with different subjects, permitting setups where multiple subjects run the task simultaneously. Second, these stipulations also require that, for every stimulus period, there should be a latency period initiated concurrently with the stimulus period and terminated by the termination of the stimulus period.

```
invalid :-
    holds_in(soa(S), T11, T12),
    holds_in(stimulus(S), T21, T22),
    T11 < T21,
    \+ T12 < T21.
invalid :-
    holds_in(soa(S), T11, T12),
    holds_in(stimulus(S), T21, T22),
    T21 < T11,
    \+ T22 < T11.
invalid :-
    holds_in(stimulus(S), T0, T1),
    \+ (holds_in(latency(S), T0, T2), T2 =< T1).
```

Note that our effect axioms, taken together with the preceeding validity constraints, guarantee that, for each subject, SOA and latency periods alternate, starting on an SOA period and ending on a latency period. They also ensure that, for each subject, the latency fluent never holds outside the bounds of stimulus periods for that subject.

Finally, we define some constraints that directly touch our experimental variables, the SOA and latency durations. The first clause below ensures that SOAs are no longer than 2000ms and no shorter than 500ms. The second clause ensures that latency periods are no longer than 500ms, when the timeout event should occur.

```
invalid :-
    holds_in(soa(_S), T0, T1),
    (T1 - T0 < 500; 2000 < T1 - T0).
invalid :-
```

[1]This is, admittedly, not the most elegant solution as we refer to future events in the effect axiom precondition. A more elegant solution would have been available if we were counting trials. In that scenario, the effect axiom could have been conditioned on whether a sufficient number of trials had been run.

```
    holds_in(latency(_S), T0, T1),
    500 < T1 - T0.
```

As a final check on the resulting specification, we can ask whether it can be used to correctly define the experimental measures. The reader is invited to check that the following clauses correctly compute SOAs and response latencies given the XTS that we have developed.

```
soa(Subject, Time) :-
    holds_in(soa(Subject), Time0, Time1),
    Time is Time1 - Time0.

rt(Subject, Time) :-
    holds_in(latency(Subject), Time0, Time1),
    Time is Time1 - Time0.
```

## Discussion

In this paper, we have developed a logic program that, when given an experimental event log, decides whether the data has the form we would expect if it had been produced by an SRT. In writing this program, we encoded the structure of the SRT in a formal representation at a high level of granularity. This exercise serves two purposes. First, it presents a step-by-step demonstration of the XTS development process and a concrete example of the proposed task representation. Second, it concretely substantiates the key premise for motivating XTSs in the first place, that even the simplest of experimental tasks have intricate narrative structure, by making explicit many details of this structure in the SRT. Indeed, explicitation of task structure is the primary motivation for developing XTSs, as it has the potential to curb omissions or errors in communications, implementations, and analyses. It is worth noting that the XTS presented here is still in need of some further elaboration. For instance, it may be necessary to add constraints to verify that SOAs are distributed appropriately within the 500-2000 msec range and add other checks against mal-formed records (e.g., two participants working on the same machine at the same time). Writing an XTS is a difficult and non-trivial endeavor that requires careful analysis of the target task.

Formalizations of psychological theory and experimental designs (in the statistical sense) are viewed rather positively (Eronen & Romeijn, 2020; Marewski & Olsson, 2009), but there appears to be little work specifically targeting the formalization of the procedural details of experimental protocols (though see Balduccini & Girotto, 2010). Aside from task analysis techniques, which were discussed in the introduction, experiment source code may, in some sense, be viewed as formalizing experimental protocols. Some software packages for implementing psychological experiments define rather intricate abstractions (e.g., Krause & Lindemann, 2014). Nevertheless, XTSs seem better suited for the documentation, abstraction, and validation roles targeted here as they are very flexible, can abstract away irrelevant implementation details, and are explicitly normative formalizations rather than procedural specifications. In general, XTSs can help detect incorrect experiment implementations by checking experimental event logs for structural and narrative integrity and double as documentation for experimental protocols.

There are several possible refinements to the proposed methodology which warrant further exploration. Some specifications may require soft constraints or robustness to occasional failures. For instance, checks addressing random data, such as the aforementioned check on the distribution of SOA durations must tolerate occasional spurious failures. Furthermore, it would be beneficial for XTS programs to return information on the causes of a failed validation. Such reporting functionality was omitted here to keep the presentation simple, but the development and debugging processes would have greatly benefited from the inclusion of such a feature. Finally, although SEC was sufficient for the purposes of the present demonstration, it may prove too limited for more advanced uses of XTSs. In the future, it may be worth developing a more powerful, standardized, open-source tool for writing XTSs to ensure harmonization and correctness.

The present proposal invites a formal way of thinking about psychological experimental tasks that is well-aligned with theoretical computer science. This opens some new and interesting avenues for theoretical and computational work, not to mention possibilities for deeper integration between two disciplines of cognitive science. Namely, by associating experimental tasks with algorithms that characterize them, we raise the possibility to bring to bear analytical tools from computer science on experimental tasks. For instance, it may be possible to formally trace and catalog similarities, differences, and other structural relationships between experimental paradigms by constructing and algorithmically analyzing a catalog of XTSs. Such a catalog would also be a valuable reference that may facilitate the development of more widely-scoped theories.

A particularly interesting prospect in the direction outlined above is that of applying automated reasoning to the design and analysis of experimental tasks in relation to psychological hypotheses. In defining XELs and XTSs, we have focused exclusively on operational aspects of psychological experiments, yet we need not stop there. For instance, if psychological theories are encoded in logic-based knowledge representations, it may be possible to, for instance, formally check experimental paradigms against the hypotheses they are designed to test or to generate experimental paradigms based on user-specified hypotheses. A promising sign in this regard is that logic progamming techniques have already had some success representing and modeling psychological theories (Balduccini & Girotto, 2010, 2011; Inclezan, 2015).

Being directly inspired by and having parallel goals to the use of formal verification in computer science, the present effort is subject to some important challenges associated with the use of such techniques. Indeed, the proposal can be situated in the broader context of a cluster of efforts attempting to apply formal verification techniques in various fields

that study the behavior of complex systems, ranging from human-automation interaction (Bolton, Bass, & Siminiceanu, 2013) to complex biomolecular processes (Calder & Hillston, 2009). For the present proposal, there seems to be three main issues under the general headings of usability and adoption. The first is a barrier to entry arising from the cost of acquiring the necessary technical knowledge and skills to use the event calculus and Prolog (or possible alternative logical formalism and logic programming paradigm). The second is the apparent necessity to write a lot of 'boilerplate' code to cover what seem to be obvious commonsense constraints, like the fact that a single user cannot simultaneously use two machines in an SRT. The third is the danger that XTSs may end up being incomplete in significant ways. These challenges must be addressed if the potential of the current proposal is to be realized.

One meta-challenge here is that it is not totally clear how each of the three general challenges will exactly manifest themselves in practice. Therefore, continued experimentation with the proposed methodology is a prerequisite for meaningful progress. Nevertheless, it is possible to offer some general directions for future work aimed towards addressing the three challenges. There seem to be two general ways in which the first challenge may be tackled. The first is to work on reducing the barrier to entry by developing user-friendly tools, and the second is to demonstrate the utility of the technique through application. To address second challenge, common patterns in experimental task design can be abstracted and collected in a library, thereby reducing the need to write 'boilerplate' constraints. As for the third challenge, it is easy, in theory, to demonstrate that an XTS is incompleted by constructing a passing XEL that is clearly flawed. However, performing such tests in a systematic way may be difficult and costly, and automation may only be possible through the development of dedicated heuristics. All considered, the proposed approach seems to touch on an interesting direction for further interdisciplinary work in cognitive science.

## Acknowledgements

I would like to thank Selmer Bringsjord for his encouragement, without which this paper would not have been written. Many thanks are also due to Mike Giancola and the reviewers for their valuable comments and suggestions.

## References

Balduccini, M., & Girotto, S. (2010). Formalization of psychological knowledge in answer set programming and its application. *Theory and Practice of Logic Programming*, *10*(4-6), 725–740. doi: 10.1017/S1471068410000384

Balduccini, M., & Girotto, S. (2011). ASP as a cognitive modeling tool: Short-term memory and long-term memory. In M. Balduccini & T. C. Son (Eds.), *Logic programming, knowledge representation, and nonmonotonic reasoning.* Berlin: Springer.

Bartko, I. (1986). *Prolog programming for artificial intelligence*. Reading, MA: Addison-Wesley.

Bolton, M. L., Bass, E. J., & Siminiceanu, R. I. (2013). Using formal verification to evaluate human-automation interaction: A review. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *43*(3), 488-503. doi: 10.1109/TSMCA.2012.2210406

Calder, M., & Hillston, J. (2009). Process algebra modelling styles for biomolecular processes. In C. Priami, R.-J. Back, & I. Petre (Eds.), *Transactions on Computational Systems Biology XI* (pp. 1–25). Berlin, Heidelberg: Springer Berlin Heidelberg.

Deary, I. J., Liewald, D., & Nissan, J. (2011). A free, easy-to-use, computer-based simple and four-choice reaction time programme: The Deary-Liewald reaction time task. *Behavior Research Methods*, *43*, 258–268.

Diaper, D. (2004). Understanding task analysis for human-computer interaction. In D. Diaper & N. Stanton (Eds.), *The handbook of task analysis for human-computer interaction.* Lawrence Erlbaum Associates.

Eronen, M. I., & Romeijn, J.-W. (2020). Philosophy of science and the formalization of psychological theory. *Theory & Psychology*, *30*(6), 786-799. doi: 10.1177/0959354320969876

Inclezan, D. (2015). An application of answer set programming to the field of second language acquisition. *Theory and Practice of Logic Programming*, *15*(1), 1–17. doi: 10.1017/S1471068413000653

Krause, F., & Lindemann, O. (2014). Expyriment: A python library for cognitive and neuroscientific experiments. *Behavior Research Methods*, *46*(2), 416–428.

Marewski, J. N., & Olsson, H. (2009). Beyond the null ritual: Formal modeling of psychological processes. *Zeitschrift für Psychologie/Journal of Psychology*, *217*(1), 49–60.

Mueller, E. T. (2008). Event calculus. In F. van Harmelen, V. Lifschitz, & B. Porter (Eds.), *Handbook of knowledge representation.* Amsterdam, Netherlands: Elsevier B.V.

Niemi, P., & Näätänen, R. (1981). Foreperiod and simple reaction time. *Psychological bulletin*, *89*(1), 133.

Shanahan, M. (1999). The event calculus explained. In M. Wooldridge & M. Veloso (Eds.), *Artificial intelligence today.* Springer.

Wielemaker, J., Schrijvers, T., Triska, M., & Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, *12*(1-2), 67–96.