# Knowledge-Assisted Reasoning of Model-Augmented System Requirements with Event Calculus and Goal-Directed Answer Set Programming*

Brendan Hall[1] Sarat Chandra Varanasi[2] Jan Fiedor[3] Joaquín Arias[4]
Kinjal Basu[2] Fang Li[2] Devesh Bhatt[1] Kevin Driscoll[1] Elmer Salazar[2]
Gopal Gupta[2]

[1]Honeywell Advanced Technology, Plymouth, USA
[2]The University of Texas at Dallas, Richardson, USA
[3]Honeywell International s.r.o & Brno Univ. of Technology, Brno, Czech Republic
[4]Universidad Rey Juan Carlos, Madrid, Spain
[1,3]{brendan.hall, jan.fiedor, devesh.bhatt, kevin.driscoll}@honeywell.com
[2]{sarat-chandra.varanasi, kinjal.basu, fang.li, ees101020, gupta}@utdallas.edu
[4]joaquin.arias@urjc.es

We consider requirements for cyber-physical systems represented in constrained natural language. We present novel automated techniques for aiding in the development of these requirements so that they are consistent and can withstand perceived failures. We show how cyber-physical systems' requirements can be modeled using the *event calculus* (EC), a formalism used in AI for representing actions and change. We also show how answer set programming (ASP) and its query-driven implementation s(CASP) can be used to directly realize the event calculus model of the requirements. This event calculus model can be used to automatically validate the requirements. Since ASP is an expressive knowledge representation language, it can also be used to represent contextual knowledge about cyber-physical systems, which, in turn, can be used to find gaps in their requirements specifications. We illustrate our approach through an altitude alerting system from the avionics domain.

## 1 Introduction

Developing effective requirements is crucial for success in building a system. The earlier the requirements are validated, the fewer problems system developers will encounter later. Current automation of requirements engineering tasks attempt to ensure their consistency and adequacy, namely, that they can withstand perceived failures. However, such automated support remains limited. In this work, we present novel automated techniques for aiding the development of model-augmented requirements that are adequate—to the extent that adequacy can be established—and consistent. Thus, we can have more confidence in the requirements. We limit ourselves to requirements for cyber-physical systems, particularly those in avionics. We assume that requirements are generated within the MIDAS (Model-Assisted Decomposition and Specification) [11] environment, and are expressed in CLEAR (Constrained Language Enhanced Approach to Requirements) [10], a constraint natural requirement language based on EARS [14].

*UT Dallas authors are supported by NSF (grants IIS 1718945, IIS 1910131, IIP 1916206), Amazon and DoD. Support for this project also comes from the FIT BUT internal project FIT-S-20-6427 and by the H2020 ECSEL project Arrowhead Tools. We would also like to dedicate our work to the memory of second author's Father.

Hossein Hojjat and Bishoksan Kafle (Eds.): 8th Workshop on
Horn Clauses for Verification and Synthesis (HCVS 2021)
EPTCS 344, 2021, pp. 79–90, doi:10.4204/EPTCS.344.6

Our main contribution in this work is to show how the *Event Calculus* (EC) [19, 20] and *Answer Set Programming* (ASP) [9] can be used to formalize constrained natural language requirements for cyber-physical systems and perform knowledge-assisted reasoning over them. ASP is a logic-based knowledge representation language that has been prominently used in AI. Our work builds upon recent advances made within the s(CASP) system [2], a query-driven (or goal-directed) implementation of predicate ASP that supports constraint solving over reals, permitting the faithful representation of time as a continuous quantity. The s(CASP) system permits the modeling of event calculus directly [3]. An advantage of using the event calculus—in contrast to automata and Kripke structure-based approaches—is that it can directly model cyber-physical systems, thereby avoiding "pollution" due to (often premature) design decisions that must be made otherwise. The event calculus is a formalism—a set of axioms—for modeling dynamic systems and was proposed by artificial intelligence researchers to solve the *frame problem* [19, 20]. Our methods have been developed within the MIDAS framework of Honeywell [11]. The MIDAS framework embodies the essence of the *Property Management Model* [16] within a model-augmented requirement ecosystem, which integrates constrained natural language requirements within a layered information model expressed in Object Process Model (OPM) [7]. The details of MIDAS, PMM and OPM are not important here, instead, the primary goal of this work is to explore how constrained natural language requirements, specified within MIDAS, using the CLEAR notation, can be automatically reasoned about and analyzed using the event calculus and query-driven answer set programming. Specifically, we explore:

1. How ASP-based model checking (over dense time) can validate specified system behaviors wrt system properties.
2. How application of *abductive reasoning* can extend ASP-based model checking to incorporate domain knowledge and real-world/environmental assumptions/concerns.
3. How knowledge-driven analysis can identify typical requirement specification errors, and/or requirement constructs which exhibit areas of potential/probable risk.

The rest of the paper is organized as follows. Section 2 introduces the enabling technologies of answer set programming (ASP), a goal-directed implementation of ASP called s(CASP), and the event calculus. Section 3 discusses the CLEAR notation for specifying requirements, and illustrates it with an example. Section 4 presents an application developed within Honeywell Corporation using our approach. We also illustrate requirement defect discovery using s(CASP) for property-based model-checking as well as discuss how more general knowledge of potential requirements defects may detect defects that traditional techniques may not be able to find. Section 5 presents our conclusions and future work.

The main contribution of our research is to show how the event calculus and goal-directed answer set programming together serve as a promising framework for modeling and reasoning over cyber-physical and avionics systems' requirements. We also demonstrate the utility of abductive reasoning to find the assumptions under which a property holds or does not hold (along with automatically providing justification in both cases) for requirements specification. Given that ASP is a knowledge representation language, we also illustrate how knowledge about system context (e.g., processing platform may malfunction causing them to be reset) can be utilized to automatically find gaps in requirements as well.

# 2 Enabling Technologies

## 2.1 Answer Set Programming

Answer Set Programming (ASP) is a declarative paradigm that provides the stable model semantics for logic programs (with negation-as-failure). ASP is a highly expressive paradigm that can elegantly ex-

press complex reasoning methods, including those used by humans, such as default reasoning, deductive and abductive reasoning, counterfactual reasoning, constraint satisfaction, etc. [4, 9].

The rules in an ASP program are of the form: `p :- q1, ..., qm, not r1, ..., not rn.` where $m \geq 0$ and $n \geq 0$. Each of `p` and `qi` ($\forall i \leq m$) is a literal, and each `not rj` ($\forall j \leq n$) is a *naf-literal* (`not` is a logical connective called *negation-as-failure* or *default negation*). The literal `not rj` is true if proof of `rj` *fails*. Negation as failure allows us to take actions that are predicated on failure of a proof. Thus, the rule `r :- not s.` states that `r` can be inferred if we fail to prove `s`. Note that in the rule above, `p` is optional. Such a headless rule is called a constraint, which states that conjunction of `qi`'s and `not rj`'s should yield *false*. Thus, the constraint `:- u, v.` states that `u` and `v` cannot be both true simultaneously in any model of the program (a model is called an answer set).

ASP also allows for assumption-based reasoning via *abduction* [12]. Abduction is a form of reasoning concerned with the generation and evaluation of explanatory hypotheses. In abductive reasoning, given the premise $P \Rightarrow Q$, and the observation $Q$, one surmises or *assumes* (*abduces*) that $P$ holds. More generally, given a theory $T$, an observation $O$, and a set of abducibles $A$, then $E$ is an explanation of $O$ (where $E \subset A$) if:

1. $T \cup E \models O$
2. $T \cup E$ is consistent

We can think of abducibles $A$ as a set of assumptions. Abduction allows us to find “gaps" in our knowledge that must entail a property in order for that property to hold.

## 2.2 The s(CASP) System

Considerable research has been done on answer set programming since the inception of the stable model semantics that underlies it [9]. A major problem with ASP implementations is that programs have to be grounded and SAT-solver-based implementations such as CLASP [8] used to execute the propositionalized program to find the answer sets. There are multiple problems with this SAT-based implementation approach, which include exponential blowup in program size, having to compute the entire model, and not being able to produce a justification for a conclusion. Goal-directed implementations of ASP [2], called s(ASP) and s(CASP), work directly on predicate ASP programs (i.e., no grounding is needed) and are query-driven (similar to Prolog). The s(ASP) and s(CASP) systems only explore the parts of the knowledge-base that are needed to answer the query, and they provide a proof tree that serves as justification for the query. The s(ASP) and s(CASP) systems support predicates with arbitrary terms as arguments as well as constructive negation [2].

In the work reported here, we will mainly use the s(CASP) system that additionally supports constraint solving over reals, which is important for reasoning faithfully about (continuous) time. The s(CASP) system is the key technology for representing and analyzing CLEAR requirements modeled with the event calculus. The s(CASP) system also directly supports abductive reasoning and can provide justification for a query.

## 2.3 Event Calculus

The Event Calculus [19, 20] is a formalism for modeling dynamic systems which originated from the field of AI where it was devised to solve the *frame problem*. The event calculus is organized around events and fluents. We use the event calculus formalism to model an evolving world in a cyber-physical system. The EC models the world that is changing due to *events* that are happening that, in turn, influence *fluents*, where a fluent is a variable that changes with time such as location of an object or temperature of the boiler. EC provides axioms for stating the conditions under which events *happen* or conditions under which events *initiate*, *terminate*, or *release* a fluent. EC also provides axioms for when a fluent *holds*. EC models the laws of inertia that hold in the real world. For instance, the location (a fluent) of

**BEC1.** $StoppedIn(t_1,f,t_2) \equiv \exists\, e,t\ (Happens(e,t) \wedge\ t_1 < t < t_2 \wedge (Terminates(e,f,t) \vee Releases(e,f,t)))$

**BEC2.** $StartedIn(t_1,f,t_2) \equiv \exists\, e,t(Happens(e,t) \wedge\ t_1 < t < t_2 \wedge (Initiates(e,f,t) \vee Releases(e,f,t)))$

**BEC3.** $HoldsAt(f_2,t_2) \leftarrow Happens(e,t_1) \wedge Initiates(e,f_1,t_1) \wedge Trajectory(f_1,t_1,f_2,t_2) \wedge \neg StoppedIn(t_1,f_1,t_2)$

**BEC4.** $HoldsAt(f,t) \leftarrow InitiallyP(f) \wedge \neg StoppedIn(0,f,t)$

**BEC5.** $\neg HoldsAt(f,t) \leftarrow InitiallyN(f) \wedge \neg StartedIn(0,f,t)$

**BEC6.** $HoldsAt(f,t_2) \leftarrow Happens(e,t_1) \wedge Initiates(e,f,t_1) \wedge\ t_1 < t_2 \wedge \neg StoppedIn(t_1,f,t_2)$

**BEC7.** $\neg HoldsAt(f,t_2) \leftarrow Happens(e,t_1) \wedge Terminates(e,f,t_1) \wedge\ t_1 < t_2 \wedge \neg StartedIn(t_1,f,t_2)$

Figure 1: Formalization of BEC (Basic Event Calculus) axioms [17].

an object does not change unless the event "move" happens. Event calculus uses reasoning over time, as events are occurrences in time. The Event Calculus is elegantly modeled using ASP, in particular, with the query-driven s(CASP) system due to its support for real-time constraints.

The basic event calculus axioms are shown in Fig. 1. The `happens/2` predicate models the occurrence of an event $e$ at a particular time instance $t_i$, while `holdsAt/2` specifies the condition under which a fluent $f$ holds at time $t_j$. The axioms are easy to follow. More details can be found elsewhere [17, 3].

In a cyber-physical system, events correspond to actuator actions and fluents correspond to effects of actuators. Fluents can also describe the conditions under which events can occur. A fluent is an entity whose value may change over time and, thus, in case of a CPS, they reflect changes in sensor readings as well as internal component(s) state. Fluents may be impacted by events, e.g, furnace shutoff (event) may lead to temperature (fluent) to drop. *The state of a system is represented by the values of all its fluents.* The event calculus simply models how events lead to changes in fluents. Fluents, in turn, may trigger events, e.g., temperature reaching 80 degrees may cause furnace shutoff. Thus, *one can model a cyber physical system directly in the event calculus without having to make any representational assumptions*. This is in contrast to other formalisms such as automata-based or Kripke structure-based approaches where one has to first discern the states of the system followed by the transitions between these states and then model it. The discerning of states and transitions for these methods is done too early in the design process leading to (possibly erroneous) assumptions being incorporated into the design. *In the event calculus, states are implicit, as a state is defined as consisting of values of all fluents at any given moment. Because of the implicit nature of system states, a state needs to be constructed only at the time we wish to verify something, e.g., a property.*

# 3 CLEAR Notation for Requirements

Constrained Language Enhanced Approach to Requirements (CLEAR) extends the Easy Approach to Requirements Syntax (EARS) developed initially by Alistair Mavin [14]. EARS is an industrially pragmatic approach based on using five structured templates and keywords. Studies [15, 13] have shown that use of EARS reduces requirements errors while improving requirement quality and readability. The EARS keywords and templates are illustrated in Figure 2.

Although the EARS approach is often presented as a syntactic guide to writing requirements, we have found the power of EARS to be more related to the *mindset* than *syntax*. The different perspectives that are explored using EARS-based keywords stimulate a lot of discussion relating to the core intent of the requirement under consideration. The separation and clarification of event and state semantics, using the `WHILE` and `WHEN` constructs are useful to constrain and define the behavioral details of cyber-physical systems at different levels of abstraction.

The EARS approach also calls for the separation of the nominal and off-nominal specification and

the IF keyword is reserved for defining the required response to an unwanted or exceptional condition. In practice we have found that it is often the insufficient understanding and naive assumptions regarding system failure scenarios that lead to omissions of critical requirements.

EARS designates optional and configurable behaviors/attributes with the WHERE keyword. Once again, this is also an area of potential vulnerability regarding requirement completeness. Significant hazards can emerge from misconfiguration, as illustrated by the A400M incident in Seville [21]. Hence, separating concerns helps address such issues in a more systematic fashion.

CLEAR embodies the full spirit of EARS and refines the terms used within EARS constructs to improve specificity. CLEAR also introduces additional constructs such as *Upon Initialization* to force a systematic examination of known problem areas, such as system initialization. CLEAR is used today to support automated analyses (e.g. requirement consistency checking) and software level test generation [5]. Through MIDAS, it is intended to move and automate such analysis to the system level.

### 3.1 Mapping Requirements to the Event Calculus

We discuss the mapping of CLEAR-based requirements [10] to the Extended Event Calculus formalization of Shanahan[20]. CLEAR is a notation for expressing requirements for cyber physical systems based on EARS (Easy Approach to Requirement Syntax) widely adopted in the avionics industry. For simplicity, we limit ourselves to discussing only EARS. EARS provides natural language sentence templates for an engineer specifying a system to follow for writing requirements. Keywords 'WHEN', 'WHERE' and 'IF'-'THEN' are used in these templates and play a major role. We make use of the extended event calculus formalism because it enables events to have duration. For cyber-physical real-time systems, response times are important, hence formally budgeting the allocation of time throughout the levels of function & temporal decomposition are primary concerns. Figure 2 summarizes the EARS notation. Details about EARS can be found elsewhere [14, 13].

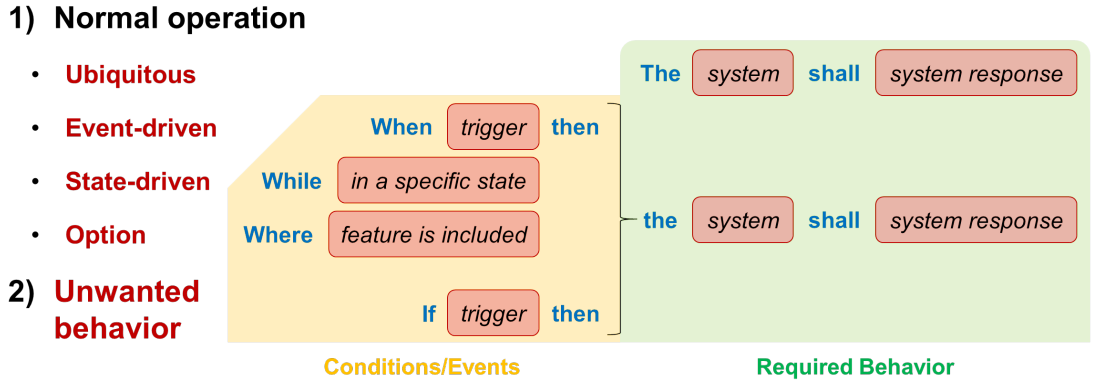


Figure 2: EARS Templates

Conceptually, the mapping of the extended event calculus to CLEAR is relatively straightforward, given that EARS already informally separates event and state semantics. To formalize the mapping, we only need to formally bound the level of temporal abstraction over which the CLEAR requirements are defined. The EARS keyword WHEN then needs to define the requirements in relation to the *upward* or *downward* perspective, with the downward facing view having the form of state invariant constraints, and the upward facing view declaring actionable behaviours. For actionable behaviours, the level of temporal abstraction also guides which of the EARS keywords to apply to the specification as outlined below:

- **WHEN**: discrete event-driven specification is used when a single observation at the level of temporal abstraction is sufficient to verify the required behavior. That is, WHEN establishes causal

relationship.

- **WHILE**: continuous state-driven specification is used when multiple observations at the level of temporal abstraction are needed to verify the required behavior. `WHILE` may also be used as a state qualifier for event-driven specifications, in cases where there are stateful conditions guarding the event.

Within MIDAS, functional influence manifests as the stateful changes to objects that surround the functional intent boundary. This maps very cleanly to the extended event calculus, once the state of the said objects at the function boundary are encoded as *fluents* within the event calculus formalism. Hence, to map the MIDAS CLEAR actionable requirements to the extended event calculus, we simply perform the following (details of MIDAS are not important here):

1. Declare each MIDAS object state as a fluent.
2. Declare `happens` predicate that characterize the discrete changes in state that triggers the `WHEN` condition.
3. Declare `initiates` and `terminates` predicates to bind the happens predicates to the influenced external state changes, with `holdsAt` state qualifiers for all `WHILE` preconditions .
4. Declare `trajectory` predicates to characterize continuous changes that evolve over multiple intervals of the temporal precision.

We illustrate the mapping of an EARS requirement related to aircraft design:

> **While** the aircraft is on-ground, **when** the requested door position becomes open, the door control system **shall** change the state of the cargo door from closed to open **within** 10 secs.

Mapping this to the EARS templates (Fig. 2) we see that the aircraft is on-ground is a precondition, the requesting of the cargo door to open is the trigger, and the opening of the cargo door within 10 seconds is the **system response**, requiring the change of environmental state, with an assumed level of temporal precision. In this requirement the functional intent manifests as a change in the physical state of the cargo door, in response to a change in state that reflects the pilot requesting the door to open. Hence, the event calculus formalism of this requirement is as follows:

```
fluent(door_requested_position(RP)). % Map object states to fluents
fluent(door_state(DS)).
fluent(aircraft_state(AS)).
happens(pilot_requests_door_to_open, T) :- % Map change of state trigger to happens
    -holdsAt(door_requested_position(open), T1),
    holdsAt(door_requested_position(open), T2), T2 #=< T1+10, T1 #< T, T #=< T2.
initiates(pilot_requests_door_to_open, door_state(open), T) :-
    holdsAt(aircraft_state(is_on_ground), T). % Map changes influenced by event
terminates(pilot_requests_door_to_open, door_state(closed),T) :-
    holdsAt(aircraft_state(on_ground), T). % Map changes influenced by event
door_response_is_correct_condition :- % A typical query verifying door behaves correctly
    -holdsAt(door_requested_position(open), TB), -holdsAt(door_state(open), TB),
    holdsAt(aircraft_state(on_ground), TH), holdsAt(door_state(open), TE),
    TB #=< TH, TB #< TE, TE #=< TB+10.
```

According to the semantics of the Extended Event Calculus [20], a fluent is considered to be initiated at any time point during the event duration. This is semantically consistent with the additional *within* constraint that the CLEAR requirement notation [10] provides. Note also that the extended event calculus formalism does not distinguish between the sensing of the environmental change and the resulting time

of establishing environmental influence. This is also consistent with the level of functional and temporal decomposition at the current level of abstraction. Hence, when validating and discharging properties and constraints above this level of specification, the event calculus model will explore the impact of the function influence starting at any point within the event duration, which in our example is 10 seconds.

Note also that the initiated action, i.e., the changing of the door state, is guarded by the aircraft state of 'on ground'. Should this hold when the respective triggering event happens, the initiates predicate will fail, and therefore there will be no external functional influence. The mapping of IF is the same as the `WHEN`, given that they are both conditioned on events. The primary difference is that IF characterizes triggering off-nominal state transitions such as component failure scenarios.

As noted above actionable `WHILE` requirements are used to characterize continuous behaviours and/or constraints that need to be expressed over multiple intervals at the current level of temporal abstraction. They map to trajectories within the extended event calculus formalism. We show an example and its mapping to EC below:-

> **While** the door is opening, the rate of change in door position shall increase positively with a maximum rate of 5 degrees per second.

```
trajectory(door_opening,T1,door_position(B),T2) :-
        holdsAt(door_position(A),T1),
        holdsAt(door_position(B),T2), A #< B, B #=< A+5, T2 #= T1+1.
door_rate_ok :-  % Constraint on rate at which door opens is true throughout
        trajectory(door_opening,T1,door_position(P),T2),
        holdsAt(door_opening,TH), T1 #=< TH, TH #=< T2.
```

ASP with EC can also be used to test the adequacy of the requirements to the extent possible. In the case of avionics, the knowledge is simple and relates to Single Event Upsets (or SEUs) [18] which can induce random resets of avionic software systems. We can model the destructive power of the SEU using a *reset* event. The *reset* event overrides the constructed internal state of the system, forcing the initialization state to be re-established. Our goal in introducing this reset is to move the designer to consider how robust the system initialization logic is to such transient resets. In ASP, an extraneous event (e.g., reset) is represented as an *abducible* (a reset may or may not happen). We would want to know if our system will still behave correctly in presence of this extraneous event. The knowledge assumes that the reset event can only override the cyber system (software state) and does not affect the continuous state of the physical world. Whether our system will behave correctly or not can be established with the abductive reasoning supported by s(CASP). This form of abductive reasoning has significant potential in helping establish the completeness of requirements for cyber-physical systems.

# 4 Demonstration Case Study

To explore the feasibility of our approach, we applied it to several systems under development at Honeywell. The primary goal of these case-studies is to explore how specialist domain knowledge can be incorporated into the requirement review process to help assess requirement sufficiency. Rather than encode hard-coded knowledge of specific cases, we are striving to illustrate the application of more general reusable knowledge. Our findings have been encouraging, and we have been able to supply useful feedback to live engineering teams. We illustrate our ideas with one of the systems developed.

## 4.1 Altitude Alerting System

The altitude alerting system should issue alerts about aircraft altitude error thresholds in a timely fashion. Figure 3 shows the alerts that should be turned on/off within one second as the absolute altitude

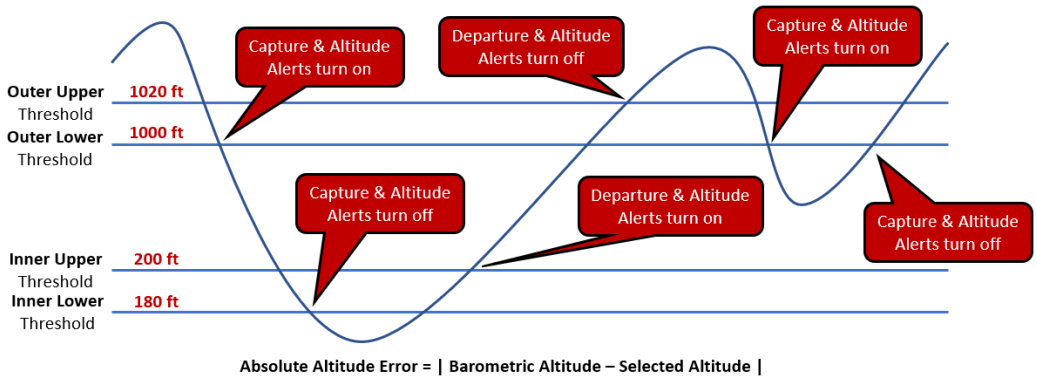


Figure 3: Altitude Alerts w.r.t Altitude Error Thresholds

crosses alert thresholds. We first present the various notions associated with the altitude alert system's requirements, and show their mapping from CLEAR to Event Calculus and ASP

During the operation of the aircraft, the pilot selects a target altitude. The alert system monitors the aircraft's barometric altitude trajectory and issues alerts when this barometric altitude becomes potentially hazardous in relation to the selected altitude. If the error crosses certain threshold values, then the system should promptly warn the pilot so that she can take appropriate action. Similarly, the alerting system can also withdraw the alerts if other threshold values are met. The two primary alerts are *capture alert* and *departure alert* explained in Figure 3. In the following, `lt` and `gt` are abbreviated for 'less than' and 'greater than', respectively. The predicate `error_becomes_lt(1000,T)` asserts if the altitude error becomes *less than* 1000 ft. at time `T`.

**REQ1:**[1] *When the **absolute_altitude_error**[2] becomes less than 1000 ft, the Altitude Alerting System shall initiate the **capture_alert** and **altitude_alert** within 1 second.*

The set of constraints, in s(CASP), in the body of `happens(init_cap_alert)` capture the notion that the event has to occur within one second of the error dropping below 1000 ft. This shows the ease of expressing continuous real-valued constraints in s(CASP).

```
happens(init_cap_alert,T) :-
error_becomes_lt(1000,T1),T1#<T,T#<T1+1.
happens(init_alt_alert,T):-
error_becomes_lt(1000,T1),T1#<T,T#<T1+1.
initiates(init_cap_alert,cap_alert_on,T).
terminates(init_cap_alert,cap_alert_off,T).
initiates(init_alt_alert,alt_alert_on,T).
terminates(init_alt_alert,alt_alert_off,T).
```

**REQ2:** *When the **absolute_altitude_error** becomes greater than 200 ft, the Altitude Alerting System shall initiate the **departure_alert** and **altitude_alert** within 1 second.*

```
happens(init_dep_alert,T) :-
error_becomes_gt(200,T1),T1#<T,T#<T1+1.
initiates(init_dep_alert,dep_alert_on,T).
terminates(init_dep_alert,dep_alert_off,T).
happens(init_alt_alert,T) :-
error_becomes_gt(200,T1),T1#<T,T#<T1+1.
```

**REQ3:** *When the **absolute_altitude_error** becomes either less than 180 ft or greater than 1020 ft, the Altitude Alerting System shall terminate all **altitude_alerts** within 1 second.*

[1]It should be noted that these requirements are significantly simpler than the original internal baseline which consisted of a state-based specification. This simplification was enabled by MIDAS' conceptual modeling approach, which forces the specification to be expressed using conceptual abstractions, over object states that surround the boundary of intent.

[2]absolute_altitude_error is a derived fluent

```
happens(term_all_alerts, T) :-
error_outside(180,1020,T1),T1#<T,T #<T1+1
initiates(term_alerts,dep_alert_on,T).
terminates(term_alerts,dep_alert_off,T).
initiates(term_alerts,cap_alert_on,T).
terminates(term_alerts,cap_alert_off,T).
initiates(term_alerts,alt_alert_on,T).
terminates(term_alerts,alt_alert_off,T).
```

**REQ4:** *While the **altitude_selection_knob** has not been '**adjusting**' within the previous 5 seconds, the Altitude Alerting System shall determine **absolute_altitude_error** as the absolute difference between the **selected_altitude** and the **barometric_altitude**, otherwise **absolute_altitude_error** shall be 0.*

```
holdsAt(altitude_abs_error(E),T) :-
        holdsAt(barometric_alt(B),T), holdsAt(selected_alt(S),T),
        abs(S,B,E),T1 #= T-5, not happensIn(adjust(Value),T1,T).
holdsAt(altitude_abs_error(0), T) :-
        happens(adjust(Value), T1, T1), T #>= T1, T #=< T1 + 5
happensIn(Event, T1, T2)          :- happens(Event, T), T1 #=< T, T #=< T2.
```

**REQ5:** *Upon initialization, the Altitude Alerting System shall consider the **altitude_selection_knob** to be '**adjusting**'.*

```
initiallyP(altitude_adjusting).
initiates(reset, altitude_adjusting, T).
```

## 4.2 Simulation of Alerting System Runs from Initial Conditions

The coding of the system requirements in event calculus and ASP is a logic program that can be executed on the s(CASP) system. We can run various queries to check if certain properties (e.g., safety properties) that we expect to hold, are entailed by the encoding. We can construct various scenarios (narratives, in event calculus parlance) and check that our requirements specifications are indeed consistent, i.e., when we do simulation runs, we do obtain a model. For example, we assume the following initial state of the system and compute when the alerts will be raised (`initiallyP(F)` means that initially fluent `F` is true and `initiallyN(F)` means initially fluent `F` is false). The fluent `start` is used to define the rate of change of barometric altitude as an event calculus trajectory[3].

With appropriate narratives[4] for the altitude alerting system, its end-to-end behavior can be understood. To find out the time at which `departure_alert_on` fluent holds, we issue the query Q1: `?- holdsAt(dep_alert_on, T).` which yields the answer `T > 30`. This is because the altitude error becomes greater than 200 ft at `T = 30`.

```
initiallyP(cap_alert_off).
initiallyP(dep_alert_off).
initiallyP(alt_alert_off).
initiallyP(altitude_adjusting).
initiallyP(selected_alt(32000)).
initiallyN(selected_alt(V)) :- V \= 32000.
trajectory(start,T1,barometric_alt(B),T2) :-
      B #= 32200 + (T2 - T1) * 10.
```

## 4.3 Adequacy of Requirements using Knowledge of Environmental Upsets

We can also test the behavior of the above scenario augmented with domain knowledge. Our goal is test the adequacy of the requirements using prior knowledge of environmental upsets. Adequacy means that in the presence of known failures, the requirements are still consistent, i.e., a model exists for them. In this case, the knowledge is simple and relates to Single Event Upsets (or SEUs) [18] which can induce random resets of avionic software systems. We can model the destructive power of the SEU using a *reset* event. The *reset* event overrides the constructed internal state of the system, forcing the initialization

[3]We have given encoding of trajectory in Basic EC, the encoding of trajectory changes only slightly in Extended EC

[4]We excluded `initiallyN(cap_alert_on)`, `initiallyN(dep_alert_on)`, `initiallyN(alt_alert_on)` for lack of space as part of the narrative given

state to be re-established. Our goal in introducing this reset is to 'move the designer to consider how robust the system initialization logic is to such transient resets. In ASP, an extraneous event (e.g., reset) is represented as an abducible (a reset may or may not happen). We would want to know if our system will still behave correctly in presence of this extraneous event. The knowledge assumes that the reset event can only override the cyber system (software state) and does not affect the continuous state of the physical world. In case of the altitude alerting system, the reset event can only affect the alert status and the altitude adjusting status, but cannot affect the barometric altitude of the aircraft or the selected altitude value. This in turn maps to the reset event terminating only the fluents that describe the internal software state of the system. The reset will not terminate fluents that describe the state evolution of continuous quantities involving the real world. In general, fluents that are outside the scope software system boundary are not affected by reset. The following generic rule specifies that the reset event terminates every fluent associated with the cyber system and initiates it to an initial value.

```
terminates(reset, Fluent, T)  :- internal_state(Fluent).
initiates(reset, Default, T)  :- default(Fluent, Default), internal_state(Fluent).
```

**Illustrating a Failing Run:** Let the predicate *in_alerting_zone*$(T)$ denote that the altitude error conditions for altitude alert ($REQ2$) are satisfied. We show that the system fails to issue critical alerts in the presence of an SEU-induced reset. We can check whether the altitude alerts are missed while in an alerting zone using the query `?- in_alerting_zone(T), holdsAt(altitude_alert_off, T).` With correct assumptions, the query would return 'no models' in s(CASP). By assuming an incorrect requirement in place of $REQ5$, the above query returns a binding in s(CASP), thereby showing a missed alert. The incorrect requirement, $REQ5'$, is stated as: *Upon initialization, the Altitude Alerting System shall consider the* ***altitude_selection_knob*** *to be* ***not*** *'**Adjusting**'*. More than likely, $REQ5$ may not even be stated by a system designer, and if stated, it is likely that it will be in the form of $REQ5'$, as one would *assume* that the pilot would not worry about adjusting the altitude knob during plane's takeoff.

Assume that the plane is cruising at 32,200 feet at time $t$ and selected altitude value is at 32300 ft. The plane's barometric altitude increases at the rate of 10 ft per second. At $t+30$, the altitude error becomes greater than 200 ft. Normally, the system should issue altitude alert at $t+30$. However, a reset occurs at $t+30$. The reset sets the altitude to not adjusting as per $REQ5'$ and also turns the altitude alert off. Due to $REQ5'$, the altitude is treated as not adjusting for any time $T > t+30$. Thus, although *in_alerting_zone*$(T)$ is true for $T > t+30$, the departure alert is never turned back on. Therefore, an alert that should be issued to the pilot is missed by the system. On the other hand, when $REQ5$ is used, the altitude error is forced to be zero upon reset. This ensures that at $T = t+35$, (5 seconds from $t+30$) the error becomes 200 ft, thereby making *in_alerting_zone*$(t+35)$ true. Again, per $REQ2$, the altitude alert would be issued and no alert is missed. Adequacy of requirements with reset is checked easily using query (Q2):
`?-happens(reset,T),T1#>=T+5,in_alerting_zone(T1),holdsAt(alt_alert_off,T2),T2#>T1.`

The above query (Q2) asks s(CASP), that if in the event of a reset, is the altitude alert turned on after 5 seconds? If Q2 succeeds, then the requirements are adequate. If Q2 fails (say when we use $REQ5$ instead of $REQ5'$) then the requirements are not adequate. This example shows the ease of validating requirements using EC and s(CASP) while using knowledge of SEUs. In summary, our ASP-coded EC model of the requirements are used to check for consistency and that they satisfy certain properties much in the spirit of model checking. *This ASP and EC model can also be elegantly used for checking adequacy of requirements against known failure patterns. The failure patterns can be induced into the model through the use of abduction directly supported in s(CASP).* This is possible due to near-zero "semantic gap" between the requirements and their mapping to ASP and EC. Checking adequacy is not direct in other approaches based on automata and Kripke structures.

| Query | s(CASP) | CLINGO |
|---|---|---|
| Query Q1, Normal Run | 0.930s | > 40 min |
| Query Q2, With Reset | 2.077s | > 40 min |
| Query Q3 With Reset | 0.081s | > 40 min |

Table 1: Query Execution Times

### 4.4 Performance

The performance of s(CASP) for above queries is given in Table 1. The queries were run on a quad code Intel i7-10510U processor at 1.8Ghz and 8GB RAM. For comparison, we also attempted to run the queries on CLINGO [8] that is based on grounding and SAT-solving. Due to large size of grounded program produced from discretization of time and altitude values, the queries generally out of memory. Note that Q3 is Q2 but rerun with REQ5 replaced by REQ5'.

## 5 Conclusions and Future Work

This paper presents a novel, systematic approach to formalizing requirement modeling, which enables the integration and leveraging of domain knowledge toward a computer-aided system validation, verification and assurance. We see this work as a first step towards a new generation of Knowledge-Assisted System Engineering (KASE) tooling. Our goal with this class of tooling is to significantly reduce system development costs by systematically addressing requirements specification defects much earlier in the development process. Our approach is unique. It enables a formal examination of preliminary conceptual models of intended behaviours, to assess adequacy and consistency with respect to world assumptions, anticipated failure modes, and environmental conditions. Our approach further enables the application of domain knowledge and prior lessons learned (known deficiency areas) to be fused with emerging specifications to more rapidly mature design robustness.

The primary catalyst for our success is not only the simplicity and elegance of the EC, ASP, and the s(CASP) system, which allows us to approach human levels of reasoning, it is also the intuitive formalism of EC being hidden behind the CLEAR natural language style of specification, that enables the power of formal methods to be introduced with minimal training and required specialist knowledge. We hope this will remove some of the barriers to broader spread of formal method adoption [6].

To the best of our knowledge, the work reported here is novel, as the event calculus has not been used outside of AI. Its application to system assurance here is novel. Even within AI, EC has not been widely adopted because of the lack of sound, query-driven implementations of negation-as-failure that can handle time faithfully as a continuous quantity. Our s(CASP) implementation of ASP solves both these problems. The work reported here also has practical value, as its results have benefited engineering teams at a major aerospace company. Compared to model checking techniques, our models are closer to the requirement specification, as states do not have to be discerned in advance. Moreover, reasoning over continuous time in EC is realized directly in the s(CASP) system.

There are many potential avenues for follow-on areas of research. First, we have already internally established the feasibility to automatically generate the ASP formal model from the MIDAS model and CLEAR requirements, using various methods, including simple definite clause grammars to more state-of-the-art symbolic natural language processing approaches [1]. We are also exploring formal architectural reasoning using models of defined functional intent and architectural intent with the abductive inference of component failures, as well as modeling more complex cyber physical systems using the method reported in this paper. Finally, we plan to apply our techniques to requirements specification analysis of more systems, in a manner similar to UPPAAL (`https://uppaal.org/casestudies/`).

## References


[1] Kinjal Basu et. al (2021): *Knowledge-driven Natural Language Understanding of English Text and its Applications*. In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, 2021*, AAAI Press, pp. 12554–12563.

[2] Joaquín Arias, Manuel Carro, Elmer Salazar, Kyle Marple & Gopal Gupta (2018): *Constraint answer set programming without grounding*. *TPLP* 18(3-4):337-354, doi:10.1017/S1471068418000285.

[3] Joaquín et al. Arias (2019): *Modeling and Reasoning in Event Calculus Using Goal-Directed Constraint Answer Set Programming*. In: *LOPSTR'19*, Springer, pp. 139–155, doi:10.1007/978-3-030-45260-5_9.

[4] C. Baral (2003): *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, doi:10.1017/CBO9780511543357.

[5] Devesh Bhatt, Brendan Hall et al. (2018): *The CLEAR Way To Transparent Formal Methods*. In: *4th Workshop on Formal Integrated Development Environment (F-IDE), FLoC 2018*.

[6] Jennifer Davis et al. (2013): *Study on the barriers to the industrial adoption of formal methods*. In: *International Workshop on Formal Methods for Industrial Critical Systems*, Springer, pp. 63–77, doi:10.1007/978-3-642-41010-9_5.

[7] Dov Dori (2011): *Object-process methodology*. In: *Encyclopedia of Knowledge Management, Second Edition*, IGI Global, pp. 1208–1220, doi:10.4018/978-1-59904-931-1.ch116.

[8] Martin Gebser et al. (2011): *Potassco: The Potsdam answer set solving collection*. *Ai Communications* 24(2), pp. 107–124, doi:10.3233/AIC-2011-0491.

[9] M. Gelfond & Y. Kahl (2014): *Knowledge representation, reasoning, & design of intelligent agents: The answer-set programming approach*. Cambridge Univ. Press, doi:10.1017/CBO9781139342124.

[10] B. Hall, D. Bhatt et al. (2018): *A CLEAR Adoption of EARS*. In: *IEEE EARS Workshop*, pp. 14–15, doi:10.1109/EARS.2018.00010.

[11] B. Hall, J. Fiedor & Y Jeppu (2020): *Model Integrated Decomposition and Assisted Specification (MIDAS)*. In: *INCOSE Int'l Symp.*, 30(1), Wiley, pp. 821–841.

[12] G. H. Harman (1965): *The Inference to the Best Explanation*. *The Philosophical Review* 74(1), pp. 88–95, doi:10.2307/2183532.

[13] Alistair Mavin & Philip Wilkinson (2010): *Big ears (the return of" easy approach to requirements engineering")*. In: *2010 18th IEEE International Requirements Engineering Conference*, IEEE, pp. 277–282, doi:10.1109/RE.2010.39.

[14] Alistair Mavin et al. (2009): *Easy approach to requirements syntax (EARS)*. In: *2009 17th IEEE International Requirements Engineering Conference*, IEEE, pp. 317–322, doi:10.1109/RE.2009.9.

[15] Alistair Mavin et al. (2016): *Listens learned (8 lessons learned applying EARS)*. In: *2016 IEEE 24th International Requirements Engineering Conference (RE)*, IEEE, pp. 276–282, doi:10.1109/RE.2016.38.

[16] Patrice Micouin (2014): *Model Based Systems Engineering: Fundamentals and Methods*. John Wiley & Sons, doi:10.1002/9781118579435.

[17] Erik T. Mueller (2014): *Common Sense Reasoning: An Event Calculus Based Approach (2nd Edition)*. Morgan Kaufmann.

[18] Eugene Normand (1996): *Single-event effects in avionics*. *IEEE Transactions on nuclear science* 43(2), pp. 461–474, doi:10.1109/23.490893.

[19] M Sergot & R Kowalski (1986): *A logic-based calculus of events*. *New Generation Computing* 4(1), pp. 67–95, doi:10.1007/BF03037383.

[20] Murray Shanahan (1999): *The event calculus explained*. In: *Artificial intelligence today*, Springer, pp. 409–430, doi:10.1007/3-540-48317-9_17.

[21] WikePedia: *2015 Seville Airbus A400M crash*. `https://en.wikipedia.org/wiki/2015_Seville_Airbus_A400M_crash`.