Knowledge-Assisted Reasoning of Model-Augmented System Requirements with Event Calculus and Goal-Directed Answer Set Programming (Extended Abstract)

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 Introduction

Developing effective requirements is crucial for success in building a system. The more complete, consistent, and feasible the requirements, the fewer problems system developers will encounter later. Current automation of requirements engineering tasks attempt to ensure completeness, consistency, and feasibility. However, such automated support remains limited. In this work, we present novel automated techniques for aiding the development of model-augmented requirements that are complete (to the extent possible), consistent, and feasible—where consistency and feasibility are validated. 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) [5] environment, and are expressed in CLEAR (Constrained Language Enhanced Approach to Requirements) [4], a constraint natural requirement language based on EARS [6].

Our main contribution in this work is to show how the *Event Calculus* [8] (EC) and *Answer Set Programming* (ASP) [3] 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 [1], 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 elegantly and directly [2]. A major 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* [8]. The primary goal of this work is to explore how constrained natural language requirements, specified within MIDAS [5] 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 to systematically capture design and intent within the MIDAS framework.
2. How ASP-based model checking (over dense time) can validate specified system behaviors wrt system properties.

© Hall, Varanasi, et al.
This work is licensed under the
Creative Commons Attribution License.

3. How application of *abductive reasoning* can extend ASP-based model checking to incorporate domain knowledge and real-world/environmental assumptions/concerns.
4. How knowledge-driven analysis can identify typical requirement specification errors, and/or requirement constructs which exhibit areas of potential/probable risk.

The talk will be organized as follows. We will give motivation for our work and discuss the importance of writing requirements that are consistent and complete. We will present how MIDAS enables a formal flow-down of functional intent through different stages of design refinement. We will summarize the two faces of requirements (outward and inward facing), as they support validation and verification objectives (respectively). We will then discuss the enabling background technologies (EC and ASP), before presenting how they can be integrated within the MIDAS platform to support our goals. We will illustrate our approach using an altitude alerting case study from an actual aerospace system and discuss adjacent real-world examples to show how one can use generalized knowledge within other system contexts. We will 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.

2 Mapping Requirements to the Event Calculus

We present the mapping of CLEAR-based requirements to the Extended Event Calculus formalization of Shanahan[8]. We adopt 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.

Conceptually, the mapping of the extended event calculus to CLEAR is relatively straightforward, given that EARS (on which CLEAR is based) 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:

1. Declare each MIDAS OPM 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 complex requirement with an example:

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 seconds.

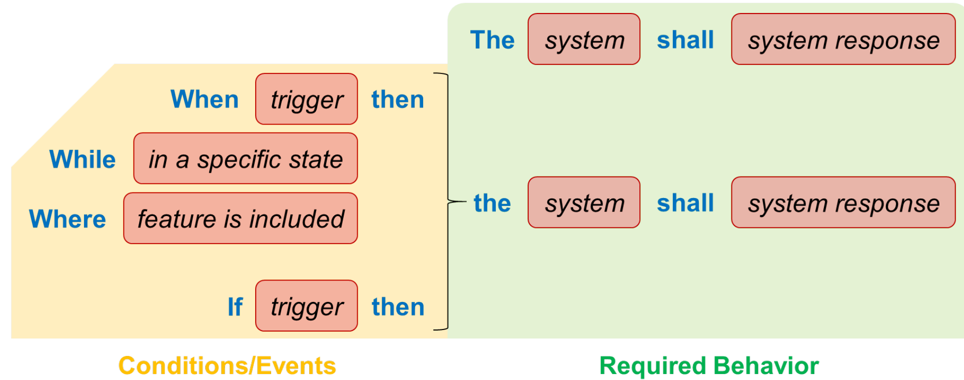

Figure 1: EARS Templates

Mapping this to the EARS templates (Fig. 1) 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:

```
% Declare fluents for respective object states. Parametrized state fluents used for simpler encoding
fluent(door_requested_position(RP)).
fluent(door_state(DS)).
fluent(aircraft_state(AS)).
% Declare `happens' predicate to reflect the change of state trigger over the duration of the event
happens(pilot_requests_door_to_open, T) :-
      -holdsAt(door_requested_position(open), T1),
      holdsAt(door_requested_position(open), T2), T2 #=< T1+10, T1 #< T, T #=< T2.
% Declare `initiates' and `terminates' predicates that map the changes influenced by the system
% response over the duration of the event
initiates(pilot_requests_door_to_open, door_state(open), T) :-
      holdsAt(aircraft_state(is_on_ground), T).
terminates(pilot_requests_door_to_open, door_state(closed),T) :-
      holdsAt(aircraft_state(on_ground), T).
% A typical query for verifying a property (door operation behaves correctly) is shown below.
door_response_is_correct_condition :-
    -holdsAt(door_requested_position(open), TB), -holdsAt(door_state(open), TB), TB #< TE,
    holdsAt(aircraft_state(on_ground), TH), holdsAt(door_state(open), TE), TE #=< TB+10, TB #=< TH.
```

According to the semantics of the Extended Event Calculus [8], a fluent is considered to be initiated at any time point during the event duration. This is semantically consistent with the *within* constraint of the CLEAR requirement. 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, i.e., the driving level

of *"the what"*, 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. Hence , these may lead to trajectories within the extended event calculus formalism. Consider the simple example 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.

At a temporal abstraction level of 1 second this becomes:

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

ASP and the event calculus can also be used to test the completeness of the requirements to the extent possible. In the case of avionics, the knowledge is simple and relates to Single Event Upsets (or SEUs) [7] 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. We are unable to give more details due to lack of space, however, abductive reasoning within ASP has significant potential in helping establish the completeness of requirements for avionics cyber-physical systems.

References

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

[2] Joaquín Arias, Zhuo Chen, Manuel Carro & Gopal Gupta (2019): *Modeling and Reasoning in Event Calculus Using Goal-Directed Constraint Answer Set Programming*. In: *LOPSTR'19*, Springer, pp. 139–155.

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

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

[5] 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.

[6] Alistair Mavin, Philip Wilkinson, Adrian Harwood & Mark Novak (2009): *Easy approach to requirements syntax (EARS)*. In: *2009 17th IEEE International Requirements Engineering Conference*, IEEE, pp. 317–322.

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

[8] Murray Shanahan (1999): *The event calculus explained*. In: *Artificial intelligence today*, Springer, pp. 409–430.