TPLP **24** (4): 844–862, 2025. © The Author(s), 2025. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S1471068424000280

*Early Validation of High-Level System Requirements with Event Calculus and Answer Set Programming**

ONDŘEJ VAŠÍČEK
Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
(*e-mail:* ivasicek@fit.vut.cz)

JOAQUIN ARIAS
Universidad Rey Juan Carlos, Móstoles, Spain
(*e-mail:* joaquin.arias@urjc.es)

JAN FIEDOR
Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
Honeywell International S.R.O., Brno, Czech Republic
(*e-mail:* ifiedor@fit.vutbr.cz)

GOPAL GUPTA
Computer Science Department, UT Dallas, Richardson, TX, USA
(*e-mail:* gupta@utdallas.edu)

BRENDAL HALL
Ardent Innovation Labs, Eden prairie, MN, USA
(*e-mail:* bren@ardentinnovationlabs.com)

BOHUSLAV KŘENA
Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
(*e-mail:* krena@fit.vutbr.cz)

BRIAN LARSON
Multitude Corporation, St Paul, MN, USA
(*e-mail:* brl@multitude.net)

∗ We are grateful to anonymous reviewers for their insightful comments and suggestions for improvement. The Czech team was supported by the project 23-06506S of the Czech Science Foundation and the FIT BUT internal project FIT-S-23-8151. Joaquin Arias was supported by grant VAE (TED2021-131295B-C33) funded by MCIN/AEI/10.13039/501100011033 and by the "European Union NextGenerationEU/PRTR", by grant COSASS (PID2021-123673OB-C32) funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe". Gopal Gupta was partially supported by US NSF Grants IIS 1910131 and grants from industry through the UT Dallas Center for Applied AI and Machine Learning.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

SARAT CHANDRA VARANASI
GE Aerospace Research, Niskayuna, NY, USA
(*e-mail:* SaratChandra.Varanasi@ge.com)

TOMÁŠ VOJNAR
Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
Faculty of Informatics, Masaryk University, Brno, Czech Republic
(*e-mail:* vojnar@fi.muni.cz)

submitted 20 August 2024; accepted 13 September 2024

Abstract

This paper proposes a new methodology for early validation of high-level requirements on cyber-physical systems with the aim of improving their quality and, thus, lowering chances of specification errors propagating into later stages of development where it is much more expensive to fix them. The paper presents a transformation of a real-world requirements specification of a medical device—the Patient-Controlled Analgesia (PCA) Pump—into an Event Calculus model that is then evaluated using Answer Set Programming and the s(CASP) system. The evaluation under s(CASP) allowed deductive as well as abductive reasoning about the specified functionality of the PCA pump on the conceptual level with minimal implementation or design dependent influences and led to fully automatically detected nuanced violations of critical safety properties. Further, the paper discusses scalability and non-termination challenges that had to be faced in the evaluation and techniques proposed to (partially) solve them. Finally, ideas for improving s(CASP) to overcome its evaluation limitations that still persist as well as to increase its expressiveness are presented.

KEYWORDS: requirements validation, event calculus, answer set programming, s(CASP)

1 Introduction and background

Early validation of specifications describing requirements placed on cyber-physical systems (CPSs) under development is essential to avoid costly errors in later stages of the development, especially when the systems undergo certification. However, there is a lack of suitable automated tools and techniques for this purpose. A crucial need here is that of a small semantic gap between the requirements and the formalism used to model them for the purposes of validation. A larger semantic gap makes it more difficult to transform the requirements into a model, and, most importantly, any validation on such a model drifts away from validating the requirements themselves and closer to validating that particular model—influenced by design and implementation decisions. Furthermore, when reasoning about safety-critical systems, it is necessary—both from engineering and legal points of view—that the tools used must be able to explain the result of the validation.

As described by Mueller (2014), Event Calculus (EC) is a formalism suitable for commonsense reasoning. The semantic gap between a requirements specification and its EC encoding is near-zero because its semantics follows how a human would think of the requirements. Using Answer Set Programming (ASP) and the s(CASP) system for goal-directed reasoning in EC, the work Varanasi et al. (2022) has demonstrated the versatility of EC for modeling and reasoning about CPSs while providing explainable results. However, the CPS presented is still a rather toy system only.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

In this work, we develop a model, presented in Section 3,[1] of the core operation of the Patient-Controlled Analgesia (PCA) Pump by Hatcliff et al. (2019)—a real safety-critical device. The model operates in a way similar to an early prototype of the system and, thus, can be used to reason about its behavior. However, due to the nature of EC, the behavior of the model is very close to the behavior described by the requirements themselves. This allows us to reason about the requirements without tainting the reasoning by implementation or design decisions, which would be necessary when using lower-level models or a physical prototype.

Our work has resulted in the discovery of a number of issues in the PCA pump specification. Using automated reasoning, we were able to discover inconsistencies between the requirements specification and the use cases and exception cases based on which the requirements were created (Section 4.1). We were further able to detect a safety property violation which can lead to an overdose of the patient (Section 4.2). Such discoveries could otherwise occur much later in the development process. We have discussed and confirmed the issues with the authors of the specification.

We present a number of challenges encountered during the translation of the requirements to EC encoded in s(CASP) and during the subsequent evaluation, based on deductive as well as abductive reasoning, which was often too costly or non-terminating. We have applied and, in multiple cases, also newly developed various techniques that helped us resolve many of these challenges. These include extensions of the axiomatization of the EC and special ways of translating certain parts of the specifications (Sections 5.1 and 5.2), which we believe may be inspiratory when modeling and evaluating other systems too. Further, we present an original approach to abductive reasoning with incrementally refined abduced values in order to assure consistency of the abduced values whenever abduction on the same value is used multiple times in the reasoning tree (Section 5.3). Next, we proposed a mechanism for caching predicate evaluations (failure-tabling and tabling of ground sub-goal success) that was added into s(CASP) as a prototype leading to a significant increase in performance (Section 5.4). We also describe a way of separating the reasoning about the trigger and the effect of certain complexity-inducing triggered events into multiple reasoning runs where each run produces new facts to be used in the subsequent ones (Section 5.5), which reduces their performance impact. Finally, we propose two new lines of work (Section 6), including a more systematic treatment of caching.

1.1 Related work

Above, we have emphasized the suitability of EC for reasoning about requirements specifications due to its low semantic gap against them. In comparison, the semantics of automata-based approaches, such as timed automata in UPPAAL by Larsen et al. (2018), require one to "design" explicit states and transitions and may lead to decomposition of the system into sub-systems each with their own automaton. Current industrial model-based engineering approaches, such as those based, for example, on Matlab Simulink models and tools like HiLiTE by Bhatt et al. (2010), are only suitable for validation

[1] For the reader's convenience, the files described/used in the paper are available and linked to a GitHub repository available at https://github.com/ovasicek/pca-pump-ec-artifacts/.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

of low-level requirements. This is due to the low-level nature of the models they use, especially when automated generation of code from the models is required.

Apart from the EC-based approach introduced by Varanasi et al. (2022), which this work builds upon, there are other ones which aim to target automated validation of high-level requirements put on CPSs. The work by Crapo et al. (2017) is based on ontologies and uses theorem proving, which traditionally requires significant manual work. The work by Arnaud et al. (2021) is based on transforming CPS specifications from templated-English into process algebras extended with real-time aspects, however, no continuous variables (apart from time) are dealt with and no experimental results are presented, which makes it difficult to judge the scalability of this approach. A more detailed comparison of the approaches is an interesting future work.

A transformation of a CPS into EC was considered in the already mentioned work by Varanasi et al. (2022). However, it considered a simple Train-Gate-Controller system only. We expand on that work by transforming a more complex, real-life specification of the PCA pump, which has led to the discovery of a number of issues that did not manifest in the simpler system. We tackle the issues by introducing techniques for avoiding non-termination and improving performance when reasoning about EC in s(CASP). In addition, we further propose a way to check consistency between levels of the specification and to leverage abductive reasoning.

Finally, we note that there are of course other ASP solvers than the s(CASP) system we used. Notably, grounding-based ASP solvers, such as Clingo by Gebser et al. (2019), are well known. However, such solvers are, unfortunately, not suitable for reasoning about fluents with large or continuous value domains due to the explosion in the grounding and a need to discretize the time. In our preliminary attempts at modeling the PCA pump using Clingo, the solver could only reason about narratives with very restricted value domains of all fluents and with a small number of large time steps without running out of memory on a machine with 64 GB of RAM while taking close to an hour of execution time. Further, the need to discretize time requires approximation of time steps for reaching exact values of continuous fluents during periods of continuous change, which can lead to inaccurate behavior of the model. In comparison, the grouding-free nature of s(CASP) supported by constraint solvers allowed us to reason over continuous time, and increasing the value domain of a fluent typically did not affect the solution time needed. Consequently, s(CASP) was able to reason about the same narratives as our preliminary Clingo model using only up to 50 MB of memory and around 5 min of execution time. A thorough comparison of the solvers is out of scope of this work. Some comparisons have already been made by Arias et al. (2018) and by Varanasi et al. (2022). A very interesting future work would be revisiting the PCA pump model using Clingo once sufficient advancements are made in avoiding the explosion in the grounding size, especially since Clingo does not suffer from non-termination issues, which make things much more complicated in s(CASP).

2 Preliminaries

This section describes (i) s(CASP), a goal-directed implementation of ASP with Constraints, and the Event Calculus (EC), a formalism for reasoning about events and change, and (ii) an open-source PCA pump specification, which we use as a real use case.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

2.1 The s(CASP) system and the Event Calculus

The s(CASP) system, presented by Arias et al. (2018), extends the expressiveness of ASP systems, based on the stable model semantics by Gelfond and Lifschitz (1988), by including predicates, constraints among non-ground variables, uninterpreted functions, and, most importantly, a top-down, query-driven execution strategy. These features make it possible to return answers with non-ground variables (possibly including constraints among them) and to compute partial models by returning only the fragment of a stable model that is necessary to support the answer to a given query. Answers to all queries can also include the full proof tree, making them fully explainable.

In s(CASP), thanks to the constructive negation, `not p(X)` can return bindings for `X` for which the call `p(X)` would have failed. Thanks to the interface of s(CASP) with constraint solvers, sound non-monotonic reasoning with constraints is possible.

Like other ASP implementations and unlike Prolog, s(CASP) handles non-stratified negation and returns the corresponding (partial) stable models, for example, for the program `p :- not q. q :- not p`, under the stable model semantics there are two possible models for this *even loop* (Lifschitz, 2019), with either `p` or `q` being true. Even loops are used in s(CASP) to implement abductive reasoning via the `#abducible` directive, where we automatically search for suitable values of the predicates in the corresponding even loop so we can satisfy the main query. We use abduction in Section 4.2 to detect a violation of a critical safety property in the PCA pump requirements.

The Event Calculus (EC) is a formalism for reasoning about events and change by Mueller (2014), of which there are several axiomatizations. There are three basic concepts in EC: *events*, *fluents*, and *time points*: (i) an event is an action or incident that may occur in the world, for example, the dropping of a glass by a person is an event, (ii) a fluent is a time-varying property of the world, such as the altitude of a glass, (iii) a time point is an instant of time. Events may happen at a time point; fluents have a truth value at any time point, and these truth values are subject to change upon an occurrence of an event. In addition, fluents may have quantities associated with them as parameters, which change discretely via events or continuously over time via trajectories.

For example, the event of *dropping* a glass initiates the fluent that captures that the glass is *falling*, which enables a trajectory that determines the decreasing value of a fluent that represents the glass's *height* above the ground, and the event of *catching* a glass terminates the fluent that the glass is *falling*, which disables the trajectory. An EC description consists of a universal theory and a domain narrative (see the book by Mueller (2014) for details). The theory is a conjunction of EC axioms, for example, axiom BEC6 states that a fluent f is true at a time t_2 if it is initiated by some event e occurring at some earlier time t_1 and it is not stopped between t_1 and t_2:

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

The domain narrative consists of the causal laws of the domain and the known events and fluent properties. Mueller (2014), in his book in Example 14, reasons about turning a light switch on and off using the event $Happens(e,t) \equiv (e = TurnOn \wedge t = 1/2) \vee (e = TurnOff \wedge t = 4)$ that states that the *TurnOn* event will happen exclusively at time $t = 1/2$ and that *TurnOff* will happen exclusively at $t = 4$.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

Two key factors contribute to the s(CASP)'s ability to model EC: the preservation of non-ground variables during the execution and the integration with constraint solvers. Using the translation rules introduced by Arias et al. (2022), one can translate the BEC axioms by Mueller (2014) into s(CASP) programs that follow the logic programming convention: constants and predicate symbols start with a lowercase letter, variables start with an uppercase letter, and constraints can be written with a prefix (`#<`). For example, the BEC6 axiom and events are translated as:

```
holdsAt(F, T2) :- T1 #< T2, initiates(E, F, T1),
  happens(E, T1), not stoppedIn(T1, F, T2).
happens(turn_on, 1/2).
happens(turn_off, 4).
```

2.2 The Patient-Controlled Analgesia (PCA) Pump project

The Open PCA Pump Project, introduced by Hatcliff et al. (2019) and available at https://openpcapump.santoslab.org/, provides a full set of realistic artifacts used in the development process of a *Patient-Controlled Analgesia Infusion Pump*, which is a safety-critical medical device. The artifacts were created at the behest of the US Food and Drug Administration to provide an open-source example of model-based systems engineering for industry, and a subject matter for researchers. The primary function of the device is to automatically and safely deliver the appropriate amount of pain-relief drugs to a patient via infusion into their bloodstream. The pump needs to do so without delivering an amount that would harm the patient, it needs to notify clinicians about hazards, and it needs to maintain safe operation even when failures occur or when hazards are detected. The delivery parameters, such as drug flow rates or maximum safe doses, are either prescribed by a physician or specified in a drug library.

In this paper, we use version 1.0.0 of the PCA specification (Open-PCA-Pump-Requirements.pdf in GitHub, which we reference as [PCA page N] in the following when referring to page N). The PCA pump delivers drug using four different types of delivery: (i) *Basal delivery* is the baseline which delivers drug using a small flow rate during normal operation. It is the initial type of delivery after starting the pump. (ii) *Patient-requested bolus* is an extra dose which can be requested by the patient via a button. Upon a valid request, the PCA pump delivers a prescribed amount of drug called VTBI (volume-to-be-infused) using a higher flow rate in addition to the baseline basal flow rate and then returns to basal delivery. (iii) *Clinician-requested bolus* is a second, similar, extra dose of the same VTBI spread over many minutes. It differs in that it can only be requested by a clinician and that they can select a duration for the bolus. (iv) *KVO delivery* (Keep-Vein-Open) is an emergency delivery with the smallest flow rate to prevent clotting of the needle in response to certain alarms.

The specification defines a number of alarms and how the PCA pump should respond to them. Most of the alarms are related to hardware failures or physical issues detected by sensors, while others are raised by the logic of the PCA pump, for example, to prevent an overdose of the patient.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

3 Modeling the PCA pump requirements in EC under s(CASP)

The requirements are specified in unconstrained natural language, which makes automated processing difficult, and so their transformation to an EC model was done manually. An automated transformation from more structured requirements is part of our future work. We have modeled the PCA pump based on Chapter II. *Requirements* [PCA page 54]. Our main focus was on the core functionality of the PCA pump, defined in Section 5 *PCA Pump Function* [PCA page 55], and we omitted a portion of requirements stated in other sections (e.g., non-essential features and physical properties). We do not cover the entire transformation here due to space limitations. Below, we demonstrate it on several representative examples. All source files are available at https://github.com/ovasicek/pca-pump-ec-artifacts/, general principles of modeling using EC are explained by Mueller (2014), and a similar transformation of the Train-Gate-Controller has been shown by Varanasi et al. (2022).

The centerpiece of the PCA pump is the total amount of drug that has been delivered. We represent it by a continuous fluent `total_drug_delivered(X)`. Its value is constant while the pump is stopped, and it changes gradually at a given rate while the pump is running. The gradually changing value is given by the chosen type of drug delivery—pump stopped (no delivery), basal, patient bolus, clinician bolus, or KVO (Section 2.2). Each of the delivery types needs to be represented by an EC trajectory and the logic of the PCA pump then determines which trajectory is active at what time.

We demonstrate the transformation on requirements defined for the delivery of a *patient-requested bolus* [PCA page 55] (implemented in 04-patient_bolus_trajectory.pl). Other delivery modes were transformed in a similar fashion.[2]

> **R1:** Upon patient's press of the PCA pump's patient-button, a prescribed bolus volume-to-be-infused, VTBI, of the drug loaded in the pump is delivered to the patient.

R1 introduces the patient bolus delivery mode in general. We define a fluent to represent the delivery state, and its start/end events and their effects. Then, we define a trajectory to determine the value of the `total_drug_delivered(X)` fluent while this delivery is active.

R1a

```
fluent(patient_bolus_delivery_enabled).
event(patient_bolus_delivery_started).   event(patient_bolus_delivery_stopped).
initiates(patient_bolus_delivery_started, patient_bolus_delivery_enabled, T).
terminates(patient_bolus_delivery_stopped, patient_bolus_delivery_enabled, T).

trajectory(patient_bolus_delivery_enabled, T1, total_drug_delivered(Total), T2) :-
  basal_and_patient_bolus_flow_rate(FlowRate),
  holdsAt(total_drug_delivered(StartTotal), T1),
  Total #= StartTotal + ((T2 - T1) * FlowRate).
```

And finally, we define that the bolus ends automatically once it delivers the full VTBI. This is represented by a `patient_bolus_completed` event and its trigger rule.

[2] See 04-basal_delivery_trajectory.pl, 04-clinician_bolus_trajectory.pl, 04-kvo_delivery_trajectory.pl, 04-pump_state.pl, and other relevant files at https://github.com/ovasicek/pca-pump-ec-artifacts/.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

```
event(patient_bolus_completed).                                          R1b
happens(patient_bolus_completed, T2) :-   initiallyP(vtbi(VTBI)),
  holdsAt(patient_bolus_drug_delivered(VTBI), T2).
happens(patient_bolus_delivery_stopped, T) :- happens(patient_bolus_completed, T).

fluent(patient_bolus_drug_delivered(X)).
trajectory(patient_bolus_delivery_enabled,T1, patient_bolus_drug_delivered(X),T2):-
  patient_bolus_only_flow_rate(FlowRate),
  X #= (T2 - T1) * FlowRate.
```

The patient_bolus_completed event triggers once the amount of drug delivered via the bolus delivery rate reaches VTBI. This value is represented by a new fluent and its trajectory, which allow easier tracking of the progress of the bolus by counting its value from zero. The new fluent is not affected by any events, it is given by the trajectory only.

R2: A patient-requested bolus shall be delivered at its prescribed rate, F_{bolus}, in addition to the basal flow rate, F_{basal}, but no more than the max. flow rate for the pump, F_{max}.

For R2, we define a predicate which computes the flow rate based on the values of system parameters which are represented using constant fluents.

```
basal_and_bolus_flow_rate(Cropped) :-   initiallyP(pump_flow_rate_max(Max)),   R2
  initiallyP(patient_bolus_flow_rate(Bolus)),   initiallyP(basal_flow_rate(Basal)),
  Combined #= Bolus + Basal,   min(Combined, Max, Cropped).

patient_bolus_only_flow_rate(BolusOnly) :-   basal_and_bolus_flow_rate(Cropped),
  initiallyP(basal_flow_rate(Basal)),   BolusOnly #= Cropped - Basal.
```

R6: Any alarm stops patient-requested bolus delivery either halting pump or switching to KVO rate as defined in Table 4 [PCA page 59].

To implement R6, we trigger the occurrence of the end of the bolus when any_alarm happens (which is itself triggered by specific alarms) while the bolus is active.

```
happens(patient_bolus_halted, T) :-   happens(any_alarm, T),                R6
  holdsAt(patient_bolus_delivery_enabled, T).
happens(patient_bolus_delivery_stopped, T) :-   happens(patient_bolus_halted, T).
```

R3: Patient-requested bolus shall not be delivered more often than a prescribed minimum time between patient-requested bolus, Δ_{prb}.

R5: Patient-requested bolus shall not be delivered if infusing prescribed VTBI will exceed hard limits retrieved from the drug library for the volume of drug infused over a period of time. Pump rate shall be reduced to KVO and a max dose warning be issued.

R3 and R5 define cases when a requested bolus should be denied. This can be implemented by making the occurrence of patient_bolus_delivery_started conditional.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

```
happens(patient_bolus_delivery_started, T) :-                    R3/5
  happens(patient_bolus_requested, T),
  not_happens(patient_bolus_denied_too_soon, T),
  not_happens(patient_bolus_denied_max_dose, T).
```

R3 is implemented by checking if any bolus delivery was enabled in the past while too close to the current request.

```
happens(patient_bolus_denied_too_soon, T) :-                     R3
  happens(patient_bolus_requested, T),
  initiallyP(min_t_between_patient_bolus(MinGap)),
  TLast #< T,   TLast #> T - MinGap,
  holdsAt(patient_bolus_delivery_enabled, TLast).
```

R5 is implemented by checking the total amount of drug delivered within the max dose time window (e.g., in the last hour). The delivery mode needs to be changed accordingly when a warning is triggered. The code below is simplified for space reasons, details can be found in 04-patient_bolus_trajectory.pl.

```
happens(patient_bolus_denied_max_dose, T) :-                     R5
  happens(patient_bolus_requested, T),
  initiallyP(vtbi_hard_limit_over_time(VtbiLimit, TimePeriod)),
  holdsAt(total_drug_delivered(CurrentTotal), T),
  TstartPeriod #= T - TimePeriod,
  holdsAt(total_drug_delivered(TotalAtStartPeriod), TstartPeriod),
  TotalInPeriod #= CurrentTotal - TotalAtStartPeriod,
  TotalInPeriod #> VtbiLimit.

happens(max_dose_warning,T) :-   happens(patient_bolus_denied_max_dose, T).
happens(basal_stopped,T) :-   happens(patient_bolus_denied_max_dose, T),
  holdsAt(basal_delivery, T).
happens(kvo_started,T) :-   happens(max_dose_warning, T).
```

4 Reasoning about the PCA pump requirements using s(CASP)

The specification defines a number of use cases (UC) and exception cases (ExC) in Section 4 *System Operational Concepts* [PCA page 13]. The requirements and the UC/ExCs should be mutually consistent. We simulate the behavior of UC/ExCs using the EC model, which was created based on the requirements, with the expectation that the model should behave exactly as defined in the UC/ExCs. An example is shown in Section 4.1. If the behavior of the model is inconsistent with the UC/ExCs, then we have produced evidence of the requirements being inconsistent with the UC/ExCs (up to correctness of the transformation). The capabilities of s(CASP) are not limited to simulating the behavior of the pump but allow us to reason about its general properties. In Section 4.2, we reason about preventing an overdose of the patient—a critical safety property.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

4.1 Validating consistency of use/exception cases and the requirements

We show an example of using s(CASP) reasoning to simulate *UC2: Patient-Requested Bolus* in order to validate its consistency with the requirements specification.

UC2: Patient-Requested Bolus

Pre: 1. Steps 1 to 14 of Normal Operation Use Case (UC1) completed.
2. Basal rate being infused.
3. Prescribed minimum time between boluses has elapsed.

Post: 1. Resume basal rate infusion.

Step: 1. Patient presses bolus request button.
2. Time since last bolus compared with prescribed min. time between boluses (*see* ExC1).
3. If not too soon, begin infusing VTBI (*unless* ExC13: Maximum Safe Dose).
4. After prescribed VTBI has been infused, resume basal rate infusion.

A use case consists of pre-conditions, a sequence of steps, and post-conditions. We create a narrative of event occurrences based on the pre-conditions and input events from the steps. Then, we form a query based on the post-conditions and triggered events from the steps. The below excludes the initialization of system parameters, for example, that the VTBI is $1\,ml$ and the bolus flow rate is $1\,ml.min^{-1}$. The implementation can be found in uc2.pl and general_utils.pl (utility predicates `holdsIn/3` and `holdsAfter/2`).

```
happens(start_button_pressed,    60).                                  % Pre  1   UC2
happens(patient_bolus_requested, 120).                                 % Step 1
?- holdsIn(basal_delivery_enabled,                          60, 120), % Pre  2
   initiallyP(min_t_between_patient_bolus(MinT)), T1 #= 120 - MinT, % Pre  3
   not_holdsIn(patient_bolus_delivery_enabled,               T1, 120), % Pre  3
   not_happens(patient_bolus_denied_too_soon,                    120), % Step 2
   not_happens(patient_bolus_denied_max_dose,                    120), % Step 3
   happens(patient_bolus_delivery_started,                       120), % Step 3
   initiallyP(vtbi(VTBI)),  happens(patient_bolus_completed,      T2), % Step 4
   holdsAt(patient_bolus_drug_delivered(VTBI),                    T2), % Step 4
   happens(basal_delivery_started,                                T2), % Step 4
   holdsAfter(basal_delivery_enabled,                             T2). % Post 1
```

The occurrence times of input events are randomly chosen. The model should behave according to the UC for any event times. In the future, we plan on allowing narratives with events at variable timepoints `T` (currently, fixed narratives are required). The above query succeeds, meaning that the model and the requirements are consistent with UC2.

4.1.1 Results of experiments with consistency validation

We have simulated all relevant UCs and ExCs from the PCA pump specification on a 2.67 GHz Xeon CPU, using at most 40 MB of memory. Selected representative results are shown in Table 1, the rest can be found in Appendix A.3 (available online as *supplementary material* at the TPLP archive).

Some of the cases appear in multiple variants of the narrative. For instance, in UC3, a clinician-requested bolus can be delivered uninterrupted (UC3a) or it may be suspended

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

Table 1. *Results of simulation of relevant use cases and exception cases*

	Use case name	Variant	Result	Time (s)
UC2	Patient-requested bolus	No variants	OK	3.17
UC3a	Clinician-requested bolus	Not suspend	OK	2.84
UC3b	Clinician-requested bolus	Suspended and resumed	OK	37.13
ExC7a-f	Over-flow rate alarm	Defined in ExC step 1	FAIL	1.53-4.62
ExC13a	Maximum safe dose	During basal delivery	FAIL	25.62
ExC13b-c	Maximum safe dose	During each bolus	OK	31.61-53.45
ExC21	Reservoir empty	No variants	OK	40.99

by a patient-requested bolus and resumed afterward (UC3b). To save space, we aggregate the measurements of variants of the same case that led to the same result. All implementations can be found in the narratives_and_queries folder.

All UCs were simulated successfully, but quite a few ExCs failed. This has led to the discovery of a number of issues in the specification, such as inconsistencies in alarm responses or defined constants. In particular, Step 1 of ExC7c [PCA page 34] says that an alarm should be raised if the drug flow rate exceeds the prescribed rate *for longer than 10 seconds*, while the requirement R6.4.0(4) [PCA page 58] defines *1 minute* instead. Very similar issues were found in ExC7e and other ExCs. Further, the second post-condition of all variants of ExC7 expects *infusion to be halted*, but Table 4 [PCA page 59] requires a *switch to KVO delivery*.

Cases UC3b and ExC13a-c are significantly slower than the others due to their narratives containing multiple bolus requests (2–3), while the others only contain one or none. In general, we have observed the biggest (exponential) increase in execution time when increasing the number of bolus requests, that is, the number of system input events. ExC21 is also slower despite featuring no bolus requests due to its use of full reasoning about the level of the drug reservoir, which is discussed in Section 5.5.

4.2 Validating the requirements wrt. general properties

We use *ExC13: Maximum Safe Dose* as an example of reasoning about general properties of the system and, later, to demonstrate abductive reasoning capabilities of s(CASP). ExC13 defines that the pump should prevent an overdose of the patient by reducing the drug flow rate. This is a general property, and so ExC13 had to be implemented in three narratives based on whether the overdose would occur during (a) basal delivery, (b) a patient-requested bolus, or (c) a clinician-requested bolus. The implementations can be found in ec13a.pl, ec13b.pl, and ec13c.pl . For example, ExC13b contains 3 patient bolus requests, while the max dose prescription is defined to allow 2.5 boluses in 4 hours. The implemented narrative consists of `happens(start_button_pressed,60)` and three instances of `happens(patient_bolus_requested,T)` for `T` equal to `300`, `340`, and `380`. The third bolus would cause an overdose if delivered. This overdose is prevented by the requirement R5.2.0(5) (discussed as R5 in Section 3), according to which the bolus

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

will be denied. A similar measure is defined for a clinician-requested bolus by R5.3.0(7) [PCA page 56].

The query `?- vtbi_hard_limit_exceeded_at_T_by_X(T, X)` (implemented in analysis_utils.pl) checks the amount of drug delivered within the max dose time window with the end of the window at time `T`. It succeeds if the maximum dose was exceeded and will return by how much via `X`. This query returns no models on ExC13b meaning that an overdose did not happen at any time `T`. However, if we modify the narrative by changing the bolus request times to `100`, `140`, and `180` (switching from ExC13b to ExC13a), then the query succeeds with bindings `T #>295,T #< 681/2` and `X #>0,X #<1/2`. This overdose happens during basal delivery due to a missing requirement (discussed in Section 4.2.2).

4.2.1 Utilizing abductive reasoning

In order to detect the basal overdose issue in the previous section, we had to be "lucky" enough to define a narrative in which the violation manifests, in the same way as with regular testing. To address this, we utilize the abductive reasoning capabilities of s(CASP).

Ideally, we would like abduction to check whether an overdose can occur in some narrative without any prior restrictions. However, this is currently not possible in s(CASP) due to non-termination issues related to reasoning in continuous time. Such abduction is part of our future work. Instead, we fix a skeleton of a narrative (i.e., a sequence of input events to happen) and abduce values of various parameters of the narrative. In particular, we abduce the overdose parameters of the PCA pump via the predicate `initiallyP(vtbi_hard_limit_over_time(VtbiLimit, TimePeriod))`, that is, we abduce both the max dose volume and the size of the max dose window, which allows the reasoner to explore a broad spectrum of overdose scenarios despite being restricted to a fixed narrative of event occurrences. We also apply restrictions on the abducible values in order to keep them meaningful, such as that the time period must be longer than the duration of a single bolus and that the max dose volume must be big enough to fit a full period of basal delivery. Using such abduction, we run the overdose query on UC2 (discussed in Section 4.1) to demonstrate that a regular "sunny day" narrative can be used to detect the overdose issue (implemented in overdose-uc2-abduction.pl). The query returns 4 different worlds of possible overdose. One of them, as an example, uses abduced values `initiallyP(vtbi_hard_limit_over_time(91/10 #=< V #< 101/10, 91))` with 3 query bindings, one of which is `141 #< T #=< 151` and `0 #< X #=< 1`. The max dose was abduced so that less than one bolus was allowed. However, the bolus requested in UC2 was delivered, which caused an overdose during subsequent basal delivery.

4.2.2 An overdose error in the PCA pump requirements

The overdose issue is caused by enough boluses being delivered early in the timeline, particularly, close to the start of the pump. The cause of the issue is that there is no overdose protection measure specified for basal delivery. This is a missing requirement

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

Table 2. *Overdose querying on ExC13 and UC2*

	Use case name	Variant	Model	Result	Time (m)
ExC13b	Maximum safe dose	Patient bolus	Original	OK	14.11
ExC13c	Maximum safe dose	Clinician bolus	Original	OK	20.85
ExC13a	Maximum safe dose	During basal	Original	Overdose	15.29
			fixed	OK	3.34
UC2	Patient-requested bolus	Abduction	Original	Overdose	38.01
			fixed	OK	48.11

that causes a violation of a critical safety property potentially causing harm to the patient depending on the overdose volume and the particular drug used. According to the authors of the PCA pump this is an unintentional omission.

According to the requirement R5.2.0(5) (discussed as R5 in Section 3), when a patient requests a bolus, the pump should reason about how much drug would be delivered within the max dose time window at the end of the currently requested bolus *if* it was delivered. However, such reasoning only considers the contents of the max dose window in the past and does not consider what will follow in the future under normal operation. When a patient requests a bolus at a time close enough to the start of the pump, then the max dose window starts at a time smaller than the start time of the pump and, thus, includes a period of zero drug delivery. With a large enough max dose time window, enough boluses could be delivered to get close to the maximum safe dose. However, as the max dose window moves forward with time, the period of zero drug delivery is pushed out by the now in-progress basal delivery. And since basal delivery has no overdose protection measures, then it will keep running even if the maximum safe dose is exceeded.

After fixing this issue by implementing the missing requirement (discussed in Appendix A.1, available online as *supplementary material* at the TPLP archive), the abductive query on UC2 no longer succeeds, meaning that an overdose was not found in the fixed model. The two versions of the model can be found in model-original.pl and model-fixed.pl. Of course, this is not a sound proof of no overdose being possible—a different overdose might be discoverable via different abducibles or narrative.

4.2.3 Results of experiments with validation of general properties

Table 2 shows results and execution times of querying overdose on variants of ExC13 (discussed in Section 4.2) and of using abduction on UC2 (discussed in Section 4.2.1). Execution of the overdose queries takes much longer than the simulation queries from Table 1 (minutes instead of seconds) due to the higher complexity of the overdose query. However, the abductive queries are the slowest ones due to the higher complexity of abduction in general but also due to the limitations of its current implementation in s(CASP), discussed in Section 5.3.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

5 Techniques used to empower s(CASP) reasoning

This section describes the techniques that we apply to avoid non-termination of s(CASP) reasoning (Sections 5.1 and 5.2), the approach that we have proposed to overcome limitations of s(CASP) abduction (Section 5.3), and techniques proposed to significantly improve reasoning performance (Sections 5.4 and 5.5).

5.1 Improved implementation of the Event Calculus axioms

Our implementation of the BEC axioms (in bec_scasp-pca_pump.pl) differs from the one by Arias et al. (2022) in two aspects in order to avoid non-termination. First, inspired by Varanasi et al. (2022), we use a custom implementation of the `not` keyword. Namely, we implement negated predicates, such as `not_stoppedIn/3`, as simplified versions of the *dual rules* that s(CASP) generates to compute the negated predicates. These simplified versions contain only the dual rules that are relevant for the intended evaluation of the negated predicates. Second, we introduce new predicates `can_initiates/2`, `can_terminates/2`, `can_releases/2`, and `can_trajectory/4`. These are created by preprocessing the source code and introducing a new fact `can_initiates(E,F).` for each fact and/or rule `initiates(E,F,T) :- some_body(E,F,T).` (and likewise for others). Our implementation of the BEC6 axiom (cf. Section 2.1) using these new predicates follows:

```
holdsAt(Fluent, T2) :- T1 #< T2, can_initiates(Event, Fluent),
  happens(Event, T1), initiates(Event, Fluent, T1), not_stoppedIn(T1, Fluent, T2).
```

This construction is motivated by an observation that, in our experiments, proving the original predicate `initiates` first often leads to non-termination due to its sub-goals, while proving `happens` first often leads to non-termination and enlarges the search space due to the unconstrained `Event`. On the other hand, proving the sub-goal-free `can_initiates` first has proven reliable in avoiding non-termination and pruning the search space by constraining `Event`. A similar approach was used by Shanahan (2000).

5.2 Modeling non-termination-prone self-ending trajectories

The main challenge during the modeling of the PCA pump was non-termination caused by trajectories which we refer to as self-ending. A trajectory, defined by a rule with a head `trajectory(F1,T1,F2,T2)`, starts when its control fluent `F1` is initiated at some time `T1`, and the body of the rule then determines how the value of its continuous fluent `F2` may be computed for any time `T2`, where `T1 #< T2`, until `F1` is terminated. The trajectory is *self-ending* if `F1` may be terminated at some time `T2` while the trajectory is active by some event `E` that gets triggered when the value of `F2` satisfies a certain predefined condition. We call such an event a *self-end event* of the given trajectory. In the PCA pump, almost all trajectories are self-ending, for example, bolus deliveries terminate themselves based on how much drug they deliver. For example, the `trajectory(clinician_bolus_delivery_enabled(Duration),T1, total_drug_delivered(X),T2)`, defined in a similar way as was shown in Section 3,

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

is self-ending, and one of its self-end events is `clinician_bolus_halted_max_dose` because its trigger rule depends on the value of `total_drug_delivered` (the code below is simplified for space reasons, details can be found in 04-clinician_bolus_trajectory.pl):

```
1 happens(clinician_bolus_halted_max_dose, T2) :-
2   initiallyP(vtbi_hard_limit_over_time(VtbiLimit, TimePeriod)),
3   T1 #= T2 - TimePeriod,   holdsAt(total_drug_delivered(TotalT1), T1),
4   VtbiLimit #= TotalT2 - TotalT1,   holdsAt(total_drug_delivered(TotalT2), T2).
```

The above rule will, currently, cause non-termination in s(CASP) (trace on GitHub). It is triggered when the amount of drug delivered within the max dose time window reaches the maximum allowed dose. The cause of the issue is that a different trajectory, in this case representing KVO delivery, can be used to determine the value of `total_drug_delivered` while at the same time the start event of that trajectory, `KVO_started`, is triggered by the event `clinician_bolus_halted_max_dose`. This particular loop is created because KVO delivery is being considered as a way to prove the value of `total_drug_delivered` at time `T2`. However, clinician bolus delivery is the only type of delivery which can lead to success because only one delivery can be active at a time, and if clinician bolus delivery was not active, then we would not need to reason about triggering its halt.

To avoid this issue, we introduce a new predicate `holdsAt/3` and a new axiom for EC. We use the new predicate to force the use of the right trajectory when proving the value of `total_drug_delivered` at `T2` at line 4 (defined above) by adding `clinician_bolus_delivery_enabled(_)` as a parameter to `holdsAt`. The new axiom is the same as the BEC3 axiom, except for the addition of `Fluent1` as the third parameter:

```
1 holdsAt(Fluent2, T2, Fluent1) :-
2   can_trajectory(Fluent1, T1, Fluent2, T2),   can_initiates(Event, Fluent1),
3   happens(Event, T1),   initiates(Event, Fluent1, T1),
4   trajectory(Fluent1, T1, Fluent2, T2),   not_stoppedIn(T1, Fluent1, T2).
```

Specifying `Fluent1` ensures that only the trajectories controlled by that fluent will be considered when trying to prove `Fluent2`. In general, the `holdsAt/3` predicate should be used in self-end event trigger rules when one needs to prove the value of a continuous fluent while its self-ending trajectory is active.

5.3 Abduction using incremental refinement to enforce consistent models

Abductive reasoning in s(CASP) can abduce a different value of an abducible every time the abducible is reached in the reasoning tree. This is, however, unsuitable when some constant or a tuple of constants, representing, for example, values of some parameters of the modeled system or of some scenario in which it is evaluated, is to be abduced.

Since the above problem appears in our model, we have proposed its solution suitable for abducing numerical values. It is based on repeatedly refining the values abduced at different points in the reasoning tree—through repeatedly tightening constraints on possible values of the abducibles—until the same values are obtained everywhere (or the abduction fails). The solution consists of two phases; the first one follows:

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

1. Run an abductive query of $predicate(p_1, ..., p_n)$ where p_i are variables to abduce.
2. **For each** model m produced by the query:
   (a) **For each** parameter p_i, the model m will contain some number y of value intervals $I_1^{p_i}, ..., I_y^{p_i}$ abduced at different points of the reasoning tree.
   (b) **For each** p_i, compute the intersection $I^{p_i} = I_1^{p_i} \cap ... \cap I_y^{p_i}$.
   (c) **If** I^{p_i} is empty for some p_i or if the exact combination of $I^{p_1}, ..., I^{p_n}$ has been seen before (globally across Step c) or if a predefined cut-off depth has been reached, **then** discard m and **end recursion**.
   (d) **Else if** for all p_i, $I_1^{p_i} = ... = I_y^{p_i}$, **then** m is a **consistent model** with intervals of values $I^{p_1}, ..., I^{p_n}$ for $p_1, ..., p_n$. Add $(m, I^{p_1}, ..., I^{p_n})$ to the **result** and **end recursion**.
   (e) **Else** make a new query by restricting the abducible value of each parameter p_i to I^{p_i}, and recursively perform Steps 1 and 2 using the new query.

The result of the first phase will be a set of models, each containing a tuple of intervals $I^{p_1}, \ldots, I^{p_n}$ of possible values of $p_1, \ldots, p_n$. However, in order to obtain one concrete witness of the result of the query, one cannot just take any tuple of values $v_1, \ldots, v_n$, where $v_i \in I^{p_i}$, since the values of the different parameters may depend on each other. Therefore, in the second phase of our solution, for each of the models, we proceed as follows. We select a value $v_n \in I^{p_n}$ and repeat the first phase with this value fixed, leading to new intervals $J_1^p \subseteq I_1^p, \ldots, J_{n-1}^p \subseteq I_{n-1}^p$. We then likewise gradually select and fix values for the parameters $p_{n-1}, \ldots, p_2$. For p_1, it is not needed to repeat the process since it is the last interval to pick a value from and, therefore, any choice will be valid.

We use the above approach in Section 4.2.1 to abduce the initial value of a constant fluent `initiallyP(vtbi_hard_limit_over_time(V,P))` with two variable parameters (the implementation is available in incremental_abduction.sh). The described approach can find witnesses of a property violation but, due to the cutoff bound in Step 2(c) of the first phase, it cannot guarantee that the property is not violated. It is also inefficient due to repeated executions which explore nonrealistic parts of the state space, and, further, each execution is slower than our other experiments because it cannot use the experimental cache we introduced to optimize s(CASP) reasoning (discussed in Section 5.4). However, despite the inefficiency, it was able to detect an error in the requirements specification (discussed in Section 4.2.2) in reasonable time. Introducing an efficient solution to this problem into s(CASP) is part of our future work.

5.4 Prototype cache for predicate proof results

The runtimes presented in Tables 1 and 2 (except for abduction) were measured using s(CASP) version 0.24.04.04 under a new, preliminary implementation of tabling that caches the first (un)successful evaluation of specific predicates. These predicates are selected using the `#table_once` directive, in a similar way as mode-directed tabling, described by Guo and Gupta (2008) and Arias and Carro (2019b), and implemented in several Prolog interpreters. Under this *cache mode*, when one of the selected predicates fails to be proved as a ground sub-goal of any rule, s(CASP) caches the failure (failure-tabling), and similarly, when the evaluation of the sub-goal succeeds, the success is cached. Subsequent attempts at proving the ground cached predicate will then use

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

these results instead of attempting to prove it again. Note that since s(CASP) implements non-monotonic reasoning, the result is only valid while the current assumptions are valid—therefore, the result stays cached until current assumptions change.

Using the cache on our test suite, we have observed a reduction of execution time by up to 95 % on individual test narratives with an overall average of 66 % across the whole test suite while still obtaining the same models (up to the cached parts of the proof tree). We believe that a more sophisticated implementation of tabling, based on TCLP by Arias and Carro (2019a), would increase the performance without losing soundness (note that for non-grounded sub-goals, we may lose other valid answers by storing only one answer, affecting the soundness of negated sub-goals) or completeness.

In our experiments, we cached all EC predicates by including the file cache.pl with the corresponding directives. Due to the nature of EC, proving anything at a timepoint requires reconstructing the whole history from time zero to the given timepoint. Therefore, the history which had to be proven for the value of a fluent at `T1` will potentially have to be re-proven again for `T2` where `T1 < T2`. The new cache is especially useful to prevent repeated *failing attempts* to prove a predicate such as `not_stoppedIn`. We found that even reasoning about simple narratives would attempt *and fail* to prove `not_stoppedIn` many times for the exact same parameters. When using cache, the predicate will only fail to be proven once for each set of parameters.

5.5 Decoupling triggers and effects of events into multiple executions

Based on our experiments, we believe that triggered events, especially the ones that can terminate a trajectory, very significantly contribute to the solving complexity. This holds even for narratives in which such events are never actually triggered—because the reasoning keeps trying to prove their trigger due to their potential effects. To reduce the performance impact of triggered events, we propose below an approach that targets particularly those of such events that may only trigger once per narrative, such as certain alarms in the PCA pump. The idea is to use a *multi-run reasoning* in which we decouple the trigger of such an event `E` from its effect. This is done by removing all effects of `E` and moving them to a newly introduced event `E_EFFECT` instead (see an example in Appendix A.2, available online as *supplementary material* in the TPLP archive). We then use one dedicated run to check whether `E` happens at some time `T` in the given narrative. If not, `E_EFFECT` will stay undefined in further reasoning. If `E` does happen at `T`, we introduce a fact `happens(E_EFFECT,T).`, which will then allow further reasoning to take the effect of `E` into account without having to reason about its trigger.

We use the above approach for alarms related to the drug reservoir contents—`empty_reservoir_alarm` and `low_reservoir_warning` (defined in 08-drug_reservoir.pl).[3] Each of the alarms can only happen once in a narrative in response to the level of drug in the reservoir reaching a certain threshold since the reservoir cannot be refiled during a narrative. This approach was needed because implementing alarms related to the drug reservoir contents caused an unbearable slowdown for some narratives—the worst case in our test suite was a slowdown from 6.7 mins to 6.3 hours,

[3] Implemented in such a way that, for each narrative, we can choose to ignore the drug reservoir reasoning (when deemed not relevant), or to use the multi-run approach, or to re-enable the full (slow) reasoning.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

in a narrative where neither of the alarms happens. This is due to the fact that prior to implementing these alarms the PCA model only contained three triggered events that could terminate a trajectory, each terminating only one trajectory, and adding the new alarms introduced two new triggered events which together can terminate *all* trajectories. Indeed, the effect of `low_reservoir_warning` is stopping any drug delivery and switching to KVO delivery and the effect of `low_reservoir_empty` is stopping the pump entirely.

We use the proposed approach for both the low reservoir warning and the empty reservoir alarm *at the same time* for a total of three executions (cf. three_runs.sh): the first query introduces a new fact for the low reservoir, the second one introduces a new fact for the empty reservoir while using the fact from the first query, and the third and final query considers both of the new facts. For a narrative based on UC2 which reasons about both the low reservoir warning and the empty reservoir alarm, the three run approach takes 12 s while a single run with full reasoning takes 35 minutes (cf. empty_reserv-uc2-multirun-* and empty_reserv-uc2-onerun.pl). We are experimenting with a similar approach for incremental reasoning about all triggered events as future work.

6 Conclusions and future lines of work

Our work demonstrated that Event Calculus (EC) can be used to model the requirements specification of a non-trivial, real-life cyber-physical system in s(CASP) and the reasoning involved can lead to discovering issues in the requirements while producing valuable evidence toward their validation. Indeed, we have discovered a violation of a critical safety property in a well-studied specification, acknowledged by its authors.

Our future work involves two directions. The first includes improvements to s(CASP) by integration and efficient implementation of our abductive reasoning semantics, improvements to prototype caching, and avoiding non-termination. A common non-termination case is the "toggle" scenario where a system toggles between two fluents affected by respective toggle events. A meta-reasoner in s(CASP) specialized to EC would be more efficient and better at avoiding non-termination. The second direction involves software engineering to make our approach more general and practically usable, including the replacement of unconstrained natural language requirements with structured languages like MIDAS, by Hall et al. (2020), for capturing requirements of industrial projects. This should provide enough structure and context to the requirements in order to enable a more general and at least semi-automated transformation of the requirements into EC, which would make our approach easier to adopt and use.

Supplementary material

The supplementary material for this article can be found at http://dx.doi.org/10.1017/S1471068424000280.

References

ARIAS, J. AND CARRO, M. 2019a. Description, implementation, and evaluation of a generic design for tabled CLP. *Theory and Practice of Logic Programming* 19a, 3, 412–448.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press

ARIAS, J. and CARRO, M. 2019b. Incremental evaluation of lattice-based aggregates in logic programming using modular TCLP. In Proc. of PADL'19 – 21st International Symposium on Practical Aspects of Declarative Languages, Springer, Vol.11372, LNCS, 98–114.

ARIAS, J., CARRO, M., CHEN, Z. AND GUPTA, G. 2022. Modeling and reasoning in event calculus using goal-directed constraint answer set programming. *Theory and Practice of Logic Programming* 22, 1, 51–80.

ARIAS, J., CARRO, M., SALAZAR, E., MARPLE, K. AND GUPTA, G. 2018. Constraint answer set programming without grounding. *Theory and Practice of Logic Programming* 18, 3-4, 337–354.

ARNAUD, M., BANNOUR, B., LAPITRE, A. AND GIRAUD, G. 2021. Investigating process algebra models to represent structured requirements for time-sensitive CPS. In *Proc. of SEKE'21 – The 33rd International Conference Software Engineering & Knowledge Engineering*, Pittsburgh, (Virtual Conference) United States.

BHATT, D., MADL, G., OGLESBY, D. AND SCHLOEGEL, K. 2010. Towards scalable verification of commercial avionics software. In *Proc. of AIAA Infotech@Aerospace*.

CRAPO, A., MOITRA, A., MCMILLAN, C. AND RUSSELL, D. 2017. Requirements capture and analysis in ASSERT(TM). in *Proc. of RE'17 – 25th International Requirements Engineering Conference*, IEEE.

GEBSER, M., KAMINSKI, R., KAUFMANN, B. AND SCHAUB, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming* 19, 1, 27–82.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proc. of 5th International Conference on Logic Programming*, 1070–1080.

GUO, H.-F. ANDGUPTA, G. 2008. Simplifying dynamic programming via mode-directed tabling. *Software: Practice and Experience* 38, 1, 75–94.

HALL, B., FIEDOR, J. AND JEPPU, Y. 2020. Model integrated decomposition and assisted specification (MIDAS). *INCOSE International Symposium* 30, 1, 821–841.

HATCLIFF, J., LARSON, B., CARPENTER, T., JONES, P., ZHANG, Y. AND JORGENS, J. 2019. The open PCA pump project: an exemplar open source medical device as a community resource. *ACM SIGBED Review* 16, 2, 8–13.

LARSEN, K. G., LORBER, F. and NIELSEN, B. 2018. 20 Years of UPPAAL enabled industrial model-based validation and beyond. In *ISoLA'18 – 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation* 2018, Springer, 11247, LNCS,

LIFSCHITZ, V. 2019. *Answer Set Programming*. Cham: Springer Cham.

MUELLER, E. T. 2014. *Commonsense Reasoning: An Event Calculus Based Approach*. Burlington, MA: Morgan Kaufmann.

SHANAHAN, M. 2000. An abductive event calculus planner. *The Journal of Logic Programming* 44, 1-3,207–240.

VARANASI, S. C., ARIAS, J., SALAZAR, E., LI, F., BSUA, K. AND GUPTA, G. 2022. Modeling and verification of real-time systems with the event calculus and s(CASP). In *Proc. of PADL'22 – Practical Aspects of Declarative Languages*, LNCS, Vol. 13165. Springer, 181–190.

https://doi.org/10.1017/S1471068424000280 Published online by Cambridge University Press