An Approach and Tool for Reasoning about Situated Cyber-Physical Systems

Alexandru Nedelcu
ialexandru.nedelcu@gmail.com
Drexel University

Marcello Balduccini
mb3368@drexel.edu
Drexel University

Abstract. By "situated" cyber-physical systems (CPS) we mean CPS that are located in some physical environment. The term is aimed at highlighting the two-way interaction between CPS and environment, and the subtle, serendipitous, and sometimes unexpected correlations that emerge because of the coexistence in and with a physical environment. For example, a CPS that has an elevated surface temperature will tend to heat the air around it, which in turn may influence the readings of temperature sensors on nearby CPS, but this conclusion can be drawn only by considering the corresponding physical laws. Unfortunately, these interactions are normally not captured by the modeling methodologies of CPS, but doing so can lead to improved anomaly detection and robustness of designs. In this paper, we present a method for modeling the interactions of situated CPS, define corresponding reasoning algorithms, and introduce a tool that integrates in a state-of-the-art CPS design tool and automates the reasoning processes.

1 Introduction

In this paper, we describe work on modeling and reasoning about cyber-physical systems (CPS), which is motivated by an increased need for the security of safety critical systems (SCSs). SCSs are an important class of CPS whose failure could lead to life loss and major material or environmental damages. Well-known examples are aircrafts, spacecraft, medical devices and nuclear plants. Engineering advancements have allowed these systems to grow in complexity to the extent that they create opportunities for cyber-attacks. Attacks on SCSs often involve changing their operation in malicious ways, and interfering with the sensor readings, so that no anomaly is suspected. Besides the cyber threat SCSs are exposed to, anomalies can also result from poor design or implementation, and can be just as dangerous as cyber-attacks. The design of individual CPS and of systems of CPS is rather well-understood, and languages for specifying such designs have been created. However, in spite of the fact that CPS are defined as systems capable of computations and *interactions with the environment*, what is less understood are, in fact, the subtle, serendipitous, and sometimes unexpected correlations that emerge because of the coexistence of a set of CPS in, and with, the physical environment in which they are located.

For example, a CPS that has an elevated surface temperature will tend to heat the air

adfa, p. 1, 2015.

around it, which in turn may influence the readings of temperature sensors on nearby CPS, but the link between the two CPS can be established only if one considers the corresponding physical laws of heat propagation. These interactions are normally not captured by the modeling methodologies of CPS.

To highlight this notion, in this paper we talk about "*situated"* CPS, to stress the fact that the CPS are located in a physical environment, with which they have a full-fledged two-way interaction.

There is interest in the CPS community for being able to model and reason about such interactions because it can lead to increased robustness of designs and to improved runtime anomaly detection. The challenge, however, is that state-of-the-art CPS modeling techniques are not equipped for representing the evolution of a domain of interest over time, and for reasoning about it.

In this paper, we propose a method for overcoming these limitations. The method is based on mapping of CPS designs in the Architecture Analysis and Design Language (AADL), a mainstream specification language, to descriptions of dynamic domains. The descriptions are represented using techniques from reasoning about actions and change (RAC) and Answer Set Programming (ASP). The physical environment is, too, modeled as a dynamic domain. Reasoning about interactions of CPS in, and with, the environment are thus reduced to reasoning about the evolution of dynamic domains.

To the best of our knowledge, ours is the first attempt to reason about emerging interactions among CPS and physical environment and using mainstream CPS modeling languages. In (Shiraishi, 2010), a cruise control system is designed in AADL, by considering its subcomponents, interactions with the other vehicle systems such as Engine System or Brake Control System, and sensors to measure various metrics such as vehicle speed. Their approach is limited to the interaction that are explicitly modeled by the system's designer and does not consider the same breadth of questions-answering tasks that we consider here.

(Chen, Nugent, Mulvenna, Finlay, & Hong, 2008) describes an approach for reasoning about smart homes, based on the events occurring in the environment. This is accomplished by using Event Calculus, which captures events and their effects. Their approach is mostly focused on planning (finding sequences of actions needed to achieve a goal) and human behavioral modeling, rather than query-answering related to the detection of discrepancies and bridging the gap between CPS modeling techniques and reasoning about actions and change. They also use a different representation language for reasoning about actions and change (Event Calculus vs ASP).

Next, we provide further motivation of our work by presenting two real-world examples of failures of SCSs. In both cases, prevention requires explicitly reasoning about the physical environment and about how it links multiple CPS – at design time in the first case and at run time in the second.

Fukushima Nuclear Disaster[1]. On Friday, 11 March 2011, Japan underwent a damaging 9.0 magnitude earthquake. The Fukushima nuclear plant reactors, located 11km away from the coast of the Pacific Ocean, behaved well throughout the earthquake.

[1] http://www.world-nuclear.org/info/Safety-and-Security/Safety-of-Plants/Fukushima-Accident/

After the earthquake, a devastating tsunami occurred, which led to many more damages. The 15 meter waves affected power grid, and consequently, the capacity of the grid was only able to supply eight out of eleven reactor cooling systems in the region. The cooling systems automatically switched their energy source from the power grid to the back-up generators, as it was expected. However, the tsunami reached the nuclear plant causing the generators to fail, leaving the reactor with no energy supply to run its cooling system.

This example highlights a design flaw related to the failure to take into account the ramifications of environmental events. The designer of the plant did not take into account that a sufficiently strong earthquake could cause a tsunami with waves high enough to reach the generators.

Air France Flight 447 Crash[2]. On June 1st, 2009, the Air France 447 flight from Rio de Janeiro to Paris was flying through harsh weather during night time and was experiencing turbulences. The pilots decided to reach a higher altitude to ensure a more stable flight. As the aircraft was climbing, the speed sensors indicated normal speeds, which led the pilots to believe the ascent was proceeding successfully. In reality, the aircraft slowed down to the point that it stalled and eventually crashed. After the black box was recovered, investigators found out that the speed sensor formed ice around it, and was malfunctioning; thus providing the pilots with incorrect readings.

The main problem was the pilots' lack of awareness. They relied on information provided by the aircraft, but the information was incorrect. The pilots could have been warned, and the situation likely avoided, if the aircraft's computers had been able to notice the discrepancy between the readings of the speed sensors and the information that other CPS were using for their own specific tasks – for example, the GPS coordinates retrieved by the plane's transponder. To accomplish this, the observing system would have had to have a more holistic view of the plane's components, together with an understanding the physical correlation between GPS coordinates and speed.

Unfortunately, in actual systems, there are often too many possible correlations for the designers to explicitly enumerate them all, or even to be aware of them. What is needed is a way of detecting such discrepancies directly from the designs of the CPS and from a general-purpose model of the surrounding physical environment.

2 Background Information

2.1 Architecture Analysis and Design Language

AADL (SAE, 2012) was inspired from the DARPA funded language MetaH, which was designed to describe embedded systems. MetaH was able to describe both software and hardware. Among the software components were threads, subprograms and data, whereas the hardware components include processor, memory, bus, and devices. MetaH had also a predefined list of properties that could be used to describe any of these components. AADL inherits these features from MetaH.

[2] http://www.airfrance447.com/

AADL is designed to support architectural description of complex SCSs, such as aircrafts, automotive electronics, and robotics. These systems highly rely on the non-functional requirements like safety, fault toleration, security, throughput and security. AADL can be used to create an architectural model of these complex systems and test for non-functional requirements even before the development begins.

AADL describes component properties and how they interact with each other. The final purpose is to be able to run multiple analyses on these models to prove that the non-functional requirements will be met, so that changes could be made early in the engineering development process. Multiple analysis tools were developed to test for timing, fault and error behaviors and safety. The language can be extended by adding new property types and user-defined "annexes".

The Open Source AADL Tool Environment (OSATE2[3]) that extends the Eclipse Integrated Development Environment and contains various analyses that are intended for end users to test their AADL models. Furthermore, it is a framework for developers who can implement new analyses based on AADL or its annexes syntax.

2.2 Answer Set Programming and Dynamic Domains

We begin by defining the syntax and semantics of ASP (Gelfond & Lifschitz, 1991); (Niemela & Simons, 2000). Let Σ be a signature containing constant, function and predicate symbols. Terms and atoms are formed as usual in first-order logic. A literal is an atom or its strong negation ¬a. A rule is a statement of the form: $l_o \leftarrow l_1, \ldots, l_m$, not $l_{m+1}, \ldots,$ not l_n where l_i's are literals and *not* is the so-called default negation. The intuitive meaning of the rule is that *a reasoner who believes* $\{l_1, \ldots, l_m\}$ *and has no reason to believe* $\{l_{m+1}, \ldots, l_n\}$*, must believe* l_o. Rules of the form $h \leftarrow$ not h, $l_1, \ldots,$ not l_n are abbreviated into $\leftarrow l_1, \ldots,$ not l_n, and called constraints. The intuitive meaning of a constraint is that $\{l_1, \ldots, l_n\}$ must not be satisfied. A program is a set of rules over Σ. A consistent set S of literals is closed under a rule if $l_o \in S$ whenever $\{l_1, \ldots, l_m\} \subseteq S$ and $\{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$. Set S is an answer set of a *not*-free program Π if S is the minimal set closed under its rules. The *reduct*, Π^S, of an arbitrary program Π w.r.t. S is obtained from Π by removing every rule containing an expression *not l* s.t. $l \in S$ and by removing every other occurrence of *not l*. Set S is an answer set of Π if it is the answer set of Π^S.

For the formalization of CPS and of the physical environment, we use techniques from reasoning about actions and change. Fluents are terms denoting the properties of interest of the domain (whose truth value typically depends on time). For example, inState(aircraft1, flight) may represent the fact that the aircraft is in flight. A fluent literal is either a fluent f or its negation ¬f. Elementary actions are also first-order terms. For example, *takeOff(aircraft)* means that the aircraft takes off, and changes its state from *ground* to *flight*. A compound action is a set of elementary actions, denoting their concurrent execution. A set S of fluent literals is consistent if $\forall$ f, $\{f, \neg f\} \not\subseteq S$ and complete if $\forall f, \{f, \neg f\} \cap S \neq \emptyset$. The set of the possible evolutions of a domain is represented by a transition diagram, i.e., a directed graph whose nodes – each labeled by a

[3] https://wiki.sei.cmu.edu/aadl/index.php/Osate_2

complete and consistent set of fluent literals – represent the states of the domain, and whose arcs – labeled by sets of actions – describe state transitions. A state transition is identified by a triple, $<\sigma_0, a, \sigma_1>$, where σ_i's are states and a is a compound action.

Transition diagrams can be compactly represented using an indirect encoding based on the research on action languages (Gelfond & Lifschitz, 1998). We adopt the variant of writing such encoding in ASP – see, e.g., (Balduccini, Gelfond, & Nogueira, 2000). The encoding relies on the notion of a trajectory $<\sigma_0, a_0, \sigma_1, a_1, \ldots>$, The states in a trajectory are identified by integers (0 is the initial state). The fact that a fluent f holds at a step i is represented by atom holds(f, i). If ¬f is true, we write ¬holds(f, i). Occurrences of elementary actions are represented by an expression occurs(a, i). ASP rules (also called laws in this context) describe the effects of actions. An action description is a collection of such rules, together with rules (called inertia axioms) formalizing the inertial behavior of fluents.

3 Approach

As we mentioned above, our approach aims to make it possible to cross-validate the behavior of a set of CPS by leveraging the implicit links established by the physical environment in which they coexist. Our approach relies on the following:

1. **An architecture of the CPS in AADL format** provides an unambiguous architectural knowledge base of the main CPS and its containing sub-systems, their hardware and software components and the interactions among them. For the purpose of reasoning, the CPS is viewed as a dynamic domain and its AADL specification is translated to collections of causal laws encoded in ASP.
2. **A description of the physical world** defines elements of the physical world, such as medium (e.g. air, water), space positioning (e.g. CPS X is 4 meters away from CPS Y), environmental properties (e.g. density, humidity), physical laws (e.g. gravitation law, wave propagation), and constraints (e.g. an object has only one position a time). In order to correlate the behavior of the environment to the CPS, we link CPS's sensors and actuators to the corresponding properties of the environment (e.g. the pressure is a property of the environment and a CPS may sense it by means of a barometric sensor). The physical world is represented, too, as a dynamic domain and its evolution described by means of causal laws.
3. **A query** is a question about the system, such as "is it possible for CPS X to produce output O given that CPS Y is in state S?" We reduce answering a query to checking for the existence of a trajectory satisfying the given requirements. A query is encoded in ASP in the form of observations about fluents and occurrences of actions.

The AADL system description(s) and the physical world model form a knowledge base that captures the composite behavior of the complete system.

The first step of the process is to translate the AADL description to ASP. At the current stage, we have defined a translation for a small, but representative fragment of AADL. The translation process is described in more detail later.

The second step is to provide a model of the physical world that includes all elements

that may impact the behavior of the CPS. Ideally, one should be able to achieve this by providing a *general theory of the physical environment*. At this early stage of the work, however, our goal is still to viability the utility of this modeling approach, and thus we allow for the physical model to be developed ad-hoc for a given scenario, and for the choice of the model elements to be influenced by the desired type of cross-validations.

The third step is to provide a query encoded as a set of ASP rules, with the expectation that the observations about fluents and actions provided in the query will be checked against the expected evolution of the complete system, and any anomalies detected. For example, a query (in English) could be: "Is it possible for the altimeter on the dashboard (where the dashboard and associated sensory and computational devices constitute a CPS) to report that an aircraft is flying at 2,000ft and for the pressurization system to be actively pressurizing the cabin?" Note that such a query can be answered only by taking into account the link, present in the physical world, between altitude and atmospheric pressure.

Lastly, the AADL model is combined into a single ASP program together with physical world model and query, and with general axioms capturing the reasoning processes. Finding answers to the query is reduced to finding answer sets of this program. In the case of the sample query above, answering it can be reduced to checking for the existence of a state in which the corresponding fluents hold. It is not difficult to see that, if the equipment is in working order, no state exists in which the altimeter reports 2,000ft and the pressurization system is actively working. A possible line of reasoning is that a dashboard CPS providing a reading of 2,000ft implies an actual aircraft altitude of 2,000ft. Using a model of the physical environment, one can derive that the atmospheric pressure at that altitude is 14psi – which is close to the standard atmospheric pressure measured at the sea level. With such high atmospheric pressure, a model of a typical pressurization system will predict that the system remains inactive. Hence, if everything is in normal working conditions, it is impossible for both observations to hold.

During the design phase, engineers can pose queries to check whether undesirable conditions may occur, either because of the interaction of individual CPS through the physical world (such as a CPS' vibrations affecting another CPS' measurements because of poor insulation) or because of ramifications of events in the physical world (e.g., earthquakes causing tsunamis in certain geographical areas). At runtime, queries can be used to check for anomalies, which may be due to malfunctioning equipment (such as a faulty altimeter or pressurization system in the previous example) or to voluntary tampering. The latter case is of particular interest in cyber security, as it allows for information from multiple CPS to be used for cross-validation, under the hope that an attacker will lack complete knowledge of all of the CPS and of how they are arranged in the physical world and of the layout and properties of their physical locations (e.g., thickness, material, and location of any surrounding walls).

4 Case Study

To demonstrate our approach, let us consider a simple example: suppose that in a room, there is a personal computer (PC) with fan that rotates when the computer is on, and a microphone, located 3 meters away from the computer and independent from it (i.e. it is used for an entirely different reason, such as part of the landline telephone[4]). Both the computer and the fan are part of the same physical environment depicted in Figure 1. Our goal is to use knowledge about the CPS and about the properties of the environment to reason about possible links between noises picked up by the microphone and the on/off state of the computer. For example, assuming that the room is otherwise empty, is it possible that the computer is off and yet the microphone detects a noise? To answer questions of this kind in a systematic manner, one needs to have a suitable, uniform representation of CPS and environment, and a way to reason about the evolution of the domain and about how it relates to the query.

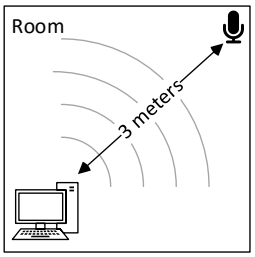

Figure 1. A room containing a computer and a microphone

Next, we describe how we model the computer and the fan in AADL, discuss the translation process to ASP, the encoding of the physical world in ASP, and end by demonstrating how non-trivial queries about the scenario can be answered.

AADL Model of PC and Microphone

In the following, we describe features and behaviors that we are interested in capturing, give their formal AADL representation and then show the translation to ASP.

Modeling a (cyber-physical) system: A computer, from a high level perspective, is a (cyber-physical) system that has inputs (or "features" in AADL terminology), different states of operation (or "modes" in AADL terminology), and properties. A system is defined by the AADL construct:

```
system computer
    features, modes, and properties are specified here
end computer;
```

Our translation procedure maps this to an ASP fact:

```
system(computer).
```

which intuitively states that "computer" is a system.

Modeling the inputs: Next, we capture the effect of switching the power button `on` and off. In AADL this can be encoded by means of event signals carried by event ports. A `turnOnEvent` signal is assumed to mean that the power button has been switched

[4] An actual example is a smart night lamp that integrates various sensors, including a microphone: http://www.pocketables.com/2015/04/leeo-smart-alert-nightlight-review.html.

to `on`. Similarly, when the power button is switched to `off`, a `turnOffEvent` signal is generated[5]. Thus, we list two corresponding event ports in the description of features.

```
features
    turnOnEvent: in event port;
    turnOffEvent: in event port;
```

Our translator procedure maps this information to the ASP facts:

```
feature(computer, turnonevent).
eventPortFeature(turnonevent).
featureDirection(turnonevent, in).
% similarly we translate the turnOffEvent feature
```

The first two lines indicate that the `turnOnEvent` is an event port, whereas the third line describes its direction.

Modeling modes: The on/off state of the computer is captured by specifying two corresponding *modes* and the initial state of the system (set to off):

```
modes
    off: initial mode;
    on: mode;
```

Our translator procedure maps this information to ASP facts:

```
mode(computer, off).
mode(computer, on).
holds(currentMode(computer, off), 0).
```

The first two lines define the modes; the last line defines the initial mode. This information is encoded by introducing a fluent `currentMode`, which specifies the current mode of a system, and by saying that the current mode at step 0 is `off`.

Modeling mode-transitions: a system's behavior over time is represented in AADL by mode transitions. In our case, we are interested in capturing the following mode transitions: if a `turnOnEvent` signal is received while the PC is `off`, then the PC's current mode becomes `on`. Similarly, when `turnOffEvent` received while the computer is on, the PC's current mode changes to `off`.

```
modes
    off: initial mode;
    on: mode;
    off -[turnOnEvent]-> on;
     on -[turnOffEvent]-> off;
```

Mode transitions can be naturally rendered by means of dynamic laws, e.g.:

```
holds(currentMode(computer, on), S+1) :-
    holds(currentMode(computer, off), S),
    occurs(eventPortSignal(turnonevent), S),
    step(S).
```

The first line specifies that the mode at step S+1 is off if a `turnOnEvent` occurs, at step S, while the computer is in the `on` mode. The encoding of the semantics of modes is completed by a state constraint stating that a device can only be in one mode at any time:

[5] There are multiple ways to represent the functionality of a switch. The one chosen here allows us to show important AADL concepts.

```
-holds(currentMode(SYSTEM, MODE1), S) :-
    holds(currentMode(SYSTEM, MODE2), S),
    MODE1 != MODE2,
    mode(SYSTEM, MODE1), mode(SYSTEM, MODE2),
    system(SYSTEM), step(S).
```

Modeling mode-specific properties: AADL allows for the value of a property to depend on the current mode. In our example, such *mode properties* are:

- The fan rotational speed is `0RPM` when the computer is `off`, or `150RPM` when the computer is `on`.
- The fan sound is `42dB` when the computer is `on` and `0dB` when it is `off`[6].
- The input voltage is `12V` and intensity `0.08A` when the computer is `on`, and `0` when it is `off`.

```
properties
    fan_rpm => 150.0 RPM in modes (on);
    fan_rpm => 0.0 RPM in modes (off);
    fan_sound => 42.0 dB in modes (on);
    fan_sound => 0.0 dB in modes (off);
    fan_input_voltage => 12.0 V in modes (on);
    fan_input_intensity => 0.08 A in modes (on);
    fan_input_voltage => 0.0 V in modes (off);
    fan_input_intensity => 0.0 A in modes (off);
```

The dependencies between modes and properties can be naturally encoded as state constraints. For example, property `fan_rpm` can be formalized by the rules:

```
holds(property(computer, fan_rpm, 150), S) :-
       holds(currentMode(computer, on), S), step(S).
holds(property(computer, fan_rpm, 0), S) :-
       holds(currentMode(computer, off), S), step(S).
```

The representation uses another general fluent, `property(o,p,v)`, which states that object `o`'s property `p` has value `v`. Specifically, the first statement says that the fan rotates at 150 rpm when the computer is on, whereas the second statement states that there is no fan rotation when the computer is off.

Modeling computed properties: A computed property is a property whose value is calculated at run-time based on some arithmetic expression, typically based on the inputs and properties of the system. For demonstration purposes, we model the power consumption of the fan as a property whose value depends on intensity and voltage:

```
properties
    fan_power => compute(power_expression);
    power_expression => "%fan_input_voltage% *
                                    %fan_input_intensity%";
```

The definition of AADL does not include a specification of the syntax of the expression that occurs in a compute statement. Hence, we introduce a custom notation in which the argument of the compute statement is the name of an arithmetic expression, and the expression itself uses literals surrounded by "%" to denote names of properties.

[6] It is indeed possible to calculate the fan sound from the rotation speed and other physical properties, but we adopt this simpler representation to simplify the presentation.

Once again, a computed property can naturally be represented using a state constraint. In the ASP encoding, we use an auxiliary relation `computed_value` for the calculation of the value of the expression:

```
holds(property(computer, fan_power, X), S) :-
        computed_value(power_expression, X, S),
        step(S).
computed_value(power_expression, X0  *  X1, S) :-
        holds(property(computer,fan_input_voltage, X0),S),
        holds(property(computer,fan_input_intensity,X1),S),
        step(S).
```

The value of the property is computed as the product of the voltage and intensity at a given step S. While this formalization approach works well for simple models, it is likely that efficient computation in the presence of more complex models will require the adoption of more advanced techniques for numerical computations, such as Constraint Answer Set Programming (Balduccini M. , July 2009).

The microphone is modeled using the same methodology. The microphone takes as an input the sound in the environment, represented as a data signal `sound_in` and produces a boolean output indicating whether there is noise in the room. Assuming that the sound levels the microphone can perceive are between 0 and 155dB, the connection between input and output can be modeled by the expression $output = \frac{input+145}{150}$, where the line indicates integer division. The expression will yield 0 if the sound is less than 5dB, and 1 if the sound is above 5dB. The corresponding AADL model uses a computed property.

```
device microphone
    features
    sound_in: in data;
    properties
        mic_sound_out => compute(expression);
        expression => "(%features::sound_in% + 145)/150";
end microphone;
```

The ASP encoding is similar to the one shown above.

Representation of the Physical World

In our approach, the physical world is modeled directly as a dynamic domain using techniques from reasoning about actions and change. For this case study, we focus on modeling the propagation of sound in the room.

It is important to recall that our ultimate goal is to be able to identify correlations in a collection of CPS even when they were not explicitly considered by the designers of the CPS. Our conjecture is that this can be accomplished by providing a sufficiently general model of the physical environment, and establishing proper links between the properties of the CPS and physical properties.

To keep the environmental model general, we rely on a set of input relations that specify general classes of objects and properties, of which the elements of the CPS are

specific instances. Input relations used in our case study specify that `computer` is a sound source, and that the property of the model of the `computer` that specifies the sound (level) it emittes is `fan_sound`:

```
soundSource(computer).
soundPropertyOf(computer, fan_sound).
```

Next is the formalization of the phenomenon of sound propagation. The encoding uses a relation of the form `distance_between(OBJ1, OBJ2, DISTANCE_UNITS)` to specify the distance between two objects. For example, if the distance between the computer and the microphone is `3` units, we write:

```
distance_between(computer, microphone, 3).
distance_between(X,Y,DIST) :- distance_between(Y,X,DIST).
```

Next, we formalize the sound propagation in the environment by using the Inverse Square Law to describe that sound power drops in intensity with the square of the propagated distance:

$$P(distance) = \frac{P_{init}}{distance^2}$$

We use this law to determine how the sound from each individual source propagates over distance. This is represented by fluent `sound_level(SOURCE, DISTANCE, SOUND_POWER)` where the distance is relative to the sound source. The corresponding sound level is determined by the state constraint:

```
holds(sound_level(SOURCE, DISTANCE, SOUND_LEVEL / (DISTANCE *
                  DISTANCE)), S) :-
      holds(property(SOURCE, SOUND_PROP, SOUND_LEVEL), S),
      soundSource(SOURCE),
      soundPropertyOf(SOURCE, SOUND_PROP),
      distance(DISTANCE),
      step(S).
```

Note how the state constraint is parametrized to the properties used in the model of the sound source. Next, we calculate the total sound present at a location in the environment as the sum of the contributions of the individual sound sources. This notion is captured by fluent `sound_level_at(DESTINATION, SUM)` and defined by another state constraint.

```
holds(sound_level_at(DESTINATION, SUM), S) :-
    distance_between(DESTINATION, SOURCE2, D2),
    soundSource(SOURCE),
    D2>0,
    holds(sound_level(SOURCE2, D2, V2), S),
    step(S),
    SUM = #sum{
       VALUE: holds(sound_level(SOURCE, DISTANCE, VALUE),S),
       distance_between(DESTINATION, SOURCE, DISTANCE),
       soundSource(SOURCE)
    }.
```

Lastly, we encode the persistency of `currentMode` over time, by defining it as a fluent (lines 1-3) and by including the inertia axioms, which capture the inertial behavior of fluents (only the one for `holds(F, S+1)` is shown to save space):

```
fluent(currentMode(SYSTEM, MODE) :-
          system(SYSTEM), mode(MODE, SYSTEM).
```

```
holds(F, S+1) :-
          fluent(F), step(S),
          holds(F, S), not -holds(F, S+1).
```

Representing and Answering Queries

Recall that our goal is to enable the cross-validation of the behavior of a set of CPS by leveraging the implicit links established by the physical environment in which they coexist. In this context, queries are sets of observations about the history of the domain, of which one wants to determine the plausibility. The query-answering mechanism follows the lines of the diagnostic reasoning approach from (Gelfond M. B., Jul. 2003), and relies on the idea of checking whether the transition diagram defined by the model of the domain (CPS *and* physical world) contains a trajectory that is compatible with the given observations. The reasoning task is encoded by the general-purpose axioms:

```
holds(F,0) :- obs(F,t,0).
-holds(F,0) :- obs(F,f,0).
occurs(A,S) :- hpd(A,S).

% Reality Check Axioms

:- obs(F,f,S), not -holds(F,S).
:- obs(F,t,S), not holds(F,S).
```

An atom of the form `obs(F,t,S)` states that fluent F was observed to be true at step S (false in the case of `obs(F,f,S))` and `hpd(A,S)` says that action A was observed to occur at S. Above, the first 2 rules say that any fluent observed to be true at step 0 must be taken to be true in the initial state of any trajectory considered (similarly for observations about fluents that are false). The third rule states that any action observed to occur must be accounted for in the trajectory. Finally, the *Reality Check Axioms* state that any prediction made about the evolution of the system must match the observations provided. We refer the reader to (Gelfond M. B., Jul. 2003) for more details on the rationale of these axioms.

The final step is to provide the queries. For our case study, we consider four queries, which are simple but require non-trivial reasoning about the connections among CPS and environment.

1) Can the computer be off at step 0 and the microphone hear a sound at step 1?

```
obs(property(microphone, mic_sound_out, 1), t, 1).
```

Note that there is no need to provide an observation about the computer being on at step 0, since that is the initial state of the CPS. It is not difficult to see that the program consisting of the observation together with all of the rules shown above is inconsistent, which indicates that it is not plausible for the computer to be off and for the microphone to hear a sound. Intuitively, the inconsistency of the program results from the fact that, if the computer is off, the computer's model predicts that the fan will generate no sound. In turn, the physical model predicts that the sound level at the location of the microphone will be 0. Finally, the model of the microphone predicts that the boolean output of the microphone is 0, which, by virtue of the Reality Check Axioms, contradicts the

observation provided.

2) Can the computer be turned on at step 1 and the microphone hear a sound at step 2?

```
hpd(eventPortSignal(turnonevent), 0).
obs(property(microphone, mic_sound_out, 1), t, 2).
```

In this case the ASP program is consistent: since the computer is turned on at step 1, the computer's model predicts its current mode to be on at step 2. With reasoning similar to the one carried out for the first query, it is not difficult to predict that the boolean output of the microphone at step 2 will be 1, which matches the observation provided.

3) Is it possible that the system is switched on at step 1, the microphone hears a noise at step 2 and the microphone is 6 units away from the computer?

This query can be encoded by replacing the fact about the distance between the two CPS by with the fact `distance_between(computer, microphone, 6).` Additionally, we provide the observations:

```
hpd(eventPortSignal(turnonevent), 1).
obs(property(microphone, mic_sound_out, 1), t, 2).
```

In this case, the program is inconsistent, indicating the implausibility of the observations. This is due to the fact that the microphone is too far from the computer.

4) Can the computer be switched on at step 1, off at step 2 and the microphone hear a sound at step 3?

```
hpd(eventPortSignal(turnonevent), 0).
hpd(eventPortSignal(turnoffevent), 1).
obs(property(microphone, mic_sound_out, 1), t, 2).
```

This query can be answered by carrying out reasoning similar to the one of the first query. The program is inconsistent, which indicates the implausibility of the observations. The challenge here is that the formalization must be capable of dealing with the effects of multiple changes of mode of the computer due to the two subsequent actions. The key to making this possible is the use of a principled formalization based on action languages.

5 Tool

To test our approach and to make it accessible to engineers, we have implemented a tool that automates the above processes by (1) automatically translating AADL models to ASP, (2) allowing the user to provide a physical model, and (3) checking the plausibility of the queries provided. The tool has been developed as a plugin for the AADL tool environment, OSATE. OSATE, which is itself a component of the Eclipse Integrated Development Environment, provides engineers with access to AADL model analysis tools, such as fault impact analysis, fault tree analysis, and functional hazard assessment. The tool relies on OSATE's ability to parse AADL models and to extract

information from them.

The interaction with the tool follows the workflow shown in Figure 2. First, the user needs to save the AADL models in AAXL format (an XML encoding of AADL). The tool parses the AAXL files using the functions provided by OSATE, and translates it to ASP as shown in the previous sections. At this point, the tool presents a window (Figure 4), allowing the user to view or change the ASP representation of the model. The application also allows the user to provide the model of the environment and the query by filling in the designated text areas. When the user request the execution of the query, the system uses the clingo-4.4 solver to check whether the corresponding ASP program is consistent and displays the output of the computation in a new window (Figure 3). The window shows one answer set if the program is consistent, and otherwise states that the program is inconsistent.

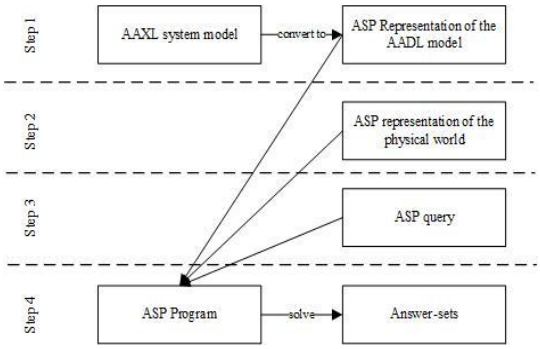

Figure 2. A step-by-step interaction with the tool, from a user's standpoint

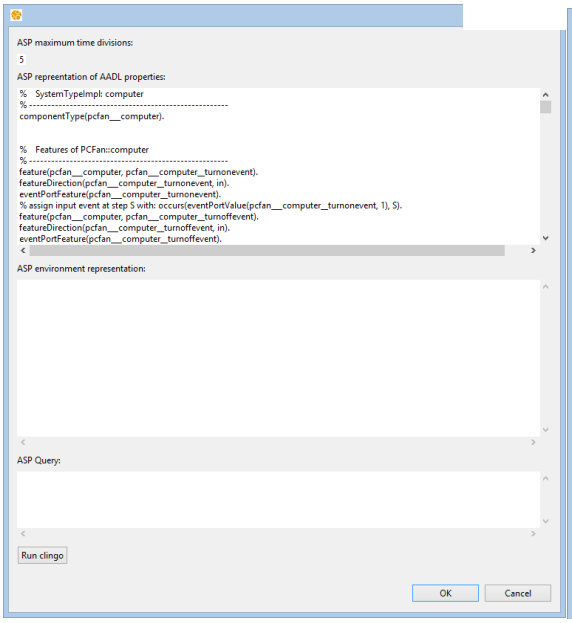

Figure 4. Pop-up showing the automatic translation of the AADL models

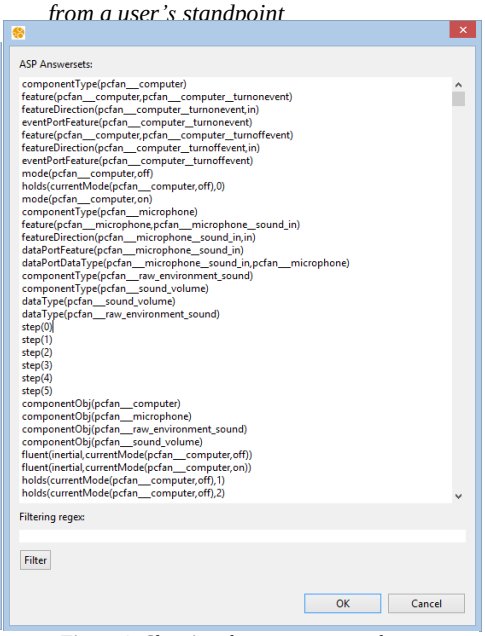

Figure 3. Showing the answer set to the query

6 Conclusions

This paper presented a method for modeling, and reasoning about, the interactions of situated CPS that relies on creating a uniform model of the CPS involved and of the physical environment they are in. To the best of our knowledge, this is not possible with the state-of-the-art techniques used by the CPS community. Our approach enables the cross-validation of the behavior of a set of CPS by leveraging the implicit links established by the physical environment in which they coexist. As we discussed, this ability is fundamental to ensure robustness, resiliency, and security of SCSs.

The approach relies on modeling the corresponding systems by means of action languages and ASP. Answering queries about the interactions is reduced to checking for the consistency of a suitable ASP program. A graphical tool was also developed, which integrates in a state-of-the-art CPS design tool and automates the use of our technique.

Acknowledgments. We would like to thank Matthew Barry for introducing us to the Architecture Analysis and Design Language and for useful discussions on possible links with Answer Set Programming.

References

Balduccini, M. (July 2009). Representing Constraint Satisfaction Problems in Answer Set Programming. *ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09)*, 15.

Balduccini, M., Gelfond, M., & Nogueira, M. (2000). A-Prolog as a tool for declarative programming. Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000).

Chen, L., Nugent, C., Mulvenna, M., Finlay, D., & Hong, X. (2008). Using Event Calculus for Behaviour Reasoning and Assistance in a Smart Home. *6th International Conference On Smart Homes and Health Telematics*.

Gelfond, M. B. (Jul. 2003). Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP)*, 42.

Gelfond, M., & Lifschitz. (1998). Action Languages.

Gelfond, M., & Lifschitz, V. (1991). Classical Negation in Logic Programs and Disjunctive Databases.

Niemela, I., & Simons, P. (2000). Logic-Based Artificial Intelligence. Kluwer Academic Publisher.

SAE, I. (2012). Standard AS5506B: Architecture Analysis & Design Language (AADL). Retrieved from http://standards.sae.org/as5506b/

Shiraishi, S. (2010). An AADL-Based Approach to Variability Modeling of Automotive Control Systems. *Conference: Model Driven Engineering Languages and Systems - 13th International Conference*, 4.

Tan, Y., Vuran, M., Goddard, S., Yue, Y., Song, M., & Ren, S. (Apr. 2010). A Concept Lattice-based Event Model for Cyber-Physical Systems. *1st International Conference on Cyber-Physical Systems*, 7-10.