Semantic analysis and verification of context-driven adaptive applications in intelligent environments

Davy Preuveneers¹ · Wouter Joosen¹

Received: 27 January 2016 / Accepted: 9 March 2016 / Published online: 13 April 2016
© Springer International Publishing Switzerland 2016

Abstract With the advent of the Internet of Things (IoT) and Cyber-Physical Systems (CPS), assistive applications and smart environments can tap into a growing amount of context information to learn about their users, their surroundings and typical behavior. This information is useful to adapt intelligently, autonomously and non-intrusively to a myriad of circumstances. However, trustworthiness and reliable adaptation to changes in line with user expectations—especially to situations that the developers did not anticipate—remain key concerns. Understanding the impact of changes from a developer (design time) and system (runtime) perspective, and ensuring that no undesired side effects take place are two non-trivial research challenges to increase the adoption of such applications. Given the limited tool support for anticipating change at design time and runtime, we present our *change impact analysis* (CIA) methodology—found in the formal semantic modeling of intelligent environments and rule-based application behavior—to contribute to the development and deployment of reliable context-aware adaptive applications. We validate our contributions on non-trivial smart home and office scenarios, and demonstrate how our framework helps increase trust in intelligent environment applications by anticipating change implications upfront at design time and by minimizing the occurrence of undesired side effects at runtime.

Keywords Semantic verification · Change impact analysis · Context · Reliability · Rule-based adaptation · Intelligent environments

1 Introduction

Smart systems and intelligent environments aim at making the life of their occupants more comfortable. A support network of context-aware applications and services proactively address user needs and assist people with their activities of daily living (ADL) [30]. By unobtrusive sensing, collecting, reasoning and acting on various kinds of personal and situational information, context-aware systems can take well-informed decisions and manifest autonomous behavior that makes these environments truly smart. The adaptive nature of such systems is often characterized by context-driven rules that define how these systems should adapt in particular circumstances. Over the past decade, context-aware adaptive applications have grown in sophistication from simple mobile applications for single users [29,34] to highly distributed multi-user applications that are offered as services running in the cloud [27].

Situational awareness is an important feature of intelligent environments [43], but a key challenge with building context-aware and adaptive systems is that the developer's assumptions of what is intelligent situation-dependent behavior may not be the same as the expectations of the individuals that end up using these systems. For context agnostic systems that mostly work in isolation without humans in the loop, the developer has a good understanding of the intended operational circumstances, and changes can be *planned* at design time or during deployment. Intelligent context-aware applications, however, have their behavior largely triggered and driven by external influences and events in the user's envi-

✉ Davy Preuveneers
davy.preuveneers@cs.kuleuven.be

Wouter Joosen
wouter.joosen@cs.kuleuven.be

¹ iMinds-DistriNet-KU Leuven, Celestijnenlaan 200A,
3001 Heverlee, Belgium

ronment. As a result, dealing with *change* becomes far more challenging. Indeed, changes in application behavior occur not only after modifications to the program logic or configuration by the software developer (i.e. *internal* changes), but also when the user's preferences or the operational circumstances of the intelligent environment (i.e. *external* changes) evolve. Furthermore, simple adjustments to an application may trigger changes elsewhere in the environment, which in turn may trigger cascading and possibly conflicting adaptations [33]. As the potential consequences of unexpected side effects of subsequent changes may not always be clear upfront and as software systems for intelligent environments become increasingly complex, the need for better design and runtime support to analyze adaptations and understand the impact of unanticipated changes grows. Such tools and frameworks would not only be helpful for software developers, but also as part of the intelligent system itself to better anticipate impact at runtime before enacting autonomous decisions to adapt.

Robust intelligent systems [45] ensure acceptable performance, not only under ordinary conditions but also under unusual conditions that stress the developers' assumptions of the system. However, it is difficult for a developer *at design time* to discover and eliminate all errors when the application will be active outside the intended limited context. As a result, fragile context-aware applications are subject to subtle or more severe errors that only make their presence known in unusual circumstances *at runtime*. To build reliable and trustworthy intelligent environments, it is paramount that (1) developers can analyze the implications of changes early in the design phase, and that (2) software systems can ascertain at runtime whether enacting certain changes or reconfigurations will cause these systems to evolve from one consistent reliable state to the next, while mitigating undesired side effects and unexpected behavior along the way as much as possible.

In this work, we focus on intelligent environment applications whose dynamic behavior is driven by context-aware decisions and adaptation rules [44] (following the typical *event-condition-action* paradigm). The complexity of guaranteeing robustness is caused by the fact that ensuring consistency among adaptation rules is not trivial as many rules, system and environment features may interact with one another [19]. This problem is further aggravated when users can customize *at runtime* the context-driven adaptation rules that modify the behavior of an intelligent application or environment.

We build on and extend our previous research [38,41,44] to offer a new integrated methodology—found in the formal modeling of the adaptive behavior of systems with description logics—to improve the reliability of applications at design time and runtime. We revisit best practices on change impact analysis [1] (CIA) in the software

Fig. 1 The change impact analysis control loop of an adaptive application in an intelligent environment

engineering domain, and enhance it to contribute to the development of reliable context-aware adaptive applications and intelligent environment systems. In the software evolution domain, change impact analysis is a well-established methodology to manage change, but it is usually carried out at the level of source code modifications. The challenge that we address in this work is that state-of-practice change impact analysis methodologies are not sufficiently equipped to reason upon the impact of (un)planned changes, especially for dynamically adapting software systems such as those that are frequently found in intelligent environments.

The design-time and runtime framework we propose to address the aforementioned change impact analysis challenges leverages the OWL 2 RL¹ ontology language [16,22] to formally and semantically represent not only the intelligent environment in which the application operates, but also the application-specific rule-based adaptive behavior. From a practical perspective, this unified framework simplifies reasoning upon change, but it also facilitates the development lifecycle transition from design time to runtime as the same framework components are also used as part of the application's control loop (see Fig. 1). As such, the semantic application model [32] and adaptation rules are the foundations of our change impact analysis framework.

The contributions of our work are threefold, and they can be summarized as a unified design-time and runtime change impact analysis framework with the following key benefits:

1. Analyze the consistency of rule-based adaptation behavior at design time. Contrary to our previous work [38],

¹ http://www.w3.org/TR/owl2-profiles/#OWL_2_RL.

we do not use the SPIN [18] model checker but use a description logic-based approach instead.

2. Improve at design time the understanding of the implications of changes [42] by integrating and unifying domain knowledge about the environment such that the system can plan and make better informed adaptation decisions at runtime.
3. Verify the outcome of adaptation rules [41] by testing asserted axioms reflecting desired objectives and observations at runtime.

These contributions also support our design for failure [25] approach to specify what a system should do to minimize the impact of a failure, leading to a complementary strategy for fault detection, identification and recovery.

In Sect. 2 we provide an overview of related work on enabling technologies, tools and methodologies to improve the reliability of intelligent environments. To illustrate the complexity of change impact analysis for context-aware applications in intelligent environments, we present non-trivial motivating use cases and scenarios in Sect. 3, including an overview of changes that our encompassing impact analysis methodology for reliable intelligent environments supports. In Sect. 4, we elaborate further on the implementation of our design time and runtime framework. We reflect back on our work in Sect. 5 before concluding with our final thoughts and suggestions for future work in Sect. 6.

2 Related work

In this section, we review relevant related work on tools and methodologies to improve the reliability of intelligent environments and assisting technologies. The motivation for this kind of research is that intelligent environments and systems are often too naively or too optimistically developed, assuming that users always understand what these systems are doing or that systems know what users want.

The complexity of making our environment intelligent, non-intrusive, proactive and moreover intuitive to use, has proven to be very high. As systems interact with each other in sophisticated ways [23], the complexity grows which adversely impacts the reliability [10] of the intelligent environment. Recent work by Augusto et al. [2] confirms that little is reported in the literature on methodologies to increase the reliability of software for safe and trustworthy intelligent environments.

Model checking with software tools like SPIN [18] and Alloy [21] can aid with the development of intelligent systems by verifying consistency, safety and reliability properties of the system. Model checking has been around for more than a decade to formally verify models of distrib-

uted software systems, and the motivation for using model checking to verify context-driven applications is straightforward. Coronato et al. [5, 12] explored the formal specification and verification of ubiquitous computing applications to analyze critical requirements regarding functional correctness, reliability, availability, security, and safety. The SPIN model checker was used by Augusto et al. [3, 4] and Preuveneers et al. [38] to increase the reliability of intelligent environments through simulation and symbolic verification. The advantage of tools like SPIN is that they can produce a counter example if a particular system property may be violated. However, from a practical point of view, there are often too many possible configurations and contextual situations that an application can be in. Counter examples can be unrealistic in practice due to the inability to incorporate rich domain knowledge in the system model that the model checker can use to verify a system. Additionally, if the system model to be verified involves complex situations with multiple context variables and context-aware rules and assertions, it is fairly likely that the exponential state space explosion will become a critical concern to verify the consistency of the context-aware adaptation rules. It is also not straightforward to accurately model external influences in an intelligent environment.

From the software engineering domain, there is a significant body of work related to software change impact analysis, with Arnold's book on *Software Change Impact Analysis* [1] highlighting many important topics, including the need for *traceability* between requirements and design elements of the software architecture, and the representation of *dependencies* that determine the consequences of changes. Lenhert [24] presented a taxonomy of impact analysis methods. Based on a broad literature review, he presents a multitude of criteria—ranging from the scope of analysis, the granularity of changes and impact, up to the availability of tool support—to classify and compare various impact analysis approaches. A sound representation of dependencies is instrumental to analyze and measure the impact of changes. German et al. [13] proposed the concept of change impact graphs to determine the impact prior to the actual enactment of code changes. Their method recursively defines the dependency graph $G(f)$ of a function f based on the other functions that can be reached by f .

Other works, such as those by Lock et al. [26] and by Tang et al. [46], focus on how to use probabilistic causal relationships to assess how changing requirements and design decisions may affect design elements of a software architecture. These kinds of works go beyond rule-based inference in the analysis phase, and for example, leverage Bayesian Belief Networks (BBN) to quantify the likelihood of change impact. Briand et al. [9] proposed an UML-based method for impact analysis. Their model-based methodology is used to predict

the cost and the complexity of changes to decide whether to actually implement these changes in a next iteration of a software release. It first starts with a consistency validation phase of the UML models of the system. The change impact analysis itself is then carried out between two different versions of the UML model. Their framework offers a set of change detection rules with respect to a given change taxonomy. Model elements that directly or indirectly undergo changes are formally represented by impact analysis rules defined in the Object Constraint Language (OCL). Other works on impact analysis that analyzes software changes with reasoning-based expert systems and data-mining techniques were proposed by Hassan et al. [17] and Gethers et al. [14].

Reflecting back on the related work discussed in the previous sections, we concluded that there is a clear gap that needs to be filled to better support the development of reliable intelligent environments. These observations can be summarized as follows:

- It is difficult to integrate domain knowledge about the intelligent environment in the system verification process to rule out counter examples that are unrealistic in practice.
- Many tools and methodologies only focus on the design phase of the application, verifying abstract system models that are not guaranteed substantive enough to deliver assurance at runtime.
- Tools that analyze software artifacts (e.g. annotated source code with pre- and post-conditions, UML diagrams, etc.) verify in a modular fashion, making it hard to account for external influences.

Indeed, while model checking allows formal verification of the state of an application at design time, the abstract models are decoupled from the application. The outcome of formal verification at a higher level of abstraction needs to be verified against to concrete system at runtime or through simulation, making it difficult to reason upon change in concrete software systems.

With change impact analysis techniques that operate on software artifacts rather than high-level abstract application or system models, there is a tighter link with the actual application logic, but support for formal verification resulting in strong assertions about the application's context-aware behavior is limited. Additionally, both approaches suffer from the lack of incorporating domain knowledge about the intelligent environment to avoid theoretical outcomes that are not realistic in practice.

In this work, we aim to bridge these gaps by borrowing concepts from the state-of-the-art on dependency graphs, and applying formal semantic application models to reason more

effectively on the impact of adaptation rules that drive the intelligent context-aware behavior of the application.

3 Motivating smart home and smart office automation use case scenarios

To ensure their widespread adoption, intelligent applications and services must be trustworthy, and hence robust against unforeseen human interventions, exceptional operational situations and unexpected events. This is especially a concern when context-aware applications move from lab environments—where they typically operate under ideal circumstances—to the real world where they are no longer under the supervision of technical experts. A lack of understanding on how these applications work may lead to frustration with the end-users if these applications do not not behave as they expected.

In this section, we introduce motivating use cases to illustrate the impact of change in non-trivial context-aware applications, after which we will elicit relevant requirements for our change impact analysis methodology. The scenarios presented here build upon our earlier work [11, 38, 41, 42].

3.1 Smart home automation

Smart home automation systems can manage the lighting, the home theater and entertainment system, the thermostat regulation and the security system. They have sensors to measure the temperature at different places in the house, motion sensors to detect someone's presence, ambient light sensors, sensors for outdoor use, as well as switches and dimmers to control the curtains and window blinds, the lighting, the heating and the air conditioning in each room. Sensor values will drive the adaptive behavior of the home automation system. In previous work [38], we elicited example rules that typically govern such a home environment:

- *Heating* The heating should only be turned on if it is less than 19 °C inside. Sensors in the house will monitor the environment and the thermostat will aim for a temperature of 21 °C.
- *Lighting* When it is getting dark, the lights should be turned on only in those rooms with people present. The lights should be turned off automatically if no presence is detected for more than 5 min.
- *Air conditioning* If it is getting more than 27 °C in the house, then turn on the air conditioning until the temperature drops to 24 °C. Motion sensors will scan the room for occupancy and when no movement is detected it will send a signal to the air conditioning unit to switch it off.

```

1 // Turn the heating on and off, and aim for a temperature of 21 degrees Celsius
2 if (temperaturein < 19 °C) then heating = on
3 if (temperaturein > 21 °C) then heating = off
4
5 // Turn on the lights if it is getting dark with people inside, and auto turn off if nobody inside for 5 mins
6 if (illuminancein < 500 lux) and (motion = true) then lights = on
7 if (motion = false)Δt=5min then lights = off
8
9 // Turn on the air conditioning if it is getting hot. Turn it off if nobody present or if cool enough
10 if (temperaturein > 27 °C) then airconditioning = on
11 if (temperaturein < 24 °C) then airconditioning = off
12 if (motion = false) then airconditioning = off
13
14 // Upgrade the home automation system with outdoor sensors that manipulate the lights and curtains
15 if (illuminanceout > 10000 lux) then curtains = open
16 if (illuminanceout > 10000 lux) then lights = off
17 if (temperatureout > 30 °C) then curtains = close

```

Listing 1 Smart home automation rules adapting the heating, lights, airconditioning and curtains

In a second phase, additional outdoor sensors are added to the automation system, with the objective to update the system with the following automation preferences:

- *Lightning* If it is daylight outside, open the curtains or window blinds and also turn off the lights.
- *Air condition* If it is more than 30 °C outside, close the curtains and window blinds to reduce the temperature increase inside the house.

The above rules (as depicted in Listing 1) are a typical illustration of how a layman end-user (i.e. not a technology or domain expert) would specify the preferred behavior of the smart system. Although these rules seem obvious at first sight, there are dependencies between these high-level automation objectives that might interfere with one another.

Later on, we will use the DogOnt ontology² to semantically model the above home automation concepts. This ontology offers an extensive vocabulary (see Table 1 for an overview) to model intelligent domotic environments, with a technology and network-independent description of buildings, rooms, sensors, appliances, and other architectural elements [7].

3.2 Smart office automation

The smart office environment is set up with sensors of different types (people counter, light, humidity, power consumption, temperature, etc.) that collect data from the office

environment. The building is also equipped with a variety of actuators that can manipulate the environment. These include chillers, cooling towers, ice tanks, pumps, lightning, and sun shields. Individuals can set their comfort preferences. The HVAC system, however, is partially managed centrally for the whole building following a fixed program, but additional air conditioning units are available on each floor that can be configured locally and individually.

Table 1 DogOnt ontology metrics

Metric	Count
Axiom	9454
Logical axioms	7127
Classes	940
Object properties	36
Data properties	59
Individuals	369
SubClassOf axioms	2664
EquivalentClasses axioms	2
DisjointClasses axioms	2431
SubObjectPropertyOf axioms	4
InverseObjectProperties axioms	12
FunctionalObjectProperty axioms	4
InverseFunctionalObjectProperty axioms	1
ObjectPropertyDomain axioms	34
ObjectPropertyRange axioms	35

² <http://elite.polito.it/ontologies/dogont.owl>.

This smart office use case [41] is very similar to the previous scenario, but somewhat more sophisticated as the main purpose is now to balance *comfortableness* and *energy savings* for the office environment. Energy savings can be directly measured by monitoring the total power consumption and estimating the electric bill based on the various tariffs selected for the building. The comfortableness can be measured with the Predicted Mean Vote (PMV) and Predicted Percentage Dissatisfied (PPD) [20] measures, based on the spatial calculation of temperature and humidity. In fact, the PMV is the average score of a large group of people on a seven-point thermal sensation scale (+3 = hot, 0 = neutral, −3 = cold). The value can be computed based on the metabolic rate of the individual, clothing insulation, air temperature, mean radiant temperature, air velocity and humidity. The PPD—a function of the PMV—is a value representing the percentage of thermally dissatisfied people who feel too hot or too cold.

```

1 // Turn the central air conditioning on at 8 AM
2 if (time = 8:00 AM) then centralairco = on
3
4 // If everybody has left, turn off the lights
5 if (people.count = 0) then light = off
6
7 // Change the fan speed if the temperature rises
8 if (temperature ≥ 24 °C) then fanspeed = high
9
10 // If it is getting dark inside, turn on the lights
11 if (illuminance < 500 lux) then light = on
12
13 // ... and open the sun shields
14 if (illuminance < 500 lux) then sunshields = open
15
16 // Turn down the fan speed if nobody is present
17 if (motion = false) then fanspeed = low

```

Listing 2 Smart office automation rules adapting the heating, lights, airconditioning and curtains

Listing 2 provides a subset of the rules that drive the behavior of the smart office automation system. As with the smart home scenario, the complexity of ensuring robustness is caused by the fact that many rules may interact with one another. For example, closing the sun shields does not only affect the temperature inside, but also the luminosity. Both the temperature and the motion detector may influence the fan speed in conflicting ways. Both the lights and sun shields can affect the luminosity in a building to reach a certain level of comfortableness. However, from an energy savings' bill, it may be cheaper to open the sun shields rather than turning on the lights everywhere.

3.3 Rule-based adaptive behavior of context-aware applications in intelligent environments

The previous motivating use case scenarios are examples of intelligent environments and context-aware applications that dynamically adapt to satisfy new functionality requirements, to optimize performance and to accommodate to variable runtime conditions. In general, a context-aware system is usually a runtime infrastructure for deploying applications enhanced with a context middleware that is capable of acquiring and aggregating context information, allowing context-aware applications to act autonomously and making computing and communication transparent and non-intrusive to the users in day to day activities. These basic principles are depicted in Fig. 2. An *event-condition-action* (ECA) rule that models context-aware behavior states that if a particular situation emerges, a predefined action should be carried out. In related work, one usually does not explicitly state the overall objective of the rule, making it very hard whether the rule has been effective or not at runtime due to unexpected rule interactions. We therefore extend the notion of ECA rule-based behavior with two additional concepts: *goal* and *effect*. We actually build upon *design-by-contract* principles in software engineering:

- *pre-condition* a predicate that must be true just prior to the execution of a method. It indicates the assumption that certain things are true about the environment.
- *post-condition* a predicate that must always be true just after the execution of a method. It indicates the changes made to the state of the world by invoking the method.

In our work on rule-based adaptive and intelligent systems, the method to be executed is the *action* part of the *event-condition-action* rule:

events: if (conditions) then action ensure goal

The *condition* defines a pre-condition on the *events* for the *action* of this rule to be carried out. Note that even though the condition of an event-condition-action rule is true, this does not mean that the corresponding adaptation will always be executed. The predicate states a pre-condition, but no obligation to execute. This gives us the opportunity to choose among multiple matching rules that might achieve the same desired effect, but with a different outcome on the optimization criteria, i.e. saving energy and increasing the comfort levels.

The *goal* defines the desired post-condition after the rule was executed. Contrary to design-by-contract best practices, we do not and cannot guarantee that the post-condition will always hold after executing the action—i.e. that the

Fig. 2 Impact analysis of rule-based adaptive behavior of context-aware applications

Table 2 Description logic (DL) representation of DogOnt ontology concepts and other classes, object properties, individuals used in the smart home model

Classes	Object properties	Individuals
Building $\sqsubseteq$ BuildingEnvironment	isIn $\equiv$ contains ⁻	building : Building
Flat $\sqsubseteq$ BuildingEnvironment	$\exists$ contains Thing $\sqsubseteq$ BuildingEnvironment	temperature : TemperatureSensor
Room $\sqsubseteq$ BuildingEnvironment	$\top \sqsubseteq \forall$ contains (BuildingEnvironment $\sqcup$ HousePlants)	flat : Flat
LivingRoom $\sqsubseteq$ Room	$\top \sqsubseteq \leq 1$ state	livingroom : LivingRoom
Device $\sqsubseteq$ BuildingThing	$\exists$ state rdfs:Literal $\sqsubseteq$ Device	heating : HeatingSystem
HousePlants $\sqsubseteq$ Device	$\top \sqsubseteq \forall$ state xsd:string	contains(building, flat)
HVACSystem $\sqsubseteq$ HousePlants	$\top \sqsubseteq \leq 1$ value	contains(flat, livingroom)
HeatingSystem $\sqsubseteq$ HVACSystem	$\exists$ value rdfs:Literal $\sqsubseteq$ Sensor	isIn(heating, livingroom)
TemperatureSensor $\sqsubseteq$ HVACSystem	$\top \sqsubseteq \forall$ value xsd:int	isIn(temperature, livingroom)
TemperatureSensor $\sqsubseteq$ Sensor		

effect of a rule reaches the intended goal—due to the presence of interfering rules and components. Indeed, errors and inconsistencies cannot be avoided for systems with humans-in-the-loop, because people definitely do not always perform as designed (as envisioned by the developer of the system).

3.4 Semantic specification of rule-based behavior

As mentioned earlier, we model the rule-based adaptive behavior in a formal representation using the RL profile of the OWL 2 ontology language [16,22]. As the smart home environment itself is also semantically described with OWL 2

ontologies, we greatly simplify the interoperability between (1) the design and (2) the adaptive behavior of an intelligent environment. Furthermore, the semantic specification allows us to:

1. Separate domain knowledge about the environment from the rule-based automation behavior.
2. Use a single description logic reasoning engine to analyze rule interactions and the impact of change.

Table 2 depicts a subset of DogOnt concepts and other classes, object properties and individuals we used to model the smart home automation. These concepts represent a temperature sensor driving the heating system in the living room

of a flat. We will use the first two rules of the smart home use case (see Listing 1) to illustrate how the adaptation rules are semantically represented with the SWRL syntax. The red color represents primitive data values, the green color represents specific individuals by name, whereas parameters starting with a question mark (?) represent unbounded variables. The color blue is used for standard SWRL functions:

R1: value(temperature, ?v) ^{swrlb:lessThan}(?v, 19)
-> state(heating, "on"^^xsd:string)

R2: value(temperature, ?v) ^{swrlb:greaterThan}(?v, 21)
-> state(heating, "off"^^xsd:string)

```

1  <rdf:Description rdf:about="http://davy.preuveneers.be/ontologies/2015/12/cia#v">
2    <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#Variable"/>
3  </rdf:Description>
4  <rdf:Description>
5    <swrla:isRuleEnabled rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">true</swrla:isRuleEnabled>
6    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"></rdfs:comment>
7    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string">R1</rdfs:label>
8    <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#Imp"/>
9    <swrl:body>
10      <rdf:Description>
11        <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#AtomList"/>
12        <rdf:first>
13          <rdf:Description>
14            <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#DatavaluedPropertyAtom"/>
15            <swrl:propertyPredicate rdf:resource="http://davy.preuveneers.be/ontologies/2015/12/cia#value"/>
16            <swrl:argument1 rdf:resource="http://davy.preuveneers.be/ontologies/2015/12/cia#temperature"/>
17            <swrl:argument2 rdf:resource="http://davy.preuveneers.be/ontologies/2015/12/cia#v"/>
18          </rdf:Description>
19        </rdf:first>
20        <rdf:rest>
21          <rdf:Description>
22            <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#AtomList"/>
23            <rdf:first>
24              <rdf:Description>
25                <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#BuiltinAtom"/>
26                <swrl:builtin rdf:resource="http://www.w3.org/2003/11/swrlb#lessThan"/>
27                <swrl:arguments>
28                  <rdf:Description>
29                    <rdf:type rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#List"/>
30                    <rdf:first rdf:resource="http://davy.preuveneers.be/ontologies/2015/12/cia#v"/>
31                    <rdf:rest>
32                      <rdf:Description>
33                        <rdf:type rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#List"/>
34                        <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#int">19</rdf:first>
35                        <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
36                      </rdf:Description>
37                    </rdf:rest>
38                  </rdf:Description>
39                </swrl:arguments>
40              </rdf:Description>
41            </rdf:first>
42            <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
43          </rdf:Description>
44        </rdf:rest>
45      </rdf:Description>
46    </swrl:body>
47    <swrl:head>
48      <rdf:Description>
49        <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#AtomList"/>
50        <rdf:first>
51          <rdf:Description>
52            <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#DatavaluedPropertyAtom"/>
53            <swrl:propertyPredicate rdf:resource="http://davy.preuveneers.be/ontologies/2015/12/cia#state"/>
54            <swrl:argument1 rdf:resource="http://davy.preuveneers.be/ontologies/2015/12/cia#heating"/>
55            <swrl:argument2 rdf:datatype="http://www.w3.org/2001/XMLSchema#string">on</swrl:argument2>
56          </rdf:Description>
57        </rdf:first>
58        <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
59      </rdf:Description>
60    </swrl:head>
61  </rdf:Description>

```

Listing 3 RDF/XML-based SWRL representation of the OWL 2 RL smart home rule R1

The above two rules change the state of the heating system depending on the value of the temperature sensor. The above rules are triggered when the named individual *temperature*, i.e. a specific temperature sensor, passes one of both thresholds.

In Listing 3, we depict how the OWL 2 RL smart home adaptation rule R1 is represented in RDF/XML-based SWRL format. Note that this translation is carried out by our framework and not manually specified by the developer. This RDF/XML representation may help with the integration of other tools only supporting this format. This way, we can also support interoperability and integration with other OWL 2 ontologies that are used to formally and semantically describe the intelligent environment.

We can generalize rules *R1* and *R2* by considering any temperature sensor in the living room as follows:

R3: TemperatureSensor(?s) ^ contains(livingroom, ?s) ^ value(?s, ?v) ^ **swrlb:lessThan**(?v, 19)
 -> state(heating, "on"^^xsd:string)

R4: TemperatureSensor(?s) ^ contains(livingroom, ?s) ^ value(?s, ?v) ^ **swrlb:greaterThan**(?v, 21)
 -> state(heating, "off"^^xsd:string)

In the above rules *R3* and *R4*, we require that the living room *contains* the temperature sensor. This means that any temperature sensor matching this location condition may trigger one of both rules. In our smart home model (see Table 2), we declared that:

isIn(temperature, livingroom)

This means that the *temperature* sensor *is in* the *livingroom*. For these two rules *R3* and *R4*, our framework automatically leverages the following object property axiom defined in the smart home model:

isIn $\equiv$ contains⁻

This axiom means that the first object property is the inverse of the latter object property and vice versa.

The attentive reader will have noticed that the adaptation rules *R3* and *R4* may cause unexpected behavior. For example, if multiple sensors satisfy the location condition but have inconsistent temperature values (e.g. a first sensor returns 18 °C and the second returns 22 °C), then it is not clear whether the heating system will be in the *on* or *off* state. We elaborate further on these aspects in the following subsections.

3.5 Temporal and computational aspects of rules

In the smart home scenario, one adaptation rule turns off the lights when there has been no motion detected for at least 5 min.

if (motion = false) _{$\Delta t=5min$} then lights = off

At design time, this rule is treated in the same way as all the other rules. At runtime, we add temporal rules to reset a timer.

Rather than handling temporal aspects in the application logic, we formally represent time by leveraging the *SWRL Temporal Ontology* [28] and its built-in functions for representing and querying temporal information. The following SWRL rule is triggered if the motion sensor has detected someone's presence:

motion(detector, true) ^ **temporal:add**(?d, "now"^^string, 5, **temporal:Minutes**) -> **sqwrl:select**(?d)

This rule is in fact an SQWRL query. SQWRL is a semantic query-enhanced web rule language based on the SWRL language to query OWL ontologies, and here, we use it to compute and select the new deadline *?d* being 5 min from now. We use this new deadline *?d* to constrain the above rule using the *notBefore* built-in of the SWRL temporal ontology:

motion(detector, false) ^ time(detector, ?t) ^ **notBefore**(?t, ?d) -> state(lights, "off"^^xsd:string)

Without going into further details, we leverage similar built-in functions to compute the energy consumption and comfort levels of the smart office scenario whenever the state of an appliance has been changed.

For more details on the available built-ins, we refer to section 8 of the *SWRL: A Semantic Web Rule Language Combining OWL and RuleML* specification.³

3.6 Analyzing rule interactions and inconsistencies

In the following subsections, we will give examples of inconsistencies that can emerge due to dependencies or conflicts among rules.

-
- 1 temperature01: TemperatureSensor
 - 2 temperature02: TemperatureSensor
 - 3 livingroom: LivingRoom
 - 4 isIn(temperature01, livingroom)
 - 5 isIn(temperature02, livingroom)
 - 6 value(temperature01, 18)
 - 7 value(temperature02, 22)
-

Listing 4 Conflicting context conditions and actions

3.6.1 Conflicting context conditions and actions

When multiple rules affect the same system component, inconsistencies in the contextual triggers may cause a non-

³ <http://www.w3.org/Submission/SWRL/>.

deterministic component state. These inconsistencies are often due to unbounded and/or overlapping contextual circumstances. Consider the example in Listing 4, the previous adaptation rules would set the heating to both the *on* and *off* state.

$\top \sqsubseteq \leq 1 \text{ state}$

In OWL functional syntax, this is the same as asserting that *state* is a functional data property:

```

1 OWLOntologyManager om = OWLManager.createOWLOntologyManager();
2 OWLDataFactory df = om.getOWLDataFactory();
3 OWLOntology o = om.loadOntologyFromOntologyDocument(new File("cia.owl"));
4
5 ReasonerFactory f = new ReasonerFactory() {
6     protected OWLReasoner createHermitOWLReasoner(Configuration configuration, OWLOntology ontology) {
7         configuration.throwInconsistentOntologyException = false;
8         return new Reasoner(configuration, ontology);
9     }
10 };
11
12 OWLReasoner r = f.createReasoner(o);
13 if (!r.isConsistent()) {
14     OWLObjectRenderer renderer = new DLSyntaxObjectRenderer();
15     BlackBoxExplanation exp = new BlackBoxExplanation(o, f, r);
16     HSTExplanationGenerator multExplainer = new HSTExplanationGenerator(exp);
17     Set<Set<OWLAxiom>> explanations = multExplainer.getExplanations(df.getOWLThing());
18     System.out.println("Axioms causing the inconsistency: ");
19     for (Set<OWLAxiom> explanation : explanations) {
20         for (OWLAxiom causingAxiom : explanation) {
21             System.out.println(" " + renderer.render(causingAxiom));
22         }
23     }
24 }

```

Listing 5 Explaining OWL ontology inconsistencies with Hermit

Note however that this by itself does not cause the ontology to be inconsistent. OWL uses an *open world assumption*, which means that an ontology may not contain all information and some information may be unknown. If an ontology does not enforce unique name assumption (UNA), then it does not implicitly treat all individuals as distinct. In these cases, it is possible for two concepts or individuals with different names to be inferred to be equivalent.

In our example with rules *R3* and *R4*, it means that named individuals with different names—such as the *temperature01* and *temperature02*—may represent the same individual unless it is explicitly stated that they are different. That is why for each temperature sensor, we state that the named individuals are different (in OWL functional syntax rendering):

DifferentIndividuals(:temperature01 :temperature02)

An individual can have multiple instances of a datum or object property. However, in Table 2 we asserted that the *state* data property can have only one unique value for each individual:

FunctionalDataProperty(:state)

A similar axiom was asserting for the *value* data property. However, for the asserted axioms in Listing 4, the adaptation rules *R3* and *R4* would infer the following two axioms for the named individual *heating*:

DataPropertyAssertion(:state:heating"off"^^xsd:string)
DataPropertyAssertion(:state:heating"on"^^xsd:string)

These results violate the *FunctionalDataProperty* axiom of the *state* data property. In the following subsection, we will discuss how to analyze the (theoretical) existence of individuals and properties that cause inconsistencies without having to manually assert the value axioms of the different temperature sensors.

Note that other rules with satisfiable antecedents P_i (i.e. the context conditions) but conflicting consequences Q_j (i.e. the smart home or office automation actions) can lead to similar ontology inconsistencies. This means that several conflicting adaptations Q_j are triggered at the same time:

```

if  $P_1$  then  $Q_1$ 
if  $P_2$  then  $Q_2$ 
:
if  $P_n$  then  $Q_n$ 

```

For example, a simple rule that turns off the heating when a window sensor detects that the window has been opened will result in the same inconsistent state for the heating system if it is cold inside (triggering rule R_3).

```

state(window, "open"^^xsd:string)
-> state(heating, "off"^^xsd:string)

```

With an ontology reasoner like HermiT [15], we can ask for an explanation when ontology inconsistencies are detected. In Listing 5, we provide example Java code for finding the axioms that give rise to the ontology inconsistency. For the given example in Listing 4 and rules R_3 and R_4 , the above code produces the following output showing the same description logic axioms we mentioned to earlier:

```

Axioms causing the inconsistency:
   $\top \sqsubseteq \leq 1$  state
  state(heating, on)
  state(heating, off)

```

Discussion In this example, we used two temperature sensors in the same room with two values that were significantly different so that each triggered two different conflicting adaptation rules leading to an inconsistent state of the heating system. One might argue that this is an unrealistic scenario. However, a single broken sensor that returns erroneous temperature values is a simple example that gives rise to such a scenario.

3.6.2 Automated causality detection of inconsistencies

In Listing 4, we manually defined the values of two temperature sensors to trigger the adaptation rules with conflicting consequences. However, defining multiple test scenarios with different input values is very cumbersome. Without sufficient coverage, they may not lead to finding all the inconsistencies. We are rather interested in analyzing whether there are any facts (i.e. asserted axioms on named individuals of temperature sensors) that would trigger rule antecedents P_i with inconsistent consequences Q_j (i.e. causing the state of the heating to be both *on* and *off* at the same time). To check whether such individuals may exist, we automatically combine the antecedents P_i of these rules in an SQWRL query:

```

R5: TemperatureSensor(?s) ^ contains(livingroom, ?s)
^ TemperatureSensor(?t) ^ contains(livingroom, ?t)
differentFrom(?s, ?t) -> sqwrl:select(?s, ?t)

```

We can simplify the conjunction of the antecedents of both adaptation rules due to the following axioms and assumptions:

- The *value* property is a functional data property.
- Each rule has an antecedent that is feasible.

As each temperature sensor can have only one (unique) value that may trigger one of the rules, we can eliminate the *swrlb:greaterThan* and *swrlb:lessThan*, as well as the *value(?s, ?v)* and *value(?t, ?w)* conditions. However, due to the open world assumption the individuals s and t may refer to the same temperature sensor. That is why we have to assert that such individuals s and t must be different. The query in rule R_5 results in the following outcome:

```

Query results [2]:
1) s = :temperature02, t = :temperature01
2) s = :temperature01, t = :temperature02

```

For Listing 4, the query results in two equivalent results, meaning that the individuals *temperature01* and *temperature02* can have (at least theoretically) values that cause an inconsistent heating state.

Discussion If our scenario had just one temperature sensor in the living room, the SQWRL query would not produce any results, meaning that there is no risk of inconsistency for the rules R_3 and R_4 . However, this may change when the user decides to install additional appliances and sensors in the intelligent environment. This example shows that without having to change the rules, our change impact analysis framework can also detect inconsistencies due to additional sources of context information that may influence the behavior of the intelligent environment.

3.6.3 Modeling the impact on the environment

Consistent behavior of the individual components does not guarantee reliable functioning of the home automation system as a whole. For example, the temperature in a room can be affected by the heating, the air conditioning and the state of the curtains. Obviously, turning on the heating and the air conditioning at the same time is not a good idea. However, these implicit dependencies are nowhere modeled in the above rules. In our approach, these hidden dependencies are made explicit by exploiting both the semantic model of the intelligent environment and the adaptation rules.

Thus far, we used rules to describe the state of the actuators (e.g. the heating system) in terms of sensory input (e.g. temperature sensors). We pursue the same approach to describe the causality between the state of actuators and the *expected impact* on the environment. For example, if the heating in a room is turned on, we expect the temperature in that room not to decline:

```

HeatingSystem(?h) ^ isIn(?h, ?l) ^ state(?h, "on")
^ env(?l, ?e) -> impact(?e, tempNonDeclining)

```

If a room has an open window, and it is colder outside, we expect the temperature inside to drop:

```
Window(?w) ^ isIn(?w, ?l) ^ state(?w, "open")
^ TemperatureSensor(?s1) ^ TemperatureSensor(?s2)
^ isIn(?s1, ?l) ^ value(?s1, ?v1) ^ isIn(?s2, outside)
^ value(?s2, ?v2) ^ swrlb:lessThan(?v2, ?v1)
^ env(?l, ?e) -> impact(?e, tempDeclining)
```

An equivalent rule exists to represent the impact of an open window when it is warmer outside:

```
Window(?w) ^ isIn(?w, ?l) ^ state(?w, "open")
^ TemperatureSensor(?s1) ^ TemperatureSensor(?s2)
^ isIn(?s1, ?l) ^ value(?s1, ?v1) ^ isIn(?s2, outside)
^ value(?s2, ?v2) ^ swrlb:lessThan(?v1, ?v2)
^ env(?l, ?e) -> impact(?e, tempRising)
```

The temperature impact is represented with an object property that characterizes a monotonic environmental influence (e.g. **tempDeclining**, **tempNonRising**, **tempSteady**, **tempNonDeclining**, **tempRising**) given a particular state of certain sensors and actuators. An intelligent system needs to avoid a configuration where both the **tempDeclining** and **tempRising** impact properties are set at the same. Other temperature impact combinations for a given environment do not characterize an invalid global state, but the current (steady) state may not be ideal or optimal.

The goal of the above examples is to illustrate how the expected impact on the environment is represented given the state of certain sensors and appliances. Furthermore, an environment can be impacted in various ways (e.g. the temperature, humidity or brightness in a given room) and that is why the impact is not considered to be a functional object property as was the case for the *state* data property discussed earlier. That is why we cannot declare for the *impact* object property the following axiom, which would mean an environment individual would only have one object property:

FunctionalObjectProperty(:impact)

By modeling the impact on the environment given the state of certain sensors and actuators, we can detect inappropriate behavior or failing hardware at runtime (e.g. an erroneous sensor reading). In rules R1 and R2, we assume that if the temperature is below 19 °C, the heating will be turned on. If there is a failure in the actuator, the temperature will not increase. Such failure can be detected with the above rule that will monitor the expected impact at runtime.

3.7 Modeling of incompatible change impacts

With the aforementioned approach to represent the impact on an intelligent environment, we can also explicitly model

invalid global states of the system or the environment—independent of the context-driven adaptation rules—to guarantee that certain inconsistent behavior does not occur or is detected early at runtime. We basically define which conditions lead to undesirable configurations by analyzing the impact of the state of certain actuators and appliances.

For example, imagine we want to avoid a situation where a window is open and where the heating in the same room is turned on. We could then declare this invalidate state with the following rule:

```
HeatingSystem(?h) ^ isIn(?h, ?l) ^ state(?h, "on")
^ Window(?w) ^ isIn(?w, ?l) ^ state(?w, "open")
-> invalidatestate(?h)
```

We could have another rule to avoid a similar energy wasting situation where the window is open and the air conditioning is turned on:

```
AirConditioning(?a) ^ isIn(?a, ?l) ^ state(?a, "on")
^ Window(?w) ^ isIn(?w, ?l) ^ state(?w, "open")
-> invalidatestate(?a)
```

In fact, next to the above incompatibility rules we can add yet another one to declare that both the air conditioning and the heating should not be turned on at the same time in the same room:

```
HeatingSystem(?h) ^ isIn(?h, ?l) ^ state(?h, "on")
^ AirConditioning(?a) ^ isIn(?a, ?l) ^ state(?a, "on")
-> invalidatestate(?a)
```

It is clear that directly listing all combinations leading to inconsistent global states is a very tedious and cumbersome way of working. Instead, we use the monotonic impact object properties of before to indirectly declare the situations which should be avoided. OWL 2 offers a construct called *DisjointObjectProperties*, which allows to assert that several object properties are pairwise incompatible:

DisjointObjectProperties(:hasParent :hasSpouse)

Unfortunately, this ontology construct is only supported by SROIQ reasoners, and that is why we resort to the following SQWRL query:

```
Environment(?e) ^ impact(?e, ?i) -> sqwrl:select(?e, ?i)
```

after which we programmatically check for incompatible impact property values ?i that would violate the monotonic change of an environmental property (e.g. a combination of a *rising* and *declining* temperature impact property).

Fig. 3 Using Protégé 5 (beta 21) to formally and semantically represent an intelligent environment, including the rule-based adaptive behavior using the SWRL and SQWRL plugins

4 Implementation of design-time tool support and runtime middleware

In this section, we will discuss implementation details of our change impact analysis framework, including the Protégé-based design-time support as well middleware building blocks. As mentioned earlier, both are built on top of the OWL 2 RL ontology specification.

4.1 Design-time support with Protégé 5

To support the developer with change impact analysis at design time, we leverage the Protégé⁴ tool (see Fig. 3) for the formal semantic representation of all domain and design aspects of the intelligent environment, including all software artifacts of the application. The advantage of semantically modeling an application with ontologies—compared to UML—is that implicit or indirect relationships between the design elements in the architecture can be automatically inferred. Reflexive, transitive or symmetric properties are powerful characteristics in the semantic definition of a dependency relationship when analyzing the propagation of changes.

We semantically instantiate all domain knowledge about the intelligent environment leveraging standard or widely used ontologies if possible. Figure 4 depicts a Protégé tool visu-

Fig. 4 Semantic modeling and instantiation of the smart home environment with the DogOnt ontology

alization of a relevant subset of the smart home represented with the DogOnt ontology. We also model the underlying application logic as a composition of software components with Protégé (as shown in Fig. 5) to formally represent:

⁴ <http://protege.stanford.edu/>.

Fig. 5 Formal modeling of component-based applications and dependencies with ontologies

1. Components and connectors within the application
2. Requirements, and components that fulfill them
3. Optional and mandatory context dependencies
4. Design assumptions and constraints

Each sensor and actuator in our intelligent environment are managed and wrapped by a software component. This way, each artifact has a unified interface to manage its life cycle (i.e. start or stop the component, inspect or change the current state, update the current configuration) and broadcast context events that are semantically enhanced with our context ontology [40] and communicated with our semantic publish/subscribe message protocol [35,36]. By modeling the knowledge we have about the intelligent system, including its design (components and connectors), we can reason about dependencies among design-time and runtime decisions, and infer whether a change means a component is modified (direct change), remains unaffected, or is affected (indirect change) because one of its context dependencies has changed.

To represent changes at design time, we version each ontology model instantiation as a developer would do with normal software code. Protégé offers a *OWL Difference Plugin* for visual inspection of ontology changes. During the design, we validate whether the modified ontology is still consistent, and use the HermiT 1.3.8.413 reasoner in Protégé 5 (beta 21) to infer whether some entities in the ontology become semantically equivalent with the *owl:Nothing* OWL class identifier. The predefined class extension of *owl:Thing*

is the set of all class instances, whereas *owl:Nothing* is always the empty set. If an entity is equivalent with *nothing*, it basically means that it does not exist. For optional components, it means the component is gone. For mandatory ones, we can conclude there must be a consistency issue.

When a design *assumption* or *trade-off* formally states that an artifact (e.g. a component or context attribute) should not change, but the meta-properties of the relationships infer an impact, unexpected side effects can be detected. Whether this is a big concern, is something that should be investigated by the software architect or developer. When a violation is detected against a design *constraint*, then the proposed change would affect the consistency of the application.

The rule-based adaptive behavior is also modeled with Protégé using the SWRLTab editor plugin. Figure 6 illustrates the two temperature rules used to automate the heating system. The plugin leverages the Drools 6.2 rule engine⁵ from JBoss to implement the OWL 2 RL ontology profile and semantic reasoning backend. More information about the underlying technologies can be found at the GitHub Wiki page of the project.⁶

4.2 Runtime support

Most of the change impact analysis tools focus on the design phase, but offer little support at runtime. This is problematic as certain undesirable impacts of unexpected changes can only be detected at runtime:

1. Inconsistent sensor values
2. Adaptations that do not reach desired goals
3. Conflicts due to end-user customizations

Our runtime framework for change impact analysis leverages the same OWL ontologies created at design time to formally and semantically represent the intelligent environment in which the application operates and the application-specific rule-based adaptive behavior. Reuse of the developer's ontology models and reasoning infrastructure greatly facilitates the design-time to runtime transition of adaptive applications in an intelligent environment.

To ensure consistency before and after deployment, the same Drools 6.2-based OWL 2 RL reasoning engine backend that Protégé leverages during the design phase is also incorporated into our framework (i.e. leaving out the user interface dependencies) to recognize the above concerns and analyze the impact of runtime changes.

This software component in charge of managing the change impact analysis at runtime complements our previous research on component-based mobile middleware for

⁵ <http://www.drools.org/>.

⁶ <https://github.com/protegeproject/swrlapi/wiki>.

context-aware adaptive applications [31,37]. This context middleware not only takes care of collecting data from sensors, but also reconfigures the distributed middleware at runtime. We use a learning [6] approach to find the best deployment given the circumstances in case multiple trade-offs and optimization constraints exist. The software component is also embedded in our SAMURAI [39] framework that was developed with the particular goal of handling the growing velocity, volume and variety of data produced in large-scale intelligent environments.

ing room. These sensors produced temperature values of 18 and 22 °C respectively, triggering two adaptation rules turned the heating on and off at the same time, hence giving rise to inconsistent behavior. While at design time, we could already analyze the theoretical possibility of such an error occurring, the middleware will now detect the actual presence of inconsistent sensor values. Listing 6 shows some of the technical implementation details of how these data values are added to the ontology at runtime. The Java code for the change impact analysis itself has been documented before in Listing 5.

```

1 OWLOntologyManager om = OWLManager.createOWLOntologyManager();
2 OWLDataFactory df = om.getOWLDataFactory();
3 OWLOntology o = om.loadOntologyFromOntologyDocument(new File("cia.owl"));
4
5 IRI i = IRI.create("http://davy.preuveneers.be/ontologies/2015/12/cia");
6 OWLDatatype xsdInt = df.getOWLDatatype(IRI.create("http://www.w3.org/2001/XMLSchema#int"));
7
8 // Add a temperature value instance to the ontology instance
9 OWLIndividual t = df.getOWLNamedIndividual(IRI.create(i + "#temperature01"));
10 OWLDataProperty v = df.getOWLDataProperty(IRI.create(i + "#value"));
11 OWLLiteral d = df.getOWLLiteral("18", xsdInt);
12 om.applyChange(new AddAxiom(o, df.getOWLDataPropertyAssertionAxiom(v, t, d)));
13
14 // Create a SWRL rule engine using the SWRLAPI
15 SWRLRuleEngine re = SWRLAPIFactory.createSWRLRuleEngine(o);
16
17 // Add two SWRL adaptation rules and fire the rule engine
18 re.createSWRLRule("R02.1", "TemperatureSensor(?s) ^ isln(?s,livingroom) ^ value(?s,?v) ^ swrlb:lessThan(?v,19)
19   -> state(heating, \"on\"^^xsd:string)");
20 re.createSWRLRule("R02.2", "TemperatureSensor(?s) ^ isln(?s,livingroom) ^ value(?s,?v) ^ swrlb:greaterThan(?v,21)
21   -> state(heating, \"off\"^^xsd:string)");
22 re.infer();
23
24 // To check for consistency, see previous code snippet in Listing 4 (from lines 5 to 24)
25
26 // Create SQWRL query engine using the SWRLAPI
27 SQWRLQueryEngine qe = SWRLAPIFactory.createSQWRLQueryEngine(o);
28 SQWRLResult r = qe.runSQWRLQuery("Q02", "state(heating, ?s) -> sqwrl:select(?s)");
29 while (r.next()) {
30     System.out.println("heating: " + r.getLiteral("s").getString());
31 }

```

Listing 6 OWL 2 RL rule and query engine

4.2.1 Inconsistent sensor values

After the context middleware has loaded the ontologies and adaptation rules, it monitors the environment and updates the sensor values in the ontology. After each sensor update or at regular intervals, the middleware checks for contextual inconsistencies that would trigger conflicting adaptation rules in the environment.

For example, in Listing 4 we documented a smart home automation scenario with two temperature sensors in the liv-

4.2.2 Adaptations that do not reach desired goals

Adaptation rules are triggered to reach a certain goal. However, there is no guarantee that these goals are actually met at runtime. For example, imagine the heating system is turned on in the middle of the winter:

```

TemperatureSensor(?s) ^ contains(livingroom, ?s)
^ value(?s, ?v) ^ swrlb:lessThan(?v, 19)
-> state(heating, "on"^^xsd:string)

```

Fig. 6 Adaptation rules in OWL 2 RL with the SWRLTab rule editor plugin for Protégé 5

As a consequence, the next rule is triggered to represent the expected impact on the environment (i.e. the living room is heating up or remains in a steady state):

```

HeatingSystem(?h) ^ isln(?h, ?l) ^ state(?h, "on")
^ env(?l, ?e) -> impact(?e, tempNonDeclining)

```

Only if the temperature drops, an error will be detected. However, if we also had an open window in the ontology, then turning on the heating will cause a consistency failure due to incompatible impacts on the environment:

```

Window(?w) ^ isln(?w, ?l) ^ state(?w, "open")
^ TemperatureSensor(?s1) ^ TemperatureSensor(?s2)
^ isln(?s1, ?l) ^ value(?s1, ?v1) ^ isln(?s2, outside)
^ value(?s2, ?v2) ^ swrlb:lessThan(?v2, ?v1)
^ env(?l, ?e) -> impact(?e, tempDeclining)

```

Again, the above theoretical problems could already be elicited during the change impact analysis at design time. It is clear that many unlikely situations may exist that will give rise to such context-dependent errors. That is why it is better to give the developer the opportunity to model these reliability concerns, but have the software in place to detect them at runtime so that the end-user can intervene.

4.2.3 Conflicts due to end-user customizations and reconfigurations

The end-user can also modify the adaptation rules by enforcing certain user preferences. This means that we not only support adding or removing rules, but also changing existing ones. Although we do not expect end-users to use Protégé to customize the rule set (but rather an application-specific front-end that maps these preferences to OWL 2 RL rules), the change impact analysis that happens at this stage is the same as procedure carried out for the developer at the design phase. However, to ensure that end-users do not break some of the core rule-based adaptation rules in the system, the developer can choose to implement these rules directly in the application or middleware, as illustrating in Listing 6.

A more challenging scenario is an intelligent environment with multiple users—such as in the smart office scenario—where each individual has different preferences for feeling comfortable. These preferences may trigger possibly conflicting adaptation rules for centrally managed HVAC systems, lighting, etc. There is no practical way to solve such problems in a general way. Our current solution allows for rule inconsistencies to occur as soft constraints as long as hard constraints are not violated (i.e. operational monetary cost should not cross a certain threshold).

5 Evaluation

We evaluated our framework for change impact analysis both at design time and at runtime. The evaluation is based on the implementation of the two use cases of Sect. 3. The complexity of the ontology-based representation of adaptive behavior in these intelligent environments is largely due to the DogOnt (see Table 1) and SWRL ontology dependencies. Each use case is respectively driven by 35 and 28 SWRL rules and SQWRL queries that are responsible for modeling:

- The context-dependent adaptation rules
- The impact of changes on the environment

The first use case represents a smart home environment (in this case an apartment) with 9 rooms, with rules driving the HVAC system, lighting, curtains, and switching off/on various home appliances and security systems through wireless remote control power switches. With 16 rooms, the smart office environment is larger in size than the smart home environment, but the rooms are much more homogeneous. With a meeting room, a coffee room and the rest being working spaces, several of the above rules only needed to be defined once, but apply to the 14 working spaces. The result is fewer rules overall for the smart office use case compared to the smart home space.

5.1 Qualitative evaluation

From a developer's perspective at design time, our framework was able to identify several scenarios that could lead to inconsistent or undesirable behavior. We previously outlined the theoretical possibility of inconsistencies of turning on the heating with an open window. These problems arise from the fact that windows are manually controlled and not under the supervision of the home automation system. To reduce the amount of warnings to deal with, we semantically annotate for each appliance whether its state is automated or manipulated by humans (or both). When components are under human control, we label undesirable impacts involving these components as warnings for the user in case such a situation emerges at runtime.

Our framework was able to identify scenarios that were not so trivial to detect without tool support. An example in the smart home environment involved adding outdoor sensors to an existing configuration. To represent the impact on the luminosity inside, we added the following adaptation rule to the system:

```
Curtain(?c) ^ isIn(?c, ?l) ^ state(?c, "closed")
^ LuminositySensor(?s1) ^ LuminositySensor(?s2)
^ isIn(?s1, ?l) ^ value(?s1, ?v1) ^ isIn(?s2, outside)
^ value(?s2, ?v2) ^ swrlb:lessThan(?v1, ?v2)
```

```
-> state(?c, "open")
```

We added a similar adaptation rule to turn off the lights if the luminosity outside is higher than 500 lux (a sunrise is around 400 lux):

```
Light(?b) ^ state(?b, "on") ^ LuminositySensor(?s)
^ isIn(?s, outside) ^ value(?s, ?v)
^ swrlb:lessThan(500, ?v) -> state(?b, "off")
```

By adding the above rules that relied on outdoor sensors, the change impact analysis identified a conflict with timer-based adaptation rules that open and close the curtains respectively at 8:00 AM and 9:00 PM. In the situation that the luminosity outside after 9:00 PM is higher than 500 lux, the two rules would cause conflicting behavior for the curtains.

To represent the impact on the environment, we stated in a similar way as for the temperature example that the luminosity inside would increase if the curtains are opened on a sunny day:

```
Curtain(?c) ^ isIn(?c, ?l) ^ state(?c, "open")
^ LuminositySensor(?s1) ^ isIn(?s1, outside)
^ value(?s, ?v) ^ swrlb:lessThan(500, ?v)
^ env(?l, ?e) -> impact(?e, luminosityRising)
```

However, to represent realistic behavior we also added a change impact rule on the environment that in a particular situation of direct sunlight (luminosity above 30000 lux), opening the curtains will lead to an increasing temperature inside:

```
Curtain(?c) ^ isIn(?c, ?l) ^ state(?c, "open")
^ LuminositySensor(?s1) ^ isIn(?s1, outside)
^ value(?s, ?v) ^ swrlb:lessThan(30000, ?v)
^ env(?l, ?e) -> impact(?e, tempRising)
```

The conflict that emerged from the above situation is that changing the state of a particular appliance (in this case the curtains) had a positive impact on the luminosity but an undesirable side effect on the temperature and interfere with adaptation strategies to reduce the temperature inside (closing the curtains on a very sunny day rather than turning on the air conditioning).

The strategy our framework pursues is that adaptation rules are not activated unless all consequences are in line with the global objectives. A more fine-grained approach would take the significance of the positive and negative impacts into account such that a small undesirable side effect can be considered acceptable if the other effects significantly outweigh the drawbacks. Unfortunately, semantic ontology reasoners are very good at analyzing the impact of changes, but these tools cannot *measure* the impact or even the likelihood of an

impact. Our approach gives a fairly black and white picture. It does not allow us to prioritize changes with respect to their impact.

5.2 Performance evaluation at runtime

In this section, we briefly discuss the computational complexity and performance overhead of the change impact analysis middleware component of our framework. The middleware component is responsible for the following activities:

1. Download the ontologies that describe the intelligent environment and the adaptation behavior.
2. Reason semantically on classes and properties to make implicit relationships explicit.
3. Evaluate the adaptation rules when relevant context events are being gathered from various sensors.
4. Monitor the environment to verify the expected impact of adaptation rules.
5. Manage updates in the adaptation rules and report conflicts after end-user customizations.

Although our context middleware supports distributed deployment configurations, the component in charge of change impact analysis is centralized because the underlying OWL 2 RL ontology reasoner cannot be scaled out easily over multiple nodes. This means that steps 1 and 2 take place once when this component is activated. Steps 3 and 4 are carried out continuously whenever new sensor data are collected by the underlying context middleware. Step 5 is triggered only when the user has updated the adaptation rules (e.g. after installing new sensors or actuators).

For the performance evaluation, we deployed our change impact analysis middleware on top of two computer systems:

- Dell Optiplex 7010 desktop with 16GB of memory and an Intel Core i7-3770 quad-core 3.40 GHz CPU.
- ODROID-U3 miniboard with 2GB of memory and a Cortex-A9 quad-core 1.7GHz CPU

Both platforms run Ubuntu 15.10 and a recent JDK 8u71 Java virtual machine for respectively the x64 and ARM platforms. The ODROID miniboard (see Fig. 7) was included in our experiments because such a platform is more cost effective and easier to integrate in an intelligent environment.

To ensure the activation of adaptation rules, we simulated the environment with 37 sensors located in different semantic rules, and the adaptive behavior being captured by 13 different rules, some of which are reused across rooms. The Java Virtual Machine on the ODROID was assigned 1 GB of heap space, allowing for intermediate reasoning results to be cached across updates. Under these circumstances, the rea-

Fig. 7 ODROID-U3 mini-board with wireless connectivity (comparable in size to Raspberry Pi though more powerful)

soning time to evaluate the semantic rules varied between 2 and 9 s.

Figure 8 shows performance results on both platforms. The high CPU load in the beginning is due to the ontology loading and semantic classification of classes and properties. After a couple of seconds, this classification (i.e. steps 1 and 2) has completed and the CPU load remains more or less stable. At this stage, steps 3 and 4 are carried out. As expected, the CPU load is higher on the miniboard compared to the desktop system. Still, a 25 % average CPU load (across all 4 CPU cores) is very reasonable. However, the numbers discussed here are specific for our use case in which we capped the environmental sensors that offer continuous updates (e.g. the temperature sensors) to maximum 2 value updates per minute. The reason for this was not the computational constraints of the ODROID board, but rather energy constraints of wireless sensors and the fact that temperature values do not change drastically in a room in less than a minute. As a result, the experiments show that it is feasible to deploy our solution on miniboards that are more practical for deployments in the real world outside the lab environment.

5.3 Comparison with the SPIN model checker

The SPIN [18] model checker has been used before [3,4,38] to improve the reliability of intelligent environment. By no means do we intend to replace this tool, but there are certain advantages of our solution compared to this model checker:

- Modeling the behavior of an intelligent environment in Promela (the modeling language of SPIN) is not straightforward. Our solution can benefit from the fact

Fig. 8 Average CPU load on Dell Optiplex system and ODROID-U3 mini-board

- that ontologies are used more often, and even allow for geospatial and temporal reasoning.
- SPIN is mainly useful for formal verification at design time. Our experiments have shown that harmful situations may theoretically exist (due to inappropriate human behavior), but may be very unlikely at runtime (e.g. failing sensors). Our solution and building blocks can be used both at design time (due to integration with Protégé 5) and at runtime (in the context-aware adaptation middleware)
 - Our solution allows customization of the adaptation rules at runtime. While end-users will not learn how SPIN works, our framework offers APIs for changing the rules after deployment. However, even in our solution we assume that the developer will offer intuitive user interfaces to indirectly modify the OWL 2 RL rules.

From a developer's point of view, there is still room for improvement to further align and optimize the co-design (i.e. the knowledge engineering and software engineering phases) of intelligent environment applications. However, these engineering aspects are beyond the scope of this work.

5.4 Discussion and validity threats

The number of scenarios in which our change impact analysis framework was tested and evaluated remains limited. The above qualitative and quantitative results are mainly based on our practical experience with the SPIN model checker and using our framework in the two case studies we outlined in the previous sections. Additional experiments with other use cases are needed to obtain statistically significant results to be able to generalize our findings under different circumstances and workloads on different platforms.

Some of the software building blocks and libraries we used in our framework for the evaluation of the semantic rules are still under development. The performance of the OWL 2 RL reasoner could be further improved by making use of the latest version of the underlying Drools rule engine. Additionally, the Drools rule engine does not scale horizontally, which means there are limitations from a scalability perspective in terms of the number of sensors and adaptation rules that can be handled on a single node. The amount of semantic data that can be handled in near real time is constrained by the available processing power and amount of memory on the node hosting our framework. However, if the changes and their impact can be contained per location, we can use location-based sharding techniques to distribute at runtime the change impact analysis workload and deal with the increased demands caused by the continuous data growth. Nonetheless, this can lead to interesting coordination issues where conflicting objectives across neighboring environments may need to be handled in a hierarchical manner.

6 Conclusion

In this work, we proposed a framework for semantic analysis and verification of context-driven adaptive applications that are deployed in intelligent environments. Our contribution focuses on improving the understanding and anticipating the impact of changes, and this is from a developer's point of view at design time as well as from a system's perspective at runtime after deployment. Our change impact analysis methodology is founded in the formal semantic modeling of intelligent environments and rule-based application behavior with the goal to avoid undesired side effects early on in the design phase. The middleware allows for detect-

ing problems that are due to the mismatches between the developer's assumptions of what are intelligent situation-dependent behavior and the user's expectations. We validated our solution on non-trivial smart home and office scenarios, and demonstrated how our framework helps increase trust by anticipating change implications upfront at design time and by minimizing the occurrence of undesired side effects at runtime. Additionally, we evaluated the performance of the system on contemporary equipment to demonstrate the practical feasibility of deploying our solution outside the lab environment.

As part of our future work, we will further investigate how to adequately quantify the positive and/or negative impact of concurrent changes such that trade-offs can be offered to the end-user without jeopardizing the reliability of the intelligent environment.

Acknowledgments This research is partially funded by the Research Fund KU Leuven.

References

1. Arnold RS (1996) Software change impact analysis. IEEE Computer Society Press, Los Alamitos
2. Augusto J, McCullagh P (2011) Safety considerations in the development of intelligent environments. In: Novais P, Preuveneers D, Corchado J (eds) ISAmI. Advances in intelligent and soft computing, vol 92. Springer, New York, pp. 197–204
3. Augusto JC, Hornos MJ (2012) Using simulation and verification to inform the development of intelligent environments. In: Botía et al. [8], pp 413–424. doi:10.3233/978-1-61499-080-2-413
4. Augusto JC, Hornos MJ (2013) Software simulation and verification to increase the reliability of intelligent environments. Adv Eng Softw 58:18–34. doi:10.1016/j.advengsoft.2012.12.004. <http://www.sciencedirect.com/science/article/pii/S0965997813000033>
5. Bakhouya M, Campbell R, Coronato A, Pietro GD, Ranganathan A (2012) Introduction to special section on formal methods in pervasive computing. ACM Trans Auton Adapt Syst 7(1):6:1–6:9. doi:10.1145/2168260.2168266
6. Bhatti ZW, Naqvi NZ, Ramakrishnan A, Preuveneers D, Berbers Y (2014) Learning distributed deployment and configuration trade-offs for context-aware applications in intelligent environments. JAISE 6(5):541–559. doi:10.3233/AIS-140274
7. Bonino D, Corno F (2008) Dogont—ontology modeling for intelligent domotic environments. In: Sheth A, Staab S, Dean M, Paolucci M, Maynard D, Finin T, Thirunarayan K (eds) The semantic web—ISWC 2008, Lecture notes in computer science, vol 5318. Springer, Berlin Heidelberg, pp 790–803. doi:10.1007/978-3-540-88564-1_51
8. Botía JA, Schmidtke HR, Nakashima T, Al-Mulla MR, Augusto JC, Azitria A, Ball M, Callaghan V, Cook DJ, Dooley J, O'Donoghue J, Egerton S, Haya PA, Hornos MJ, Morales E, Orozco JC, Portillo-Rodríguez O, González AR, Sandoval O, Tripicchio P, Wang M, Zamudio V (eds) (2012) Workshop proceedings of the 8th international conference on intelligent environments, Guanajuato, México, June 26–29, 2012, ambient intelligence and smart environments, vol 13. IOS Press. <http://www.booksonline.iospress.nl/Content/View.aspx?piid=30661>
9. Briand L, Labiche Y, O'Sullivan L (2003) Impact analysis and change management of uml models. In: Proceedings of the international conference on software maintenance, 2003. ICSM 2003, pp 256–265. doi:10.1109/ICSM.2003.1235428
10. Cook R (2000) How complex systems fail. Tech. rep. Cognitive technologies Laboratory, University of Chicago. <http://www.ctlab.org/documents/HowComplexSystemsFail.pdf>
11. Coopman T, Theetaert W, Preuveneers D, Berbers Y (2010) A user-oriented and context-aware service orchestration framework for dynamic home automation systems. In: Augusto JC, Corchado J, Novais P, Analide C (eds) Ambient intelligence and future trends—international symposium on ambient intelligence (ISAmI 2010), advances in soft computing, vol 72. Springer, Berlin Heidelberg, pp 63–70. doi:10.1007/978-3-642-13268-1_8
12. Coronato A, De Pietro G (2011) Formal specification and verification of ubiquitous and pervasive systems. ACM Trans Auton Adapt Syst 6(1):9:1–9:6. doi:10.1145/1921641.1921650
13. German DM, Hassan AE, Robles G (2008) Change impact graphs: determining the impact of prior code changes. Inf Softw Technol 51(10):1394–1408. doi:10.1016/j.infsof.2009.04.018. <http://www.sciencedirect.com/science/article/pii/S095058490900069X> (Source Code Analysis and Manipulation, (SCAM) 2008)
14. Gethers M, Dit B, Kagdi H, Poshvanyk D (2012) Integrated impact analysis for managing software changes. In: Proceedings of the 34th international conference on software engineering, ICSE '12. IEEE Press, Piscataway. <http://dl.acm.org/citation.cfm?id=2337223.2337274>
15. Glimm B, Horrocks I, Motik B, Stoilos G, Wang Z (2014) Hermit: an OWL 2 reasoner. J Autom Reason 53(3):245–269. doi:10.1007/s10817-014-9305-1
16. Grau BC, Horrocks I, Motik B, Parsia B, Patel-Schneider P, Sattler U (2008) Owl 2: the next step for owl. Web Semant 6(4):309–322. doi:10.1016/j.websem.2008.05.001
17. Hassan M, Deruelle L, Ahmad A, Basson H (2011) Towards a better change impact analysis in architecture description languages. In: Maciaszek L, Loucopoulos P (eds) Evaluation of novel approaches to software engineering. Communications in computer and information science, vol 230. Springer, Berlin Heidelberg, pp 136–147. doi:10.1007/978-3-642-23391-3_10
18. Holzmann G (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295. doi:10.1109/32.588521
19. Igaki H, Nakamura M (2010) Modeling and detecting feature interactions among integrated services of home network systems. IEICE Trans 93–D(4):822–833
20. ISO EN 7730-2005 (2005) Ergonomics of the thermal environment—analytical determination and interpretation of thermal comfort using calculation of the pmv and ppd indices and local thermal comfort criteria. Tech Rep
21. Jackson D (2002) Alloy: a lightweight object modelling notation. ACM Trans Softw Eng Methodol 11(2):256–290. doi:10.1145/505145.505149
22. Krötzsch M (2012) OWL 2 profiles: an introduction to lightweight ontology languages. In: Eiter T, Krennwallner T (eds) Reasoning web. Semantic technologies for advanced query answering—Proceedings of 8th International Summer School 2012, Vienna, Austria, September 3–8, 2012. Lecture Notes in Computer Science, vol 7487. Springer, New York, pp 112–183. doi:10.1007/978-3-642-33158-9_4
23. Leelaputra P, Matsuo T, Tsuchiya T, Kikuno T (2008) Detecting feature interactions in home appliance networks. In: SNPD. IEEE Computer Society, pp 895–903
24. Lehnert S (2011) A taxonomy for software change impact analysis. In: Proceedings of the 12th international workshop on principles of software evolution and the 7th annual ERCIM workshop on software evolution, IWPSE-EVOL '11. ACM, New York, pp 41–50. doi:10.1145/2024445.2024454
25. Lindsay B (2004) Designing for failure may be the key to success—interview by Steve Bourne. ACM Queue 2(8)

26. Lock S, Kotonya G (1999) An integrated, probabilistic framework for requirement change impact analysis. *Australas J Inf Syst* 6(2). <http://dl.acs.org.au/index.php/ajis/article/view/292>
27. Naqvi NZ, Moens K, Ramakrishnan A, Preuveneers D, Hughes D, Berbers Y (2015) To cloud or not to cloud: a context-aware deployment perspective of augmented reality mobile applications. In: Wainwright RL, Corchado JM, Bechini A, Hong J (eds) *Proceedings of the 30th annual ACM symposium on applied computing*, Salamanca, Spain, April 13–17. ACM, pp 555–562. doi:10.1145/2695664.2695880
28. O'Connor M, Das A (2011) A method for representing and querying temporal information in owl. In: Fred A, Filipe J, Gamboa H (eds) *Biomedical engineering systems and technologies. Communications in computer and information science*, vol 127. Springer, Berlin Heidelberg, pp 97–110. doi:10.1007/978-3-642-18472-7_8
29. Pauty J, Preuveneers D, Rigole P, Berbers Y (2006) Research challenges in mobile and context-aware service development. In: *Workshop on research challenges in mobile and context-aware service development, FRCSS 2006*, pp 141–148. <https://lirias.kuleuven.be/handle/123456789/133340>
30. Pirsiavash H, Ramanan D (2012) Detecting activities of daily living in first-person camera views. In: *2012 IEEE conference on computer vision and pattern recognition*, Providence, RI, USA, June 16–21, 2012. IEEE Computer Society, pp 2847–2854. doi:10.1109/CVPR.2012.6248010
31. Preuveneers D, Berbers Y (2005) Adaptive context management using a component-based approach. In: *Proceedings of the 5th IFIP WG 6.1 international conference on distributed applications and interoperable systems, DAIS'05*. Springer, Berlin, Heidelberg, pp 14–26. doi:10.1007/11498094_2
32. Preuveneers D, Berbers Y (2005) Semantic and syntactic modeling of component-based services for context-aware pervasive systems using owl-s. In: *Proceedings of first international workshop on managing context information in mobile and pervasive environments*, pp 30–39. <https://lirias.kuleuven.be/handle/123456789/133327>
33. Preuveneers D, Berbers Y (2006) Multi-dimensional dependency and conflict resolution for self-adaptable context-aware systems. In: *Proceedings of the international conference on autonomic and autonomous systems, ICAS '06*. IEEE Computer Society, Washington, DC, p 36. doi:10.1109/ICAS.2006.37
34. Preuveneers D, Berbers Y (2007) Towards context-aware and resource-driven self-adaptation for mobile handheld applications. In: *Proceedings of the 2007 ACM symposium on applied computing, SAC '07*. ACM, New York, pp 1165–1170. doi:10.1145/1244002.1244255
35. Preuveneers D, Berbers Y (2008) Encoding semantic awareness in resource-constrained devices. *IEEE Intell Syst* 23(2):26–33. doi:10.1109/MIS.2008.25
36. Preuveneers D, Berbers Y (2010) μ c-semps: energy-efficient semantic publish/subscribe for battery-powered systems. In: Snc P, Ott M, Seneviratne A (eds) *Mobile and ubiquitous systems: computing, networking, and services—7th international ICST conference, MobiQuitous 2010*, Sydney, Australia, December 6–9, 2010, Lecture notes of the institute for computer sciences, social informatics and telecommunications engineering, vol 73. Springer, New York, pp 39–50. doi:10.1007/978-3-642-29154-8_4 (revised selected papers)
37. Preuveneers D, Berbers Y (2010) Context-driven migration and diffusion of pervasive services on the osgi framework. *Int J Auton Adapt Commun Syst* 3(1):3–22. doi:10.1504/IJAACS.2010.030309
38. Preuveneers D, Berbers Y (2012) Consistency in context-aware behavior: a model checking approach. In: Bota et al.[8], pp 401–412. doi:10.3233/978-1-61499-080-2-401
39. Preuveneers D, Berbers Y, Joosen W (2016) SAMURAI: a batch and streaming context architecture for large-scale intelligent applications and environments. *JAISE* 8(1):63–78. doi:10.3233/AIS-150357
40. Preuveneers D, den Bergh JV, Wagelaar D, Georges A, Rigole P, Clerckx T, Berbers Y, Coninx K, Jonckers V, Bosschere KD (2004) Towards an extensible context ontology for ambient intelligence. In: Markopoulos P, Eggen B, Aarts E, Crowley JL (eds) *Second European symposium on ambient intelligence, LNCS*, vol 3295. Springer, Eindhoven, pp 148–159
41. Preuveneers D, Chen C, Ramakrishnan A, Feng M, Wang P, Berbers Y (2014) Design for failure: intelligent systems learning from their mistakes. In: Augusto JC, Zhang T (eds) *Workshop proceedings of the 10th international conference on intelligent environments*, Shanghai, China, June 30–July 1, 2014. *Ambient intelligence and smart environments*, vol 18. IOS Press, pp 171–182. doi:10.3233/978-1-61499-411-4-171
42. Preuveneers D, Joosen W (2015) Change impact analysis for context-aware applications in intelligent environments. In: Preuveneers D (ed) *Workshop proceedings of the 11th international conference on intelligent environments*, Prague, Czech Republic, July 15–17, 2015. *Ambient intelligence and smart environments*, vol 19. IOS Press, pp 70–81. doi:10.3233/978-1-61499-530-2-70
43. Preuveneers D, Novais P (2012) A survey of software engineering best practices for the development of smart applications in ambient intelligence. *J Ambient Intell Smart Environ* 4(3):149–162. <http://dl.acm.org/citation.cfm?id=2350776.2350779>
44. Preuveneers D, Vandewoude Y, Rigole P, Ayed D, Berbers Y (2006) Context-aware adaptation for component-based pervasive computing systems. In: *Advances in pervasive computing 2006. Adjunct proceedings of the 4th international conference on pervasive computing*, vol 207, pp 125–128. <https://lirias.kuleuven.be/handle/123456789/133938>
45. Schuster A (2008) *Robust intelligent systems*, 1st edn. Springer, New York
46. Tang A, Nicholson A, Jin Y, Han J (2007) Using Bayesian belief networks for change impact analysis in architecture design. *J Syst Softw* 80(1):127–148. doi:10.1016/j.jss.2006.04.004