A Language for Online State Processing of Binary Sensors, Applied to Ambient Assisted Living

Nic Volanschi, Bernard Serpette, Adrien Carteron, Charles Consel

▶ **To cite this version:**

Nic Volanschi, Bernard Serpette, Adrien Carteron, Charles Consel. A Language for Online State Processing of Binary Sensors, Applied to Ambient Assisted Living. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies , In press, 2 (4), pp.26. 10.1145/3287070 . hal-01947742

HAL Id: hal-01947742
https://inria.hal.science/hal-01947742

Submitted on 17 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Language for Online State Processing of Binary Sensors, Applied to Ambient Assisted Living

NIC VOLANSCHI*, Inria Bordeaux - Sud-Ouest, France
BERNARD SERPETTE, Inria Bordeaux - Sud-Ouest, France
ADRIEN CARTERON, Inria Bordeaux - Sud-Ouest, France
CHARLES CONSEL, Bordeaux INP & Inria, France

There is a large variety of binary sensors in use today, and useful context-aware services can be defined using such binary sensors. However, the currently available approaches for programming context-aware services do not conveniently support binary sensors. Indeed, no existing approach simultaneously supports a notion of state, central to binary sensors, offers a complete set of operators to compose states, allows to define reusable abstractions by means of such compositions, and implements efficient online processing of these operators.

This paper proposes a new language for event processing specifically targeted to binary sensors. The central contributions of this language are a native notion of state and semi-causal operators for temporal state composition including: Allen's interval relations generalized for handling multiple intervals, and temporal filters for handling delays. Compared to other approaches such as CEP (complex event processing), our language provides less discontinued information, allows less restricted compositions, and supports reusable abstractions. We implemented an interpreter for our language and applied it to successfully rewrite a full set of real Ambient Assisted Living services. The performance of our prototype interpreter is shown to compete well with a commercial CEP engine when expressing the same services.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; • **Computer systems organization** → *Sensor networks*; • **Information systems** → *Data streaming*; *Online analytical processing engines*; • **Human-centered computing** → Ubiquitous computing;

Additional Key Words and Phrases: Binary sensors, Allen interval algebra, Ambient assisted living, Smart homes

ACM Reference Format:
Nic Volanschi, Bernard Serpette, Adrien Carteron, and Charles Consel. 2018. A Language for Online State Processing of Binary Sensors, Applied to Ambient Assisted Living. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 4, Article 192 (December 2018), 26 pages. https://doi.org/10.1145/3287070

1 INTRODUCTION

There is a large variety of binary sensors in use today (e.g., [15, 41]) . These include both low-cost sensors for the consumer market, such as contact sensors for detecting opening or closing doors, drawers, cabinets, etc.; motion detectors for signalling motion in some area; smoke sensors for fire alarms; vibration sensors for

*This is the corresponding author

Authors' addresses: Nic Volanschi, Inria Bordeaux - Sud-Ouest, 200, avenue de la Vieille Tour, 33405, Talence cedex, France, eugene.volanschi@inria.fr; Bernard Serpette, Inria Bordeaux - Sud-Ouest, Talence, France, bernard.serpette@inria.fr; Adrien Carteron, Inria Bordeaux - Sud-Ouest, Talence, France, adrien.carteron@aquilenet.fr; Charles Consel, Bordeaux INP & Inria, Talence, France, charles.consel@inria.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
2474-9567/2018/12-ART192 $15.00
https://doi.org/10.1145/3287070

detecting possible intrusions; and more sophisticated sensors for professional markets, such as pressure switches for detecting when a certain pressure of a fluid has been reached; pressure-sensitive floors for locating persons in indoor areas, etc. Other commonly used binary sensors are software sensors such as: connectivity sensors associated to any wireless device; at-home detectors based on a conjunction of several physical sensors, and so on. Moreover, various objects within a building are being instrumented with binary sensors, such as connected locks for signalling and changing the lock status of doors; smart switches to detect when a light or appliance is on; and so on. Even non-binary sensors such as connected plugs (signalling the electric power delivered by the plug) are frequently used in fact as binary sensors: the corresponding appliance (coffeemaker, cooker, etc.) being used or not used.

Given this wide range of binary sensors, it is no surprise that useful applications are being developed using exclusively or mostly binary sensors in several domains, including smart home services [15] and automated control in plants [41]. Specific research has been focused, for instance, on uses of binary sensors for object tracking [8], activity recognition [33, 36], or simultaneous tracking and activity recognition [42]. Such works frequently mention advantages of binary sensors such as cheap cost, low intrusiveness, and easy installation, and claim that these advantages also facilitate user acceptance.

In spite of this widespread use of binary sensors, there is no programming model specifically supporting such sensors. Nevertheless, binary sensors do have a key characteristics that unifies them and takes them apart from sensors in general: their indissociable notion of *state*. Indeed, most binary sensors we studied produce a value only when there is a significant change in the monitored environment parameter. Given that there are only two possible values (noted 0 and 1), a significant change means only switching from 0 to 1 or vice versa. As sensors usually report such switches in a timely manner, one can usually assume that the current value of the monitored parameter is the last measured value, as long as no new value has been reported. For instance, a contact sensor attached to a door signals a value when the door is open or closed (e.g., values of 1 and 0, respectively). When no value is produced the *current state* of the door can be assumed to be the last state reported. Similarly, a motion sensor produces a value of 1 when motion is detected in the monitored area, and 0 when no motion is detected anymore after some small delay. Even though this delay can usually be configured, it is usually set so as to give a timely view of the environment, where the notion of "timely" may vary according to application needs. Thus, once the motion sensor is configured correctly, applications may generally assume that the current value of the motion parameter is the last value produced, until they are notified about a change by an opposite value. Similar examples could be formulated for all kinds of binary sensors.

These requirements of supporting states and related notions such as current state or state combining operators are thus specific to applications over binary sensors. However, they only add to other key requirements associated to applications over sensors in general, such as the need for *online processing* of sensor data, and the support of effective *code reuse*. Indeed, online processing of the continuous streaming data, while it is delivered, is crucial for instance in IoT applications, due to the volume of data [34], and for the timely recognition of human activities, which is essential in many ubiquitous applications interacting with users [29]. On the other hand, code reuse is a desirable feature in general software engineering, but its need is exacerbated in domains where a large variety of similar services have to be developed. This is the case indeed in domains such as smart homes and ambient assisted living, where service logic customization, to a large number of user and home configurations, is a key requirement for achieving user acceptance [16].

This paper starts by reviewing a number of available approaches for developing ubiquitous services based on sensors. We focus here only on the context detection component of such services, which detects situations of interest in the environment. The action component of the services, which consists of reacting appropriately at such situations, is out of the scope of this paper, and can be considered in most cases as orthogonal to the detection component. When reviewing available approaches, we thus evaluate their support for developing the context detection logic, and we show that they have important limitations when applied to binary sensors.

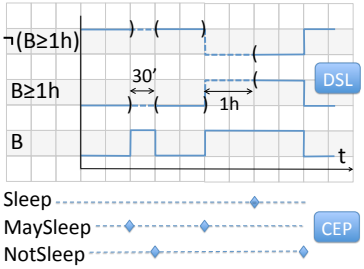

Fig. 1. Delaying notifications during sleep.

Namely, no existing approach simultaneously addresses the notion of state, provides a comprehensive set of operators for combining states from different sensors, computes these operators online on streams of upcoming data coming from sensors, keeps applications informed in a timely manner about the current value of (possibly combined) states, and provides mechanisms for creating reusable abstractions for state combination.

1.1 Outline of Our Proposal: a DSL for Online State Processing

To overcome these limitations, we present a new approach to developing ubiquitous applications based on binary sensors, supported by a domain-specific language (DSL) for online state processing. This language offers a native notion of state by modeling both physical and software sensors as boolean function of time, or equivalently, as series of time intervals. It provides a comprehensive set of operators for combining states, including the full set of Allen's time interval relations [4], complemented by operators for expressing durations, and extensible with first-class user-defined abstractions for combining states. All these operators are computed online on streams of sensor events.

The key enabling feature of our approach is that we admit *non-causal* functions as the semantics of some state operators. Indeed, many state operators could not be naturally modeled as boolean functions over time if only causal functions were considered, i.e., whose value at a given time depends only on present and past values of its argument(s).

For instance, let us consider an application that delivers notifications to the user of a smart home only when the user is surely not sleeping, based on a sensor indicating presence in the bedroom. Let us assume that the sleeping situation is defined as the user being present in the bedroom for more than one hour. This application should bufferize notifications whenever the user is sleeping (i.e. is in the bedroom since more than one hour) or *may be* sleeping (i.e., is in the bedroom since less than one hour). Figure 1 illustrates how our language allows to define the sleep state during which notifications should be bufferized. The presence in the bedroom indicated by a motion sensor B is first filtered using the expression 'B≥1h'. Note that some periods of this boolean function are represented as dotted lines. During these periods, the value of the function is not known in real time, because the function is non-causal, that is, it depends on a future event: whether state of B lasts for more than 1 hour or not. In the meantime, the function is blocked waiting the next event in stream B. When this happens, the value of this function becomes known and the function can catch up real time, represented as a solid line. The boolean negation of this function gives the moments when notifications may be delivered to the user: these are the moments when its value is 1 figured as a solid line. In all the other moments, notifications should be bufferized.

Let us furthermore assume that phone calls from the caregiver are to be muted and redirected to an answering machine whenever the user is surely sleeping. No new logic has to be developed to detect this situation, as it

corresponds to the periods when the preceding signal is 0 figured as a solid line (or, equivalently, its non-negated form is 1 figured as a solid line).

Thus, modeling operators as possibly non causal boolean functions allows to provide real-time information to applications whenever possible, and to apply standard boolean operators such as negation, in a very natural way. The details of managing periods of incertitude at different levels of a composed situation are transparently handled by the language interpreter by blocking and unblocking function evaluation according to the stream of upcoming values coming from the sensors. For comparison, the same sleep situations expressed as a complex event in a CEP language would provide the more discontinued information shown at the bottom of Figure 1. Thus, the absence of the Sleep complex event may mean either than the user is surely not sleeping (i.e. absent from the bedroom) or maybe sleeping (i.e. in the bedroom since less than one hour). To distinguish between these situations and to implement a similar behavior in CEP, two more events have to be defined: MaySleep, when the user enters the bedroom, and NotSleep, when the user leaves the bedroom. Furthermore, these three related events must be combined in two distinct ways for bufferizing notifications (between MaySleep and NotSleep), and muting phone calls from the caregiver (between Sleep and NotSleep). In real cases, more complex than this simple example, defining the three related events and the two combining logics would favor code duplication, with adverse effects on productivity and maintenance. In constrast, in our DSL the two logics are expressed uniformly in terms of a single Sleep signal, offering less discontinued information; this avoids code duplication and simplifies development.

1.2 Challenges Faced by Our DSL

Generally speaking, a domain-specific language aims at simplifying the development in a given domain, by enabling a concise, higher-level, specialized discourse, compared to a general programming language (GPL). Stemming from this main differences with respect to GPLs, there are three important challenges a DSL is facing. The first challenge is that of *expressiveness*. This does not require being able to express more programs than a GPL (which by definition can express any program). Rather, it requires covering a significant part of the addressed domain, avoiding over-restriction of the discourse. The second challenge is that of implementation *efficiency*, which requires that the higher abstraction level should not incur excessive resource consumption with respect to established solutions, to the point of limiting its practical application. Thereby, the focus of this challenge is not on measuring detailed performance figures, but rather on assessing the overall computing power needed to support the approach. This assessment is especially relevant in the target domain of pervasive computing, where solutions are desirable that apply to a large spectrum of computing environments, including those with scarce resources, towards the edge of the cloud. The third challenge is that of *conciseness*: a DSL must be concise in solving problems representative for its domain, when compared to other established solutions.

For validating our DSL against these challenges, we implemented our approach in the form of an interpreter for our language, and apply it for expressing a set of 53 real Ambient Assisted Living (AAL) services, thus demonstrating the expressiveness of our language. By comparing our approach to an established approach for developing AAL services, namely Complex Event Processing (CEP) [13], we show that our interpreter performs efficiently in both processing time and memory consumption, and exposes to applications less discontinuous information about current states. Furthermore, we assess the conciseness of our approach by comparing the form of these AAL services in the two approaches (DSL and CEP), both qualitatively in terms of abstraction level and reuse, and quantitatively in terms of code size.

Thus, in accordance with the prevalent practice in the research on *textual* DSLs [27], we use empirical validation of our implemented approach along the mentioned challenges, on a representative set of domain problems. Empirical evaluation of the approach (e.g., involving a user study) is left for future work.

1.3 Contributions

Our main contributions can be summarized as follows:

- We propose a new approach for computing context information from boolean sensors, based on the concept of online state processing, supported by a domain-specific language.
- We demonstrate through examples some of its advantages when compared to other approaches, such as less discontinuous results, less restricted compositions, intuitive semantics, powerful abstraction mechanisms, and clean handling of simultaneity.
- We validate our language by using it to redefine a full set of real AAL services.
- We present a prototype implementation of this language, and show that it exhibits competitive efficiency, in terms of both CPU and memory consumption, when compared to the Esper CEP.

After reviewing related work in Section 2, we first present some background knowledge about AAL in Section 3 before introducing our language for online state processing in Section 4, and its implementation and validation in Sections 5 and 6.

2 RELATED WORK

We first discuss here the Allen time interval relations, and their possible use for expressing temporal relations between sensor states; we specially focus on its applications to events or streams of events. Then, we review some common approaches for building ubiquitous applications based on events coming from sensors.

2.1 Allen Algebra for Event Handling

The relations between time intervals have been studied long time ago by Allen [4], who defined all the 13 possible, mutually exclusive, relations between two finite intervals $I_1 = [t_{s_1}, t_{e_1})$ and $I_2 = [t_{s_2}, t_{e_2})$, as shown in Figure 2. For example, 'I_1 Overlaps I_2' means that $I_1 \cap I_2 \neq \emptyset \wedge t_{s_1} < t_{s_2} \wedge t_{e_1} < t_{e_2}$. Note that the relations 'Starts'/'Started by', 'Ends'/'Ended by', 'Equals', and 'Meets'/'Met by' require simultaneity between several events. For example, 'I_1 Meets I_2' means that I_1 ends exactly when I_2 starts, that is, $I_1 = [t_1, t_2) \wedge I_2 = [t_2, t_3)$. Note also that all relations except 'Equals' are grouped in pairs in the figure, that are the inverse of each other. For example, I_1 During $I_2 \Leftrightarrow I_2$ Contains I_1. The 13 Allen relations cannot directly be used between sensors, because a sensor may switch its state several times, thus generating several time intervals when its current value is 1. Thus, Allen relations have to be generalized to work on two series of time intervals, instead of just two intervals, before using them for binary sensors.

In fact, the Allen relations have been generalized in the past to deal with series of non-overlapping intervals, also called non-convex intervals. The seminal work of Ligozat [31] resulted in a great number of relations between such intervals (exponentially many in the number of basic intervals). For practical reasons, a smaller set of relations called Allen* [19] has been defined later to cover only 14 relations of practical interest, that roughly correspond to the 13 Allen relations, as follows. If R is an Allen relation, then R* is the relation between two series of intervals I and J defined as:

$$I \; R^* \; J \Leftrightarrow \forall I_i \in I \; \exists J_j \in J \; . \; I_i \; R \; J_j$$

Relations in Allen* have been designed to compare periodic events, that is to answer questions such as "is my weekly schedule fully compatible with the schedule of a given recurrent activity?". For pervasive applications, it is not informative enough to ask "does a person always have breakfast after getting up?", because the corresponding answer is a boolean value for a whole smart home log. Rather, it is much more informative to ask "which days does a person have breakfast after getting up?", whose answer is a boolean function over the log period. Furthermore, this work does not consider evaluating these relations on streaming data, which is crucial for

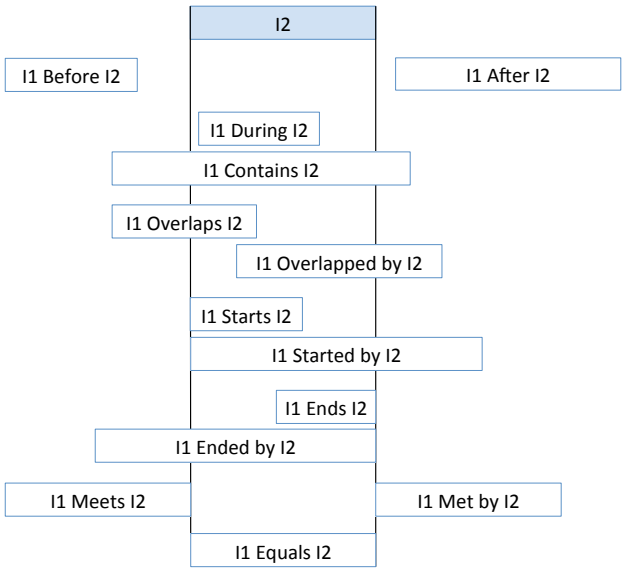

Fig. 2. Allen interval operators.

applications deployed in a smart home, reacting to detected situations. The evaluation of such queries is vastly different in an offline setting (on a complete log) *vs* online (on streaming data).

It has been shown [35] that when considering a discrete time domain, the Allen interval relations can be encoded in the linear temporal logic LTL. In fact, a subset $LTL_{\Box}$ containing only the operators $\Diamond$ (eventually) and $\Box$ (always), which are in fact reducible to each other, is sufficient. This encoding allows to reuse the many formal tools developed for LTL to expressions containing Allen and boolean operators. Unfortunately, their LTL encoding of Allen relations only works for relations between two intervals, as in the Allen logic, and not for series of non-overlapping intervals, as in our sensor states. Therefore, we have to design Allen-based operators that work on series of intervals, and to implement a special evaluation strategy that computes the value of these semi-causal operators as soon as possible on online streams of events.

2.2 Event Handing Languages and Models

CEP. Complex Event Processing (CEP) [13] is an established computing paradigm that was specifically designed for processing online streams of events, and which allows defining layers of increasingly complex event patterns out of elementary events. The CEP paradigm and its associated languages have been naturally applied since more than one decade to process data flows coming from sensors, and recently there is a growing number of CEP applications targeting sensors in the IoT and ubiquitous computing domains [14].

According to this trend, it may seem that CEP languages are the perfect match for expressing the context detection logic using sensors. In fact, when applied to binary sensors, CEP languages have the following important limitations:

No native notion of state: In CEP, binary state information must be encoded as a sequence comprising the starting and ending events of the interaction and other events, e.g., "door-is-open, *followed by* some-event-of-interest *before* door-is-closed or door-is-open"[1]. When several interactions must be combined (e.g. a door being open and nobody there for a while), this tends to produce complex formulas which are tedious to code and error-prone. Thus, a native notion of state is generally lacking in CEP languages.

Discontinued information: Complex events provide information only when they are completed. Thus, absence of a complex event may either correspond to the event being incompletely produced or definitely not started (e.g., the Sleep event in Figure 1). This kind of discontinued information is not sufficient, for instance, to an AAL notification service bufferizing alerts when the user is sleeping or may be sleeping, as discussed in the introduction; it requires duplicating the complex event to a family of related events.

Limited abstraction mechanisms: CEP is all about defining more complex events in terms of simpler events. It is indeed easy to define layers of abstraction by defining increasingly complex events as CEP formulas involving atomic and other complex events. However, most CEP languages offer a fixed set of composition operators, and it is usually not possible to easily add new operators, nor treat operators as first-class objects, for instance by passing them as parameters to other operators, much the same way as user-defined functions in general programming languages. This prevents defining a whole spectrum of useful *reusable* abstractions, which can be instantiated many times. Among others, it is usually not possible to define a new binary operator and apply it to N events by passing it as parameter to a 'reduce' operator.

Interval-based CEP. The importance of events lasting a certain time has been recognized in certain CEP languages that adopt an interval-based semantics [2]. However, only complex events may have a duration, not atomic events. The goal is to avoid some paradoxes related to the classical, time-point semantics, of complex events. For instance, in the time-point semantics of classic CEP languages, a sequence pattern 'A *followed by* (B *followed by* C)' may counter-intuitively match the sequence of events 'B, A, C', because the complex event 'B *followed by* C' is timestamped with the time of its last event, C, which follows the event A. Interval-based CEP languages (IB-CEP, for short) avoid this paradox by associating the complex event 'B *followed by* C' with the whole time interval from B to C. This also allows to use in some IB-CEP languages Allen's relations between time intervals, such as before, during, or overlaps to correlate complex events [2, 5, 12, 30]. Nevertheless, states in IB-CEP are not native and must be expressed as complex events, which leads to the same complex formulas as in classic CEP, when expressing combinations of states.

End-user event programming. A native notion of state in event programming has been investigated more recently in the End-User Programming (EUP) research community. The classic IFTTT (If This Then That) language [40] allows end users to express services (including for a smart home) in the form of triggers and actions, where triggers only refer to events. In particular, triggers mentioning a state such as 'the light is on' refer in fact to the events of state changes, e.g., 'the light gets turned on'. A user study [24] evidenced the fact that specifying services in IFTTT is difficult because the notions of event and state are frequently confused by end users. This study recommends some guidelines for future EUP languages for smart homes. Taking that work into account, an extension of IFTTT called AppsGate has been recently proposed [11], in which distinct conditional constructs refer to instantaneous events (e.g., as-soon-as, each-time) and to states lasting over a time interval (e.g., if, while). However, conditions must refer to a single event and a single state respectively, which prevents combining events or states between them and with each other. Very recently, the CCBL visual language has been proposed [38] which allows to express triggers combining several states. Combinations are done by graphically nesting state conditions, which is roughly equivalent to using boolean operators And, Or, and Not between states. There are

[1]The second occurrence of door-is-open is usually required to ensure that the first opening event is not paired with a closing event from a subsequent door opening.

no other operators combining states. An extension of CCBL [39] adds a few operators named after some Allen relations: During, Starts, Ends, and After. However, their semantics is always causal, and it does not exactly correspond to the semantics of Allen relations bearing the same names. Furthermore, other Allen relations such that 'Overlaps/Overlapped by' are not covered by CCBL, precisely because they are only known *a posteriori*.

Automata. Automata are also frequently used for describing systems reacting to events, possibly coming from sensors. A commonly used language for describing automata is the StateCharts visual language and associated tools [23], allowing to describe hierarchical finite state machines. Temporal delays are not natively handled, and must be implemented using events from external timers. Timed automata [6] add a native expression of temporal delays, and allow to formally check timing properties such as reachability in presence of time constraints.

However, combining conditions about several states in a predictable way involves producing the cross product of several automata, each expressing the state of one sensor. This operation may lead to automata of size exponential in the number of sensors, which may quickly lead to state explosion.

Besides that, although Allen relations and temporal operators can be modelled using automata, these visual languages usually do not offer powerful mechanisms of reusable abstraction, beyond copy and paste. Therefore, the sequence and timing constraints corresponding to an operator have to be repeatedly coded into automata, with the associated risk of introducing subtle variations in behavior or bugs. Indeed, user experiments [38] have shown that users, including programmers, often forget to consider all the possible transitions when defining automata for relatively simple smart home services.

Synchronous languages. Synchronous languages such as Esterel [20] propose a textual notation for a higher-level encoding of automata, to describe applications reacting to events, modeled as signals over time. Esterel includes both pure signals, that are not valued, so they can only be tested for presence or absence, and typed signals, that may have boolean values. Esterel constitutes a domain-specific, imperative programming language for computing over signals, including an abstraction mechanism called modules, similar to user-defined procedures in general programming languages. Beyond such powerful abstractions, the strength of synchronous languages is that their domain-specific nature enables ensuring strong guarantees about the correctness and real-time behaviour of the programs, including a clean and predictable handling of simultaneity between events. Esterel programs are compiled into automata, and suffer from the space explosion problem mentioned above. Some smart optimizations performed by the compiler can alleviate, but not solve this problem. There are also a number of issues for handling delays in Esterel [7], which are crucial for implementing temporal operators. Most importantly, synchronous languages allow to encode in a reusable way event-based logic provided that it is reactive, that is, expressed in a causal way. It is an interesting question whether and how non causal operators such as the ones considered here (Allen relations and temporal operators) could be encoded in a synchronous language such as Esterel.

In general, the definition of various operators as non strictly causal distinguishes our approach from most of the event handling languages discussed above, including CEP, EUP, automata-based, and synchronous languages.

2.3 Pervasive Middleware and Tools

A recent survey of the state of the art in developing context-aware systems [3] discusses several development methodologies, supporting various phases in the development lifecycle of a pervasive application, including design, implementation, and maintenance. Programming paradigms are discussed in relation to the implementation phase. They mostly concern extensions of general programming languages for handling context-sensitive behavior to a program, in the form of aspects in Aspect-Oriented Programming (AOP) or of features in Software Product Lines (SPL). The only kind of domain-specific language for context detection logic covered by the survey is trigger-action programming such as IFTTT, discussed above. Other covered domain-specific languages such as DiaSuite only concern the design phase. Middleware for pervasive systems and more specifically the Internet of

Things are discussed in some depth in another survey [34]. They present middleware as reusable solutions to frequently encountered problems like heterogeneity, inter-operability, security, dependability, and sometimes, event management. Thus, one goal of middleware is to provide a set of programming abstractions to help software developers by abstracting them from low-level aspects. However, the logic of context detection still has to be coded in a general programming language, with no specific support for expressing states, their temporal relationships, and computing such relationships online on streams of events.

3 CASE STUDY

Before presenting our solution, we briefly describe in this section the real size case study in the AAL domain that we have chosen for validating our approach. The case study involves expressing in our language the full set of AAL services in an interdisciplinary project called HomeAssist aiming to assess the benefits of pervasive technology and assistive applications for prolonging aging in place [10]. The project started with a human-centered, participative design of a set of useful assistive services for seniors [16]. The specific needs of seniors living alone were gathered by experts in aging and ergonomics from seniors and their caregivers, including both family and professional caregivers. The resulting needs concerned three key domains: everyday functioning, security, and social link. Correspondingly, a set of assistive services was defined to monitor activities of daily living (ADL), alert about security situations, and facilitate social interactions to prevent isolation. These services were implemented using AAL technology, using a minimal set of commercially available wireless sensors, placed at strategic locations in the home. In accordance to the requirements expressed by users, no cameras were included. Rather, the only sensors deployed were motion sensors signalling presence or absence in the covered area, contact sensors signalling door or drawer openings and closings, and smart plugs, measuring the power in Watts delivered by the plug, upon change. The set of wireless sensors were connected to a Vera smart home box situated in each apartment and centralizing all sensor events. An Android tablet completes this home equipment, placed at a fixed location in the home, always plugged in, and used for user interaction, mainly by delivering notifications and receiving user acknowledgements, or responses when applicable. The Vera boxes of each home are connected to the Internet, which allows assistive applications to run on a remote server. A special-purpose application store was populated with a catalog of assistive services implemented in Java, ready to be installed in each home, according to the needs of each participant.

This platform was deployed during a first field study in 17 homes of seniors living alone, aged 82 on average, during 6 months. The study showed significant improvements of users' daily autonomy, self-regulation, and empowerment [17] with respect to a control group. Following these encouraging results, a wider deployment is currently ongoing, involving 129 seniors living alone, for at least 12 months. Their homes span a wide range of configurations, from small apartments to houses with several floors. The number of sensors deployed in a home varies according to its configuration, but includes for the least the following sensors :

- Contact sensors for the entrance door, the fridge, and a cupboard in the kitchen.
- Motion sensors for the bedroom, kitchen, living room, bathroom, toilets, and the entrance hall. For small homes, these sensors cover only parts of multi-purpose rooms.
- Smart plugs corresponding to the appliances being used for meal preparation, such as coffeemaker or microwave.

These sensors are used as binary sensors. Thus, contact sensors signal a value of 1 on opening and 0 on closing; motion detectors signal a value of 1 when a motion begins, and 0 when motion stops (typically, after a timeout, e.g., 30 seconds). Smart plugs are abstracted as binary values: a value of 1 when the corresponding appliance is used, i.e., when the power exceeds a threshold, and 0 when it is not used. The threshold is a constant dependent on the appliance (for the appliances we handled, a threshold of 20W was appropriate). Similarly, battery levels

are considered as almost discharged, signalled by a 0, when their charge level is inferior to some threshold (20% in our case).

The set of services deployed on the HomeAssist platform comprises 53 AAL services. These include:

- Daily activity recognizers, for the following ADLs: wakeup and go-to-bed routines; and meal preparation (breakfast, lunch, and dinner). For instance, the wakeup routine consists in detecting, during a morning time slot, a disappearance in the bedroom shortly followed by an appearance in the kitchen. Similarly, breakfast involves detecting, during a given time slot, two customizable events, such as starting the coffeemaker and accessing the cupboard.
- Security situations monitors, detecting from simply undesired up to critical situations, such as: forgetting the fridge door open; leaving the entrance door open and unattended for some time; abnormal situations like a long period of inactivity in a frequently used room; and night wandering (leaving the entrance door open for some time during the night).
- Infrastructure monitoring services, such as: detecting incompatible sensor events like opening the cupboard while not being present in the kitchen; detecting devices failing to communicate, either at some time point or during a certain time; and signalling devices whose battery is almost discharged. Indeed, such technical AAL services are strictly necessary for a deployment of this scale, comprising a total of more than thousand sensors and lasting for one year or more.

4 A LANGUAGE FOR ONLINE STATE PROCESSING

Let us consider the class of context-aware applications over binary sensors. Each binary sensor provides a stream of binary values in $\mathbb{B} = \{0, 1\}$. These values are timestamped, for example by the smart home box when the event is received from the sensor. Events are considered *simultaneous* when they wear the same timestamp. We consider the domain of natural numbers for timestamps $\mathbb{N} = \{0, 1, 2, \ldots\}$. Sensors usually produce events only when the value changes, e.g, when a door is open or closed. Thus, we can consider that the *current value* of each sensor at time t is the last value produced by it. We further assume that the initial value of each sensor, at time 0, is known. In practice, some sensors may occasionally repeat a value due to packet losses and retransmissions, but these can be filtered out at a lower level. Therefore, at any time t, the *log* of a sensor s can be modeled as a sequence of values $s(t_i) = v_i$ for $0 \leq i \leq n$, where $t_0 = 0 < \ldots < t_i < t_{i+1} \ldots < t_n \leq t$ and $v_{i+1} = \neg v_i$ for $0 \leq i < n$. This log may be modeled by a boolean function over time $s : [0, t] \rightarrow \mathbb{B}$, whose value is $s(t') = v_i \ \forall t' \in [t_i, t_{i+1})$, and $s(t') = v_n \ \forall t' \in [t_n, t]$. Note that the intervals $[t_i, t_{i+1})$ are closed on the left and open on the right, meaning that when a new value is produced, this value replaces the old value of the function. At time t, either $t_n = t$, so the new value v_n has just been produced, or $t_n < t$, implying that no new value has been produced during $[t_n, t]$, so the value at t is still the last value v_n.

We will call a *signal* such a boolean function over time, and note it using letters p, q, r, and s, sometimes indexed by their domain, e.g., $s_{[0,t]}$

Our language for online state processing aims at expressing conditions about sensors, and uses signals for that. Thus, primitive signals are the values of sensors, as shown above. More complex signals are defined by applying operators on other signals.

4.1 Boolean Operators

The most straightforward operators are the boolean connectors 'and', 'or', and 'not', defined as the point-wise application of the standard boolean connectors at every time point of the signals used as arguments. Thus, 'not' is a unary operator that takes a signal $p_{[0,t]}$ and produces a signal $(\neg p)_{[0,t]}$ negating all its values:

$$(\neg p)(t') = \neg\, p(t'), \forall 0 \leq t' \leq t$$

'And' and 'or' are binary operators, i.e., take two signals. In general, the argument signals may not be defined up to the same moment, so the operator is defined on their intersection domain. Thus, for two signals $p_{[0,t_1]}$ and $q_{[0,t_2]}$, they produce signals $(p \wedge q)_{[0,min(t_1,t_2)]}$ and respectively $(p \vee q)_{[0,min(t_1,t_2)]}$ where:

$$(p \wedge q)(t') = p(t') \wedge q(t'), \forall 0 \leq t' \leq min(t_1, t_2)$$

$$(p \vee q)(t') = p(t') \vee q(t'), \forall 0 \leq t' \leq min(t_1, t_2)$$

It is often useful in practice to consider, for a signal p, the time intervals where $p(t') = 1$, e.g. the periods when a door is open, when motion is going on in a room, or when an appliance is being used. Thus, a signal can be viewed as a series of non-overlapping time intervals corresponding to 'states' such as open states, presence states, or in-use states. We will write for such intervals $I \in p$ or $[t_s, t_e) \in p$ when we want to refer to its start and end times. This means that $p(t_s) = 1$ and $p(t_e) = 0$. Note that the dual states (closed states, absence states, or no-use states) can be found the same way in the negated signals. Conversely, if $\mathcal{I}$ is a series of non-overlapping intervals, we may define a signal $p_{[0,t]} = \mathcal{I}$ to be 1 during intervals in $\mathcal{I}$ and 0 elsewhere, in the domain $[0, t]$.

4.2 Allen-based Operators

In many cases, it is necessary to express conditions involving temporal relations between states, such as 'fridge is open *during* absence in the kitchen', which detects an incoherence between two sensors. The 13 Allen relations exhaustively express such conditions between two time intervals (see Figure 2).

However, to cope with the fact that the last state of a signal may be infinitely long (e.g., a door may remain indefinitely open), we extend Allen's relations with the following cases of infinite intervals (and their reverse relationships) which can be decided in finite time:
$\forall t_1, t_2, t_3\ t_1 < t_2 < t_3 \rightarrow [t_2, t_3)$ During $[t_1, \infty) \wedge [t_1, t_3)$ Overlaps $[t_2, \infty) \wedge [t_1, t_2)$ Before $[t_3, \infty)$
$\forall t_1, t_2\ t_1 < t_2 \rightarrow [t_1, t_2)$ Starts $[t_1, \infty) \wedge [t_1, t_2)$ Meets $[t_2, \infty)$.

Moreover, Allen's relations cannot be directly used between two signals, because these latter contain multiple time intervals (also named states). Rather, inspired from each Allen's relation $R(I_1, I_2)$, working on two intervals, we propose to define an operator $r(p, q)$ in our DSL, working on two signals, seen as series of intervals, that detect occurrences of the relationship R in the argument signals, as follows.

$$r(p, q) = \{I_1 \in p \mid \exists I_2 \in q\ .\ I_1\ \mathrm{R}\ I_2\}$$

Essentially, operators applied to signals p and q select all the states in p that are in the corresponding Allen relationship with some state in q. For example:

- $during(p, q) = \{I_1 \in p \mid \exists I_2 \in q\ .\ I_1$ During $I_2\}$: selects all states in p strictly contained in some state of q
- $started(p, q) = \{I_1 \in p \mid \exists I_2 \in q\ .\ I_1$ Started by $I_2\}$: selects all states in p started by some state of q

We use lower case letters for our operators to distinguish them from the corresponding Allen relations, which are capitalized, and we omit the word 'by' for inverse relations. See Figures 4 and 5 for examples of the operators.

However, the Allen relations 'Before/After' have no useful meaning on signals. Indeed, selecting the states in p before some state in q means selecting virtually every state in p. Thus, they will be omitted in the following.

Note that, contrary to the original Allen relations, $during(p, q)$ and $contains(q, p)$ are not the same, the former using a condition $\forall p \exists q$ while the latter uses a condition $\forall q \exists p$. Indeed, all 4 dual combinations between during/contains and $(p, q)/(q, p)$ are different and meaningful.

4.2.1 Causality. When computing some Allen operators on two signals $p_{[0,t_1]}$ and $q_{[0,t_2]}$, the domain of the resulting signal, $[0, t_3]$, may be shorter than the intersection of its arguments' domains: $t3 \leq min(t_1, t_2)$. Indeed, some values in the resulting signal sometimes depend on future events. For instance, as illustrated in Figure 3, when signal $q_{[0,4]}$ becomes 1 at time 1, then signal $p_{[0,4]}$ becomes 1 at time 2, the signal during(p,q) can only be computed until before $t = 2$. The value at time 2 depends on the next upcoming event of either p or q at some

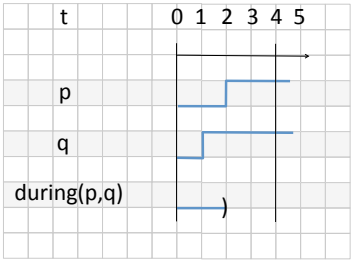

Fig. 3. Semi-causal Allen-based operator.

time $t' > 4$: if p becomes 0 first (see Figure 4, top left), during(p,q)(2)=1 because the state in p starting at 2 is strictly contained in the state of q starting at 1, whenever it may end; on the contrary, if q becomes 0 first (see Figure 4, top middle) or p and q become both 0 at the same time (see Figure 4, top right), then during(p,q)(2)=0. Thus, the during operator may be computed in this case with a delay of one event with respect to the domains of its arguments. We say that an operator is *causal* if its value never depends on future events of its arguments, i.e. its value at any time t only depends on events happening at times $t' \leq t$ [25]. We say that an operator is *semi-causal of order N* if computing its value depends on at most N events in the future of its argument signals.

It may be shown that operator during is semi-causal of order 1. However, 1 is only the *worst* possible delay for this operator. For instance, in the same case in Figure 3, the value of signal during(p,q) is known to be 0 at times 0 and 1, independently of any future events. Indeed, during(p,q)(1) is known to be 0 at time 1 (so with no delay) because during selects some states of p, and there is no state of p including time 1. Thus, an implementation of our language can compute operator during with a delay of 1 event, but it can also optimize many cases to compute it with no delay. We will see in Section 5 that our implementation performs such optimizations whenever possible.

Interestingly, the Allen-based operators defined above have different causality orders. These can be found by studying the worst cases of delay for each operator, illustrated in Figures 4 and 5. Thus, it can be shown that:

- The only causal Allen-based operator is operator 'met(p,q)' (not shown in the figures). Indeed, the value of this operator becomes 1 only when at the same time p becomes 1 and q becomes 0, so that a state of q ends exactly when a state of p begins. The value stays 1 until p becomes 0 again. Thus, the value of operator met only depends on past and present events.
- Operators during, overlapped, ends, starts, started, and equals are all semi-causal of order 1. Their worst cases are shown in Figure 4. The period when the value of these operators is unknown is shown as a dashed line. As can be seen, these periods span only one event of the arguments. For instance, the value of during(p,q)(2) depends on which of p and q falls to 0 first: if p falls first, during(p,q)(2) is 1; else, it is 0.
- Operator 'contains(p,q)' is semi-causal of order 3. Indeed, its worst cases are shown in Figure 5 (top). As can be seen, the period of incertitude is ended at the latest when the state of p ends, or as soon as a (first) state of q is strictly included in the current state of p.
- Operators overlaps, ended, and meets are semi-causal of order N, with N unbound. Indeed, their worst cases, shown in Figure 5 (middle and bottom), contain periods of incertitude spanning an unbounded number of events of signal q, and at most one event of signal p.

4.2.2 Quantified Allen-based Operators. The Allen interval logic itself has been complemented in the past [21] for practical reasons with two other operators, testing whether a time-dependent proposition P holds all the time, respectively some time, during interval I: 'Holds(P,I) $\Leftrightarrow \forall t \in I \,.\, P(t)$' and 'Occurs(P,I) $\Leftrightarrow \exists t \in I \,.\, P(t)$'. As

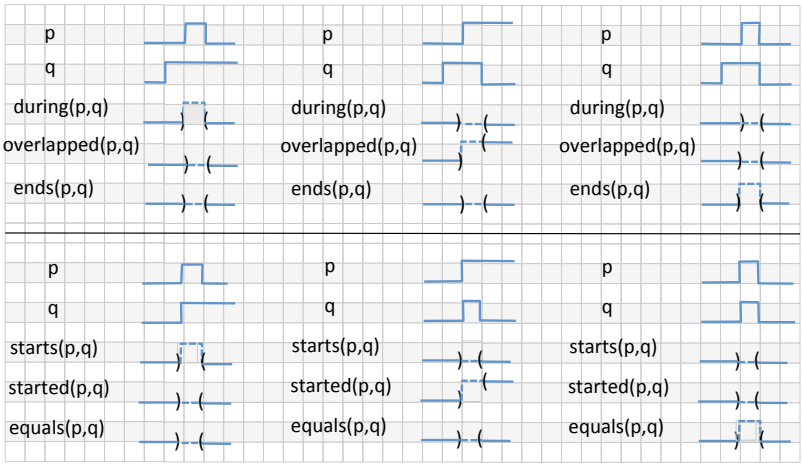

Fig. 4. Semi-causal Allen-based operators of order 1.

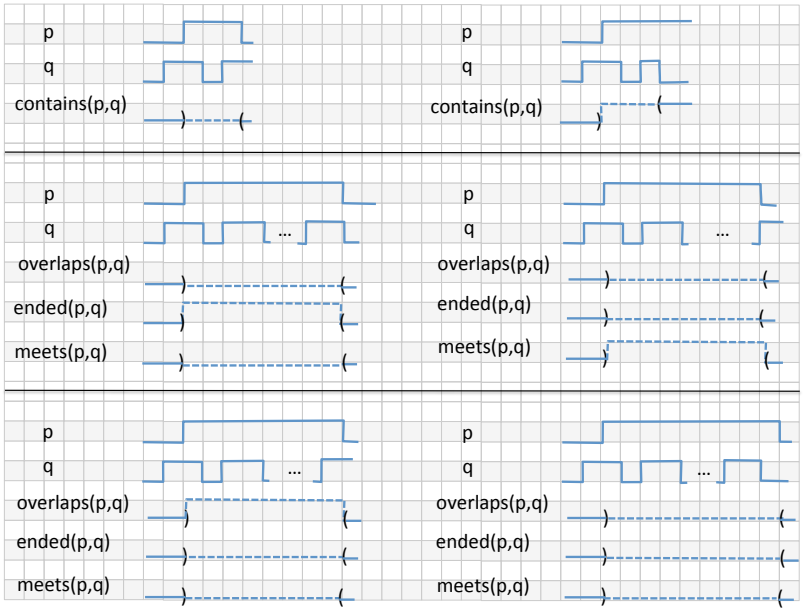

Fig. 5. Semi-causal Allen-based operators of orders greater than 1.

time-dependent propositions directly correspond to our signals, it is most natural to define similar operators in our language. Actually, we added three related operators:

- holds(p,q) = $\{I \in q \mid \text{Holds(p, I)}\}$
- occurs(p,q) = $\{I \in q \mid \text{Occurs(p, I)}\}$
- occurred(p,q) = $\{[t_1, t_e) \mid \exists [t_s, t_e) \in \text{occurs(p,q)}\ \exists t_1 \in [t_s, t_e)\ .\ p(t_1) = 1 \wedge \text{Holds}(\neg \text{p}, [t_s, t_1))\}$

The first two operators select the states in q where p holds all the time, respectively occurs at least once. Operator occurred(p,q) is a stricter version of occurs(p,q), in the sense that occurred(p,q)(t)=1 → occurs(p,q)(t)=1, which selects only the ending sub-intervals of occurs(p,q) when p has already occurred. Thus, this operator provides some extra online information to applications, signalling moments when p happens for the first time in a state q. It can be shown that holds and occurs are semi-causal of order 1, and occurred is causal.

4.3 Parameterized Operators

The arity of an operator is the number of signals it takes as arguments. All the above operators are binary, except negation, which is a unary operator. In our language, operators can take not only signals as arguments, but also other parameters, valued as integer constants.

4.3.1 Temporal Operators. Parameterized operators are useful for instance to provide upper or lower bounds for temporal constraints. For instance, the following unary temporal operators are parameterized by a delay $T > 0$:

- ge<T>(s) = $\{[t_s, t_e) \in s \mid t_e - t_s \geq T\}$
- le<T>(s) = $\{[t_s, t_e) \in s \mid t_e - t_s \leq T\}$
- geRT<T>(s) = $\{[t_s, t_s + T) \mid \exists [t_s, t_e) \in \text{ge<T>(s)}\}$
- delay<T>(s) = $\{[t_s + T, t_e + T) \mid \exists [t_s, t_e) \in s\}$

Operators 'ge' and 'le' (read as "greater/less or equal") select from a signal the states that are longer, respectively shorter than the given delay. Operator 'geRT' is a 'runtime' version of 'ge' selecting the same states, but shortened to exactly T. This operator provides as additional information the moments when the delay T has been exceeded. This information is useful for timely triggering alerts. For example the expression geRT<5 min>(Door) switches value from 1 to 0 every time the door has been left open for more than 5 minutes, as soon as this is detected, without waiting for the door to be closed.

Operator 'delay' simply delays a signal by T. Ironically, delay can be computed with no delay, i.e. it is causal, because it only depends on past events of the given signal. All the other temporal operators above are semi-causal of order 1. Indeed, the question to ask at the beginning of each state of s is whether the next event in s (which always is falling back to 0) happens before or after time T. This question can be decided either when this event happens before T, or at time T otherwise. Thus, the computing delay for these operators is not only bounded by 1 event, but also by time T.

4.3.2 Nullary Operators. Parameters are also useful for nullary operators, that is, taking no signal as arguments. In particular, AAL services for checking daily activities commonly use a time slot that is the same every day. Such a slot can be defined by a nullary operator generating a periodic wave signal.

- wave<T_s, T_1, T_0> = $\{[t_s, t_s + T_1) \mid t_s \geq T_s \wedge (t_s - T_s) \bmod (T_1 + T_0) = 0\}$

The produced signal is 0 until T_s, then loops by staying 1 during T_1, and 0 during T_0. Thus, a daily wakeup slot between 7AM and 9AM can be defined as wave(7AM, 2h, 22h). Obviously, the 'wave' operator is causal, as it does not depend on any input event.

4.4 User-defined Operators

Our language allows users to define new operators by composition of predefined or other user-defined operators, using the 'sub' keyword. This allows new abstractions to be defined and organized into abstraction layers, much the same way as procedures (also known as subroutines) allow this in programming languages.

For example, a very useful abstraction is one that allows to derive events from states, namely the beginning or ending of the state. Here are user-defined operators for that:

- sub up(s) = $s \wedge \neg delay\text{<1>}(s)$
- sub dn(s) = $up(\neg s)$

Operator 'up' selects the first tick of each state in s, which are the only moments where the signal is 1 but its copy delayed by 1 tick is still 0. Thus, it signals "up fronts" of a signal. Operator 'dn' indicates "down fronts" of a signal, which are exactly the up fronts of its negation.

As can be seen, events are not a native notion in our language, where sensors always provide state signals. However, events can be easily derived from states as user-defined abstractions. This is the dual of CEP languages, that on the contrary, define events as native data, and where states must be defined by aggregating events.

Like native operators, user-defined operators may also be parameterized. Here are some examples:

- sub cut<T>(p) = le<T>(p) ∨ geRT<T>(p)
- sub sustain<T>(p) = p ∨ met(cut<T>(¬p), p)

Operator 'cut' selects all the states in the given signal, shortened to T if longer. Operator 'sustain' extends all states in the given signal p by T. It does this by adding states of $\neg p$, cut to T. The 'met' operator is needed only to drop an eventual state of $\neg p$ starting at $t = 0$, by ensuring that these state extensions are met by a previous state of p. Note that 'sustain' collapses states distanced by less than time T.

4.5 First-Class Operators

Operators are first-class objects in our language. This means that they can be constructed on the fly and passed as arguments to higher-order operators such as 'map' or 'reduce'. For example:

- sub either(@lst) = reduce(sub(p,q) p ∨ q, @lst)
- sub either_up(@lst) = either(map(sub(p) up(p), @lst))
- sub either_dn(@lst) = either(map(sub(p) dn(p), @lst))

Operator 'either' defines an n-ary 'or', operating on a list of signals. List variables are prefixed by the character '@'. It passes an anonymous user-defined operator to the predefined 'reduce' operator, that folds a binary operator over a list of arguments. Operator 'either_up' detects any up front in a list of signals, using the n-ary or defined above, and the predefined 'map' operator, which applies an operator to each element in a list of signals. Operator 'either_dn' is similar.

For instance, assuming predefined list variables for a home giving the list of different sensors, e.g., @Any_Commfailure with the list of all sensors signalling wireless communication faults (there is one such sensor for each wireless sensor), and @Any_Batterylevel with the list of all sensors signalling battery levels (there is one such sensor for each battery-powered sensor), it is easy to watch for any up front of any sensor of a certain kind, for instance either_up(@Any_Commfailure).

4.6 Local Variables

Sometimes a sub-expression is repeated several times within a bigger expression, but it is not general enough for constituting a user-defined abstraction. In this case, a local variable can be used in our language to avoid repeatedly computing this sub-expression. For example:

- let slot = wave<7AM,2h,22h> in
  occurred(up(Cupboard), slot) ∧
  occurred(up(Coffeemaker), slot)

5 IMPLEMENTATION

We implemented a prototype interpreter of our language for online state processing. The interpreter takes two arguments: a DSL expression and a log file (in JSON format), and outputs the signal computed for the expression over the log, that is, the list of time points when the signal switches from 0 to 1 and vice versa. The implementation is written in Perl and consists of about 4000 lines of code. It implements the language as a DSL deeply embedded

[22] in Perl. This means that the abstract syntax of DSL expressions is represented as a data structure of the host language (that is, Perl), and there are functions for compiling this abstract syntax to an internal representation, for transforming this representation, and for interpreting it over a log to produce the results. Depending on the expression, the main interpreting function calls specific functions implementing the semantics of every predefined operator. User-defined abstractions are encoded as code generators, i.e., Perl functions returning DSL abstract syntax. Thus, user-defined abstractions are first expanded to code containing only predefined operators, before the expression is evaluated. For instance, the expression:

occurred(up(Coffeemaker), ¬Kitchen)

is represented in abstract syntax as a nested array:

```
[&occurred, &up("Coffeemaker"), [&not, "Kitchen"]],
```

The first element in each array is the function implementing the corresponding operator (e.g., occurred) on the subsequent elements. The user-defined abstraction 'up' is implemented as a user-defined Perl function returning DSL abstract syntax (it may be thought of as a macro), defined as follows:

```
sub up() {
  my ($p) = @_;
  return [&and, $p, [&not, [&delay(1), $p]]];
}
```

Note that parameterized operators are implemented as higher-order functions, which return functions, such as the call to '&delay(1)' above. Thus, the technique of DSL deep embedding allows using the full power of the host language for implementing advanced code generation mechanisms. In particular, anonymous user abstractions are first-class objects because they are directly represented as Perl anonymous subroutines, the encoding being straightforward.

5.1 Online State Processing

As already mentioned in Section 4.2.1, some operators are causal, so they can be computed with no delay because they depend only on present or past events, while other operators are semi-causal of order N, so they are computed with a delay of at most N events, as they may depend on at most N events in the future. A naive interpreter would implement every semi-causal operator of order N with a delay of N, its worst case. In this scheme, every expression containing a delayed operator would never be computed in real time. Moreover, this constant delay would accumulate in the case of complex expressions, in which several such operators are nested. Our interpreter does not adopt this conservative strategy, but rather optimizes the cases in which delayed operators can be computed in real time. This is based on an online evaluation strategy that allows every operator to produce values whenever possible, while other operators are blocked waiting for future events. We illustrate this opportunistic, de-synchronized strategy on the example of a user-defined operator 'flat(p,q)'. This operator flattens together all the states of p that are strictly contained in the same state of q, as shown in Figure 6: signal 'flat(p,q)', at the top of the figure, collapses the two states of p that happen completely during the second state of q. In practice, this abstraction is useful for inferring a period of continuos activity by adding up fragments of the activity within a given slot. For instance, this operator can be used for reconstructing a shower activity out of fragmentary movements detected in the bathroom, or for reconstructing a presence in a room by putting together sparse movements in that room while no action is detected elsewhere. The operator is defined as follows:

sub flat(p,q) = let d = during(p,q) in d ∨ during(¬d,q)

This expression works as follows. The first term of the disjunction, 'd=during(p,q)' selects all the states of p completely contained in some state of q. The second term of the disjunction , 'during(¬d,q)', adds to these the holes between these states that are completely contained within a state of q. This has the effect of glueing together (or collapsing) all the pieces of p within q.

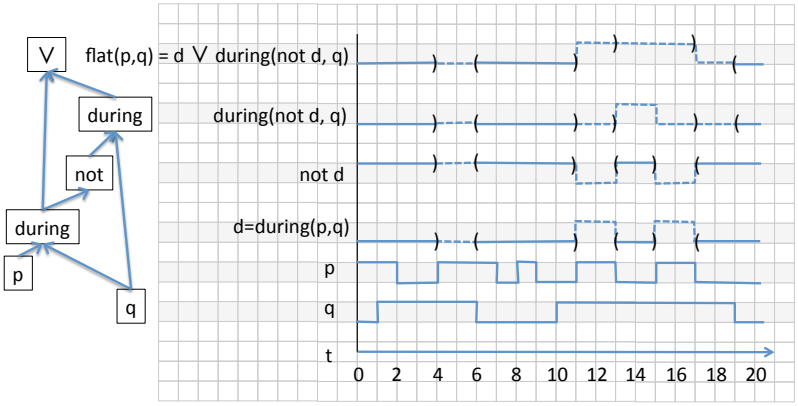

Fig. 6. Evaluation of a complex expression containing shared subtrees: flat(p,q).

This expression is represented in our interpreter by the abstract syntax tree in Figure 6 (left side), containing a shared node corresponding to the 'let' construct: the lower 'during' node. In the overall tree, the leaves are the signals p and q (which are here sensors, but could be more complex expressions), and other nodes are operators. In particular, the root of the tree represents the whole expression, whose top-level operator is 'or'. To compute the value of the whole expression as as signal, our interpreter computes a signal for each node, in a bottom-up fashion, as shown in Figure 6 (right side). All signals are computed online, as new events become available for p and q. More precisely, when time advances and an event becomes available for p and/or q, the signals of each node are updated, if possible, propagating values upwards in the tree. At any time t, signals are drawn up to various moments $t_i \leq t$. For instance, at $t = 3$, all signals are known up to t, i.e. in real time; at $t = 5$, all signals are known up to $t_i = 4$ (except the sensors p and q, which are always known in real time); at $t = 16$, signals during(p,q) and its negation are known up to $t_1 = t_2 = 15$, whereas the other two signals are known only up to $t_3 = t_4 = 13$; at time $t = 20$, all the signals are again known up to t, that is, in real time. We may notice several things in this example. Firstly, about half of the time, the value of the top-level expression is known in real time. The only times when it is delayed is between [4,6) and between [11,19). Secondly, delayed values are not necessarily computed all at once, but rather incrementally. This corresponds to closing parentheses in the middle of the dashed regions for the two top signals, when a delayed value has been computed but the signal still does not catch up real time. Thirdly, the delays of the two nested 'during' operators (each one of at most one event) are sometimes, but not always, cumulated to a delay of two events. This cumulative delay happens for the two top signals, whose values at time 13 are only computed at time 17, after two more events of p have happened. However, at time 5, their delay is of only one event, and at time 3, there is no delay at all.

By carefully handling all these delays and avoiding their cumulation when possible, our interpreter can optimize the evaluation strategy and achieve the results shown in Figure 6, where the current value of the expression is known much of the time in real time.

6 VALIDATION

As shown in Section 4, our DSL natively supports the notion of state and states combinations. As shown in Section 5, the state combination operators, including semi-causal operators, are computed online on the stream of sensor events, leading to the notion of online state processing promoted by the language and implemented

by the interpreter. Finally, the DSL supports two kinds of software reuse, that have been described in pervasive computing applications [9], where sensors are shared between several applications performing similar processing steps. In such systems, *code reuse* has been defined as instantiating common processing code multiple times, while *instance reuse* has been defined as using already instantiated processing modules. Our DSL supports code reuse with the user-defined operators, by means of the 'def' construct; these user abstractions can be instantiated multiple times with distinct arguments and/or parameters. The DSL also supports instance reuse by means of the 'let' construct, translating to shared nodes in the abstract tree, which allow the interpreter to share, instead of duplicate, computations in a complex expression (as illustrated in Figure 6).

Thus, our DSL satisfies the set of basic requirements put forward in the introduction: states, their combination, online computation, and reuse. The rest of this section validates the DSL and its implementation with respect to the further challenges described in the introduction, namely: expressiveness, efficiency, and efffectiveness.

6.1 Expressiveness

The expressiveness challenge stipulates that the DSL should not overly restrict the programs that can be coded in the language. In other terms, the DSL should cover a significant set of representative services in the target domain. This property of a DSL has been also called completeness [32].

To validate the expressiveness of our language, we have rewritten in it the context detection logic of all the 53 AAL services currently deployed on the HomeAssist platform. More precisely, the context detection logic of each AAL service was coded as an expression in our DSL, which signals the occurrences of the target situation by an up front or a down front, depending on the service. For space reasons, we cannot list them entirely, but a representative subset is given in Table 1. All the services left out are simple variations of the 13 services listed here. For instance, the Presence dependency service has to check correlation between any contact sensor or smart plug and the motion sensors covering their location. The example in the table has thus to be instantiated for other pairs of sensors. Other kinds of variations are due to the diversity of user routines, involving different delays, different appliances for preparing meals, or different rooms for performing daily activities.

As can be seen, all the AAL existing services can be written in our language as very concise formulas. This is partly due to the reuse of previously defined abstractions such as 'sustain', 'up/dn', and 'either/either_up/either_dn'. Moreover, these formulas are rather easy to explain. For instance, the wakeup routine is detected by extending every presence in the bedroom by 10 more minutes, and checking whether these states contain apparitions in the kitchen. This way, the presence in the kitchen is within 10 minutes from the presence in the bedroom, which corresponds to the specification. The use of 'occurred' in this service, as in many other services in the table, ensures that wakeup is signalled at most once per slot, and hence, at most once per day. Indeed, if the person repeatedly walks from the bedroom to the kitchen during this slot, the 'occurred' will go 1 the first time, and stay 1 until the end of the slot.

For verifying that these formulas give the correct results, we executed them on logs accumulated from the real home deployments during one year, and we verified that they give the same results as the original services, on one of these logs.

Note that all our versions executing on the logs can be executed online by feeding them through a pipe with the stream of events coming from a real home, rather than from an accumulated log. Thus, they could timely detect all the contexts used by the AAL services. However, these versions do not contain the action part of the deployed Java services, that notify the users and interact with them. Closing the loop back to the users by adding the action parts is beyond the scope of this paper, and will be studied in future work.

Once the results were validated, we could also compare the efficiency of our interpreter with that of an alternative CEP implementation of these AAL services, as described in the next section.

Table 1. The main AAL services of the HomeAssist experiment.

Service name	**Service description**	**Implementation in DSL**
Presence dependency	Detect if cupboard status changes while no presence in the kitchen	occurred(up(Cupboard) ∨ dn(Cupboard), ¬Kitchen)
Departure alert	Detect if entrance door is opened for at least 5 minutes during night time	let Night = wave<8PM,12h,12h> in occurred(dn(geRT<5min>(Door)), Night)
Door alert	Detect if entrance door is opened for at least 5 minutes while no presence in entrance	occurred(geRT<5min>(Door ∧ ¬Entrance), Door)
Long inactivity	Detect no movement in Bedroom since 24 hours	geRT<24h>(¬Bedroom)
Fridge opened	Detect if fridge remains open at least 5 minutes	geRT<5min>(Fridge)
Breakfast	Detect cupboard and coffeemaker opening (any order) during breakfast period	let slot = wave<7AM,2h,22h> in occurred(Cupboard, slot) ∧ occurred(up(Coffeemaker), slot)
Lunch reheat	Detect fridge opening and then stove use within 10 minutes, or fridge opening during stove use; all during lunch period	let slot = wave<11AM,3h,21h> in occurred(sustain<10min>(up(Fridge)) ∧ Microwave, slot)
Dinner	Detect fridge opening and microwave use (any order) during dinner period	let slot = wave<6PM,4h,20h> in occurred(up(Fridge), slot) ∧ occurred(up(Microwave), slot)
Go to bed	Detect end of presence in bathroom and then begin of presence in bedroom within 10 minutes, during go-to-bed period	let slot = wave<9PM,3h,21h> in occurred(sustain<10min>(dn(Bathroom)) ∧ up(Bedroom), slot)
Wakeup	Detect end of presence in bedroom and then begin of presence in kitchen within 10 minutes during wakeup period	let slot = wave<6AM,4h,20h> in occurred(sustain<10min>(dn(Bedroom)) ∧ up(Kitchen), slot)
Commfailure warning	Detect any sensor that fails to communicate	either_up(@Any_Commfailure))
Commfailure alert	Detect any sensor that has failed to communicate since 24 hours	either_dn(map(sub(p) geRT<24h>(p), @Any_Commfailure))
Battery alert	Detect battery level of any sensor that become less than 20%	either_dn(@Any_Batterylevel)

6.2 Implementation Efficiency

The implementation efficiency challenge stipulates that the high-level domain abstractions offered by the DSL should not impose excessive resource consumption, potentially limiting practical applicability. Thus, resource consumption sobriety may unlock the way to using the approach in various pervasive and ubiquitous environments, some of which are severely constrained in terms of CPU power and/or memory capacity. Also, in the IoT context, lightweight solutions enable the transition from cloud computing to edge computing, for saving bandwidth and improving response time, among others [37]. For all these reasons, it is important to show that our prototype implementation competes well with respect to other established approaches, both in terms of CPU load and memory footprint.

To demonstrate that our implementation is usable in practice, we compared its efficiency with that of an existing alternative approach. We chose the CEP paradigm as a baseline because it is an established paradigm for building pervasive and IoT applications that seems the closer to our approach. In particular, it also uses a

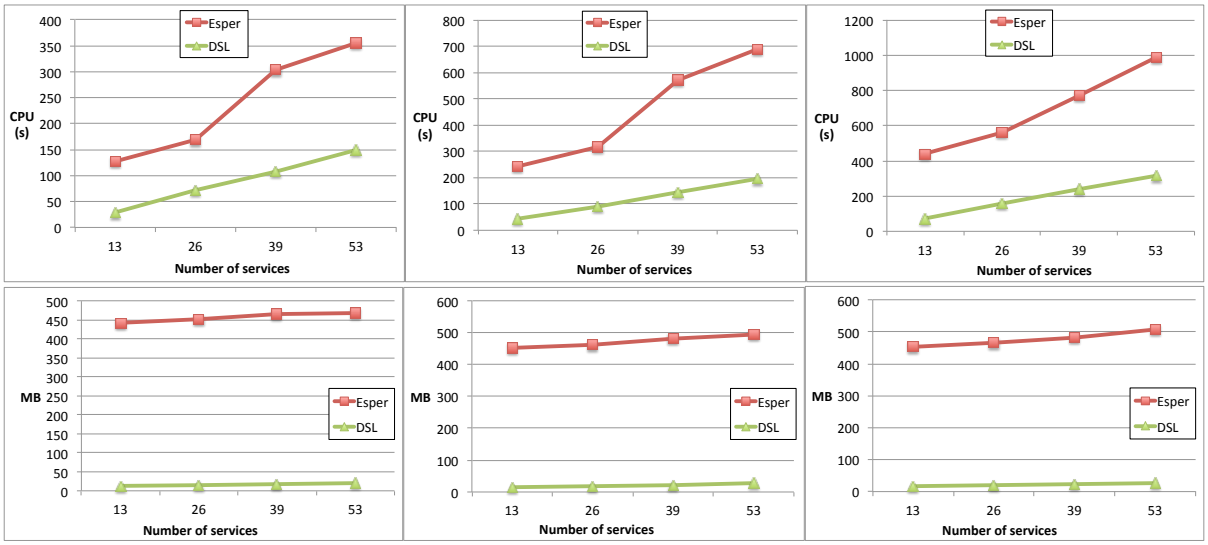

Fig. 7. CPU time (top, in seconds) and memory consumption (bottom, in MBytes) for processing a log of 1 year from three different homes (H1 left, H2 middle, and H3 right), when varying the number of services.

DSL for expressing context detection, it is evaluated online on streaming data, and it contains domain-specific optimizations for efficient execution. We chose Esper [18] as the particular CEP engine because it is a well-known, open source, commercial quality implementation of the CEP paradigm, both used in the industry and referred in many research papers.

Thus, we also implemented the 53 AAL services described above in the Esper CEP language. We compared the efficiency of our prototype interpreter and Esper on the set of AAL services according to two indicators: total processing time and memory consumption. The benchmarks were executed on real logs of one year long coming from three representative homes, called here H1, H2, and H3. The number of events in these logs heavily depends on the home configuration and user habits, and thus vary from 168,550 events (in H1), going thru 968,092 events (in H2), to 2,416,513 events (in H3). All the benchmarks were executed on a PC equipped with an Intel Core i5-3320MHz and 8Go of RAM, running the Linux kernel 4.15, Perl 5.26.1, and Esper 5.5. For a fair comparison, as our interpreter does not parallelize execution, all the Esper measurements were performed by forcing execution on a single core. Results are depicted in Figure 7.

When AAL services are executed online, processing goes on sporadically, triggered by incoming events. It is difficult to compare the performance indicators on every event. Moreover, as explained previously, our efficiency comparison aims at measuring the overall resource consumption of our approach, abstracting out from runtime execution dynamics. Thus, what we are interested in is the total CPU time spent, and the maximum memory used, for processing events over a representative period of time. This is why we rather executed the AAL services in batch mode, that is, on logs accumulated during one year. The services were executed by submitting all the events from the log with no delay. Consequently, the Esper rules using timeouts (i.e. the 'timer' and 'within' constructs) were slightly modified for batch mode to compute delays between events based on their timestamps. As far as our DSL is concerned, no such modification of the services were necessary, as our temporal operators already compute delays based on the timestamps of events. In turn, our interpreter is currently designed to evaluate a single expression. Thus, we evaluated a disjunction of all the expressionss (using the 'or' boolean operator). As

already explained in relation to Figure 6, the interpreter computes the signal of each subexpression. Thus, the signal of each service is the one computed for each child of this top-level disjunction.

The batch times indicated in Figure 7 (top) correspond to the total time for processing all the events in the corresponding log. The memory consumptions indicated in Figure 7 (bottom) correspond to the maximum memory size consumed during the processing of the whole corresponding log. Obviously, resource consumption (CPU time and memory) for processing incoming events depends on the number of services applied to them. To test the scalability of our implementation when the number of services increases, we divided our set of 53 services in 4 roughly equivalent subsets, including more and more variations of the 13 base services defined in Table 1. Thus, we measured resource consumption when including 13, 26, 39, and the full set of 53 services.

As can be observed, our interpreter is always faster then Esper, on average 3.7 times faster on these cases. It also consumes much less memory, on average 25 times less. Both processing time and memory consumption tend to increase according to linear patterns with the number of the applied services. These results can be considered as very encouraging, as our prototype interpreter is compared to an established, commercial quality CEP engine. The savings in both CPU load and memory usage suggest that processing could be migrated from the cloud into each home (e.g., on the Vera or Internet boxes).

Note that these improvements may come either from differences between the languages or approaches (our DSL vs. Esper) or from the underlying implementation platform (Perl vs. Java). Thus, we do not conclude that our DSL is intrinsically lighter than Esper. However, the source of the savings is unimportant from a practical point of view. The figures only demonstrate that our DSL *can* be implemented with better resource consumption than a widely used existing tool, and thus is applicable to less powerful computing devices.

6.3 Conciseness

The conciseness challenge concerns the effort required from developers to solve problems in the target domain. The number of lines of code for solving a domain problem is commonly used for roughly estimating the programming effort from the user, when comparing a DSL with a GPL (e.g., [26]) or between different DSLs (e.g., [28]). Validating our DSL along this axis may be done by comparing it with an established approach. In principle, the IFTTT and the CEP approached could be relevant as a baseline, because they are DSLs aimed at simplifying the development of context detection components, albeit based on events rather than states. They both aim to encode context detectors in a way much more concise than in a GPL.

However, IFTTT falls short as a baseline because it only allows to express single-trigger rules, based on a single event. Only rules "Commfailure warning" and "Battery alert" can be expressed this way. For instance, rule "Commfailure alert" is not expressible because it involves a sensor state and a temporal constraint, which can only be encoded by a start event, an end event, and a delay in between. This is why we considered again the CEP approach as the baseline.

Table 2 gives the number of effective lines of code (excluding comments, blank lines, and parentheses) for expressing the 13 AAL main services, in our DSL, and in Esper. As can be seen, the DSL form is on average 4 times more concise than the Esper form. However, this average hides some non-linear variations. Thus, very simple services such as ''Commfailure warning" or "Battery alert", consisting in testing a single sensor, are expressed as easily in both DSLs. Mid-complexity services testing one sensor with one time constraint, such as "Long inactivity", are moderately more verbose in Esper because the notion of state has to be encoded as a sequence of related start and end events, cumulated with an explicit timer event:

```
select Absence from pattern [
  every Absence=Event(location='Bedroom', type='Presence', status='false') ->
    timer:interval(24hours) and not ( Event(location='Bedroom', type='Presence', status='true') )  ]
```

Note that '`->`' is read '*followed-by*', and '`and not`' is used for expressing '*before*'. On the other extreme, services testing a combination of several sensors and cumulating temporal or time ordering constraints, such as "Door

Table 2. Code size comparison for the AAL services.

Service name	DSL (LOC)	Esper (LOC)
Presence dependency	1	5
Departure alert	2	6
Door alert	1	14
Long inactivity	1	3
Fridge opened	1	3
Breakfast	2	11
Lunch reheat	2	12
Dinner	2	5
Go to bed	2	7
Wakeup	2	7
Commfailure warning	1	1
Commfailure alert	1	4
Battery alert	1	1
Total	19	79

alert" or "Breakfast", show a much higher code size ratio, ranging between 6 and 14. In these cases, the Esper form has to explicitly list all the various event patterns that may lead to a given situation. For instance, in "Breakfast", either the cupboard is open during the breakfast slot, to extract a cup from it, or the cupboard was already open before the breakfast slot starts, which allows to extract the cup without signalling a cupboard open event.

```
select Slot, Open, Coffee from pattern [
  ( every Slot=Event(type='Calendar', status!='end') ->
      Open=Event(type='Cupboard', status='open')
      and Coffee=Event(type='CoffeeMaker', status='on')
      and not ( Event(type='Calendar', status='end') ) )
  or
  ( every Open=Event(type='Cupboard', status='open') ->
      Slot=Event(type='Calendar', status!='end')
      and not ( Event(type='Cupboard', status='close') ) ->
        Coffee=Event(type='CoffeeMaker', status='on')
        and not ( Event(type='Calendar', status='end') ) )  ]
```

In the DSL form, the two cases are unified by the 'occurred(Cupboard, slot)' operator, compactly requiring that the cupboard appears as open at some moment during the slot. Another reason for which the DSL form is shorter is the reuse of user-defined operators, as already mentioned. These examples also show that, besides being more verbose, the Esper code also is lower level, manipulating sequences of events and explicit timers, instead of states and temporal operators.

Both the quantitative comparison, concerning code size, and the qualitative comparison, concerning the discourse level, suggest that our DSL is easier to use than Esper for this set of real AAL problems.

7 DISCUSSION

We discuss here some limitations and possible extensions of our approach concerning non-binary sensors, and its relationship to probabilistic approaches for activity recognition.

7.1 Handling Non-binary Sensors

The core of our approach is limited to handling binary sensors, because operators are defined on the closed domain of boolean signals (not limited to causal operators). Non-boolean sensors can be handled only if they are translated as boolean signals in a pre-processing layer. Thus, we already use non-boolean sensors in the HomeAssist rules, associated to a threshold value: the smart plugs, and the battery level indicators. Thus, their value domain is quantized in two sub-domains, respectively higher and lower than the fixed threshold. It would be possible to handle non-boolean sensors in a slightly more general way, by dividing the value domain using N threshold values, and pre-processing the sensor into N boolean signals. Then, user-defined operators could be used to define specific value ranges. For instance, if a light sensor is pre-processed into boolean signals 'Light_gt_20' and 'Light_gt_200', one could define dim light as a value between 20 and 200 lux as follows:

- sub dim_light = Light_gt_20 $\wedge$ $\neg$ Light_gt_200

This user-defined operator could then be correlated with other sensors such as the motion detector in the bedroom to detect for example pre-sleeping activities. Such preprocessing can allow to handle many practical situations.

A more fundamental limitation concerns the output signals, which must also be binary in our approach. For instance, adding a native operator comparing two given non-binary input signals and returning 1 when the first signal is greater than the second signal is easy to implement in our framework. On the other hand, an operator computing the average between two non-binary input signals cannot be integrated easily. In general, non-binary signals may require a wide variety of numerical operators, and defining a a relevant set of core operators would probably depend on a specific application domain. In contrast, for binary sensors it is easier to define such a core subset, guided by their common notions of state, current state, and temporal relationships on state intervals.

7.2 Relation with Probabilistic Approaches

Our approach is targeted at easily specifying complex conditions about multiple sensors. It is thus convenient for detecting situations which can be described by an explicit, deterministic, formula based on the sensor values, including past, current, and future values. This approach perfectly fits, for instance, activity verification, in which the way of performing a certain activity by a given user can be declared by such an explicit formula over a limited number of high-level markers, such as using an appliance, opening a door, etc

In contrast to that, activity recognition based on machine learning [1] addresses situations that cannot be easily described in a simple and deterministic way. In other terms, the explicit formula based on sensor values is difficult to determine a priori, or impossible/impractical to express as a combination of sensor values. Learning-based approaches use a probabilistic approach, and require a great amount of annotated data for training a recognizer. Therefore, this approach performs very well in applications over sensor infrastructure providing high-volume, low-level data, such as recognizing a physical activity based on wearable accelerometers providing fine-grained motricity data.

Our approach is not applicable in such cases. On the other hand, in cases where an explicit formula can be found, it allows to implement customized detectors with no training data. The different strengths of the two approaches could be seen as complementary. Indeed, a probabilistic recognizer of physical activities could usefully be integrated as a binary sensor in our DSL to specify higher-level activity recognition. For instance:

- sub toothbrushing<T> = occurred(ge<T>(BrushMovements), Bathroom)

8 CONCLUSION

We presented the concept and implementation of a domain-specific language for processing online events coming from binary sensors, validated within the domain of AAL services. The language aims at addressing some important issues of alternative approaches when applied to such binary sensors.

Firstly, our language provides a primitive notion of state, modeled as a boolean signal over time, and allows building complex conditions on several states using signal operators. Our model does not include instantaneous events, but these can be derived from states by user-defined operators that detect the beginning and end of a state.

Secondly, it innovates by introducing operators that are not always causal, but which can provide results in many cases without delay, by case-specific optimizations. This design sacrifices strict causality to provide an occasionally delayed, but retrospectively continued signal. Therefore, the standard boolean operators and, or, and not are freely applicable to any signals, with their usual semantics. In contrast, in CEP languages for instance, the And operator is a relational join, and negation can only be applied in specific combinations: for instance, checking the absence of event e between events e_1 and e_2.

Thirdly, our language introduces first class operators for combining signals, which can be defined by the user, parameterized, and passed as arguments to other operators.

We validated the expressiveness of our language on a set of 53 real assistive services. These services were originally developed in Java and deployed in many real homes. We rewrote these services both in our language and in the Esper CEP language, and we reexecuted those services on one of the logs accumulated during one year, with the same results as the original services. Furthermore, we compared the performance of the rewritten services in our language and in Esper, and shown that a prototype implementation of our language compares well with this commercial-grade CEP implementation, both in terms of total CPU load and maximum memory consumption. This demonstrates that our approach is usable in practice. The savings in both total CPU load and memory usage also suggest that it is applicable to a larger class of computing environments, compared to CEP, towards the edge of the cloud. We also showed that our DSL allows to encode context detectors more concisely than CEP, partly due to the native domain concepts such as states, and partly due to effective code reuse.

Our approach is well suited for detecting situations that can be described as exact combinations of boolean states. Thus, the approach is intrinsically targeted to binary sensors. However, non-binary sensors can be integrated to some extent by quantizing their outputs into binary signals, using threshold values. Software sensors implemented by more complex technologies, such as statistical recognizers based on machine learning, or CEP-based modules, can also be integrated as boolean sensors, provided there is enough training data, or computing capacity respectively.

In future work, it would be useful to investigate a minimal subset of the operators to be implemented as predefined in our language, so that all the other operators be defined in a library. On the applications side, we plan to implement more services, for instance to detect 'weak signals' in AAL logs, signalling long-terms evolutions such as signs of decline due to aging. There is a real need expressed by experts in aging for such services. Such applications could advance our language to include new operators, e.g. performing statistical analyses over a signal. One challenge is to express such operators as boolean-valued, to preserve composability.

Our DSL simplifies the development of ubiquitous applications over binary sensors, by allowing programmers to specify context detection logic as concise formulas. This could be a promising base for designing an end-user development language. As a first step in this direction, we plan to evaluate the readability of formulas in our language by non-technical stakeholders in the aging domain, such as caregivers.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their detailed and constructive comments, which allowed to substantially ameliorate the paper.

REFERENCES

[1] Zahraa Said Abdallah, Mohamed Medhat Gaber, Bala Srinivasan, and Shonali Krishnaswamy. 2015. Adaptive mobile activity recognition system with evolving data streams. *Neurocomputing* 150 (2015), 304 – 317. https://doi.org/10.1016/j.neucom.2014.09.074 Bioinspired

and knowledge based techniques and applications The Vitality of Pattern Recognition and Image Analysis Data Stream Classification and Big Data Analytics.

[2] Raman Adaikkalavan and Sharma Chakravarthy. 2003. SnoopIB: Interval-Based Event Specification and Detection for Active Databases. In *Advances in Databases and Information Systems*, Leonid Kalinichenko, Rainer Manthey, Bernhard Thalheim, and Uwe Wloka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–204.

[3] Unai Alegre, Juan Carlos Augusto, and Tony Clark. 2016. Engineering context-aware systems and applications: A survey. *Journal of Systems and Software* 117 (2016), 55 – 83. https://doi.org/10.1016/j.jss.2016.02.010

[4] James F. Allen. 1983. Maintaining Knowledge About Temporal Intervals. *Commun. ACM* 26, 11 (Nov. 1983), 832–843. https://doi.org/10.1145/182.358434

[5] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. 2010. A Rule-based Language for Complex Event Processing and Reasoning. In *Proceedings of the Fourth International Conference on Web Reasoning and Rule Systems (RR'10)*. Springer-Verlag, Berlin, Heidelberg, 42–57. http://dl.acm.org/citation.cfm?id=1894568.1894575

[6] Johan Bengtsson and Wang Yi. 2004. *Timed Automata: Semantics, Algorithms and Tools.* Springer Berlin Heidelberg, Berlin, Heidelberg, 87–124. https://doi.org/10.1007/978-3-540-27755-2_3

[7] T. Bourke and A. Sowmya. 2010. Delays in Esterel. In *SYNCHRON 2009 (Dagstuhl Seminar Proceedings)*, Albert Benveniste, Stephen A. Edwards, Edward Lee, Klaus Schneider, and Reinhard von Hanxleden (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2010/2434

[8] Yann Busnel, Leonardo Querzoni, Roberto Baldoni, Marin Bertier, and Anne-Marie Kermarrec. 2011. Analysis of Deterministic Tracking of Multiple Objects using a Binary Sensor Network. *ACM Transactions on Sensor Networks* 8 (2011), 0. https://hal.inria.fr/inria-00590873

[9] Guanling Chen, Ming Li, and David Kotz. 2008. Data-centric middleware for context-aware pervasive computing. *Pervasive and Mobile Computing* 4, 2 (2008), 216 – 253. https://doi.org/10.1016/j.pmcj.2007.10.001

[10] Charles Consel, Lucile Dupuy, and Hélène Sauzéon. 2017. HomeAssist: An Assisted Living Platform for Aging in Place Based on an Interdisciplinary Approach. In *Proceedings of the 8th International Conference on Applied Human Factors and Ergonomics (AHFE 2017)*. Springer.

[11] Joëlle Coutaz and James L Crowley. 2016. A First-Person Experience with End-User Development for Smart Homes. *IEEE Pervasive Computing* 15 (May 2016), 26 – 39. https://doi.org/10.1109/MPRV.2016.24

[12] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, Jean Bacon, Peter R. Pietzuch, Joe Sventek, and Ugur Çetintemel (Eds.). ACM, 50–61. https://doi.org/10.1145/1827418.1827427

[13] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* 44, 3, Article 15 (June 2012), 62 pages. https://doi.org/10.1145/2187671.2187677

[14] Miyuru Dayarathna and Srinath Perera. 2018. Recent Advancements in Event Processing. *ACM Comput. Surv.* 51, 2, Article 33 (Feb. 2018), 36 pages. https://doi.org/10.1145/3170432

[15] Dan Ding, Rory A. Cooper, Paul F. Pasquina, and Lavinia Fici-Pasquina. 2011. Sensor technology for smart homes. *Maturitas* 69, 2 (2011), 131 – 136. https://doi.org/10.1016/j.maturitas.2011.03.016

[16] Lucile Dupuy, Charles Consel, and Hélène Sauzeon. 2016. Self Determination-Based Design To Achieve Acceptance of Assisted Living Technologies For Older Adults. *Computers in Human Behavior* 65 (Sept. 2016). https://doi.org/10.1016/j.chb.2016.07.042

[17] Lucile Dupuy, Charlotte Froger, Charles Consel, and Hélène Sauzéon. 2017. Everyday Functioning Benefits from an Assisted Living Platform amongst Frail Older Adults and Their Caregivers. *Frontiers in Aging Neuroscience* 9 (Sept. 2017), 1–12. https://doi.org/10.3389/fnagi.2017.00302

[18] EsperTech. [n. d.]. Esper Reference. http://esper.espertech.com/release-5.5.0/esper-reference/html/index.html. Accessed: 2018-05-12.

[19] Cyril Faucher, Jean-Yves Lafaye, and Frédéric Bertrand. 2012. *Putting Non Convex Interval Mutual Relation Models into Practice.* Technical Report. Université de La Rochelle, France. https://hal.archives-ouvertes.fr/hal-00685182 22 pages.

[20] Abdoulaye Gamatié. 2010. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification.* Springer New York, New York, NY, Chapter Synchronous Programming: Overview, 21–39. https://doi.org/10.1007/978-1-4419-0941-1_2

[21] Malik Ghallab and Amine Mounir Alaoui. 1989. Managing Efficiently Temporal Relations Through Indexed Spanning Trees. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2 (IJCAI'89)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1297–1303. http://dl.acm.org/citation.cfm?id=1623891.1623963

[22] Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 339–347. https://doi.org/10.1145/2628136.2628138

[23] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (June 1987), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9

[24] Justin Huang and Maya Cakmak. 2015. Supporting Mental Model Accuracy in Trigger-action Programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 215–225.

https://doi.org/10.1145/2750858.2805830
[25] David Janin and Bernard Paul Serpette. 2016. *Timed Denotational Semantics for Causal Functions over Timed Streams.* Research Report. LaBRI - Laboratoire Bordelais de Recherche en Informatique. https://hal.archives-ouvertes.fr/hal-01402209
[26] GabriÃ Konat, Michael J Steindorfer, Sebastian Erdweg, Eelco Visser, et al. 2018. PIE: A Domain-Specific Language for Interactive Software Development Pipelines. *The Art, Science, and Engineering of Programming Journal* 2, 3 (2018). https://doi.org/10.22152/programming-journal.org/2018/2/9
[27] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. 2016. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology* 71 (2016), 77 – 91. https://doi.org/10.1016/j.infsof.2015.11.001
[28] Tomaž Kosar, Pablo E. Martınez López, Pablo A. Barrientos, and Marjan Mernik. 2008. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology* 50, 5 (2008), 390 – 405. https://doi.org/10.1016/j.infsof.2007.04.002
[29] Narayanan C. Krishnan and Diane J. Cook. 2014. Activity recognition on streaming sensor data. *Pervasive and Mobile Computing* 10 (2014), 138 – 154. https://doi.org/10.1016/j.pmcj.2012.07.003
[30] Ming Li, Murali Mani, Elke A. Rundensteiner, and Tao Lin. 2011. Complex Event Pattern Detection over Streams with Interval-based Temporal Semantics. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System (DEBS '11)*. ACM, New York, NY, USA, 291–302. https://doi.org/10.1145/2002259.2002297
[31] Gerard Ligozat. 1991. On Generalized Interval Calculi. In *Proceedings of the 9th National Conference on Artificial Intelligence, Anaheim, CA, USA, July 14-19, 1991, Volume 1.*, Thomas L. Dean and Kathleen McKeown (Eds.). AAAI Press / The MIT Press, 234–240. http://www.aaai.org/Library/AAAI/1991/aaai91-037.php
[32] Arne Nordmann, Nico Hochgeschwender, Dennis Leroy Wigand, and Sebastian Wrede. 2016. A Survey on Domain-Specific Modeling and Languages in Robotics. *Journal of Software Engineering in Robotics* 7, 1 (2016), 75–99.
[33] Fco. Javier Ordóñez, Paula de Toledo, and Araceli Sanchis. 2013. Activity Recognition Using Hybrid Generative/Discriminative Models on Home Environments Using Binary Sensors. *Sensors* 13, 5 (2013), 5460–5477. https://doi.org/10.3390/s130505460
[34] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. 2014. Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys Tutorials* 16, 1 (First 2014), 414–454. https://doi.org/10.1109/SURV.2013.042313.00197
[35] Grigore Roşu and Saddek Bensalem. 2006. Allen Linear (Interval) Temporal Logic – Translation to LTL and Monitor Synthesis. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 263–277.
[36] J. Saives, C. Pianon, and G. Faraut. 2015. Activity Discovery and Detection of Behavioral Deviations of an Inhabitant From Binary Sensors. *IEEE Transactions on Automation Science and Engineering* 12, 4 (Oct 2015), 1211–1224. https://doi.org/10.1109/TASE.2015.2471842
[37] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (Oct 2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198
[38] Lénaïc Terrier, Alexandre Demeure, and Sybille Caffiau. 2017. CCBL: A Language for Better Supporting Context Centered Programming in the Smart Home. In *The 9th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (PACM on Human-Computer Interaction)*, Vol. 1. Lisbonne, Portugal. https://hal.archives-ouvertes.fr/hal-01534805
[39] Lenaïc Terrier, Alexandre Demeure, and Sybille Caffiau. 2017. CCBL: A new language for End User Development in the Smart Homes. In *Proceedings of IS-EUD 2017*. 82–87. https://pure.tue.nl/ws/files/69763287/IS_EUD2017_extended_abstracts.pdf#page=83
[40] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. 2014. Practical Trigger-action Programming in the Smart Home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 803–812. https://doi.org/10.1145/2556288.2557420
[41] Le Yi Wang, Ji-Feng Zhang, and G. G. Yin. 2003. System identification using binary sensors. *IEEE Trans. Automat. Control* 48, 11 (Nov 2003), 1892–1907. https://doi.org/10.1109/TAC.2003.819073
[42] D. H. Wilson and C. Atkeson. 2005. Simultaneous Tracking and Activity Recognition (STAR) Using Many Anonymous, Binary Sensors. In *Pervasive Computing*, Hans W. Gellersen, Roy Want, and Albrecht Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 62–79.

Received May 2018; revised August 2018; accepted October 2018