International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Bringing Runtime Verification Home

A Case Study on the Hierarchical Monitoring of Smart Homes using Decentralized Specifications

Antoine El-Hokayem · Yliès Falcone

Received: date / Accepted: date

Abstract We use runtime verification (RV) to check various specifications in a smart apartment. The specifications can be broken down into three types: behavioral correctness of the apartment sensors, detection of specific user activities (known as activities of daily living), and composition of specifications of the previous types. The context of the smart apartment provides us with a complex system with a large number of components with two different hierarchies to group specifications and sensors: geographically within the same room, floor or globally in the apartment, and logically following the different types of specifications. We leverage a recent approach to decentralized RV of decentralized specifications, where monitors have their own specifications and communicate together to verify more general specifications. We leverage the hierarchies, modularity and re-use afforded by decentralized specifications to: (1) scale beyond existing centralized RV techniques, and (2) greatly reduce computation and communication costs.

Keywords runtime verification · monitoring · decentralized systems · decentralized specifications · smart home · internet of things

This work is supported by CPS4EU, a project funded from the H2020-ECSEL-2018-IA call – Grant Agreement number: 826276. The authors thank the Amiqual4Home (ANR-11-EQPX-0002) team for providing the collected data.

Antoine El-Hokayem
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG,
38000 Grenoble, France
E-mail: antoine.el-hokayem@univ-grenoble-alpes.fr

Yliès Falcone
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG,
38000 Grenoble, France
E-mail: ylies.falcone@univ-grenoble-alpes.fr

1 Introduction

Sensors and actuators are used to create "smart" environments which track the data across sensors and human-machine interaction. One particular area of interest consists of homes (or apartments) equipped with a myriad of sensors and actuators, called *smart homes* [24]. Smart homes are capable of providing added services to users. These services rely on detecting the user behavior and the context of such activities [19], typically detecting activities of daily living (ADL) [58,21] from sensor information. Detecting ADL allows to optimize resource consumption (such as electricity [1]), improve the quality of life for the elderly [51] and users suffering from mild impairment [59].

Relying on information from multiple sources and observing behavior is not just constrained to activities. It is also used with techniques that verify the correct behavior of systems. *Runtime Verification* (RV) [43,49,6,7,8] is a lightweight formal method which consists in verifying that a run of a system is correct wrt a specification. The specification formalizes the behavior of the system typically in logics (such as variants of Linear Temporal Logic, LTL [55]) or finite-state machines. Based on the provided specification, monitors are automatically synthesized to run alongside the system and verify whether or not the system execution complies with the specification. RV techniques have been used for instance in the context of automotive [23] and medical [50] systems. In both cases, RV is used to verify communication patterns between components and their adherence to the architecture and their formal specifications.

While RV can be used to check that the devices in a smart home are performing as expected, we show it can be extended to monitor ADL, and complex behavior on the activities themselves. We identify three classes of specifications for applying RV to a smart home. The first class pertains to the system behavior. These specifications are used to

check the correct behavior of the sensors, and detect faulty sensors. Ensuring that the system is behaving correctly is what is generally checked when performing RV. However, it is also possible to use RV to verify other specifications. The second class consists of specifications for detecting ADL, such as detecting when the user is cooking, showering or sleeping. The third class pertains to user behavior. These specifications can be seen as meta-specifications for both system correctness and ADL, they can include safety specifications such as ensuring that the user does not sleep while cooking, or ensuring that certain activities are only done under certain conditions.

However, standard RV techniques are not directly suitable to monitor the three classes of specifications. This is mainly due to scalability issues arising from the large number of sensors, as typically RV techniques rely on a large formula to describe specifications. Synthesizing centralized monitors from certain large formulas considered in this paper is not possible using the current tools. Instead, we make use of RV with decentralized specifications [29,32], as it allows monitors to reference other monitors in a hierarchical fashion. The advantage of this is twofold. First, it provides an abstraction layer to relate specifications to each other. This allows specifications to be organized and changed without affecting other specifications, and even to be expressed with different specification languages. Second, it leverages the structure and layout of the devices to organize the hierarchies. On the one hand, we have a geographical hierarchy resulting from the spacial structure of the apartment from a given device, to a room, a floor, or the full apartment. On the other hand, we have a logical hierarchy defined by the interdependence between specifications, i.e. ADL, specifications that use other ADL specifications, and specifications that combine sensor safety with ADL specifications. For example, informally, consider checking two activities: sleeping and cooking, which can be expressed using formulae $\varphi_{\mathtt{s}}$ and $\varphi_{\mathtt{c}}$ respectively. A monitor that checks whether the user is sleeping and cooking requires to check $\varphi_{\mathtt{s}} \wedge \varphi_{\mathtt{c}}$ and as such will replicate the monitoring logic of another monitor that checks $\varphi_{\mathtt{s}}$ alone, instead of re-using the output of that monitor. The formula will be written twice, and changing the formula for detecting sleeping requires changing the formula for the monitor that checks both specifications.

At this point we mention that RV with decentralized specifications resembles other RV techniques that distributes the monitoring process such as decentralized RV for synchronous [12,22,47] or asynchronous [53,17,38] systems, and predicate detection in distributed systems [54]. While such approaches do consider monolithic specifications (and proceed to split them accordingly), they are less restrictive on the assumptions on the monitoring architecture and communication. It is the previously described setting of RV with decentralized specifications that allows the approach of this paper to scale in the context of smart apartments.

Overall, we see our contributions as follows[1]:

- We apply RV with decentralized specifications to analyze traces of over 36,000 timestamps spanning 27 sensors in a real smart apartment (Sect. 2.1).
- We show how to go beyond system properties, to specify ADL using RV, and more complex interdependent specifications defined on up to 27 atomic propositions (Sect. 2.2).
- We leverage the hierarchies, modularity and re-use afforded by decentralized specifications (Sect. 3) to both be able to synthesize monitors and to reduce overhead when monitoring complex interdependent specifications (Sect. 6.1).
- We improve the existing data structures used for monitoring decentralized specifications, to account for large traces (Sect. 5).
- We use RV to effectively monitor ADL and identifying some insights and limitations inherent to using formal LTL specifications to determine user behavior (Sect. 6.2).
- We elaborate on the advantages of modularity by adapting parts of the specification to the *Activity Recognition with Ambient Sensing* (ARAS) [2] dataset (Sect. 6.3).

This paper extends existing work published in the proceedings of the the international conference on Runtime Verification (RV 2018) [31] with the following:

- Providing a more detailed explanation of decentralized specifications and their dependency hierarchies (Sect. 3.2);
- Providing full details on trace generation, sensor polling, and trace replay using `THEMIS` (Sect. 4);
- Enhancing the existing data structures of [29] to support large traces, by elaborating on data structures, their operations, and strategies for garbage collection and lazy evaluation in Sect. 5;
- Extending the evaluation section to include additional days where the trace is replayed, to illustrate changes in user behavior in Sect. 6.2, adding more details for modifying the specification to improve precision and recall, and also illustrating adaptability to new environments by porting the specification to the ARAS dataset in Sect. 6.3.

2 Writing Specifications for the Apartment

2.1 Devices and Organization

We consider an actual apartment, with multiple rooms, where activities are logged using sensors. Amiqual4Home [48] is an experimental platform consisting of a smart apartment, a rapid prototyping platform, and tools for observing human activity.

[1] An artifact [28] that contains data, documentation, and software, is provided to replicate and extend on the work.

2.1.1 Overview of Amiqual4Home

The Amiqual4Home apartment is equipped with 219 sensors and actuators spread across 2 floors. Amiqual4Home uses the OpenHab 6 integration platform for all the sensors and actuators installed. Sensors communicate using the KNX, MQQT and UPnP protocols sending measurements to OpenHab over the local network, so as to preserve privacy. The general layout of the apartment consists of 2 floors: the ground and first floors. On the ground floor (resp. first floor), we have the following rooms: `entrance`, `toilet`, `kitchen`, and `livingroom` (resp. `office`, `bedroom`, and `bathroom`). Between the two floors, there is a connecting `staircase`. This layout reveals a tree-like geographical hierarchy of components, where we can see the rooms at the leaves, grouped by floors then the whole apartment. While in effect all device data is fed to a central observation point, it is reasonable to consider the hierarchy in the apartment as a simpler model to consider hierarchies in general, as one is bound to encounter a hierarchy at a higher level (from houses, to neighborhoods, to smart cities, etc.). Furthermore, hierarchies appear when integrating different providers for devices in the same house.

Reusing the Orange4Home Dataset

Amiqual4Home has been used to generate multiple datasets that record all sensor data, this includes an ADL recognition dataset [48] (ContextAct@A4H), and an energy consumption dataset [25] (Orange4Home). In this paper, we reuse the dataset from [25]. The case study involved a person living in the apartment and following (loosely) a schedule of activities spread out across the various rooms. The schedule was set out by the authors of [25]. Figure 1 displays the suggested schedule of activities for Tuesday, Jan 31 2017. This allows us to nicely reconstruct the schedule from the result of monitoring the sensors. Furthermore, the person living in the home provided manual annotations of the activities done, which helps us assess our specifications. We chose to use the Orange4Home dataset over the ContextAct@A4H one as it involves only one person living in the house at a time which simplifies specifying and validating specifications.

2.1.2 Monitoring Environment

In total, we formalize 22 specifications that make use of up to 27 sensors, and evaluate them over the course of a full day of activity in the apartment. That is, we monitor the house (by replaying the trace) from 07:30 to 17:30 on a given day, by polling the sensors every 1 second, creating a trace of a total of 36,000 timestamps. Specifications are elaborated in Sect. 2.2 and expressed as decentralized specifications [29] (recalled in Sect. 3.2). Traces are replayed using the `THEMIS` tool [30] which supports decentralized specifications and provides a wide range of metrics. We elaborate on the trace replay in Sect. 4.

2.2 Property Groups

We now express the specifications that describe different behaviors of components in the smart apartment. Specifications can be subdivided into 3 groups: system-behavior specifications, user-behavior specifications, and meta-specifications on both system and user behavior. The considered specifications are listed in Table 1.

2.2.1 System Behavior

The first group of specifications consists in ensuring that the system behaves as expected. That is, verifying that the sensors are working properly. These properties are the subject of classical RV techniques [34, 16] applied to systems. For the scope of this case study, we verify light switches as system properties. We verify that for a given room i, whenever the switch is toggled, then the light must turn on until the switch is turned off. We verify the property at two scopes, for a given room, and the entire apartment. While this property appears simple to check, it does highlight issues with existing centralized techniques applied in a hierarchical way. We develop the property in Sect. 3.1, and show the issues in Sect. 3.2.

2.2.2 ADL

The second group of specifications is concerned with defining the behavior of the user inferred from sensors. The sensors available in the apartment provide us with a wealth of information to determine the user activities. The list of activities of interest is detailed in [46] and includes activities such as cooking and sleeping. By correctly identifying activities, it is possible to decide when to interact with the user in a smart setting [1], provide custom care such as nursing for the elderly [51], or help users who suffer from mild impairment [59]. Inferring activities done by the user is an interesting problem typically addressed through either data-based or knowledge-based methods [21]. The first method consists in learning activity models from preexisting large-scale datasets of users' behaviors by utilizing data mining and machine learning techniques. The built models are probabilistic or statistical activity models such as Hidden Markov Model (HMM) or Bayesian networks, followed by training and learning processes. Data-driven approaches are capable of handling uncertainty, while often requiring large annotated datasets for training and learning. The second method consists in exploiting prior knowledge in the domain of interest to construct activity models directly using formal logical reasoning, formal models, and representation. Knowledge-driven approaches are semantically clear, but are typically poor at handling uncertainty and temporal information [21]. We elaborate on such limitations in Sect. 6.2. Writing specifications can be seen as a knowledge-based approach to describe the behavior of sensors. As such, we believe that runtime verification is useful to describe an activity as a specifi-

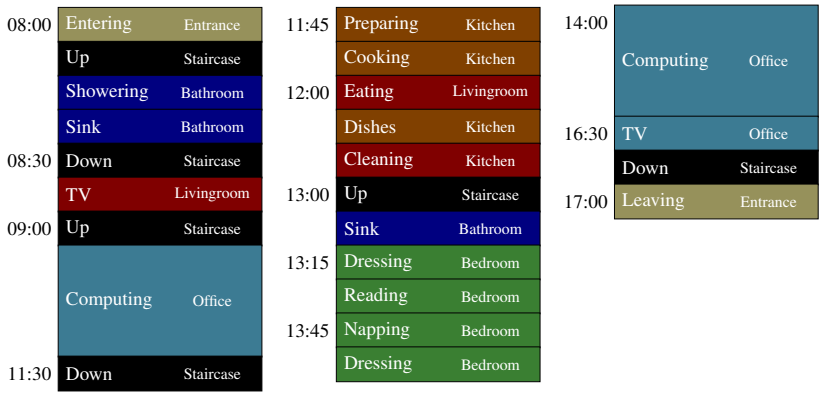


Fig. 1: Suggested Schedule (Tuesday, Jan 31 2017)

Table 1: Specifications considered in this paper. (*) indicates added ADL specifications. G indicates specification group: system (S), ADL (A), and meta-specifications (M). $|\mathrm{AP}|^{\mathrm{d}}$ (resp. $(|\mathrm{AP}|^{\mathrm{c}}$): atomic propositions needed to specify specification in decentralized (resp. centralized) specifications. d is the maximum depth of monitor dependencies.

G	Scope	Name	Description	$\lvert\mathrm{AP}\rvert^{\mathrm{d}}$	$\lvert\mathrm{AP}\rvert^{\mathrm{c}}$	d
S	Room	`sc_light`(i)	light switch turns on light ($i \in [0..3]$).	2	2	1
M	House	`sc_ok`	All light switches are ok.	4	8	2
A	Toilet	`toilet`*	Toilet is being used.	1	1	0
A	Bathroom	`sink_usage`	Sink is being used.	1	2	1
A	Bathroom	`shower_usage`	Shower is being used.	1	2	1
A	Bedroom	`napping`	Tenant is sleeping on the bed.	1	1	1
A	Bedroom	`dressing`	Tenant is dressing, using the closet.	2	3	1
A	Bedroom	`reading`	Tenant is reading.	3	5	2
A	Office	`office_tv`	Tenant is watching TV.	1	1	1
A	Office	`computing`	Tenant is using the computer.	1	1	1
A	Kitchen	`cooking`	Tenant is cooking food.	2	2	1
A	Kitchen	`washing_dishes`	Tenant is cleaning dishes.	2	3	1
A	Kitchen	`kactivity`*	Using cupboards and fridge.	4	9	1
A	Kitchen	`preparing`	Tenant is preparing to cook food.	2	11	2
A	Living	`livingroom_tv`	Tenant is watching TV.	2	2	1
A	Floor 0	`eating`	Tenant is eating on the table.	2	2	1
M	Floor 0	`actfloor`(0)	Activity triggered on floor 0.	6	16	3
M	Floor 1	`actfloor`(1)	Activity triggered on floor 1.	7	11	3
M	House	`acthouse`	Activity triggered in house	2	27	4
M	House	`notwopeople`	No 2 simultaneous activities on different floors.	2	27	4
M	House	`restricttv`	No watching TV for more than 10s.	2	3	3
M	House	`firehazard`	No cooking while sleeping.	2	3	2

cation over sensor outputs. We formalize a specification for the following ADL activities described in [25] (see Table 1). We re-use the traces to verify that our detected activities are indeed in line with the proposed schedule. Figure 2 displays the reconstructed schedule after detecting ADL with runtime verification. Each specification is represented by a monitor that outputs (with some delay) for every timestamp (second) verdicts $\top$ or $\bot$. To do this, the monitor finds the verdict for a timestamp t then respawns to monitor $t+1$. Verdict $\top$ indicates that the specification holds, that is, the activity is being performed. The reconstructed schedule shows the eventual outcome of a specification for a given timestamp ignoring delay. In reality some delay happens based on the specification itself, and the dependencies on other monitors.

2.2.3 Meta-specifications

Specifications of the last group are defined on top of the other specifications. That is, we refer to a meta-specification as a specification that defines the interactions between various specifications. While one can easily define specifications by defining predicates over existing ones, such as checking that the light switch specification holds in all rooms or whether or not detecting an activity was performed on a specific floor or globally in the house, we are more interested in specifications that relate to each other. We consider a meta-specification that reduces fire hazards in the house. In this case, we specify that the tenant should not cook and sleep at the same time, as this increases the risk of fire. In addition to mutually excluding specifications, we can also constrain the behavior of existing specifications. For ex-

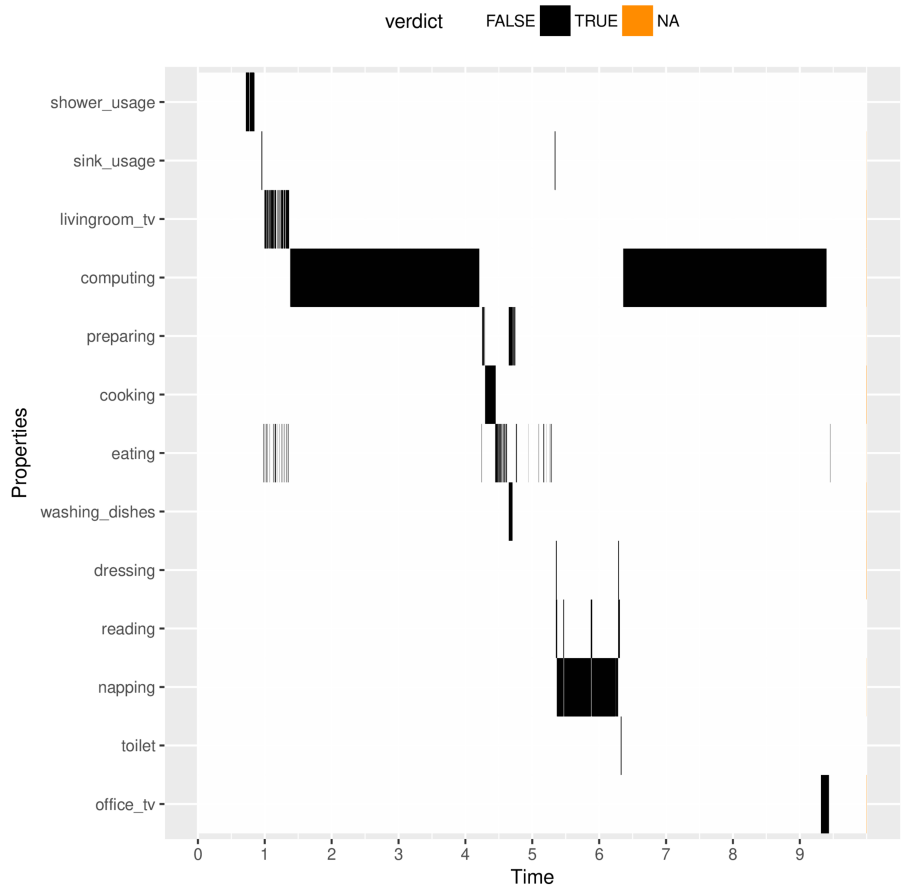


Fig. 2: Detected ADL for Tuesday, Jan 31 2017. Time is in hours starting from 7:30.

ample, we can specify a specification regulating the duration of watching TV to be at most 10 timestamps.

3 Monitoring the Apartment

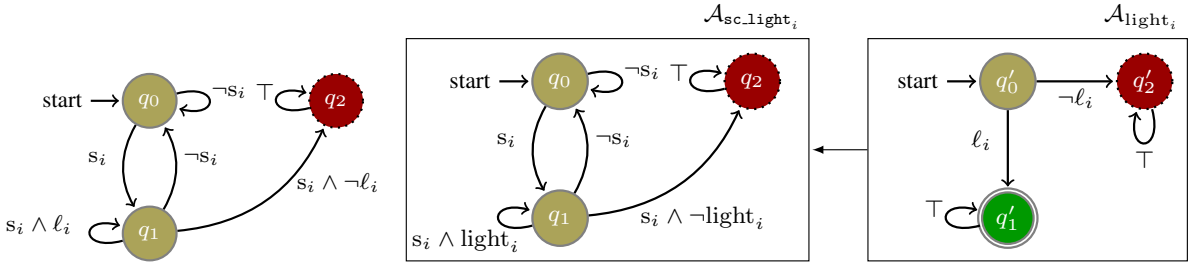
We show how we monitor the apartment using decentralized specifications, while highlighting their advantages.

3.1 Monitor Implementation

To monitor the apartment, we use LTL3 monitors [16]. LTL3 [14,15] is a variant of the standard Linear Temporal Logic (LTL) [55] giving a semantics to finite traces. An LTL3 monitor is a complete and deterministic Moore automaton where states are labeled with the verdicts in a domain $\mathbb{B}_3 = \{\top, \bot, ?\}$. Verdicts $\top$ and $\bot$ respectively indicate that the current execution complies and does not comply with the specification, while verdict ? indicates that the verdict has not been determined yet. Verdicts $\top$ and $\bot$ are called final, as once the monitor outputs $\top$ or $\bot$ for a given trace, it cannot output a different verdict for any suffix of that trace. Using LTL3 monitors for representing specifications allows us to take advantage of the multiple RV tools that convert different specification languages to LTL3 monitors. For our monitoring, we use the `THEMIS` tool [30] which is able to use both `ltl2mon` [16] and `LamaConv` [44] to generate monitors. `ltl2mon` generates LTL3 monitors from LTL formulae, while `LamaConv` supports a wider range of languages such as Regular Expressions, Omega Regular Expressions, LTL, LTL with past (pLTL), Regular LTL (RLTL) and RLTL with past (pRLTL), and Structured Assertion Language for Temporal Logic (SALT) [13].

Example 1 (Check light switch) Let us consider property `sc_light`(i) (sensor check light): "Whenever a light switch is triggered in a room i at some timestamp t, then the light must turn on at $t+1$ until the switch is turned off again". Figure 3a shows the Moore automaton that represents the property. Starting from q_0 with verdict ?, the automaton verifies that the property is falsified (as it is a safety property). That is, upon reaching q_2 the verdict will be $\bot$ for all possible extensions of a trace.

For the scope of this paper and for clarity, we use LTL extended with two (syntactic) operators, mostly to strengthen and relax time constraints. We consider the operator *eventually within* t ($\Diamond_{\leq t}$) which considers a disjunction of next operators. It is defined as: $\Diamond_{\leq t} ap \stackrel{\text{def}}{=} ap \vee \bigcirc ap \vee \bigcirc\bigcirc ap \vee \ldots \bigcirc^t ap$, where ap is an atomic proposition. Intuitively, the eventually within states that ap holds within a given number



(a) Centralized (b) Decentralized

Fig. 3: Monitor(s) for $\texttt{sc\_light}(i)$, for a given room i in the house. The verdicts associated with the states are $\bot$: dotted red , $\top$: double green, and ?: single yellow.

of timestamps. Operator $\Diamond_{\leq t}$ allows us to relax the time constraints for a given atomic proposition. Similarly, we consider the operator *globally within* t ($\Box_{\leq t}$) which is the dual of the previous operator: $\Box_{\leq t} ap \stackrel{\text{def}}{=} ap \wedge \bigcirc ap \wedge \bigcirc \bigcirc ap \wedge \bigcirc^t ap$.

Example 2 (Check light switch modalities) The property expressed in Ex. 1 can be expressed in LTL as: $\texttt{sc\_light}(i) \stackrel{\text{def}}{=} \Box(\text{s}_i \implies \bigcirc(\ell_i \,\text{U}\, \neg \text{s}_i))$. The property can be modified with the extra operators relax or constrain the time on the light. The relaxed property $\texttt{sc\_light}'(i) \stackrel{\text{def}}{=} \Box(\text{s}_i \implies \Diamond_{\leq 3}(\ell_i \,\text{U}\, \neg \text{s}_i))$ allows the right-hand side of the implication to hold within any of the next 3 timestamps instead of immediately after. The bounded property $\texttt{sc\_light}''(i) \stackrel{\text{def}}{=} \Box(\text{s}_i \implies \Box_{\leq 3}(\ell_i))$ states that the light is on starting from the timestamp the switch is turned on and the subsequent two (for a total of 3). An example of such a property is the restriction on watching TV for a specific duration (Table 1) where $\texttt{restricttv} \stackrel{\text{def}}{=} \Box(\texttt{tv} \implies \Diamond_{\leq 10} \neg \texttt{tv})$.

3.2 Decentralized Specifications

While simple specifications can be expressed with both LTL and automata, it quickly becomes a problem to scale the formulae or account for hierarchies (see Sect. 3.3). As such, we use decentralized specifications [29].

Overview.

Decentralized specifications consider a system of multiple components $\mathcal{C} = \{\mathcal{C}_1 \ldots \mathcal{C}_n\}$, where the set of all atomic propositions (noted AP) (i) has a partition over all components, i.e., $AP = AP_1 \cup \ldots \cup AP_n$ such that $\forall i, j \in [1..n], i \neq j \implies AP_i \cap AP_j = \emptyset$, and (ii) each component has at least one atomic proposition to monitor (i.e., $\forall i \in [1..n], AP_i \neq \emptyset$). Details for assigning sensor information as atomic propositions for this case study are presented in Sect. 4.2. Furthermore, we have a set of monitor labels AP_{mons} (called *monitor references*), that associates each monitor with a label. For this case study, each specification in Table 1 is assigned a monitor labeled by its name. Each monitor $\mathcal{A}_{\text{lbl}}$ ($\text{lbl} \in AP_{\text{mons}}$) is a Moore automaton (detailed in Sect. 3.1) and is assigned to a single component. A monitor $\mathcal{A}_{\text{lbl}}$ assigned to component $\mathcal{C}_j \in \mathcal{C}$ utilizes the alphabet $AP_{\text{lbl}} = AP_j \cup (AP_{\text{mons}} \setminus \{\text{lbl}\})$. That is, it contains the atomic propositions local to the component (in AP_j), and the references to all dependent monitors excluding itself ($AP_{\text{mons}} \setminus \{\text{lbl}\}$). A decentralized trace is a partial function that assigns each component and timestamp with an event. A monitor reference is evaluated as if it were an oracle. That is, to evaluate a monitor reference lbl at a timestamp t, the monitor referenced ($\mathcal{A}_{\text{lbl}}$) is executed starting from the initial state on the trace starting at t. The atomic proposition lbl at t takes the value of the final verdict reached by the monitor.

Example 3 (Decentralized light switch) Figure 3b shows the decentralized specification for the check light property from Ex. 1. We have two monitors $\mathcal{A}_{\texttt{sc\_light}_i}$ and $\mathcal{A}_{\text{light}_i}$. They are respectively attached to the light switch and light bulb components. In the former, the atomic propositions are either related to observations on the component (s_i, switch on), or references to other monitors (light_i). The light switch monitor first waits for the switch to be on to reach q_1. In q_1, at some timestamp t, it needs to evaluate reference light_i by running the trace starting from t on monitor $\mathcal{A}_{\text{light}_i}$. Monitor $\mathcal{A}_{\text{light}_i}$ then reads the value of ℓ_i at t from the trace, and moves to q'_1 or q'_2 depending on its value, and sends the verdict $\top$ or $\bot$ respectively back to monitor $\mathcal{A}_{\texttt{sc\_light}_i}$. The returned verdict is associated with the reference light_i for timestamp t allowing monitor $\mathcal{A}_{\texttt{sc\_light}_i}$ to evaluate its own transition at t.

Assumptions.

The assumptions of decentralized specifications on the system are as follows: no monitors send messages that contain wrong information; no messages are lost, they are eventually delivered in their entirety but possibly out-of-order; all components share one logical discrete clock marked by round numbers indicating relevant transitions in the system specification. While security is a concern in the smart apartment setting, the first two assumptions are met in this case study

as the apartment sensor network operates on the local network, and we expect monitors to be deployed by the sensor providers, and users of the apartment. Furthermore, the last assumption is also met in the setting of the smart appartment, as all sensors share a global clock. This is evidenced by the obtained traces from Orange4Home [25].

Hierarchical dependencies.

Decentralized specifications allow us to analyze the dependencies between various monitors, and organize them in logical hierarchies represented as directed acyclic graphs (DAGs). The DAGs help us relate specifications to other specifications and analyze the inter-dependent behavior of monitors. We elaborate on the benefits of the hierarchical dependencies in Sect. 3.3.

Example 4 (Hierarchical dependencies) Figure 4 presents the dependency DAG of specification `preparing`. We can see that specification `preparing` depends directly on both specifications `kactivity` and `cooking`. Specification `kactivity` depends on specifications `cubpoard`, `sink_water`, `presence`, and `fridge_door`, as it depends on the tenant being present in the kitchen, opening or closing cupboards or the fridge, or using the sink. The later specifications do not depend on other specifications but on direct observations from the components. We note that while `presence` is not used in this case study to determine the cooking activity, since a tenant can start cooking and leave the kitchen. One could imagine that specifications can share dependencies, as such the hierarchy is indeed best represented as a DAG. Let us consider the monitor checking specification `cupboard`. Since we have 5 cupboard doors, we have 5 sensors in total (1 for each door). The monitor observing the 5 different observations simply checks if one is open and relays its verdict upwards, transmitting only the summary of observations instead of the totality. In this example, the hierarchy can be seen starting from different sensors on the same component, and expanding geographically to the different components in the room (kitchen).

3.3 Advantages of Decentralized Specifications

3.3.1 Modularity and Re-use

Monitor references in decentralized specifications allow specifications writers to modularize behavior. Given that a monitor represents a specific specification, this same monitor can be re-used to define more complex specifications at a higher level, without consideration for the details needed for this specification. This allows specification writers to reason at various levels about the system specification.

Let us consider the ADL specification `cooking` (resp. `sleeping`) which specifies whether the tenant is cooking (resp. sleeping) in the apartment. One can reason about the meta-specification `firehazard` using both `cooking` and `sleeping` specifications without considering the lower level sensors that determine these specifications, that is:

$$\texttt{firehazard} \stackrel{\text{def}}{=} \Box(\texttt{sleeping} \implies \neg\texttt{cooking}).$$

While we can define `cooking` as:

$$\texttt{cooking} \stackrel{\text{def}}{=} \texttt{kitchen\_presence} \wedge \Diamond_{\leq 5}(\texttt{kitchen\_cooktop} \vee \texttt{kitchen\_oven}).$$

Additionally, any specification that requires either `sleeping` or `cooking` specifications can re-use the verdict outputted by their respective monitors. For example, specifications $\texttt{actfloor}(0)$ and $\texttt{actfloor}(1)$ require the verdicts from monitors associated with `cooking` and `sleeping`, respectively, since cooking happens on the ground floor while sleeping on the first floor. Furthermore, we can disjoin $\texttt{actfloor}(0)$ and $\texttt{actfloor}(1)$ to easily specify that there is some activity in the house, $\texttt{acthouse} \stackrel{\text{def}}{=} \texttt{actfloor}(0) \vee \texttt{actfloor}(1)$. While specification `acthouse` can be seen as a quantified version of $\texttt{actfloor}(i)$, we can use modular specifications for behavior, for example we can verify the triggering of an alarm in the house within 5 timestamps of detecting a fire hazard, i.e. $\texttt{checkalert} \stackrel{\text{def}}{=} \texttt{firehazard} \implies \Diamond_{\leq 5}(\texttt{firealert})$.

In addition to providing a higher level of abstraction and reasoning about specifications, the modular structure of the specifications present three additional advantages.

1. The first is that sub-specifications can change without affecting the meta-specifications, that is if the sub-specification `cooking` is changed (possibly to account for different sensors), no changes need to be propagated to specifications `firehazard`, $\texttt{actfloor}(0)$, `acthouse`, and `checkalert`.
2. The second advantage is controlling duplication of computation and communication, as such sensors do not have to send their observations constantly to all monitors that verify the various specifications. Specification `cooking` requires knowledge from the kitchen presence sensor, the kitchen cooktop (being enabled) and the kitchen oven. Without any re-use these three sensors (presence, cooktop, and oven) need to send their information to monitors checking: `firehazard`, $\texttt{actfloor}(0)$, `acthouse`, and `checkalert`.
3. The third advantage is a consequence of modeling explicitly the dependencies between specifications. This allows the monitoring to take advantage of such dependencies and place the monitors that depend on each other closer depending on the hierarchy, either geographically (i.e., in the same room or floor) or logically (i.e., close to the monitors of the dependent sub-specifications). Furthermore, knowing the explicit dependencies between specifications allows the user to choose a placement for their monitors, adjusting the placement to the system architecture. In the case a placement is not possible, it is

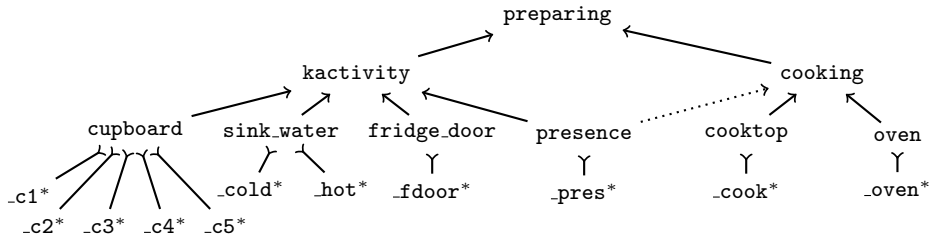


Fig. 4: Dependencies for `preparing`. * indicates an atomic proposition of a component.

possible to create intermediate specifications that simply relay verdicts of other monitors, to transitively connect all components that are not connected.

3.3.2 Abstraction from Implementation

One setback for learning-based techniques to detect ADL is their specificity to the environment. That is, the training set is specific to a house layout, user profile (i.e., elderly versus adults) [45].

Decentralized specifications define modular specifications that can be composed together to form bigger and more complex specifications. By using references to monitors, we leave the implementation of the specification to be specific for the house or user profile. Using our existing example, `cooking` is implemented based on the available sensors in the house, which would change for different houses. However, the meta-specifications such as `firehazard` can be defined independently from the implementation of both `cooking` and `sleeping`.

Furthermore, using monitor references, which are treated as oracles, opens the door to utilizing existing techniques in the literature based on other formalisms (not based on automata). That is, as a reference is expected to eventually evaluate to $\top$ or $\bot$, any decision procedure can be incorporated to form more complex specifications. For example, one can use the various machine learning techniques [19,45, 58] to define monitors that detect specific ADLs, then reference them in order to define more complex specifications.

3.3.3 Scalability

Decentralized specifications allow for a higher level of scalability when writing specifications, and also when monitoring. By using decentralized specifications, we restrict the atomic propositions of monitors to (i) the local atomic propositions of the components they are attached to and (ii) references to other monitors (see Sect. 3.2). This greatly reduces the number of atomic propositions to consider when synthesizing the monitor and reduces its size, as the subspecifications are offloaded to another monitor.

For example, let us consider writing specifications using LTL formulae. The classical algorithm that converts LTL to Moore automata is doubly exponential in the size of the formula counted in terms of atomic propositions (to form events) [16]. Therefore, reducing both the size of the formula and the number of atomic propositions used in the formula helps significantly when synthesizing the monitors, allowing us to scale beyond the limits of existing tools. For a large formula, and the larger formulas considered in this paper, it becomes impossible to generate a central monitor using the existing synthesis techniques. Decentralized specifications provide a way to manage the large formula by subdividing it into subformulas. The decomposition ensures that the formula evaluates to the same verdict given the same observations, at the cost of added delay.

Example 5 (Synthesizing the check light monitor) Recall the system property $\texttt{sc\_light}(i)$ in Ex. 2 responsible for verifying that in a room i a light switch does indeed turn a light bulb on until it is turned off. We recall the LTL specification $\texttt{sc\_light}(i) \stackrel{\text{def}}{=} \square(\mathrm{s}_i \implies \bigcirc(\ell_i \,\mathrm{U}\, \neg \mathrm{s}_i))$. To verify the property across n rooms of the house, we formulate a property $\texttt{sc\_ok} \stackrel{\text{def}}{=} \bigwedge_{i\in[0..n]} \texttt{sc\_light}(i)$. In the case of a decentralized specification the formula will reference each monitor in each room, leading to a conjunction of at n atomic propositions. However, in the case of a centralized specification, the specification needs to be written as: $\texttt{sc\_ok}^{\text{cent}} \stackrel{\text{def}}{=} \bigwedge_{i\in[0\ldots n]} \square(\mathrm{s}_i \implies \bigcirc(\ell_i \,\mathrm{U}\, \neg \mathrm{s}_i))$, which is significantly more complex as a formula consisting of $4n$ operators (to cover the sub-specification), along n conjunctions, and defined over each sensor and light bulb atomic propositions ($2n$). Given that monitor synthesis is doubly exponential, both `ltl2mon` [16] and `lamaconv` [44] require significant resources and time to generate the minimal Moore automaton (in our case[2], both tools where unable to generate the monitor for $n = 3$ after an hour to timeout).

We note that this effect on synthesis can be greatly beneficial in our case as formulae appear smaller than they actually are. Our usage of the shorthand operators $\Diamond_{\leq t}$ and $\square_{\leq t}$, when applied to a formula φ, results in a new formula where φ appears t times, hence contributing to a much larger expansion.

3.3.4 Limitations

Decentralized specifications revolve around monitors sending feedback as boolean verdicts. This enables the advantageous scalability and abstraction level when monitoring

[2] On an Intel(R) Core(TM) i7-6700HQ CPU, using 16GB RAM, and running openjdk 1.8.0_172, with ltl2mon 0.0.7.

smart homes. However, the expressiveness of decentralized specifications is limited. For instance, contrarily to [53], we cannot compare values of sensors in different rooms to establish a verdict based on some function of those values at a given moment (i.e., power consumption in a given room exceeds that of another room). We believe that, in the future, leveraging stream-based RV approaches (see Sect. 7.3) will allow monitors to output values in a domain richer than Boolean.

4 Trace Replay with THEMIS

To perform monitoring we use THEMIS [30] which is a tool for defining, handling, and benchmarking decentralized specifications and their monitoring algorithms. For replaying the trace, we perform monitoring by defining a start time, an end time and a polling interval. For this case study, for a given date, we use 07:30 as start time, 17:30 as an end time, and a 1-second polling interval.

We first overview THEMIS in Sect. 4.1. Then, in Sect. 4.2, we elaborate on the trace format provided in the public dataset, and our adaptation for replay to perform the monitoring. In brief, the process consists of extracting each sensor data converting it to observations (atomic propositions and verdicts), and passing the observation to a logical component for multiple related sensors. Later in Sect. 5, we introduce extra considerations when monitoring large traces.

4.1 THEMIS

Overview.

THEMIS [30] is a tool to facilitate the design, development, and analysis of decentralized monitoring algorithms; developed using Java and AspectJ. It consists of a library and command-line tools. THEMIS provides an API, data structures, and measures for decentralized monitoring. These building blocks can be reused or extended to modify existing algorithms, design new algorithms, and elaborate new approaches to assess existing algorithms. THEMIS encompasses existing approaches [11,22] that focus on presenting one global formula of the system from which they derive multiple specifications, and in addition supports any decentralized specification [32].

Monitoring.

THEMIS defines two phases for a monitoring algorithm: setup and monitor. In the first phase, the algorithm creates and initializes the monitors, connects them to each other so they can communicate, and attaches them to components so they receive the observations generated by components. In the second phase, each monitor receives observations at a timestamp based on the component it is attached to. The monitor can then perform some computation, communicate with other monitors, abort monitoring or report a verdict. The two distinct phases separate the monitor generation (monitor synthesis) problem from the monitoring [29], giving algorithms the freedom to generate monitors and deploy them on components, while integrating with existing tools for monitor synthesis such as [16,44]. The monitors used in this case study use similar logic than *choreography* [22], as they are defined over a shared global clock. All monitors start monitoring at $t = 0$. A monitor checks the compliance of the specification for a given timestamp t, which could take a fixed delay d to check. After reaching the delay at $t + d$, the monitor reports the verdict for t to all other monitors that depend on it, and starts monitoring the specification again for $t + 1$ (i.e., it *respawns*). As such, the communication between monitors consists of sending verdicts for given timestamps.

4.2 Generating the Trace

4.2.1 Provided Trace

The trace from [25] is given as a database with a table for each sensor. We extract each table as a *csv* file for each sensor. The provided sensor data is stored as entries of values associated with timestamps, representing the changes in the sensor data across time. Typically, a new entry is provided whenever a change in the sensor data occurs. The provided data range over Boolean-like, integer, or real domains.

4.2.2 Generating Atomic Propositions

The sensor data needs to be processed to create observations, as LTL3 monitors (see Sect. 3.1) operate on atomic propositions. Each sensor is implemented as an input (*Periphery* in THEMIS) to a logical component. For example, for the shower water, we use both cold and hot water sensors but define only a single component (“shower water”), from an RV perspective, “hot” and “cold” are multiple observations passed to the “shower water” component. To process different sensor data, we implemented two peripheries: *SensorBool* and *SensorThresh*. The first periphery parses Boolean values from the *csv* file associated with timestamps. The processing assigns Boolean values $\top$ (resp. $\bot$) based on sensor data such as: ”ON” (resp. ”OFF”), and ”OPEN” (resp. ”CLOSED”). The second periphery reads real (double) values, and returns a Boolean based on whether the number is below or above a certain threshold. Both peripheries associate each atomic proposition with the generated Boolean to generate an observation.

4.2.3 Synchronizing Traces

The provided dataset only provides sensor updates, that is, the data only contains timestamps and values for a sensor when the value changes. Our monitoring strategy, however, requires polling the devices at given fixed time intervals. Since the system has a global clock, to synchronize observations, our periphery implementations synchronize on a date at the start and an increase (in our case 1 second) and a default Boolean value for the observation. When polled, the

periphery returns the default value if nothing is observed yet, or the last value observed otherwise. The last value observed is updated when changes occur in the *csv* file. In short, we interpolate values between changes to return the oldest value before a change.

4.2.4 Determining the Polling Rate

We leverage the global clock of the system to evaluate the specification synchronously for all components. As such, we need a fixed interval to poll the monitors in order to evaluate the specification, that is, we take the necessary transition in each of the automata. We refer to this interval as the *polling rate*. The polling rate determines the frequency of evaluation of the specification; the higher the rate, the more rounds, and the more monitors process and communicate. To determine the minimal rate, we consider the rate of change for all sensors involved in the specification. We are interested in ensuring that no sensor changes twice in between the evaluation of the specification. To do so, we write a simple program that processes the trace files for each sensor in an input specification, to determine the rate of change. Listing 1 shows an example output on the 27 sensors used for ADL detection. It shows the atomic proposition associated with the sensor, the sensor type, the trace file, the fastest change rate (min), and the slowest change rate (max), and whether or not it is skipped. The rates are provided in milliseconds. Then, we aggregate over all sensors by computing the fastest and slowest. Sensors are not included in the aggregate computation (i.e., skipped) if no change appears in their entire trace file. In this case, we choose 1 second as our polling rate, as no sensor will change twice within a second.

5 Consideration for Large Traces

Managing the trace length (36,000) is an issue for the monitoring techniques presented in [29]. Since the associated monitors rely on eventual consistency [56], in some cases, they wait for input for the length of the trace, which requires a lot of memory. This was not an issue for the small traces (of length 100) used to compare algorithms originally, but becomes a significantly larger issue when monitoring a real apartment.

Two data structures are introduced in [29] to support monitoring decentralized specifications: *memory* and *execution history encoding* (EHE). We briefly review them in Sect. 5.1 along with their key operations so we can we present a garbage collection strategy for the memory data structure in Sect. 5.2 and an expansion strategy for the EHE in Sect. 5.3. The memory footprint for monitors consists of the sizes of their *memory* and *EHE*. Both our improvements aim at reducing their size for long traces. Theoretical details for the data structures and monitoring are in [29].

Note that replaying the large traces, without garbage collection, is not possible as we would run out of memory due to the sheer size of the trace and the space needed to store information. The expansion strategy, when used appropriately, further reduces memory consumption for large EHEs (see Sect. 6.1.3).

5.1 Monitoring Data Structures and Their Operations

The data structures *memory* and EHE operate over *atoms*, where an atom is an encoding of atomic propositions. The encoding used for monitoring the apartment consists of a pair of timestamp and atomic proposition. For example, the atom $\langle 23, \mathrm{s}_1 \rangle$, is used to refer to the truth value of switch 1 at timestamp 23.

5.1.1 Memory

The *memory* buffers all observations the monitor received from the component it is associated with, and the monitors it depends on. The memory is a partial function (noted $\mathcal{M}$) that associates atoms with verdicts. For example, the memory $\mathcal{M} = [\langle 23, \mathrm{s}_1 \rangle \mapsto \top, \langle 23, \mathrm{s}_2 \rangle \mapsto \bot]$ states that at timestamp 23, switch 1 was enabled while switch 2 was disabled. An underlying operation used to perform monitoring is denoted by eval, which takes a Boolean expression of atoms, and a memory. Function eval attempts to rewrite the expression by replacing the value of the atoms present in the memory by their associated verdict, then simplifies the expression (using Boolean simplification). The memory stores all observations and is used to rewrite expressions when performing monitoring.

Example 6 (eval) For the expression $e = \langle 23, \mathrm{s}_1 \rangle \vee \langle 23, \ell_1 \rangle$ and memory $\mathcal{M} = [\langle 23, \mathrm{s}_1 \rangle \mapsto \top]$, applying $\mathrm{eval}(e, \mathcal{M})$ will first rewrite e to $\top \vee \langle 23, \ell_1 \rangle$, which is then simplified to $\top$.

5.1.2 Execution History Encoding

We recall from Sect. 3.1 that monitors are Moore automata that check decentralized traces. Since we are dealing with partial information due to the decentralized nature of monitors, the EHE encodes the execution of the underlying automaton, keeping track of potential states when receiving partial observations. In brief, an EHE can be modeled as a partial function ($\mathcal{I}$) that associates a timestamp t and a state q of the automaton with a boolean expression e. Whenever e holds (i.e., $\mathcal{I}(t, e)$), we are sure that the automaton is in state q at timestamp t. The Boolean expression e is evaluated using the content of the monitor's *memory* data structure using eval. The size of the EHE grows to account for timestamps and potential reachable states as the system executes (as time passes). The main function that extends the EHE to new timestamps is mov. Function mov takes the current EHE, along with its last stored timestamp, and an arbitrary timestamp in the future, and expands the entries by generating the expressions up to the future timestamp using the structure of the automaton and reachability. As such,

Listing 1 Rates of change for sensor data. The highlighted sensors are skipped since their data never change.

```
livingroom_table              SensorBool      28.csv     Min: 3000       Max: 230704000    (ms) [OK]
kitchen_dishwasher            SensorThresh    167.csv    Min: 2190810000 Max: 2190810000   (ms) [SKIP]
office_deskplug               SensorThresh    119.csv    Min: 6000       Max: 231159000    (ms) [OK]
office_tv                     SensorBool      283.csv    Min: 420000     Max: 343980000    (ms) [OK]
livingroom_couch              SensorBool      45.csv     Min: 3000       Max: 247031000    (ms) [OK]
kitchen_presence              SensorBool      269.csv    Min: 2000       Max: 230702000    (ms) [OK]
kitchen_c1                    SensorBool      300.csv    Min: 1000       Max: 259080000    (ms) [OK]
kitchen_c2                    SensorBool      315.csv    Min: 1000       Max: 431493000    (ms) [OK]
kitchen_c3                    SensorBool      316.csv    Min: 1000       Max: 259095000    (ms) [OK]
kitchen_c4                    SensorBool      317.csv    Min: 1000       Max: 259051000    (ms) [OK]
kitchen_c5                    SensorBool      355.csv    Min: 1000       Max: 779361000    (ms) [OK]
kitchen_sink_hotwater         SensorThresh    184.csv    Min: 12000      Max: 260085000    (ms) [OK]
kitchen_sink_coldwater        SensorThresh    189.csv    Min: 12000      Max: 260501000    (ms) [OK]
bedroom_closet_door           SensorBool      339.csv    Min: 7000       Max: 605093000    (ms) [OK]
bedroom_luminosity            SensorThresh    120.csv    Min: 1000       Max: 254250000    (ms) [OK]
kitchen_cooktop               SensorThresh    36.csv     Min: 7000       Max: 260333000    (ms) [OK]
bathroom_shower_coldwater     SensorThresh    22.csv     Min: 12000      Max: 345139000    (ms) [OK]
bathroom_shower_hotwater      SensorThresh    201.csv    Min: 12000      Max: 345066000    (ms) [OK]
kitchen_fridge_door           SensorBool      314.csv    Min: 1000       Max: 260749000    (ms) [OK]
livingroom_tv                 SensorBool      282.csv    Min: 840000     Max: 344040000    (ms) [OK]
toilet                        SensorThresh    254.csv    Min: 12000      Max: 518222000    (ms) [OK]
bathroom_sink_coldwater       SensorThresh    86.csv     Min: 12000      Max: 260437000    (ms) [OK]
bathroom_sink_hotwater        SensorThresh    264.csv    Min: 25000      Max: 25000        (ms) [SKIP]
kitchen_oven                  SensorThresh    232.csv    Min: 2191235000 Max: 2191235000   (ms) [SKIP]
bedroom_drawer_1              SensorBool      357.csv    Min: 1000       Max: 345825000    (ms) [OK]
bedroom_drawer_2              SensorBool      358.csv    Min: 2000       Max: 515617000    (ms) [OK]
bedroom_bed_pressure          SensorThresh    349.csv    Min: 1000       Max: 342361000    (ms) [OK]

                               (Detected Rate)           Min: 1000       Max: 779361000    (ms)
```

to create an EHE $\mathcal{I}'$ from another one $\mathcal{I}$ containing current information at timestamp $\mathrm{t_{cur}}$ with information up to timestamp $\mathrm{t_{future}}$, we use $\mathcal{I}' = \mathrm{mov}(\mathcal{I}, \mathrm{t_{cur}}, \mathrm{t_{future}})$. Expanding the EHE when information is missing leads to large expressions in the EHE which require a larger memory to store and a longer time to simplify. As such, it is important to ensure that mov is called when sufficient information is present to resolve the EHE.

5.2 Memory Garbage Collection For Large Traces

We optimized data structure *memory* (which is used to store observations) to add garbage collection. To do so we have created a new implementation (*MemoryIndexed*) that indexes observations by timestamp. When the monitor concludes with a final verdict for timestamp t, and respawns to monitor timestamp $t+1$, all observations associated with a timestamp lesser than or equal to t are removed from the memory. That is, the new memory $\mathcal{M}'$ is constrained to $\mathrm{dom}(\mathcal{M}') = \mathrm{dom}(\mathcal{M}) \setminus \{\langle \mathrm{t}'', \mathrm{ap}\rangle \in \mathrm{dom}(\mathcal{M}) \mid t'' < t\}$ (where dom indicates the domain of the partial function). This ensures that older information is discarded as the monitoring moves with time.

5.3 Lazy EHE Expansion

The EHE data structure is designed to be as general as possible, and keeps expanding while it has not detected the state the automaton is in. For large trace sizes, this can cause an EHE to grow quickly to consume all available memory and prevents monitoring from completion. That is, the monitor expands the EHE using mov, causing the expressions to grow exponentially [32], when no information is provided to the monitor.

This is prominently the case when monitoring safety properties. Safety properties such as $\mathtt{p} \stackrel{\mathrm{def}}{=} \Box(\mathtt{ap})$ will only conclude when the value of ap is $\bot$. So long as the value of ap is $\top$, the monitor checking p does not reach a final verdict, and does not report it to its parent. Consequently, a monitor that checks a safety property that is never violated, incurs a delay that is as long as the trace size. One approach is to limit the expansion of the EHE to a fixed length (assuming a fixed maximal delay), and use a sliding window to maintain the limit. This approach, however, may cause monitoring not to conclude in cases where monitoring requires more time than that of the window. To solve this issue and provide the user with more control, we allow the user to specify the expansion condition for the EHE as an additional Boolean formula that is determined by communication. This allows us to expanding the EHE based on the communication patterns between monitors.

5.3.1 Scope

We recall from Sect. 3.2 that, for a given monitor labeled lbl, its alphabet AP_{lbl} consists of atomic propositions of dependent monitors and the alphabet of the attached component. For this enhancement, we consider monitors which only depend on other monitors, i.e., when $AP_{\mathrm{lbl}} \subseteq AP_{\mathrm{mons}}$. We can see, when looking at dependencies in Fig. 4, that most monitors eventually rely only on lower-level monitors which themselves rely on component observations. As such, most high-level specifications for the smart home, and in particular safety properties (formulated as meta-specifications in

Table 1), rely on other monitors which evaluate different specifications, and thus only depend on monitors.

5.3.2 Communication AP

For a monitor that only depends on other monitors, its alphabet consists of monitor references (i.e., $AP_{\text{lbl}} \subseteq AP_{\text{mons}}$). For each dependent monitor (labeled dep), we create two atomic propositions, one if the received verdict is $\top$ (noted $\top_{\text{dep}}$) and one if it is $\bot$ (noted $\bot_{\text{dep}}$). The resulting alphabet is $AP^{\text{com}}_{\text{lbl}} = \{\top_{\text{dep}}, \bot_{\text{dep}} \mid dep \in AP_{\text{lbl}}\}$. The expansion condition (noted $\varphi^{\text{trigger}}_{\text{lbl}}$) is thus a Boolean expression over the alphabet $AP^{\text{com}}_{\text{lbl}}$.

5.3.3 Evaluating the Expansion Condition

To evaluate the added atomic propositions, we define function resolve which takes as input an expansion condition $\varphi^{\text{trigger}}_{\text{lbl}}$, a memory $\mathcal{M}$, and a timestamp t as follows:

$$\begin{aligned}&\text{resolve}(\varphi^{\text{trigger}}_{\text{lbl}}, \mathcal{M}, t) = \\ &\quad \texttt{match } \varphi^{\text{trigger}}_{\text{lbl}} \texttt{ with} \\ &\qquad \mid \top_{\text{dep}} \in AP^{\text{com}}_{\text{lbl}} \rightarrow \text{eval}(\langle t, \text{dep}\rangle, \mathcal{M}) = \top \\ &\qquad \mid \bot_{\text{dep}} \in AP^{\text{com}}_{\text{lbl}} \rightarrow \text{eval}(\langle t, \text{dep}\rangle, \mathcal{M}) = \bot\end{aligned}$$

Function resolve performs pattern matching to convert the communication atomic proposition to an expression capable of being evaluated using eval, checking if the monitor returned verdict $\top$ and $\bot$ at timestamp t for $\top_{\text{dep}}$ and $\bot_{\text{dep}}$, respectively. We note that when the atom is not found in the memory, both $\top_{\text{dep}}$ and $\bot_{\text{dep}}$ do not hold.

5.3.4 Triggering the Expansion

Given a current time t_{cur} for which we last expanded the EHE, we determine the maximum possible expansion for the EHE by looking for the atom in the memory with the highest timestamp, noted t_{max}. Next, we define function resolved, which takes as input an expansion condition, a memory, a current timestamp and a maximum timestamp and generates the timestamps for which the EHE must be expanded.

$$\begin{aligned}&\text{resolved}(\varphi^{\text{trigger}}, \mathcal{M}, t_{\text{cur}}, t_{\text{max}}) = \\ &\quad \{t_{\text{cur}} < t \leq t_{\text{max}} \mid \text{resolve}(\varphi^{\text{trigger}}, \mathcal{M}, t) = \top\}\end{aligned}$$

Finally, we pick the maximum of the timestamps and expand the EHE accordingly.

Remark 1 (Wildcard Trigger.) It is common to observe a expansion condition that involves, for a given monitor (labeled lbl), all the atoms found in the checked specification. The expansion condition is then a disjunction of all atoms (i.e., $\bigvee_{\text{ap} \in AP^{\text{com}}_{\text{lbl}}}(\text{ap})$). To avoid evaluating such large expression, particularly when many dependencies exist (for example, meta-specifications `actfloor(0)` and `actfloor(1)`), we provide an optimization flag for a monitor to only trigger expansion upon receiving messages from other monitors.

Example 7 (Combination of safety properties and expansion) Consider the three monitors m_0, m_1 and m_2 that check for the following specifications:
- $\square(\neg$`firehazard`$)$,
- $\square(\neg$`notwopeople`$)$,
- and $\text{m}_0 \wedge \text{m}_1$.

We can see that in this case m_0 and m_1 only output verdicts when the property is falsified. That is, monitor m_2 which depends on both, has to normally expand its own EHE as time passes awaiting information that will only become available when the specification of either is falsified (i.e., `firehazard` or `notwopeople` evaluate to true in either monitors m_0 or m_1, respectively). As such, we can specify the expansion condition for monitor m_2 to be $\bot_{\text{m}_0} \vee \bot_{\text{m}_1}$: so long as no $\bot$ is communicated from either m_0 or m_1, the EHE is not expanded, as it cannot be falsified.

6 Assessing the Monitoring of the Apartment

Monitoring the smart apartment requires leveraging the interdependencies between specifications to be able to scale, beyond monitoring system properties, to more complex meta-specifications (as detailed in Sect. 2.2). We assess using decentralized specifications to monitor the apartment by conducting three scenarios. The first scenario (Sect. 6.1) evaluates the scalability and re-use advantages of using decentralized specifications presented in Sect. 3.3 by looking at the complexity of monitor synthesis, and communication and computation costs when adding more complex specifications that re-use sub-specifications. In addition, it also assesses the impact of using lazy expansion, as well as the overhead of monitoring on a per-monitor basis to account for realistic deployments. The second scenario (Sect. 6.2) evaluates the effectiveness of detecting ADL by looking at various detection measures such as precision and recall. The third scenario (Sect. 6.3) portrays the advantages of modularity by (i) adapting specification `napping` to use different sensors without modifying dependencies, and (ii) porting specification `firehazard` to a completely different environment (using the ARAS dataset [2]).

6.1 Monitoring Efficiency and Hierarchies

6.1.1 Monitor Synthesis

Table 1 displays the number of atomic propositions referenced by each specification for the decentralized ($|\text{AP}^{\text{d}}|$) and the centralized ($|\text{AP}^{\text{c}}|$) settings. Column d indicates the maximum depth of the directed acyclic graph of dependencies. We use the depth to assess how many levels of subspecifications need to be computed. When $d = 0$, it indicates that the specification can be evaluated directly by the monitor placed on the component, while $d = 1$ indicates that the monitor has to poll at most 1 monitor for its verdict (which typically relays the component observations). More generally, when $d = n$, it indicates that the specification depends

on a monitor that has at most depth $n-1$. The atomic propositions indicate either direct references to sensor observations (in the centralized setting) or references to either sensor observations or dependent monitors (in the decentralized setting). For certain specifications such as `toilet` which relies only on the water sensor in the toilet to be detected, there is no difference between using a centralized or decentralized specification, as it resolves to the observations. Reduction becomes more pronounced when specifications re-use other specifications as sub-specifications. For example, specification `acthouse` $\stackrel{\text{def}}{=}$ `actfloor`$(0) \vee$ `actfloor`(1), when decentralized, uses only 2 references (for each of the sub-specifications). However, when expanded, it references all 27 sensors used to detect activities. Additionally, specification `notwopeople` $\stackrel{\text{def}}{=} \neg($`actfloor`$(0) \wedge$ `actfloor`$(1))$ would not re-use the sub-specifications if expanded, requiring all sensors again. Henceforth, re-use greatly reduces the formula size and allows us to synthesize the monitors needed to check the formulas, as the synthesis algorithm is doubly exponential as mentioned in Sect. 3.3.

6.1.2 Assessing Re-use and Scalability

Reducing the size of the atomic propositions needed for a specification not only affects monitor synthesis, but also runtime performance, as atomic propositions represent the information needed to determine the specification (Sect. 3.3). To assess re-use and scalability, we perform two tasks and gather two measures pertaining to computation and communication, and present results in Fig. 5. The first task compares a centralized (SW-C) and a decentralized (SW-D) version of specification `sc_ok` presented in Example 5 using only 2 rooms. The second task introduces large meta-specifications on top of the ADL specifications to check scalability. Firstly, we measure the communication and computation for monitoring ADL specifications (ADL). Secondly, we introduce specifications `actfloor`(0), `actfloor`(1) and `acthouse` (ADL+H) as they require information about all sensors for ADL. Thirdly, we add specification `notwopeople` (ADL+H+2), as it reuses the same sub-specifications as specification `acthouse`. Lastly, we show all measures for all meta-specifications in Table 1 (ADL+M). We re-use two measures from [29]: the total number of simplifications the monitors are doing, and the total number of messages transferred. These measures are provided directly with `THEMIS` [30]. The total number of messages abstracts the communication (**#Msgs**), as our messages are of fixed length, they also represent the total data transferred. The total number of simplifications (**#Simplifications**) abstracts the computation done by the monitors, as they attempt to simplify Boolean expressions that represent automaton states, which are the basic operations for maintaining the monitoring data structures in [29]. Both measures are normalized by the number of timestamps in the execution (36,000). The resulting normalized measures represent the number of simplifications and messages per round. We conduct simulations over 10 different days as the person living in the house behaves slightly differently. For each day, we execute 5 simulations [3].

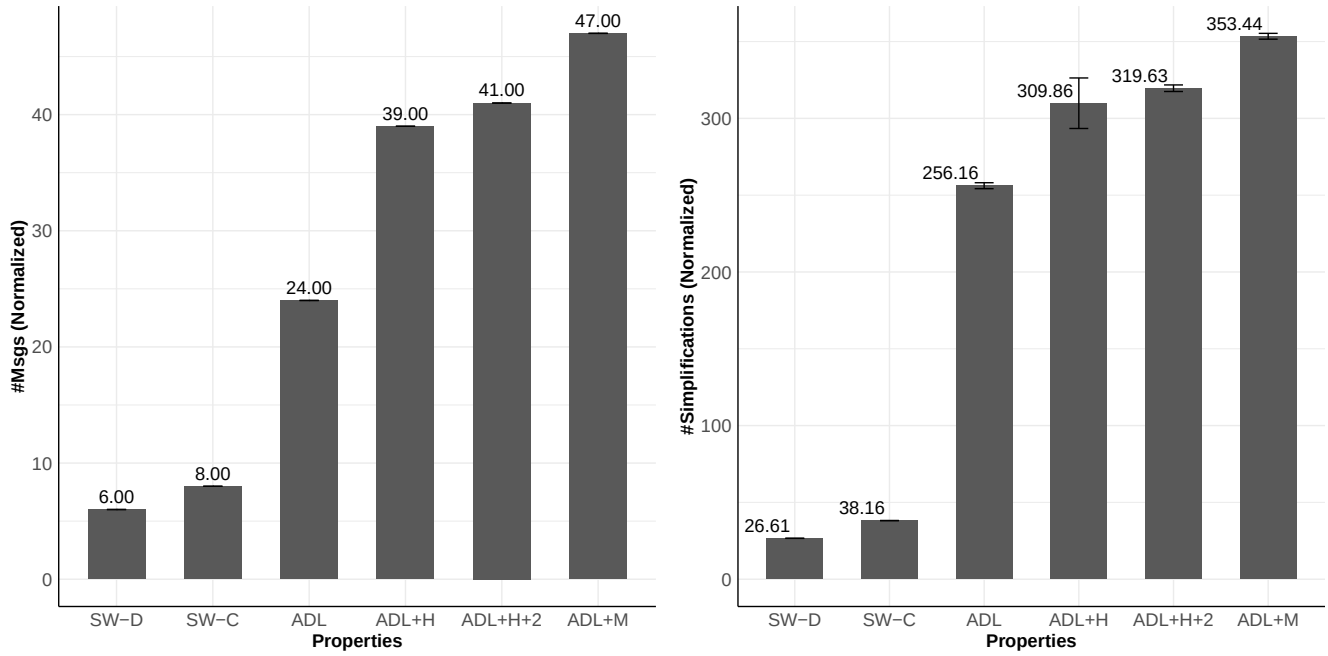
Figure 5a shows the normalized number of messages sent by all monitors. For the first task, we notice that the number of messages is indeed lower in the decentralized setting, SW-D sends on average 2 messages per timestamp less than SW-C, which corresponds to the difference in the number of atomic propositions referenced (6 for SW-D and 8 for SW-C). For the second task, we notice that on the baseline for ADL, we observe 24 messages per timestamp, a smaller number than the sensors count (27). This is because some ADL like `toilet` are directly evaluated on the sensor without communicating, and other ADL like `preparing`, re-use other ADL specifications like `kactivity`. By introducing the 3 meta-specifications stating that an activity occurred on a floor or globally in the apartment, the number of messages per round only increases by 15. This also coincides with the number of atomic propositions for the specifications (6 for `actfloor`(0), 7 for `actfloor`(1), and 2 for `acthouse`) as those monitors depend in total on 15 other monitors to relay their verdicts. This costs much less than polling 16 sensors to determine `actfloor`(0), 11 sensors to determine `actfloor`(1), and 27 (a total of 54) to determine `acthouse`. To verify this, we notice that the addition of `notwopeople` (ADL+H+2) that needs information from all 27 sensors, only increases the total number of messages per timestamp by 2. The specification `notwopeople` reuses the verdicts of the two monitors associated with each `actfloor` specification. After adding all the meta-specifications (ADL+M), the total number of messages per timestamp is 46, whihc is less than the number needed to verify adding `actfloor`, and `acthouse` in a centralized setting (54). We notice a similar effect for computation (Fig. 5b).

6.1.3 Impact of Lazy `EHE` Expansion

Figure 6 shows the maximum size of the `EHE` data structure obtained in a single run when using default wildcard triggers and custom triggers. For this scenario, we simulated a run for each optimization profile for 10 different days. The maximum size of an `EHE` presents us with the worst case memory footprint needed to hold the `EHE`[4]. We recall from Sect. 5.3, that wildcard triggers expand the `EHE` only when receiving messages from other monitors, while custom triggers designate specific expressions tailored for monitors based on their specification.

[3] The 95% confidence interval error was within 1% for the different runs for a single day

[4] Considering average `EHE` size in general would not be informative as the unbounded LTL operators are on the few meta specifications (particularly for safety).



(a) Communication.

(b) Computation.

Fig. 5: Scalability of communication and computations in decentralized specifications (95% confidence interval error bars).

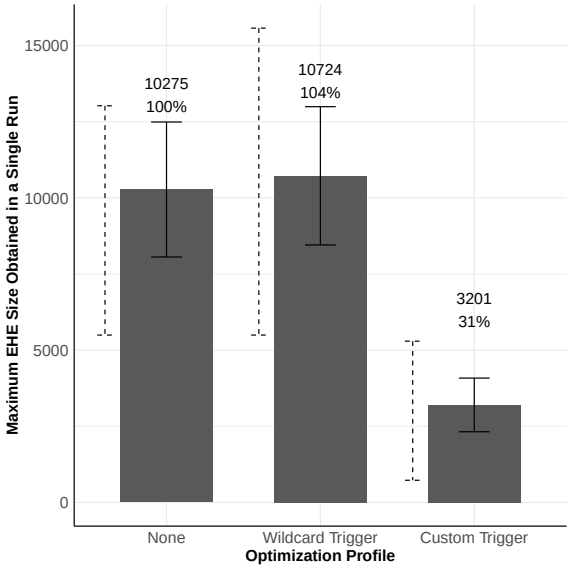


Fig. 6: Impact of lazy expansion on the size of EHE with wildcard and custom triggers. The value is the mean across traces from 10 different days with the 95% confidence interval, the dashed error bar on the left is the minimum and maximum size.

We observe that simply using the wildcard triggers does not necessarily lead to reducing the EHE size. Since waiting on communication can incur additional delay for processing EHE. However, when the triggers are relevant to the specification, we observe a decrease in worst-case EHE size, resulting in a EHE that is 31% of the non-optimized size.

6.1.4 Individual Monitor Performance

While observing the aggregate information about the entire simulation provides insights on advantages of using decentralized specifications, monitors are effectively deployed on devices, and their overhead is important to realistically assess if such deployment is reasonable. This section presents a per-monitor assessment of overhead for both runtime, memory and communication.

To assess each monitor performance we execute a simulation for 10 different days and record the performance of each monitor per round (that is 36,000 entries per day). We report the results in Table 2. Runtime is recorded for each monitor as the time it took to execute its monitoring step. Memory is measured by measuring the size of the EHE based on the expressions it contains, their length and the number of bytes needed to encode them. This approach is more accurate than attempting to use the JVM memory functions as garbage collection interferes heavily with memory measures for specific monitors. Communication is measured by observing messages that are sent and received by the monitor at a given round. Recall, that the message carries a verdict, and all messages have the same constant message size (needed to encode the monitor ID and a Boolean). For all metrics we gather the mean, a 95% confidence error mar-

Table 2: Average performance for 10 different days of individual monitors: runtime, memory, and communication (number of incoming/outgoing messages). Value in a cell contains: mean ± 95% confidence interval error (maximum value observed).

Name	Runtime (ms)	Size (B)	Incoming	Outgoing
`toilet`	1.9 ± 0.03 (635)	27 ± 0 (27)	0 (0)	1 (2)
`sink_usage`	2.0 ± 0.02 (593)	50 ± 0.16 (161)	1 (3)	1 (5)
`shower_usage`	2.1 ± 0.02 (596)	138 ± 0.17 (173)	1 (3)	1 (4)
`napping`	1.8 ± 0.02 (638)	114 ± 0.18 (167)	1 (3)	3 (81)
`dressing`	1.9 ± 0.02 (636)	90 ± 0.06 (287)	2 (6)	2 (12)
`reading`	3.4 ± 0.02 (639)	316 ± 0.61 (5,294)	3 (31)	1 (32)
`office_tv`	2.2 ± 0.03 (637)	33 ± 0 (33)	0 (0)	1 (2)
`computing`	2.1 ± 0.02 (637)	90 ± 0.14 (137)	1 (3)	1 (5)
`cooking`	2.0 ± 0.02 (637)	79 ± 0.07 (245)	2 (6)	3 (21)
`washing_dishes`	2.0 ± 0.02 (596)	95 ± 0.09 (299)	2 (5)	1 (5)
`kactivity`	3.6 ± 0.02 (621)	502 ± 0.76 (727)	4 (10)	1 (5)
`preparing`	2.4 ± 0.02 (639)	130 ± 0.06 (185)	2 (10)	1 (8)
`livingroom_tv`	1.8 ± 0.02 (412)	93 ± 0.10 (339)	2 (5)	2 (10)
`eating`	2.6 ± 0.02 (635)	174 ± 0.22 (231)	2 (6)	1 (8)
`actfloor(0)`	2.5 ± 0.02 (636)	93 ± 0.23 (703)	6 (20)	2 (16)
`actfloor(1)`	3.3 ± 0.02 (638)	280 ± 0.55 (1299)	7 (57)	2 (52)
`acthouse`	2.5 ± 0.02 (635)	158 ± 0.32 (927)	2 (28)	0 (0)
`notwopeople`	2.3 ± 0.02 (636)	134 ± 0.18 (339)	2 (28)	0 (0)
`restricttv`	2.2 ± 0.02 (635)	131 ± 0.24 (675)	2 (15)	0 (0)

gin[5], and the peak value. The peak value is important as it allows to give a worst-case assessment which corresponds to the highest resource consumption for the monitor to be deployed on an IoT device.

We observe that the computation is in the order of a few milliseconds at any given timestamp. Note that the sampling rate of the simulation is 1s, even at the worst-case (639ms), the performance is still acceptable for realistic deployment. It is important to note that `THEMIS` schedules monitors to run in parallel using all cores resources. Since monitors run in parallel, they may interfere with other monitors. However, our results give a general idea about the realistic deployment of such monitors on the devices.

Memory usage for monitors varies with delays imposed by specifications (using temporal operators) or dependencies on other specifications. For monitors operating directly at the sensor level, such as `toilet` and `office_tv`, the memory consumption of the `EHE` is quite low (27 and 33 bytes respectively), and stable, since no delay is introduced that causes their `EHE` to expand. Monitors with more complex dependencies and specifications, such as `reading`, `actfloor(i)`, may exhibit large sizes of `EHE` in the worst-case, as information may arrive all at once after being delayed by the dependent specifications. While the memory footprint is still acceptable (~5KB), these monitors are also typically deployed on larger devices as they are tasked with aggregating more complex information about a given room, floor or the entire house.

Communication patterns follow the dependencies between monitors on average. On average monitors receive a number of messages equivalent to the number of other monitors they depend on, and send on average a number of messages equivalent to the number of other monitors that depend on them. Due to delays imposed by the specification and dependencies, monitors often have to wait to find verdicts for several timestamps. when resolving after not reaching a verdict for some time, some ”burst” behavior may be seen where monitors send a large number of messages all at once to notify those that depend on them of all the verdicts for the elapsed timestamps. This can be minimized in the future by appending a large message per recipient with all verdicts instead of sending smaller ones, for each timestamp.

[5] This is omitted for communication as it was close to 0 for all monitors, since communication is stable.

6.2 ADL Detection using RV

6.2.1 Measurements

Table 3 displays the effectiveness of using RV to detect all ADL specifications on the trace of three days with different schedules. To assess the effectiveness, we compared with the provided self-annotated data from [25], where the user annotated the start and end of each activity. We measure precision, recall and F1 (the geometric mean of precision and recall). To measure precision, we consider a true positive when the verdict $\top$ of a monitor for a given timestamp fell indeed in the self-annotated interval for the activity. To measure recall, we measure the proportion of the intervals for which the monitors have determined $\top$ (using RV). This approach is more fine-grained than the approach used in [48] where the precision and recall are computed for the start and end of intervals.

6.2.2 Results

The effectiveness of detection depends highly on the specification. Our approach performs well for the specifica-

Table 3: Precision, Recall, and F1 scores of monitoring all ADL specifications on three days with different schedules.

	Tuesday, Jan 31 2017			Monday, Feb 20 2017			Tuesday, Feb 21 2017		
Specification	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
`computing`	0.98	0.99	0.99	0.94	0.99	0.96	0.99	0.99	0.99
`office_tv`	1.00	0.80	0.89	1.00	0.94	0.97		-	
`cooking`	0.88	0.88	0.88	0.90	0.93	0.92		-	
`shower_usage`	1.00	0.50	0.67		-		1.00	0.63	0.77
`washing_dishes`	1.00	0.47	0.64	0.93	0.63	0.75		-	
`livingroom_tv`	1.00	0.43	0.60		-		1.00	0.47	0.64
`dressing`	1.00	0.41	0.58	1.00	0.31	0.47		-	
`toilet`*	1.00	0.18	0.30		-		0.75	0.24	0.36
`sink_usage`	1.00	0.13	0.23	1.00	0.24	0.35	0.003	0.16	0.01
`eating`	0.61	0.35	0.44	0.70	0.73	0.71		-	
`napping`	0.43	0.95	0.60	0.38	0.94	0.54		-	
`preparing`	0.23	0.77	0.35	0.21	0.79	0.34		-	
`reading`	0.37	0.04	0.06	0.02	0.10	0.03		-	

tions `computing`, `cooking`, `office_tv`, as it exhibits high precision and high recall. The second group of specifications contains specifications such as `shower_usage`, and `livingroom_tv`. It exhibits high precision but medium recall, that is, we were able to determine around 40 to 50% of all the timestamps where the specifications held according to the person annotating, without any false positives. The third group is similar to the second group but has very low recall (13-18%) and contains the specifications `toilet` and `sink_usage`. We notice that for `sink_usage` specific user behavior can throw it off, as seen for the trace of Feb 21, we elaborate on the limitations in the next paragraph. The fourth group, which includes the specifications `napping` and `preparing`, shows high recall but a high rate of false positives. And finally, specification `reading` is not properly detected, as it has a high rate of false positives and covers almost no annotated intervals.

6.2.3 Limitations of RV for Detecting ADL

The limitations of using RV to detect ADL are due to the modeling. As mentioned in Sect. 2.2, RV can be seen as a knowledge-based approach to activity detection, as such it suffers from similar weaknesses and limitations [21]. The activity is described as a rigid formal specification over the sensor data, and this has two consequences. Firstly, since RV relies purely on sensor data, activities which cannot be inferred from existing sensors will be poorly detected or not detected at all. This is the case for `reading`, as there are no sensors to indicate that the tenant is reading. We infer reading by checking that the light is on in the room and no other specified activity holds. Secondly, given that specifications are rigid, we expect the user to behave exactly as specified for the activity to be detected, any minor deviation results in the activity not being detected (as seen on Feb 21). To illustrate this point, the specification `computing` relies on the power consumption of the plug in the office. Had the tenant been charging his phone instead of computing, the recall would have suffered greatly. Another great example of this is the `shower_usage` specification, that is captured by inspecting the water usage of the shower. The time the tenant spends getting into the shower and out of the shower will not be considered, which greatly impacts recall. The above issues are further compounded by the annotation being carried out by a person. The annotator can for example take a few seconds to annotate some events which could impact recall, especially for short intervals of activity. However, even with the inherent limitations of using knowledge-based approaches, our observed groups and results fall within the expected range, of knowledge-based approaches such as [48], and also have similar effectiveness as model-based SVM approaches such as [20]. We elaborate on how the introduced modularity from decentralized specifications can alleviate some of these issues in Sect. 6.3.

6.3 Specification Adaptation for ADL Detection

Decentralized specifications introduce numerous advantages (see Sect. 3.3) for monitoring hierarchical systems that can change. We illustrated in Sect. 6.1 the scalability of decentralized specifications with hierarchies. Decentralized specifications allows specifications to be written with references to other specifications. The references allow specifications to be modular, changing the referenced specification is transparent with no modification to the specifications that depend on it. In this section, we illustrate the advantages of modularity in two cases. In the first case, we improve the detection of the activity `napping` by adding relevant sensors. The change only requires changing the monitor for `napping`, and no change is necessary for the remaining dependent specifications. In the second case, we apply the specification `firehazard` and all its dependencies on a completely different environment using the ARAS dataset [2].

Table 4: Modifying the decentralized specification to improve detection, and adapt to new environment.

(a) Refining `napping` using the bedroom sensors: bed pressure (`weight`), presence (`pres`), and light (ℓ).

Formula	Precision	Recall	F1
$\square_{\leq 25}(\texttt{weight})$	0.43	0.95	0.60
$\square_{\leq 3}(\texttt{weight})$	0.43	0.99	0.60
$\Diamond_{\leq 3}(\texttt{weight})$	0.43	1.0	0.60
$\square_{\leq 3}(\texttt{pres} \wedge \texttt{weight})$	0.34	0.14	0.20
$\square_{\leq 3}(\neg\ell \wedge \texttt{weight})$	1.00	0.97	0.99

(b) Modifications to detect `firehazard` in ARAS.

Specification	Formula
`preparing`	$\Diamond_{\leq 3}(\texttt{m\_kdrawer} \vee \texttt{m\_fridge} \vee \texttt{m\_cupboard})$
`cooking`	`preparing`
`beds`	$\texttt{bed1} \vee \texttt{bed2}$
$\texttt{beds}'$	$\texttt{bed1} \wedge \texttt{bed2}$
`napping`	$\square_{\leq 25}(\texttt{beds})$
`firehazard`	$\texttt{napping} \implies \neg\texttt{cooking}$

6.3.1 Improving Activity Detection

We modify the specification `napping` to better capture the activity. This requires no change to specifications that depend on `napping`. Table 4a shows the changes in precision and recall, for various versions of the specification `napping`. We modify the formula to relax the time constraints on the output of the bed pressure sensor. We notice, that while this could slightly improve recall (0.95 to 1), it does not translate to any precision improvement (it remains at 0.43). We explore using additional sensors in the room to capture the specification better. Using the presence sensor proves to be detrimental as it reduces precision to 0.34 and recall to 0.14. This is reasonable, as the presence sensor is a motion detector, and when someone is sleeping there may be no motion at all. However, people typically tend to turn the lights off when sleeping. Using the additional light sensor to detect lights are off, helps us increase precision to 1 and recall to 0.99. One could see that the effect of ADL detection is behavior specific, a tenant that sleeps with lights on will have undetected sleep using our specification. Being able to change to specific parts of the specification without impacting the rest of the it provides the flexibility to tune the ADL detection to specific users and behaviors.

6.3.2 Adapting to New Environments

In Sect. 2.2 we mentioned that ADL can be challenging as the detection of the specification does not only depend on the user behavior, but also on the environment in which it is monitored. In the context of learning techniques, using information learned from one environment to apply it to detection of ADL in other environments is discussed in [45]. Since decentralized specifications provide both a hierarchical and modular approach to designing specifications, it is possible to adapt specifications to new environment, by only changing the relevant parts or dependencies, and reasoning at the appropriate level. For instance, while specifications specifying ADL may change depending on the sensors and user behavior, meta-specifications do not necessarily change. We adapt specification `firehazard` and all its dependencies in the ARAS [2] dataset. The ARAS dataset features contact, pressure, distance, and light sensors, recording the interactions of two tenants with the sensors over a period of 30 days.

Table 4b shows the changes in the decentralized specification compared with that of Amiqual4Home found in Appendix A. For activity `preparing`, we follow a similar pattern, looking at the usage of cupboards, fridge, and kitchen drawers. Thus, we adapt the formula to reflect the available sensors in the kitchen. However, the ARAS dataset does not provide any electricity sensors for appliances, nor any way to detect heat being turned on. As such it is impossible to detect cooking using any sensors. Since we cannot tell `preparing` and `cooking` apart, we define `cooking` to simply be equivalent to `preparing`. Notice how in this case, we inverted the dependency from Fig. 4 (in ARAS, `cooking` depends on `preparing`). The ARAS dataset records the behavior of *two* people, instead of just *one*. As such, activity `napping` needs to be adjusted for the *two beds*. There are two ways to do so, the first assumes either one of the tenants is napping (`beds`), and the second assumes both are napping simultaneously ($\texttt{beds}'$). We notice that the meta-specification `firehazard` remains unchanged. However, it has two different interpretations. If we use `beds`, then it is possible to trigger `firehazard` when one tenant is cooking while the other is sleeping. We verify that, and notice that it is indeed falsified in 8 days (7, 9, 16, 17-19, 24, 27). Using $\texttt{beds}'$, allows us to only capture `firehazard` when both tenants are sleeping. It is then possible to refer `napping` to `allnapping` and `anynapping`, then using `firehazard` on `allnapping`, which would apply in both scenarios.

6.3.3 Discussion

We see that modularity provides several advantages. It allows us to make local change to specifications that do not need to be propagated upwards. It also makes it possible to generalize and abstract the specification to adapt to multiple environments. Decentralized specifications allow specifications to be written in a modular and adaptable fashion, allowing specifications to be adapted to target changes in user behavior and environment. It can be seen much like component-based design [57], which separate the implementation of each component in software, from its interaction with other components.

7 Related Work

We present similar or useful techniques for detecting ADL in a smart apartment that use log analysis and complex event processing. Then, we present techniques from stream-based RV that can be extended for monitoring smart apartments.

7.1 ADL Detection Using Log Analysis

Detecting ADL can be performed using trace analysis tools. The approach in [48] defines parametric events using Model Checking Language (MCL) [52] based on the modal mu-calculus (inspired by temporal logic and regular expressions). Traces are read and transformed into actions, then actions are matched against the specifications to determine locations in the trace that match ADL. Five ADL (sleep, using toilets, cooking, showering, and washing dishes) are specified and checked in the same smart apartment as our work. While this technique is able to detect ADL activities, it amounts to checking traces offline, and a high level of post-processing is required to analyze the data. In [10], the authors describe an approach for log analysis at very large scale. The specification is expressed using Metric First Order Temporal Logic (MFOTL), and logs are expressed as a temporal structure. The authors develop a *MapReduce* monitoring algorithm to analyze logs generated by more than 35,000 computers, producing approximately 1 TB of log data each day. While this approach is designed for distributed systems, does not map dependencies, and works offline, it could be used to process and monitor rich specifications over sensor data seen as log files.

7.2 ADL Detection Using Complex Event Processing

Reasoning at a much higher level of abstraction than sensor data, the approach in [42] attempts to detect ADL by analyzing the electrical consumption in the household. To do so, it employs techniques from Complex Event Processing (CEP), in which data is fed as streams and processed using various functions to finally output a stream of data. In this work, the ADL detection is split into two phases, one which detects peaks and plateaus of the various electrical devices, and the second phase uses those to indicate whether or not an appliance is being used. This illustrates a transformation from low-level data (sensor signal) to a high-level abstraction (an appliance is being used). The use of CEP for detecting ADL is promising, as it allows for similar scalability and abstraction. However, CEP's model of named streams makes it hard to analyze the specification formally, making little distinction between specification and implementation of the monitoring logic.

7.3 ADL Detection Using Runtime Verification

Similarly to CEP but focusing on Boolean verdicts, various stream-based RV techniques have been elaborated such as LOLA [26] which are used to verify correctness properties for synchronous systems such as the PCI bus protocol and a memory controller. A more recent approach uses the Temporal Stream-Based Specification Language (TeSSLa) to verify embedded systems using FPGAs [27]. Stream-based RV is particularly fast and effective for verifying lengthy parametric traces. However, it is unclear how these approaches handle monitor synthesis for a large number of components and account for the hierarchy in the system.

7.4 Discussion

Stream-based systems such as stream-based RV [18,40] and CEP are bottom-up. Data in streams is eventually aggregated into more complex information and relayed to a higher level. Decentralized specifications also support top-down approaches, which would increase the efficiency of monitoring large and hierarchical systems. To illustrate the point, consider the decentralized specification in Fig. 3b. In the automaton $\mathcal{A}_{\texttt{sc\_light}_i}$, the evaluation of the dependent monitor $\mathcal{A}_{\ell_i}$ only occurs when reaching q_1, so long as the automaton is in q_0, no interaction with the dependent monitor is necessary. This top-down feedback can be used to naturally optimize dependencies and increase efficiency. Because of the oracle-based implementation of decentralized specifications, it is possible to integrate any monitoring reference that eventually returns a verdict. One could imagine integrating other stream-based monitors or even data-driven ADL detection approaches. The integration works both ways, as monitors can be considered a (blocking) stream of verdicts for the other techniques.

8 Conclusion and Future Work

8.1 Conclusion

Monitoring a smart apartment presents RV with interesting new problems as it requires a scalable approach that is compositional, dynamic, and able to handle a multitude of devices. This is due to the hierarchical structure imposed by either limited communication capabilities of devices across geographical areas or the dependencies between various specifications. Attempting to solve such problems with centralized specifications is met with several obstacles at the level of monitor synthesis techniques (as we are presented with large formulae), and also at the level of monitoring as one needs to model interdependencies between formulae and re-use the sub-specifications used to build more complex specifications. We illustrate how decentralized specifications tackle such systems by explicitly modeling of interdependencies between specifications. Furthermore, we illustrate monitoring specifications that detect ADL in addition to system properties and even more specifications defined over both types of specifications.

8.2 Future Work

We believe that the use of decentralized specifications could be further extended to bring monitoring closer to data (col-

lected on sensors), and make RV a suitable verification technique for *edge computing*. One challenge of the case study was to determine the correct sampling period for monitor to operate. Further investigation is required to layout the trade-offs between the sampling period, communication overhead, and energy consumption. Also, decentralization is only supported by specifications based on the standard (point-based) LTL3 semantics. We believe that the use and decentralization of richer specification languages are desirable. For instance, we consider (i) using a counting semantics able to compute the number of steps needed to witness the satisfaction or violation of a specification [5] (ii) using techniques allowing to deal with uncertainty (e.g., in case of message loss) [9] (iii) using spatio-temporal specifications (e.g. [41]) to reason on physical locations in the house, and (iv) using a quantitative semantics possibly with time [4]. Finally, we consider using runtime enforcement [33,37,36] techniques (especially those for timed specifications [35,39]) to guarantee system properties and improve safety in the house (e.g., disabling cooking equipment whenever specification `firehazard` is violated). This requires to define the foundations for decentralized runtime enforcement on the theoretical side, and provide houses and monitors with actuators on the practical side.

References

1. Aimal, S., Parveez, K., Saba, A., Batool, S., Arshad, H., Javaid, N.: Energy optimization techniques for demand-side management in smart homes. In: Advances in Intelligent Networking and Collaborative Systems, The 9th International Conference on Intelligent Networking and Collaborative Systems, INCoS-2017. Lecture Notes on Data Engineering and Communications Technologies, vol. 8, pp. 515–524. Springer (2017)
2. Alemdar, H.Ö., Ertan, H., Incel, Ö.D., Ersoy, C.: ARAS human activity datasets in multiple homes with multiple residents. In: 7th International Conference on Pervasive Computing Technologies for Healthcare and Workshops, PervasiveHealth 2013. pp. 232–235. IEEE (2013)
3. Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017. ACM (2017)
4. Bakhirkin, A., Ferrère, T., Maler, O., Ulus, D.: On the quantitative semantics of regular expressions over real-valued signals. In: Abate, A., Geeraerts, G. (eds.) Formal Modeling and Analysis of Timed Systems - 15th International Conference, FORMATS 2017, Berlin, Germany, September 5-7, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10419, pp. 189–206. Springer (2017)
5. Bartocci, E., Bloem, R., Nickovic, D., Röck, F.: A counting semantics for monitoring LTL specifications over finite traces. CoRR abs/1804.03237 (2018)
6. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457. Springer (2018)
7. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools, and final results of crv 2014. International Journal on Software Tools for Technology Transfer (Apr 2017)
8. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer (2018), https://doi.org/10.1007/978-3-319-75632-5_1
9. Bartocci, E., Grosu, R.: Monitoring with uncertainty. In: Bortolussi, L., Bujorianu, M.L., Pola, G. (eds.) Proceedings Third International Workshop on Hybrid Autonomous Systems, HAS 2013, Rome, Italy, 17th March 2013. EPTCS, vol. 124, pp. 1–4 (2013)
10. Basin, D.A., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring of temporal specifications. Formal Methods in System Design 49(1-2), 75–108 (2016)
11. Basin, D.A., Klaedtke, F., Zalinescu, E.: Failure-aware runtime verification of distributed systems. In: Harsha, P., Ramalingam, G. (eds.) 35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015. LIPIcs, vol. 45, pp. 590–603. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
12. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Formal Methods Syst. Des. 48(1-2), 46–93 (2016)
13. Bauer, A., Leucker, M.: The theory and practice of SALT. In: NASA Formal Methods - Third International Symposium, NFM 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 13–40. Springer (2011)
14. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., Tasiran, S. (eds.) Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4839, pp. 126–138. Springer (2007)
15. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010)
16. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. 20(4), 14 (2011)
17. Bonakdarpour, B., Fraigniaud, P., Rajsbaum, S., Rosenblueth, D.A., Travers, C.: Decentralized asynchronous crash-resilient runtime verification. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada. LIPIcs, vol. 59, pp. 16:1–16:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
18. Bozzelli, L., Sánchez, C.: Foundations of boolean stream runtime verification. Theor. Comput. Sci. 631, 118–138 (2016)
19. Brdiczka, O., Crowley, J.L., Reignier, P.: Learning situation models in a smart home. IEEE Trans. Systems, Man, and Cybernetics, Part B 39(1), 56–63 (2009)
20. Chen, B., Fan, Z., Cao, F.: Activity recognition based on streaming sensor data for assisted living in smart homes. In: 2015 International Conference on Intelligent Environments, IE 2015. pp. 124–127. IEEE (2015)
21. Chen, L., Hoey, J., Nugent, C.D., Cook, D.J., Yu, Z.: Sensor-based activity recognition. IEEE Trans. Systems, Man, and Cybernetics, Part C 42(6), 790–808 (2012)
22. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. Formal Methods in System Design 49(1-2), 109–158 (2016)
23. Cotard, S., Faucou, S., Béchennec, J., Queudet, A., Trinquet, Y.: A data flow monitoring service based on runtime verification for AUTOSAR. In: 14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012. pp. 1508–1515. IEEE Computer Society (2012)
24. Crowley, J.L., Coutaz, J.: An ecological view of smart home technologies. In: De Ruyter, B., Kameas, A., Chatzimisios, P.,

Mavrommati, I. (eds.) Ambient Intelligence. pp. 1–16. Springer International Publishing, Cham (2015)
25. Cumin, J., Lefebvre, G., Ramparany, F., Crowley, J.L.: A dataset of routine daily activities in an instrumented home. In: Ubiquitous Computing and Ambient Intelligence - 11th International Conference, UCAmI 2017, Proceedings. Lecture Notes in Computer Science, vol. 10586, pp. 413–425. Springer (2017)
26. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005). pp. 166–174. IEEE Computer Society (2005)
27. Decker, N., Dreyer, B., Gottschling, P., Hochberger, C., Lange, A., Leucker, M., Scheffel, T., Wegener, S., Weiss, A.: Online analysis of debug trace data for embedded systems. In: 2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018. pp. 851–856. IEEE (2018)
28. El-Hokayem, A., Falcone, Y.: THEMIS Smart Home Artifact Repository, gitlab.inria.fr/monitoring/themis-rv18smarthome
29. El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: Antoine El-Hokayem and Yliès Falcone [3], pp. 125–135
30. El-Hokayem, A., Falcone, Y.: THEMIS: a tool for decentralized monitoring algorithms. In: Antoine El-Hokayem and Yliès Falcone [3], pp. 372–375
31. El-Hokayem, A., Falcone, Y.: Bringing runtime verification home. In: Colombo, C., Leucker, M. (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11237, pp. 222–240. Springer (2018), https://doi.org/10.1007/978-3-030-03769-7
32. El-Hokayem, A., Falcone, Y.: On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation. ACM Trans. Softw. Eng. Methodol. 29(1), 1:1–1:57 (2020)
33. Falcone, Y.: You should better enforce than verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 89–105. Springer (2010)
34. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Engineering Dependable Software Systems, NATO science for peace and security series, d: information and communication security, vol. 34, pp. 141–175. ios press (2013)
35. Falcone, Y., Jéron, T., Marchand, H., Pinisetty, S.: Runtime enforcement of regular timed properties by suppressing and delaying events. Sci. Comput. Program. 123, 2–41 (2016)
36. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Bartocci and Falcone [6], pp. 103–134
37. Falcone, Y., Mounier, L., Fernandez, J., Richier, J.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods in System Design 38(3), 223–262 (2011)
38. Falcone, Y., Nazarpour, H., Jaber, M., Bozga, M., Bensalem, S.: Tracing distributed component-based systems, a brief overview. In: Colombo, C., Leucker, M. (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11237, pp. 417–425. Springer (2018)
39. Falcone, Y., Pinisetty, S.: On the runtime enforcement of timed properties. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11757, pp. 48–69. Springer (2019)
40. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: Colombo, C., Leucker, M. (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11237, pp. 282–298. Springer (2018)
41. Haghighi, I., Jones, A., Kong, Z., Bartocci, E., Gros, R., Belta, C.: Spatel: A novel spatial-temporal logic and its applications to networked systems. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control. pp. 189–198. HSCC '15, ACM, New York, NY, USA (2015)
42. Hallé, S., Gaboury, S., Bouchard, B.: Activity recognition through complex event processing: First findings. In: Artificial Intelligence Applied to Assistive Technologies and Smart Environments, Papers from the 2016 AAAI Workshop. AAAI Workshops, vol. WS-16-01. AAAI Press (2016)
43. Havelund, K., Goldberg, A.: Verify your runs. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. Lecture Notes in Computer Science, vol. 4171, pp. 374–383. Springer (2005)
44. Institute for Software Engineering and Programming Languages: LamaConv - Logics and Automata Converter Library, www.isp.uni-luebeck.de/lamaconv
45. van Kasteren, T., Englebienne, G., Kröse, B.J.A.: Transferring knowledge of activity recognition across sensor networks. In: Pervasive Computing, 8th International Conference, Pervasive 2010. Proceedings. Lecture Notes in Computer Science, vol. 6030, pp. 283–300. Springer (2010)
46. Katz, S.: Assessing self-maintenance: Activities of daily living, mobility, and instrumental activities of daily living. Journal of the American Geriatrics Society 31(12), 721–727 (1983)
47. Kazemlou, S., Bonakdarpour, B.: Crash-resilient decentralized synchronous runtime verification. In: 37th IEEE Symposium on Reliable Distributed Systems, SRDS 2018, Salvador, Brazil, October 2-5, 2018. pp. 207–212. IEEE Computer Society (2018)
48. Lago, P., Lang, F., Roncancio, C., Jiménez-Guarín, C., Mateescu, R., Bonnefond, N.: The ContextAct@A4H real-life dataset of daily-living activities - activity recognition using model checking. In: Modeling and Using Context - 10th International and Interdisciplinary Conference, CONTEXT 2017, Proceedings. Lecture Notes in Computer Science, vol. 10257, pp. 175–188. Springer (2017)
49. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
50. Leucker, M., Schmitz, M., à Tellinghusen, D.: Runtime verification for interconnected medical devices. In: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9953, pp. 380–387 (2016)
51. Majumder, S., Aghayi, E., Noferesti, M., Memarzadeh-Tehran, H., Mondal, T., Pang, Z., Deen, M.J.: Smart homes for elderly healthcare - recent advances and research challenges. Sensors 17(11), 2496 (2017)
52. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Proceedings. Lecture Notes in Computer Science, vol. 5014, pp. 148–164. Springer (2008)
53. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015. pp. 494–503. IEEE Computer Society (2015)
54. Ogale, V.A., Garg, V.K.: Detecting temporal logic predicates on distributed computations. In: Pelc, A. (ed.) Distributed Computing, 21st International Symposium, DISC 2007, Lemesos, Cyprus,

September 24-26, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4731, pp. 420–434. Springer (2007)
55. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977)
56. Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6976, pp. 386–400. Springer (2011)
57. Szyperski, C.A., Gruntz, D., Murer, S.: Component software - beyond object-oriented programming, 2nd Edition. Addison-Wesley component software series, Addison-Wesley (2002)
58. Tapia, E.M., Intille, S.S., Larson, K.: Activity recognition in the home using simple and ubiquitous sensors. In: Pervasive Computing, Second International Conference, PERVASIVE 2004, Vienna, Austria, April 21-23, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3001, pp. 158–175. Springer (2004)
59. Thapliyal, H., Nath, R.K., Mohanty, S.P.: Smart home environment for mild cognitive impairment population: Solutions to improve care and quality of life. IEEE Consumer Electronics Magazine 7(1), 68–76 (2018)

A List of Properties

Table 5 shows all property definitions used in this case study. We ommitted the smaller monitors that are trivial such as `m_kitchen_cupboard` which is a disjunction of all cupboard doors observations in the kitchen.

Table 5: Definitions of the specifications used in the case study. A specification with name prefixed with m_ is such that the corresponding monitor is directly deployed on the component.

Name	**Formula**
sc_light(i)	$\Box(\text{switch}_i \implies \bigcirc(\text{light}_i \,\mathrm{U}\, \neg\text{switch}_i), i \in [0..3]$
sc_ok	$\bigwedge_{i\in[0..3]}$ sc_light(i)
m_toilet	toilet_water
sink_usage	$\Box_{\leq 3}$(m_bathroom_sink_water)
m_bathroom_sink_water	bathroom_sink_cold $\vee$ bathroom_sink_hot
shower_usage	$\Box_{\leq 2}$(m_bathroom_shower_water)
napping	$\Box_{\leq 25}$(m_bedroom_bed_pressure)
dressing	$\Diamond_{\leq 4}$(m_bedroom_closet_door $\vee$ m_bedroom_drawers))
reading	m_bedroom_light $\wedge$ $\Diamond_{\leq 4}$($\neg$dressing $\wedge$ $\neg$napping)
office_tv	$\Diamond_{\leq 3}$(m_office_tv)
computing	$\Diamond_{\leq 3}$(m_office_deskplug)
cooking	$\Diamond_{\leq 5}$(m_kitchen_cooktop $\vee$ m_kitchen_oven)
washing_dishes	$\Diamond_{\leq 3}$(m_kitchen_dishwasher $\vee$ m_kitchen_sink_water)
kactivity	m_kitchen_presence $\wedge$ $\Diamond_{\leq 3}$(m_kitchen_sink_water $\vee$ m_kitchen_fridgedoor $\vee$ m_kitchen_cupboard)
preparing	kitchen_activity $\wedge$ $\neg$cooking
livingroom_tv	$\Diamond_{\leq 3}$(m_livingroom_tv $\wedge$ m_livingroom_couch)
eating	$\neg$m_kitchen_presence $\wedge$ $\Box_{\leq 6}$(m_livingroom_table)
actfloor(0)	cooking $\vee$ preparing $\vee$ eating $\vee$ washing_dishes $\vee$ livingroom_tv $\vee$ m_toilet
actfloor(1)	computing$\vee$dressing$\vee$napping$\vee$office_tv$\vee$reading$\vee$shower_usage$\vee$ sink_usage
acthouse	actfloor(0) $\vee$ actfloor(1)
notwopeople	$\neg$(actfloor(0) $\wedge$ actfloor(1))
restricttv_office	office_tv $\implies$ $\Diamond_{\leq 10}$($\neg$office_tv)
restricttv_living	livingroom_tv $\implies$ $\Diamond_{\leq 10}$($\neg$livingroom_tv)
restricttv	restricttv_living $\wedge$ restricttv_office
firehazard	napping $\implies$ $\neg$cooking