Runtime Monitoring of Safety and Performance Requirements in Smart Cities

Meiyi Ma
Department of Computer Science
University of Virginia
Charlottesville, Virginia, 22903
meiyi@virginia.edu

John A. Stankovic
Department of Computer Science
University of Virginia
Charlottesville, Virginia, 22903
stankovic@virginia.edu

Lu Feng
Department of Computer Science
University of Virginia
Charlottesville, Virginia, 22903
lu.feng@virginia.edu

ABSTRACT

With the increasing number of smart services implemented in smart cities, it is important yet challenging to dynamically detect service conflicts with respect to safety and performance requirements. In this paper, we propose a framework for monitoring the operation of smart cities and services at runtime. We formalize a set of typical safety and performance requirements from different domains in smart cities (e.g., transportation, emergency, and environment) using Signal Temporal Logic. We present a case study based on a smart city simulator, in which actions of smart services and their predicted effects on city states are converted into signal traces over time and monitored continuously using formal specifications. The experimental results demonstrate the feasibility of using runtime monitoring to detect various conflicts of smart services.

CCS Concepts

•Security and privacy → Logic and verification; •Applied computing → *Service-oriented architectures;* **•Computer systems organization** → Sensors and actuators;

Keywords

Smart City, Safety and Performance Requirements, Signal Temporal Logic, Runtime Monitoring

1. INTRODUCTION

Cities around the globe are becoming smarter, thanks to the prevalence of the Internet of Things (IoT) technology. Many city governments (e.g., Amsterdam, Barcelona, Chicago, and Stockholm) have devoted significant effort in implementing infrastructure for sensing and actuation, with the purpose of improving public services and quality of life. Meanwhile, companies (e.g., IBM, Oracle) have invested heavily to develop smart city IoT platforms and services. It is estimated that the global market for smart city services across domains of transportation, energy, healthcare, water, food and waste will amount to about $400 billion per year by 2020 [13]. Nevertheless, as an increasing number of smart services, developed independently by different stakeholders and operated in smart cities simultaneously, an open problem is how to detect and resolve conflicts among the smart services. Often smart cities build control centers that provide control of services and dashboard of city states, such as Rio de Janeiro's Operations Center and Smart+Connected Operations Center. They are aware of real-time city states collected from sensors, such as the air quality, traffic condition, pedestrian number, etc. Many public smart services send requests to centers before taking actions. Our solution would execute in these control centers.

There are several other approaches for conflict detection in domains other than smart cities. DepSys [11] and SIFT [7] provide comprehensive strategies to specify, detect, and resolve conflicts in a smart home setting. Preclude [12] is a semantic rule-based solution to detect conflicting health advice derived from heterogeneous sources utilizing linguistic rules and external knowledge bases. Eyephy [10] detects dependencies across healthcare interventions by simulating the effect of interventions on over 7000 physiological parameters. However, none of the existing work on conflict detection across applications formulates the conflict detection in a smart city domain or as a runtime monitoring problem using formal methods.

In our previous work, we proposed CityGuard, a safety-aware watchdog architecture, for the detection and resolution of service conflicts in smart cities [8]. CityGuard is designed to be a middle layer embedded between the infrastructure layer (sensors and actuators) and the software layer (IoT platforms and smart services) installed in the smart city control center. The vision of CityGuard is to intercept the actions from smart services, detect if potential conflicts exist ahead of time, and provide conflict resolution when necessary. However, CityGuard detects service conflicts by running simulations and using hard-encoded rules. There is no formal logic used in the analysis.

In this paper, we address this limitation by proposing a temporal logic based runtime monitoring framework for safety and performance requirements in smart cities. Runtime monitoring is an approach that monitors and analyzes the runtime behavior of systems to detect if system predicted execution traces satisfy or violate certain properties. Such properties are often captured as formal specifications, allowing for a less adhoc analysis approach than system testing or

SafeThings'17, November 5, 2017, Delft, Netherlands

DOI: https://doi.org/10.1145/3137003.3137005

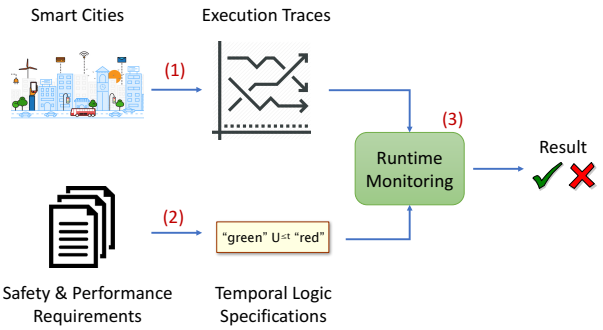

Figure 1: Overview of runtime monitoring framework for smart cities

simulation. Although runtime monitoring has been widely applied to various computer software and hardware systems, to the best of our knowledge, this is the first work to propose a temporal logic based runtime monitoring framework for smart cities.

The major contributions of this paper are:

A framework for runtime monitoring of smart cities: We describe how to create predicted traces of the operation of smart cities using an simulator, what traces and specifications to monitor, and how to use runtime monitoring for the conflict detection of smart services.

Formal specification patterns of safety and performance requirements: We present a set of specification patterns written in Signal Temporal Logic, which cover typical safety and performance requirements across the domains of transportation, emergency and environment in smart cities. We also provide specification patterns for describing conflicts in smart cities.

A case study of conflict detection in smart cities via runtime monitoring: We demonstrate the applicability of our proposed runtime monitoring framework and specification patterns via a case study based on a smart city simulator SUMO [4] and the Breach monitoring toolbox [5].

2. RUNTIME MONITORING FRAMEWORK FOR SMART CITIES

In this section, we propose a framework for runtime monitoring of smart cities. Figure 1 shows an overview of the proposed framework. First, when receiving action requests from smart services, we predict their effects on the smart city over a future time period through simulation. Second, we convert safety and performance requirements of smart cities written in English to formal specifications expressed in signal temporal logic. Then, we apply algorithmic runtime monitoring techniques to check if the execution traces satisfy or violate the specifications. We describe each of these steps in detail as follows.

Step 1: Simulating and extracting execution traces.

To reduce the complexity of runtime monitoring, we simulate requested actions and extract traces from the projected operation of smart cities. When receiving action requests from smart services, we execute actions for a time period into the future on the simulated city, where the simulator is initialized to have the same states as the real city (i.e., city states of the moment when actions are requested). Then we extract the predicted execution traces from the simulation.

There are two types of predicted traces: (1) predicted actions issued by smart services and (2) predicted states of smart cities. Smart services request actions on the actuators in smart cities. For example, smart traffic services can order the traffic lights at certain street intersection to change the value of the signals at a given time or smart event services can request blocking certain streets. Such actions can take binary or numeric values (e.g., green or red for traffic signals, four illumination levels for street lights). The actions are often issued with time intervals (e.g., keep the traffic light green for 3 minutes). The predicted states of smart cities include the level of air quality, noise, waiting time of pedestrians at intersections, average speed of traffic, etc. Actions from smart services have effects on the city states. For example, the level of air quality decreases if a new chemical factory opens; the average speed of traffic on highway decreases if there is an accident and a lane is blocked. We represent traces of service actions and city states as real-time discrete or continuous signals (e.g., traffic light signals are discrete, while noise levels are continuous).

Step 2: Specifying safety and performance requirements.

Governments and companies have been publishing various rules, policies and laws regarding the safety and performance of smart cities. These requirements are often written in English (or other languages), which cannot be directly used as formal specifications for runtime monitoring. Mining formal specifications from natural language requirements is known as a difficult problem [6]. There are several additional challenges when considering requirements of smart cities. First, some requirements are vague or incomplete, without specific information about the location, time and conditions. Second, smart cities are open systems with significant uncertainty, thus the requirements should also allow certain degree of uncertainty. To this end, we distinguish *hard constraints* that must be satisfied (e.g., actions of smart services should not cause collisions of vehicles) and *soft constraints* that allow uncertainty (e.g., actions of smart services should not increase traffic congestion by more than 10%). To help people write formal specifications for smart city applications more easily, we present a set of specification patterns as templates in Section 3. Among the various choices of temporal logic for runtime monitoring, we use *Signal Temporal Logic* (STL) [5], a formalism for specifying requirements of dense-time real-valued signals. This is because the execution traces of smart cities include discrete, continuous and timed behaviors.

Step 3: Runtime monitoring for smart service conflict detection.

We can check if predicted execution traces of smart cities violate safety and performance specifications using runtime monitoring. Such violations are often caused by conflicts among smart services. To detect service conflicts, we instantiate specification patterns presented in Section 3 and monitor relevant traces of service actions and city states. We demonstrate the feasibility of detecting smart service conflicts using runtime monitoring via case studies in Section 4. As a result, we detect the potential conflicts and safety violation among actions of smart services in advance.

3. SIGNAL TEMPORAL LOGIC BASED SPECIFICATION PATTERNS

In this section, we present formal specification patterns for a set of typical safety and performance requirements in smart cities. These requirements are based on our previous work [9, 8], which are originally taken from public documents by U.S. Department of Transportation [3] and U.S. Environmental Protection Agency [1]. Our goal is to provide specification patterns as templates to help people write formal specifications of smart cities more easily. For example, smart city practitioners who are not familiar with temporal logic can instantiate a specification pattern by filling in parameters of the requirement (e.g., locations, time intervals, thresholds). It is expected that as new services are created and deployed there will be a need for additional patterns.

3.1 Safety and Performance Specifications

We use STL [5] to write specification patterns. Table 1 shows a set of typical smart city safety and performance requirements in transportation, emergency, and environment services. For each requirement, we provide a specification pattern written in STL. The instantiated specifications can be used to monitor the operation of smart cities at runtime, as shown in Figure 1.

The general template is ⟨TL Operator⟩⟨time interval ⟩⟨event/action⟩, where ⟨TL Operator⟩ denotes the operator in temporal logic (e.g. Always, Eventually, Until), ⟨time interval is during which time this requirement sustains, ⟨event/action⟩ can be an action or event. In the following, we present templates for creating these specification patterns and describe each specification pattern in detail.

3.1.1 Transportation.

We can use the following templates to generate STL specifications for transportation by filling in parameters of time interval and event:

- Always⟨time interval⟩⟨event⟩, where ⟨event⟩ can be a single objective (e.g., "no collision") or furthered parameterized with the template ⟨traffic efficiency metric⟩⟨threshold⟩.
- Eventually⟨time interval⟩⟨event⟩, where event can be the number of congested vehicles less than a threshold.
- ⟨action1⟩Until⟨time interval⟩⟨action2⟩, where examples of actions include turn a signal light for vehicles/pedestrians to green/red.

R1-*R6* in Table 1 are examples of transportation specifications generated using these templates.

R1 is a safety requirement for vehicle collisions. Suppose there is a binary signal Collision, which takes the value True if a vehicle collision occurs. We can, therefore, write a specification $\square(\neg\mathsf{Collision})$ to represent the safety requirement that "No vehicle collision should occur".

R2 is a performance requirement: "The number of vehicles in a lane should never exceed its maximum capacity". The specification uses VehicleNumber(lane) and MaxCapacity(lane) to represent two signals over a location variable lane, which need to be instantiated.

R3 is a performance requirement that compares the traffic congestion state over a time interval (a, b), which is measured by the number of vehicles waiting in a lane. The specification uses signal Congestion(lane) to denote the traffic congestion state in certain lane after the implementation of some smart services. Congestion′(lane) is a constant obtained from historical data, denoting the average previous congestion state of that lane during that time.

R4 and *R5* are very similar to *R3*, which are all performance requirements measuring the vehicle traffic efficiency. *R4* is about the Yield (i.e., number of vehicles that are unable to cross an intersection where they do not have priority). *R5* is about the average waiting time of vehicles in a lane, denoted as WaitTime(lane).

R6 is a performance requirement about the pedestrian traffic efficiency over a time interval (a, b). The specification uses signal Pedestrian(i) to denote the number of pedestrians waiting at an interaction i after the implementation of some smart services. Pedestrian′(i) is a constant, denoting the average number of pedestrians waiting in an interaction i.

3.1.2 Emergency.

We can use one of the following templates to generate STL specifications for emergency services.

- Always⟨time interval⟩⟨object⟩⟨metric⟩⟨threshold⟩, where the object, for example, can be an ambulance, police car or fire fighter truck, and the metric can be response time and waiting time.
- Eventually⟨time interval⟩⟨action⟩, where the action can be response for an accident, or resolve an accident.
- ⟨action1⟩Until⟨time interval⟩⟨action2⟩, where examples of actions include block a lane, and vehicle/pedestrian moves in a direction/lane.

R7-*R9* in Table 1 are examples of emergency specifications generated using such templates.

R7 uses signal EmergencyWaitTime(i) to denote the waiting time of emergency vehicles at an interaction, which should always be less than 10 seconds.

R8 uses two binary signals EmergencyDirection(lane) and Blocked(lane) to represent whether the emergency vehicle is directed to a certain lane and if a lane is blocked, respectively. If both signals are True, then the emergency vehicle is directed to a blocked lane, which violates the safety requirement.

R9 uses Blocked(lane) and ¬Blocked(lane) to denote the block and unblock of a lane. It also uses the Until operator to bound the required time interval.

3.1.3 Environment.

The following template can be used to generate STL specifications for environment requirements:

- Always⟨time interval⟩⟨metric⟩⟨threshold⟩, where metrics can be the level of air pollution and noise.
- Eventually⟨time interval⟩⟨metric⟩⟨threshold⟩, where the time interval can be used to control the time to reduce the pollution level.
- ⟨action1⟩Until⟨time interval⟩⟨action2⟩, where examples of actions include the level of air pollution reaching a threshold or time and allowing vehicles crossing a school area.

Table 1: Smart City Requirements (left) and Specification Patterns (right)

Transportation	
R1: No vehicle collision should occur.	$\Box(\neg \mathsf{Collision})$
R2: The number of vehicles in a lane should never exceed its maximum vehicle capacity.	$\Box(\mathsf{VehicleNumber(lane)} < \mathsf{MaxCapacity(lane)})$
R3: Traffic congestion in a lane should not increase by more than 10%.	$\Box_{(a,b)}(\mathsf{Congestion(lane)} < 1.1 * \mathsf{Congestion'(lane)})$
R4: Traffic yield in a lane should not increase by more than 20%.	$\Box_{(a,b)}(\mathsf{Yield(lane)} < 1.2 * \mathsf{Yield'(lane)})$
R5: The average waiting time of vehicles in a lane should not increase by more than 10%.	$\Box_{(a,b)}(\mathsf{WaitTime(lane)} < 1.1 * \mathsf{WaitTime'(lane)})$
R6: The number of pedestrians waiting in an intersection by more than 200%.	$\Box_{(a,b)}(\mathsf{Pedestrian(i)} < 2 * \mathsf{Pedestrian'(i)})$
Emergency	
R7: Emergency vehicles should not wait for more than 10s at an intersection.	$\Box\ (\mathsf{EmergencyWaitTime(i)} < 10)$
R8: Emergency vehicles should not be directed to a blocked lane or area.	$\Box\neg(\mathsf{EmergencyDirection(lane)} \wedge \mathsf{Blocked(lane)})$
R9: The highway blocked by an emergency accident should be unblocked within 30 min.	$\mathsf{Blocked(lane)}\ \mathbf{U}_{(1,30)}\ (\neg\mathsf{Blocked(lane)})$
Environment	
R10: The noise level in a lane should always be less than 70 dB.	$\Box_{(a,b)}\ (\mathsf{Noise(lane)} < 70)$
R11: The noise level in a lane should be reduced to less than 60 dB after some point.	$\Diamond_{(a,b)}\ (\mathsf{Noise(lane)} < 60)$
R12: The carbon monoxide (CO) emission in a lane should always be no more than 50 mg.	$\Box_{(a,b)}\ (\mathsf{CO(lane)} < 50)$
R13: The hydrocarbons (HC) emission in a lane should always be no more than 1 mg.	$\Box_{(a,b)}\ (\mathsf{HC(lane)} < 1)$
R14: The particulate matter (PMx) emission in a lane should always be no more than 0.2 mg.	$\Box_{(a,b)}\ (\mathsf{PMx(lane)} < 0.2)$
R15: The camera should be turned on and the illumination of street light should be set at least level 3 within a time interval.	$\Diamond_{(a,b)}(\mathsf{Camera(lane)} \wedge \mathsf{Illumination(lane)} > 3)$

R10-R15 are examples of environment specifications generated using these templates.

R10 and *R11* are about the noise level, which use the signal Noise(lane) to denote the noise level in a lane. Similarly, *R12*, *R13* and *R14* use signals CO(lane), HC(lane) and PMx(lane) to denote the carbon monoxide, hydrocarbons and particulate matter emission in a lane, respectively. *R15* uses a signal Camera(lane) to represent that the camera is turned on and a signal Illumination(lane) to denote the illumination levels of street lights. Note that the numerical thresholds used in these requirements are just example parameter values and can be changed.

3.2 Smart Services Actions Conflicts

The safety and performance specifications listed in Table 1 are all about the states of smart cities (e.g., traffic congestion, air pollution). The actions of smart services can have positive or negative effects on these city states, and thus have the potential of leading to requirement violations and service conflicts. In addition, smart services may issue contradictory actions on the same actuator, which are identified as *device conflicts* in [8]. There are three types of device conflicts. We provide STL specification patterns for each type as follows.

Opposite device conflicts occur when multiple smart services issue opposite (binary) actions to the same actuator. For example, smart traffic service turns a traffic light to green, while smart pedestrian service requests the same light to red. Let $A_1, A_2, \ldots, A_n$ be multiple boolean-valued actions issued on the same actuator. We can write a specification $\Box\neg(A_1 \oplus A_2 \cdots \oplus A_n)$ to detect opposite device conflicts, where $\oplus$ denotes the exclusive or operator.

Numeric device conflicts occur when smart services issue multiple actions with different numeric parameters. For example, the smart energy service tries to set the illumination of street lights to level 1 to save energy, while the smart safety service maintains it at level 3 because the camera to monitor the community requires higher illumination. Let $A_1, A_2, \ldots, A_n$ be multiple real-valued actions issued on the same actuator. We can use the formal specification $\Box((A_1 - A_2 = 0) \wedge ... \wedge (A_{n-1} - A_n = 0))$ to monitor if these actions assign the same numeric value.

Duration device conflicts occur when smart services require actions with different time intervals. Let A_i and A_j be two actions. We consider three scenarios. First, A_j cannot be taken within m steps after A_i. Formally, we write $A_i \wedge (\Box_{(1,m)}\neg A_j)$. An example is "Traffic light should not be turned to green within 2 seconds after it is turned to red". Second, A_j cannot be taken until A_i stops (e.g., the street should be kept blocked until the accident is resolved). The formal specification is $A_i \mathbf{U}(\neg A_i \wedge A_j)$. Third, we can write $A_i \wedge (\Diamond_{(1,m)} A_j)$ to represent the time dependence between actions. For example, if the accident service blocks an area, it should release blocking eventually within m steps.

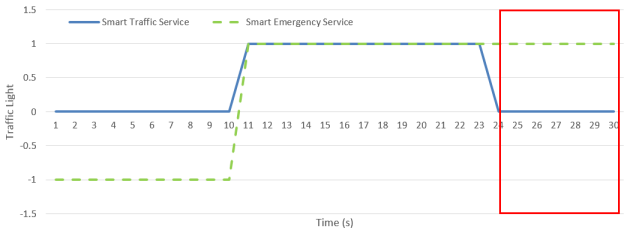

Figure 2: Action conflict caused by Smart Traffic Service (blue solid line) and Smart Emergency Service (green dashed line). Action values of the traffic signal (1: green, 0: red, -1: no action). The red square highlights the detected action conflict.

4. CASE STUDY

The case study is based on a smart city simulator SUMO [4], in which we have implemented six smart services (i.e. Smart Traffic Service, Smart Pedestrian Service, Smart Event Service, Smart Emergency Service, Smart Air Pollution Service and Smart Noise Control Service. Please refer to [8] for details.). The simulated smart city platform covers the lower half of Manhattan with 102 streets and 454 traffic lights. Simulation uses traffic data of Manhattan from 8:00 am to 2:00 am over about 6 months (9/28/2012 - 4/19/2013) [2]. We apply the Breach toolbox [5] for runtime monitoring.

Case 1: Monitor service action conflict.

In case 1, we consider two smart services. The smart traffic service adjusts a traffic signal to optimize traffic and relieve traffic congestion, while the smart emergency service requests a green light when an emergency vehicle is approaching. We monitor two traces of service actions on the same

Figure 3: Action of Smart Traffic Service causes the conflicts and violations of environment requirements. (a) illustrates the action trace on a traffic light (1: green, 0: red), while (b), (c), (d) and (e) show the traces of CO, Noise, waiting time of emergency vehicle and number of waiting pedestrian. The red squares highlight the detected conflicts.

traffic light, as illustrated in Figure 2. We use the STL specification $\square\neg(\mathsf{A_e} = 0 \wedge \mathsf{A_t} = 1) \vee (\mathsf{A_e} = 1 \wedge \mathsf{A_t} = 0)$, where $\mathsf{A_e}$ and $\mathsf{A_t}$ denote the actions of the emergency and traffic services, respectively. The result of STL runtime monitoring returns False and detects a device action conflict at time 24s (as highlighted in Figure 2). This is indeed a real device conflict. From time 1s to 10s, no action is taken by the smart emergency service. From time 11s to 23s, both services order the same action for a green light. However, at time 24s, the smart traffic service turns the signal light to red while the smart emergency service keeps it at green, which are contradictory actions.

Case 2: Monitor the effect conflict from a single service.

In this case, we monitor the effects on smart city states (CO, Noise, waiting time of emergency vehicles and the number of waiting pedestrians) caused by the action of the smart traffic service. We instantiate four specifications *R12*, *R10*, *R7*, and *R6* from Table 1. The results of STL runtime monitoring show that there are conflicts detected with *R12* and *R10*, while *R7* and *R6* are satisfied. Figure 3 illustrates the traces of service actions and city states. Figure 3 (a) shows that, at time 6s, the traffic light signal changes. Figure 3 (b) and

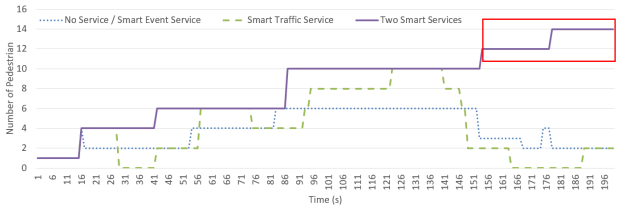

Figure 4: Conflict caused by Smart Traffic Service and Smart Event Service. Traces of the number of pedestrians waiting in an intersection, when there is no smart service or event service only (blue dotted line), traffic service only (green dashed line), or with both services (purple solid line). The red square highlights the detected conflict.

(c) show the traces of CO and Noise over time for *R12* and *R10*, both of which increase after the action of the smart traffic service. At around time 25s, conflicts are reduced with these two requirements (as highlighted by red squares in the figures). Figure 3 (d) and (e) show the traces of the waiting time of emergency vehicles and the number of waiting pedestrians. There are no highlighted conflicts in these traces, because both requirements are satisfied.

Case 3: Monitor the effect conflict from multiple services.

In this case, we monitor the additive effects of smart traffic service and the smart event service. We instantiate *R6* from Table 1, which says that the action should not increase the number of pedestrians waiting at an intersection by 20%. We simulate and monitor traces representing the number of pedestrians waiting at an intersection when there is (1) no smart service, (2) smart event service only, (3) smart traffic service only, and (4) both services. These traces are drawn in Figure 4, from which we make the following observations:

- The traces with no service and the one with smart event service are overlapped (the blue dotted line), indicating the smart event service has no effect on the number of pedestrians by itself.
- The smart traffic services affect the number of pedestrians. Sometimes (e.g., in time 29s - 41s) it decreases the number and sometimes it increases the number (e.g., in time 97s - 145s), but it does not violate the requirement.
- When running the smart traffic service and smart event service simultaneously, the number of pedestrians increases dramatically and violates the requirement at around time 155s (as highlighted in the red square).

The smart event service does not affect pedestrians on its own, because the service only blocks roads of vehicles rather than pedestrians. However, when the smart traffic service is running at the same time, it adjusts the traffic light based on the traffic condition. In order to relieve the vehicle congestion caused by a blocked street, the smart traffic service may request the signal light of one direction to be green for a longer time. As a result, pedestrians from the other direction need to wait longer.

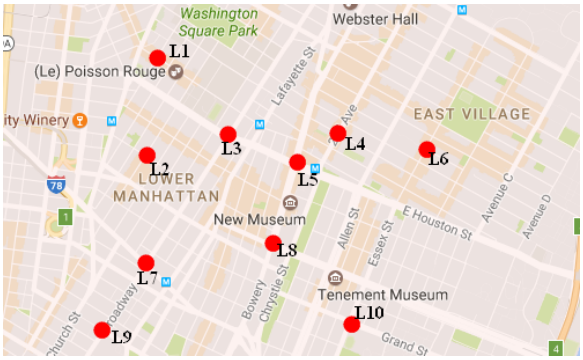

Figure 5: Simulated Manhattan with 5 smart services running at 10 different locations

Table 2: Number of Conflict Detected in 1 hour

Requirement	T1 (5-6 am)	T2 (12-1 pm)	T3 (5-6 pm)
R5: Waiting Time	86	501	831
R6: Pedestrian	19	151	189
R7: Emergency	0	6	11
R10: Noise	53	421	483
R12: CO	91	360	520

Case 4: Monitoring large scale traces.

We also run experiments on traces simulated from 5 smart services (e.g. S1, S2, S4, S5 and S6) and 10 locations over 3 hours, which involve 10 traffic signals and 118 lanes (see Figure 5). We choose three one-hour durations in the morning, afternoon, and evening for diverse traffic conditions. The traffic volume is lowest during T1 and highest during T3. Table 2 shows the number of conflicts detected using these traces for five requirements.

- The number of conflicts detected varied depending on the time. During T1, there are less number of vehicles and lower levels of noise and pollution, thus smart services are not triggered as frequently as during T3, when the traffic is congested. It also indicates the necessity of using runtime monitoring because the conflicts depend on the context.
- Although a large number of conflicts happened, some of them last for an extended period and some of them lasted only for a few seconds and then returned to normal. For example, in the case of R5 during T3, there are 831 conflicts detected. About 600 of these conflicts last less than 5 seconds, which can be a good indicator for conflict resolution.
- R5 and R6 are performance requirements, while R7, R10 and R12 are safety requirements. In the case of R7, although the number of conflicts is fewer than other requirements, it is very important to detect them.

In summary, the experimental results of the above typical cases demonstrate that we can use STL runtime monitoring to detect various conflicts caused by smart services. Particularly, runtime monitoring using STL helps detect that effect conflicts took place after the occurrence of service actions within a certain period. This is very useful for the conflict detection, because effects of service actions often only become observable after some time. In addition, STL runtime monitoring is capable of detecting conflicts caused by the additive effects of multiple smart services, which is difficult to identify manually.

5. ACKNOWLEDGMENTS

This work was funded, in part, by NSF under grants CNS-1527563 and CNS-1319302.

6. REFERENCES

[1] *Air Quality Planning and Standards, U.S. Environmental Protection Agency.* https://www3.epa.gov/airquality/index.html.

[2] *New York City Open Data.* https://nycopendata.socrata.com/.

[3] *Safety and Health, U.S. Department of Transportation.* https://www.transportation.gov/policy/transportation-policy/safety.

[4] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzewicz. Sumo–simulation of urban mobility: an overview. In *Proceedings of the 3rd International Conference on Advances in System Simulation*. ThinkMind, 2011.

[5] A. Donzé and O. Maler. Robust satisfaction of temporal logic over real-valued signals. In *Proceedings of International Conference on Formal Modeling and Analysis of Timed Systems*, pages 92–106, 2010.

[6] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner. Arsenal: Automatic requirements specification extraction from natural language. In *Proceedings of the 8th International Symposium on NASA Formal Methods*, pages 41–46, 2016.

[7] C.-J. M. Liang, B. F. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu. Sift: building an internet of safe things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 298–309, 2015.

[8] M. Ma, S. M. Preum, and J. A. Stankovic. Cityguard: A watchdog for safety-aware conflict detection in smart cities. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, pages 259–270, 2017.

[9] M. Ma, S. M. Preum, W. Tarneberg, M. Ahmed, M. Ruiters, and J. Stankovic. Detection of runtime conflicts among services in smart cities. In *Proceedings of IEEE International Conference on Smart Computing*, pages 1–10. IEEE, 2016.

[10] S. Munir, M. Ahmed, and J. Stankovic. Eyephy: Detecting dependencies in cyber-physical system apps due to human-in-the-loop. In *Proceedings of the 12th EAI International Conference on Mobile and Ubiquitous Systems*, pages 170–179, 2015.

[11] S. Munir and J. Stankovic. Depsys: Dependency aware integration of cyber-physical systems for smart homes. In *Proceedings of ACM/IEEE International Conference on Cyber-Physical Systems*, 2014.

[12] S. M. Preum, A. S. Mondol, M. Ma, H. Wang, and J. A. Stankovic. Preclude: Conflict detection in textual health advice. In *Proceedings of IEEE International Conference on Pervasive Computing and Communications*, pages 286–296. IEEE, 2017.

[13] I. UK Government Department for Business and Skills. Smart cities - background paper, 2013.