Using IoTGolog to Formalize IoT Scenarios

Shah Rukh Humayoun[1], Yael Dubinsky[2], Ragaad AlTarawneh[1]
[1]Computer Graphics and HCI Group
University of Kaiserslautern
67663 Kaiserslautern, Germany
[2]IBM Research - Haifa
Mount Carmel, Haifa 31905, Israel
[1]{humayoun, tarawneh}@cs.uni-kl.de, [2]dubinsky@il.ibm.com

***Abstract*— Internet of Things (IoT) scenarios exist in various domains and due to the vast changes in technology and systems modernization, they are constantly evolving. In order to cope with this dynamicity, there is a need to provide a robust yet flexible way to design, develop, and evaluate the IoT architecture and implementation. We present IoTGolog, a formal task modeling language to define and evaluate IoT scenarios. Using IoTGolog, characteristics of the IoT scenario as well as the tests to evaluate it are formalized. This provides accuracy and better control over the IoT evolution. IoTGolog extends our previous work on TaMoGolog and MobiGolog for general and mobile-based scenarios respectively. We illustrate IoTGolog for the case of car emergency response system.**

Keywords— Internet of Things (IoT); mobile systems; task models; formal languages; IoTGolog; TaMoGolog.

I. INTRODUCTION

Recently, we witness a growing interconnectivity amongst different smart entities (e.g., sensors, objects, systems, etc.). The interconnectivity between these smart entities enables the concept of Internet of Things (IoT) [1, 2, 18], which allows these smart entities to cooperate with each other seamlessly to exchange information and to perform tasks in more automated forms. The recent advancements in mobile systems enhanced this concept of IoT [5, 7, 16, 17], as it reduces the limitation of the physical connectivity between different entities.

The resulting applications provide *smart environments* [5, 6, 7, 8, 17] (e.g., smart homes, smart factories, smart universities, smart traffic infrastructure), which are consisted of heterogeneous smart entities, ranging from mobile systems, sensors, objects, to normal systems. Therefore, each smart environment is constructed of heterogeneous smart entities and the communication channels between them. However, it leads to a set of challenges when designing, developing, or evaluating different IoT scenarios due to a number of reasons, e.g., the dynamicity, the diversity of participating entities' types, the constantly evolution of participating entities, and the interaction between these entities, etc.

In order to cope with these challenges, there is a need to provide a robust, yet flexible way that can be used from specifying to evaluating IoT scenarios. Targeting this concern, we focus on defining IoT scenarios formally, so that to specify and model these scenarios unambiguously and accurately as well as to use the resulting formal specification for the evaluating purposes.

TaMoGolog (Task Modeling Golog) [9, 10], a formal task modeling language, was built on the top of the Golog family [3, 4, 15] of high-level programming languages. It provides well-defined syntax and semantics, enables precondition axioms of tasks, states post condition effects to variables due to the successful task execution, provides a rich set of operators, and gives the facility to express the domain knowledge in task models. TaMoGolog has been successfully used previously for defining task models at different abstraction levels [9], for defining tasks in a user-centric interactive environment [11], for performing task-based automated evaluation at the development environment level [12], and for specifying user multi-touch gestures interaction on mobile devices [14] and user interaction requirements for mobile apps [13] using its extension MobiGolog.

In this work, we extend TaMoGolog language to work over IoT Scenarios. We present IoTGolog that provides a new set of predicates, in addition to the TaMoGolog original set of predicates and constructs, to handle the dynamicity of IoT scenarios. We focus on modeling those scenarios in which users are involved. Therefore, other aspects dealing with system specifications (e.g., packet loss in the IoT network) are out of the scope of this paper. We also present a use case from the car emergency response in traffic environment to show how such a scenario can be modeled formally. Providing such unambiguous formal specifications of IoT scenarios not only helps in understanding these scenarios accurately between different involved parties, but would also enable us to learn users' behavior patterns in such environments. Further, they could be used for generating system models and performing automated testing using task-based approaches.

The remainder of the paper is structured as follows: In Section II, we provide background details of TaMoGolog that is necessary to understand the IoTGolog specification. In Section III, we introduce our IoTGolog predicates. In Section IV, we provide specification of the car emergency use case using IoTGolog. Finally, we conclude the paper and shed light on future directions in Section V.

II. BACKGROUND: TAMOGOLOG

TaMoGolog distinguishes tasks using three main categories [9, 10]: *unit* tasks (denoted as μ) that are performed in an atomic manner and are also called *atomic actions*, *waiting* tasks (denoted as ϖ) that wait either for a particular event to happen or for some set of conditions to be fulfilled, and *composite* tasks (denoted as Γ) that handle the structural behavior of the task model. Below, we list some of the important predicates. The complete set of predicates is available in [10].

- *UnitTask*(α): α is a unit task.
- *CompositeTask*(Γ): Γ is a composite task.
- *WaitingTask*(ϖ): ϖ is a waiting task.
- *Type*(t): t is a new task type (targeting some specific domain) in additional to the three main task categories.
- *Task*($\mathscr{b}$): $\mathscr{b}$ is a task, which can be a kind from the three main categories or the new one defined.
- *TaskType*($\mathscr{b}$, t): to associate a task $\mathscr{b}$ to a new type t.
- *Precondition*(α) $\equiv \Pi_\alpha$: formula Π_α defines the set of axioms that must be true at the time of execution of a unit task α.
- *Postcondition*(α, v, $\Omega_{(\alpha,v)}$) $\equiv \phi_{(\alpha,v)}$: formula $\phi_{(\alpha,v)}$ defines the effects of executing unit task α on related variable v under any condition $\Omega_{(\alpha,v)}$.
- *Fluent*(f): f is a functional or relational fluent (variable).
- *InitialState*(m) $\equiv l_m$: describes the values of variables at the beginning of a task model m through the formula l_m.
- *TaskModel*(m): m is the task model name.
- *Agent*(*agt*): *agt* represents some external entity (e.g., human or application) participating in the task model.
- *Responsible*(*agt*, α): external entity *agt* is responsible for executing task α.
- *Goal*(g, v) $\equiv \Delta_{(g,v)}$: goal g is defined by Δ_g on fluent v.
- ***proc*** $P(\vec{x})$ δ ***end***: defines the definition of composite/waiting task through procedure definition of Golog-family [3, 15] where P is the name of composite/waiting task, $\vec{x}$ represents the option of parameter passing, and δ represents the composite/waiting task definition. Calling to a composite/waiting task in another place is simply by $\Gamma(\vec{x})$.

TaMoGolog semantics [10] is based on the Golog-family of languages, in which a *unit* task executes (in atomic manner) if all precondition axioms are true; then the effects are shown on related variables (fluents) after execution. Composite tasks are performed in step-by-step executions. Each task model defines one or more paths (depending on non-determinism) to achieve a goal.

TaMoGolog provides a rich set of operators [10], mostly obtained from the Golog-family languages. Operator $\emptyset$? represents a *waiting* or *testing* condition; operator $[\Gamma_1; \Gamma_2]$ represents *sequence* in which task Γ_2 starts after task Γ_1 is finished; and operators $[\Gamma_1|\Gamma_2]$ and $[agt\ \Gamma_1|\Gamma]$ represent internal and external *nondeterministic choice*, respectively. In this case, *agt* means some external entity (application/system or human user) that decides the nondeterministic branch. Operator [**if** ϕ **then** Γ_1 **else** Γ_2] is a normal *if-then-else* choice in which ϕ is a conjunction of conditions. Operators $[\pi x.\ \Gamma(x)]$ and $[agt\ \pi x.\ \Gamma(x)]$ are *nondeterministic choice of argument* in which the system or the external entity chooses the variable binding for the task and then the task is executed accordingly. Operators $[\Gamma]^*$ and $[agt\ \Gamma]^*$ are *nondeterministic iterations* (internally and externally). Operator [**while** ϕ **do** Γ] is the usual *while-do* iteration. Operator $[\Gamma_1||\Gamma_2]$ represents the *interleaving concurrency* of tasks while operator $[\Gamma_1 \gg \Gamma_2]$ represents the *priority concurrency* in which task Γ_2 continues only if task Γ_1 is finished or is in the blocking state. Operator $[\Gamma]^{||}$ is for *concurrent iteration* while operator $[agt\ \Gamma_1 <> \Gamma_2]$ represents that the external entity *agt* decides the priority concurrency at run time. Operator $[agt\ \Gamma]^{||}$ represents the concurrent iteration decided by external entity while operator $<\phi \rightarrow \Gamma>$ represents *interrupt* where when the condition ϕ becomes true and the interrupt has control and the task Γ is started. Operator $[\Gamma_1 > \Gamma_2]$ represents failure handling, when Γ_2 executes if Γ_1 fails to finish.

III. IOTGOLOG PREDICATES FOR IOT SCENARIOS

In this section, we provide IoTGolog set of predicates to define formally IoT scenarios. These new predicates, along TaMoGolog predicates and constructs, are useful for defining different entities, participating in an IoT scenario, as well as the communication between these entities.

A. Defining IoT Entities and Communication Channels

- **Agent(agt):** *agt* represents some entity (e.g., sensor, device, application, etc.) participating in building up an IoT sceanrio.
- **InCom(agt, $\mathcal{V}(\vec{x})$) $\stackrel{\text{def}}{=}$ UnitTask(InCom(agt, $\mathcal{V}(\vec{x})$)):** A new predicate ***InCom*** (***In****ward-****Com****munication)* to define the message/notification coming from other entities in an IoT sceanrio. Here, *agt* is some other entity from which the system accepts $\mathcal{V}(\vec{x})$ set of variables. By definition, it is a unit task (or *atomic* action); therefore, it shows the execution effects on some variables. We differentiate from one *InCom* to another one based on the sending entity (i.e., the *agt*) and the variables.
- **OutCom(agt, $\mathcal{U}(\vec{y})$) $\stackrel{\text{def}}{=}$ UnitTask(OutCom(agt, $\mathcal{V}(\vec{y})$)):** A new predicate ***OutCom*** (***Out****ward-****Com****munication)* to define the message/notification going towards some other participating entity in an IoT sceanrio. Here, *agt* is the other participating entity to which the system sends $\mathcal{U}(\vec{y})$ set of variables. By definition, it is also a unit task.

In IoTGolog, these three predicates are used to define different entities and the communication channels between them in order to build an IoT scenario. Through *InCom* and *OutCom* predicates, we can define different communications between different kinds of entities or with the same kind of entities. The differentiation is made through the participating entities and the set of variables.

B. Defining Actions/Tasks' Association

- **TaskType(InCom(agt, $\mathcal{V}(\vec{x})$), exo):** *TaskType* is a TaMoGolog predicate that is used to associate a task/action to another type in addition to the basic types (i.e., *unit/atomic*, *composite*, and *waiting*). Here, we associate *InCom* action/task to *exo* type (i.e., exogenous actions or tasks). The definition of *exo* actions/tasks is taken from the ConGolog [3], which define them as actions that are generated by some external systems and the underlying system itself does not have control over their execution. In our context, the other participating entities (e.g., sensors, objects, or applications) in an IoT scenario can start *InCom* actions/tasks in order to notify inputs or to communicate with the system. Therefore, they are triggered whenever any other participating entity activates them.

- **TaskType(OutCom(agt, $\mathcal{U}(\vec{y})$), signalOut):** *signalOut* is a new action/task type in IoTGolog to indicate if an action/task is for the purpose of sending communication to other entities participating in an IoT scenario. However, it is important to note that when we define the specification from that receiving communication entity, this *OutCom* action/task works as an *InCom* action/task.

C. Defining Communication Effects

- The TaMoGolog *Postcondition* axiom is used to indicate effects of an exogenous action/task on one or more variables whenever it is triggered by some other entity in an IoT sceanrio. It is also valid for the any OutCom action/task. It is represented by:

  Postcondition(InCom(agt, $\mathcal{V}(\vec{x})$), v,
  $\Omega_{(\mathbf{InCom/OutCom}(\mathbf{agt},\mathcal{V}(\vec{x})),\mathbf{v})}$)≡ $\Phi_{(\mathbf{InCom/OutCom}(\mathbf{agt},\mathcal{V}(\vec{x})),\mathbf{v})}$

- **NotificationOut(ϕ, OutCom(agt, $\mathcal{U}(\vec{y})$)** $\stackrel{\text{def}}{=}$ **$< \phi \rightarrow$ OutCom(agt, $\mathcal{U}(\vec{y})) >$:** IoTGolog new predicate *NotificationOut* defines under what condition or set of conditions ϕ the system triggers the *OutCom* action/task in order to communicate with other entities in the IoT. This predicate is defined using the standard interrupt call construct in TaMoGolog, originally taken from ConGolog [3], which means whenever the condition or a set of conditions ϕ is fulfilled the corresponding *OurCom* action/task will be triggered.

- **NotificationEffect(ϕ, Γ)** $\stackrel{\text{def}}{=}$ **$< \phi \rightarrow \Gamma >$:** The new predicate *NotificationEffect* in IoTGolog defines the effects of some received notification from other entities in an IoT sceanrio. This predicate is also defined using the interrupt call construct, as same in the case of the *NotificationOut* predicate. However, here the resulting execution task is a composite task rather than the *OutCom* action/task. This composite task may compose of a unit task or a detailed task structure in order to act upon the received communication.

D. Describing the Whole System

On the upper level, an entity in an IoT sceanrio is consisted of $(\mathbf{T_{EXO}} \gg \mathbf{T_{EFT}} \gg \mathbf{T})$, where T_{EXO} stands for the program consisted of all exogenous actions/tasks, T_{EFT} means the program consisted of all *NotificationOut* and *NotificationEffect* predicates, and T stands for the program consisted of the entity's normal activities. These three parts work in priority concurrency, i.e., T_{EXO} has a priority concurrency over the other two parts, and T_{EXO} has a priority over T. Therefore, whenever any other entity triggers an *InCom* action/task, then the control is instantly transformed to this triggered action/task. After this, the second priority is given to any *OutCom* action/task or any task in *NotificationEffect*. The entity's routine activities managed through the T part are on the least priority in this currency.

Inside T_{EXO}, we use the same definition as defined by [3] that means executing non-deterministically chosen actions/tasks until any one of them is ready to be executed. Inside T_{EFT}, we assume all parts related to *NotificationOut* and *NotificationEffetc* predicates are in normal concurrency; hence, they execute whenever they get the control and the condition is fulfilled.

IV. Illustrating IoTGolog: Car Emergency Response System

In this section, we illustrate how to formalize IoT scenarios using IoTGolog.

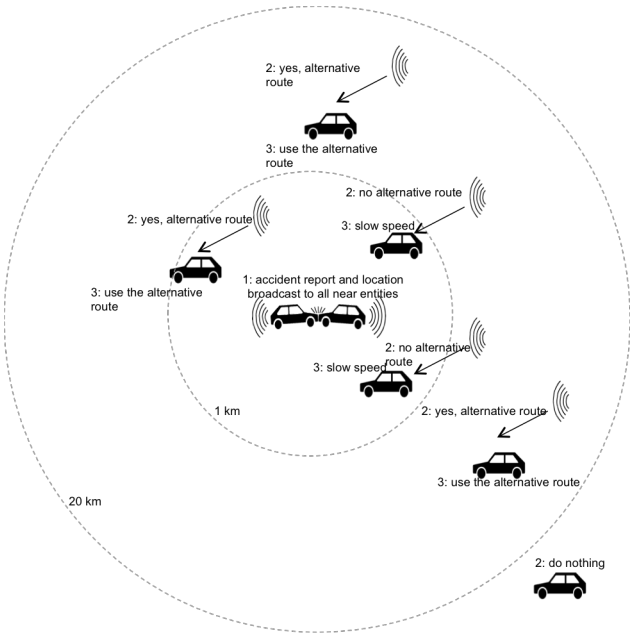

Fig. 1. An IoT scenario illustration for the car emergency response system. Here the circle of 1 km and 20 km are not equally scaled due to the space limitation.

This IoT use case we present involves several of the emergency response systems that can be activated in the case of a car accident. Specifically, we aim to alert all cars' drivers that they approach the place of an accident as follows: 1) All cars that are within the radius of 1 km and have no other recommended routes, will get an alert message that reports about the accident and asks to reduce the speed. 2) All cars that are within the radius of 20 km and have one or more alternative routes, will receive an alert message that reports about the accident and provides alternative recommendations. Further, if a car is involved in a serious accident, it will send an alert message to all other cars around it with the details of accident (e.g., report about the accident, the location, etc.). Figure 1 illustrates such a scenario.

Figure 2 provides an IoTGolog specification for this scenario. Due to the space limitation, we provide only those parts of the specification that are necessary to understand this use case. It describes that there are two types of entities participating in this scenario, i.e., cars and the route finder service. However, this specification is from the perspective of cars. A car can send or receive communication signals from other cars and the route finder service (lines 14–18).

In the case of an accident happens, the system inside the car lets it know about it (that's why *accident* is an exogenous task) and makes changes in the variable values (lines 20–26). This triggers a communication between this car and all other cars around (lines 40–41). When other cars around will receive this message through the *InCom* action (line 8), the resulting effect is shown on the variable values (lines 30–39). Due to this effect, these cars also send details to other cars around (lines 47–48) and find the solution for them (lines 50–51). In the case if the distance is less than 20 KM, then the emergency route finder operation is started (lines 53-60) through the route finder service entity. In the case of more than 20 KM distance, cars will discard the alarm message and nothing will be happened.

This IoT use case we present involves several of the emergency response systems that can be activated in the case of a car accident. Specifically, we aim to alert all cars' drivers that they approach the place of an accident as follows: 1) All cars that are within the radius of 1 km and have no other recommended routes, will get an alert message that reports about the accident and asks to reduce the speed. 2) All cars that are within the radius of 20 km and have one or more alternative routes, will receive an alert message that reports about the accident and provides alternative recommendations. Further, if a car is involved in a serious accident, it will send an alert message to all other cars around it with the details of accident (e.g., report about the accident, the location, etc.). Figure 1 illustrates such a scenario.

As we described earlier in the previous section that at the upper level, an entity in the IoT scenario has the program in the form of $(\mathbf{T}_{\boldsymbol{EXO}} \gg \mathbf{T}_{\boldsymbol{EFT}} \gg \mathbf{T})$). In Figure 3, we provide descriptions of $\mathbf{T}_{\boldsymbol{EXO}}$ and $\mathbf{T}_{\boldsymbol{EFT}}$ in our scenario formalized in Figure 2. Here, $\mathbf{T}_{\boldsymbol{EXO}}$ (lines 1–2) is managed according to the definition of exogenous tasks execution as defined in [3]. This means the system checks continuously whether there is any communication signals coming form other entities.

```
1.  % Entities and tasks initialization
2.  Agent(Car).                          Agent(RouteFinder).
3.  UnitTask(accident).                  TaskType(accident, exo).
4.  UnitTask(emergencyOFF).              CompositeTask(adoptRoute(route)).
5.  Task(reduceSpeed).                   Task(findDistance(location1, location2)).
6.  CompositeTask(emergencyRouteFinder).
7.  % Fluents (variables)
8.  Fluent(emegencyType).                Fluent(currentLocation).
9.  Fluent(emergencyLocation).           Fluent(emergencyReport).
10. Fluent(emergencyStatus).             Fluent(emergencyAlarm).
11. Fluent(destination).                 Fluent(distance)
12. Fluent(isAlternativeRoute).          Fluent(route).
13.  % Communication with entities through InCom and OutCom predicates
14. InCom(Car, (emergency, report, location)).
15. InCom(RouteFinderService, (isRoute, newRoute)).
16. OutCom(Car, (emergencyType, emergencyReport, accidentLocation)).
17. OutCom(RouteFinderService, (currentLocation, emergencyLocation,
18.                                                    destination)).
19. % Postcondition axioms for "accident" and emergencyOFF tasks
20. Postcondition(accident, emergencyAlarm, null) ≡ emergencyAlarm = TRUE
21. Postcondition(accident, emergencyType, null) ≡ emergencyType = ACCIDENT
22 Postcondition(accident, emergencyLocation, null) ≡
23.                                    emergencyLocation = currentLocation
24. Postcondition(accident, emergencyReport, null) ≡ emergencyReport = REPORT
25. Postcondition(emergencyOFF, emergencyStatus, null) ≡
26.                                                 emergencyStatus = FALSE
27. % Postcondition axioms of some Outcom and InCom tasks
28. Postcondition(OutCom(Car, (emergencyType, emergencyReport,
29.                                    accidentLocation)), hit, null) ≡ hit = FALSE
30. Postcondition(InCom(Car, (emergency, report, location)), emergencyType,
31.                                    null) ≡ emergencyType = emergency
32. Postcondition(InCom(Car, (emergency, report, location)), emergencyReport,
33.                                    null) ≡ emergencyReport = report
34. Postcondition(InCom(Car, (emergency, report, location)), emergencyLocation,
35.                                    null) ≡ emergencyLocation = location
36. Postcondition(InCom(Car, (emergency, report, location)), emergencyStatus,
37.                                    null) ≡ emergencyStatus = TRUE
38. Postcondition(InCom(Car, (emergency, report, location)), distance, null) ≡
39.                          distance = findDistance(currentLocation, location)
40. Postcondition(InCom(RouteFinderService, (isRoute, newRoute)),
41.                          isAlternativeRoute, null) ≡ isAlternativeRoute = isRoute
42. Postcondition(InCom(RouteFinderService, (isRoute, newRoute)), newRoute,
43.                                    isAlternativeRoute) ≡ route = newRoute
44. % Communication to other entities in the case of accident happened
45. NotificationOut(emergencyAlarm, OutCom(Car, (emergencyType,
46.                                    emergencyReport, emergencyLocation))
47. NotificationOut(emegencyStatus, OutCom(Car, (emergencyType,
48.                                    emergencyReport, emergencyLocation))
49. % Effect execution if the car is within or above 20 KM of accident
50. NotificationEffect(emergencyStatus ∧ distance <= 20, emergencyRouteFinder)
51. NotificationEffect(emergencyStatus ∧ distance > 20, emergencyOFF)
52. % Composite task for finding alternative route if car is within 20 KM
53. Proc emergencyRouteFinder
54.         showReport(emergencyReport);
55.         OutCom(RouteFinderService, (currentLocation, emergencyLocation,
56.                                                    destination));
57.         If ←(isAlternativeRoute) ∧ (distance < 1) then reduceSpeed;
58.                    else adoptRoute(route);
59.         emergencyOFF
60. end
```

Fig. 2. Formal specification of car emergency response system scenario using IoTGolog

When it receives one or more communication signals from other entities, the system selects one of them non-deterministically and executes it. The system repeats it until there is no one exogenous action/task remaining. In the case of $\mathbf{T}_{\boldsymbol{EFT}}$ (lines 3–11), all the *NotificationOut* and *NotificationEffetc* predicates are in parallel concurrency. They are all in a non-deterministic iteration and as the condition of any of them is fulfilled the corresponding task starts its

execution. The **T** part here consists of the entity normal activities, which is out of the scope of this paper.

1. $\mathbf{T_{EXO}} = [\pi\alpha.(Exo(\alpha) \wedge (\alpha = \text{InCom}(\text{Car}, (\text{emergency}, \text{report}, \text{location})) \vee \alpha = \text{InCom}(\text{RouteFinderService}, (\text{isRoute}, \text{newRoute})))?; \alpha]^*$
3. $\mathbf{T_{EFT}} = [$
4. NotificationOut(emergencyAlarm, OutCom(Car, (emergencyType,
5. emergencyReport, emergencyLocation)) ||
6. NotificationOut(emegencyStatus, OutCom(Car, (emergencyType,
7. emergencyReport, emergencyLocation)) ||
8. NotificationEffect(emergencyStatus ∧ distance <= 20,
9. emergencyRouteFinder) ||
10. NotificationEffect(emergencyStatus ∧ distance > 20, emergencyOFF)
11.]*

Fig. 3. Definition of T_{EXO} and T_{EFT}

In general, writing IoT scenarios in this way not only provides an unambiguous and accurate specification, which can be used as a medium of communication between different parties involved in the design and development of these IoT scenarios; but also enables the automation of tests derivation (as have been demonstrated by Humayoun et al. in [12, 14]) and in general the ability to better control the evolution of IoT environments.

CONCLUDING REMARKS

In this work, we presented IoTGolog, an extension to TaMoGolog and MobiGolog for general and mobile-based scenarios respectively, for specifying and evaluating IoT scenarios. We explained different predicates for formalizing IoT scenarios and illustrated a use case of car emergency response system to show how IoT scenarios can be modeled using IoTGolog.

In future, we intend to explore in depth using the IoTGolog to generate automatically IoT system models from the formal specification as well as to perform automated evaluation of the implemented IoT systems using task-based approaches. We have already defined one such task-based approach to test system functionality of desktop applications in [12]. We aim to extend such approach towards testing IoT scenarios. Further, we also plan to embed our IoTGolog specification support in some visual IoT wiring tools, e.g., Node-RED[1].

REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey". Computer Networks, 54(15), 2787-2805, 2010.

[2] Buckley, J. (Ed.), The Internet of Things: From RFID to the Next-Generation Pervasive Networked Systems, Auerbach Publications, New York, 2011.

[3] G. de Giacomo, Y. Lespérance, and H. J. Levesque, "ConGolog, A Concurrent Programming Language Based on the Situation Calculus", Artif. Intell. 121, 1-2 August, 2000.

[4] G. de Giacomo, Y. Lespérance, and A. R. Pearce, "Situation Calculus Based Programs for Representing and Reasoning about Game Structures", KR 2010.

[5] C. G. García, J. P. Espada, E. R. Núñez-Valdez, and V. García-Díaz, "Midgar: Domain-Specific Language to Generate Smart Objects for an Internet of Things Platform", IMIS 2014, IEEE, 352-357, 2014.

[6] M. Gerla, E-K. Lee, G. Pau, and U. Lee, "Internet of vehicles: From Intelligent Frid to Autonomous Cars and Vehicular Clouds", Internet of Things (WF-IoT), IEEE, 2014.

[7] A. Gluhak, S. Krco, M. Nati, D. Pfisterer, N. Mitton, and T. Razafindralambo, "A Survey on Facilities for Experimental Internet of Things Research", IEEE Communications Magazine, 49 (11), 58–67, 2011.

[8] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions", Future Generation Computer Systems, 29(7), 1645-1660, 2013.

[9] S. R. Humayoun, T. Catarci, and Y. Dubinsky, "A Dynamic Framework for Multi-View Task Modeling", CHItaly '11, ACM, 185-190, 2011.

[10] S. R. Humayoun, "Incorporating Usability Evaluation in Software Development Environments", PhD Thesis, Sapienza University of Rome, Rome, Italy, 2011.

[11] S. R. Humayoun, A. Poggi, T. Catarci, and A. Dix, "Task-based User-System Interaction", KI, 26 (2), 141-149, 2012.

[12] S. R. Humayoun, Y. Dubinsky, T. Catarci, E. Nazarov, and A. Israel, "A Model-based Approach to Ongoing Product Evaluation", AVI '12, ACM, 596-603, 2012.

[13] S. R. Humayoun, R. AlTarawneh, and Y. Dubinsky, "Formalizing User Interaction Requirements of Mobile Applications", BCS-HCI 2014, 382-383, 2014.

[14] S. R. Humayoun and Y. Dubinsky, "MobiGolog: Formal Task Modeling for Testing User Gestures Interaction in Mobile Applications", MOBILESoft 2014, ACM, New York, NY, USA, 46-49, 2014.

[15] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl, "GOLOG: A Logic Programming Language for Dynamic Domains", Journal of Logic Programming, vol. 33, pp. 59-83, 1997.

[16] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Sensing as a Service Model for Smart Cities Supported by Internet of Things", Transactions on Emerging Telecommunications Technologies, 25(1), 81-93, 2013.

[17] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context Aware Computing for The Internet of Things: A Survey", Communications Surveys & Tutorials, IEEE, 16(1), 414-454, 2014.

[18] F. Xia, L. T. Yang, L. Wang, and A. Vinel, "Internet of Things", International Journal of Communication Systems, 25(9), 1101-1102, 2012.

[1] Node-RED tool: http://nodered.org/