Automata-Based Timed Event Program Comprehension for Real-Time Systems

Aziz Fellah and Ajay Bandi

School of Computer Science and
Information Systems
Northwest Missouri State Univerity
Maryville, Missouri USA
Email:{afellah, ajay}@nwmissouri.edu

***Abstract*—In this paper, we extend the software space of program comprehension to real-time systems and introduce two orthogonal and hybrid paradigms that we refer to as timed event component comprehension and timed event program comprehension. The former, timed event component comprehension, with no role in the coding aspect, is a set of autonomous timed event components that provide a high-level system-specific functionality about the overall real-time system including its structure, components and their synchronized interrelationships at different level of granularity, and static and dynamic behaviors. The later, timed event program comprehension recovers high level of information from timed event component comprehension and then builds an automata-based model about the system. This process occurs before carrying any program comprehension. We show that both paradigms are intrinsically linked and neither of them can be explored in isolation. Importantly, we map the component comprehension paradigm into a distinguished component class that we refer to as timed event components (TeCmp) which, in turn, are formally modeled as timed event automata, a powerful canonical model for modeling and verifying real-time computations. Furthermore, to support this research towards an effective program comprehension geared towards real-time and embedded systems, we investigated and evaluated the effect of our approach through a practical Internet of Things (IoT) case study.**

Keywords–Program comprehension; program understanding; software modeling; real-time systems; embedded systems; IoT; timed event automata.

I. Introduction

Because of its importance in software engineering, program comprehension has emerged as a significant component in software evolution and maintenance. It is a process of understanding an existing software system before it can be properly maintained, enhanced, reused, and extended. For instance, a common situation that software developers may find themselves in is reviewing and extending their own or their teammate's code. This situation is much easier than understanding and maintaining the code of unfamiliar software systems, or reading the code of an Application Programming Interface (API)/utility library. We call these knowledge-intensive activities program comprehension, which is considered as an important aspect of the software development process. In general, new developers spend much of their time analyzing code and searching for information to understand the system under evolution. Other closely related terms are also used to describe activities related to program comprehension, such as code refactoring and reverse engineering. For years, researchers have tried to understand how developers comprehend programs during software maintenance and evolution, and assess the quality of program comprehension. To address these challenges, numerous proposals and approaches have been investigated by Storey [1], Siegmund [2], Yuan *et al.* [3], Fowkes *et al.* [4], and Lucia *et al.* [5], just to name a few that span a spectrum of activities, such as cognitive models and software visualization, empirical evaluation, mental models representation of the program, knowledge-base models, top-down and bottom-up comprehension, code semantics, and data context interaction [1], [6]–[8]. Some of these theoretical models are grounded in experimental studies and validated by experienced programmers.

In this paper and with no comprehensive overview, we attempt to lay a foundation of program comprehension for real-time systems, an area of research that has not received much attention and could be investigated in various directions. In this work, we are not claiming that we developed a general and conclusive program comprehension framework for real-time and embedded systems, but our work will add value to the existing approaches. The paper describes strategies and knowledge needed as well as the rational of this orthogonal paradigm: component and program comprehension. We will shed light on what developers should emphasize when faced with the challenging time-dimension tasks of gaining an understanding of real-time source code. This should be aligned with the original code of the designers.

Importantly, the focus of this contribution is on two orthogonal and hybrid paradigms that we introduce and refer to as *timed event component comprehension* and *timed event program comprehension*. Such a dual comprehension paradigm would help programmers with comprehending systems‘ functionality, understanding code, interweaving abstractions, and building a mental model about a piece of software as well as using effective tools to support program comprehension activities.

Timed event component comprehension provides a high level system-specific functionality about the overall real-time system including its architectural structure, static and dynamic behaviors, and synchronized interrelationships at different levels of granularity. With no role in the coding aspect, timed event component comprehension tasks are grounded on a set of autonomous functional block units that we refer to as *timed*

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-750-4

event components (TeCmp). The abbreviation of TeCmp will be used for both singular and plural terms in the concordant context of the sentence in the rest of this paper.

Timed event program comprehension recovers high-level of information from timed event component comprehension and builds an automata-based model about the system before carrying program (*i.e.,* source code) comprehension. The coordination and interaction between TeCmp is fully delegated to a special class of components that we refer to as *timed component connector* (TeCnn).

A major challenge in the proposed timed event component comprehension development is the coordination of the active components and entities that comprise real-time systems. Thus, there is a need to complement TeCmp with formalisms for coordinating, integrating, and synchronizing components which have well-defined and fixed interfaces. In addition, we collectively refer to the pair, *timed event component* and *timed event connector* models, as (TeC&C) which are formally modeled by timed event finite automata, a powerful canonical model for modeling and verifying real-time computations.

The structure of the paper is as follows. In Section II, we survey the related work and research challenges that appear in software systems related to program comprehension. In Section III, we describe timed event component-based framework which is characterized in terms of two types of components that we refer to as TeCmp and TeCnn. Both of these components are intrinsically linked and neither of them can be explored in isolation. Section IV discusses the challenges of component comprehension in real-time systems. Furthermore, this section states some definitions and concepts that can be used in subsequent sections. Section V focuses on timed event component and connector models (TeC&C) to gain an understanding of the overall system‘s inner workings in terms of the time dimension. Section VI describes time event transitions which are fundamentally important for real-time systems. It also maps the main component, such as TeCmp into timed event automata. The characteristics of the IoT irrigation case study system are presented and summarized in Section VII. We conclude the paper with some potential discussions in Section VIII.

II. Related Work

Over decades program comprehension has been characterized by several classical theories and strategies in conjunction with other complementary techniques such as software inspection, visualization [9], static and dynamic source code analysis. For instance, the knowledge-base model of [10] which is based on the problem domain, developer's experience and background knowledge. A number of mental representations at various levels of abstraction have been investigated in literature [1]–[6] [11]. The top-down model [1] which reflects the developer's mental and conceptual representations integrate domain knowledge as a starting point. On the contrary, with no prior knowledge and little experience with the domain, program comprehension starts at the source code level and builds a higher-level abstraction (bottom-up model) [11]. Knowledge-based, mental and top-down models support the timed event component comprehension paradigm. However, the bottom-up model supports timed event program comprehension paradigm. Based on the nature of events, time-driven and event-driven of [12] and [13], real-time UML (Unified Modeling Languages) has emerged as the choice of the development of real-time and embedded systems. Data context interaction architecture [8] is a software paradigm whose main goal is to bring the end user's mental models and computer program models closer. Data context interaction [8] focuses on objects and their relationships to mental models by which users and programmers add new functionalities and modify the existing ones.

Furthermore, the software system development has shifted its emphasis from traditional building and programming software systems to a component-based approach. Component-Based Development (CBD) [14]–[18] has emerged among the most feasible approaches to overcome and address the software complexity in different domain areas, and advocates the reuse of independently developed software components as a promising technique for the development of complex software systems. Importantly, individual component-based functionalities incorporate potential future reusability, hence served to increase the program comprehension.

Our approach is different from other existing conceptual and theoretical models because we are primarily focusing on the timing characteristics of the application, which is the most predominant factor in real-time and embedded systems. In general, our work partially borrows the concept of time stamps of Leslie Lamport [19], but in particular it is grounded on the foundation of timed automata of Alur [12].

III. Timed Event Component-Based Development

The component-based model [14]–[16] is used to develop software at higher abstraction levels and promotes the reuse and evolution of existing artifacts and entities developing new software systems. It is composed of a collection of functional building blocks or services that have become a system blueprint in modern software engineering development life cycle. In timed event component-based development (TeCBD), we refer to the smallest functional block unit as TeCmp. It is defined in much the same way as a standard component in CBD.

This work is based on component-based software development. In this research, (TeCBD) an emerging software development approach is based on building new software systems from the existing and reusable components. TeCBD involves three stakeholders, TeCmp, TeCnn, and interfaces, which in turn provide, get, or synchronize services. Testing these TeCBD is done first at the component level and then at the assembled unit level.

In this paper, we only focus on the key characteristics of such TeCmp. Individual TeCmp are designed and developed from a hybrid of custom and off-the-shelf (potentially reusable) components. They can be used independently or composed with other TeCmp. In real-time and embedded systems, TeCmp often perform dedicated functionalities under computing and timing constraints as they become more complex and distributed in various environments. Each TeCmp hides its implementation and complexity behind an interface and provides only its functionality to the outside environment, but their interaction and coordination are realized throughout TeCnn.

TeCmp are developed for real-time systems where the logical correctness depends on both the functionality and temporal correctness in a specific environment where the portability should be held to a minimum. Overall, TeCmp describe a syntactically constructive representation where all

tasks are grounded in a set of autonomous functional block units, capturing a common understanding of the application domain at a higher level and according to its semantics.

TeCnn, defined by the protocols, describe the interconnection between TeCmp. That is, they represent a path of interactions between TeCmp and allow transferring data from one TComp′s interface to another without compromising the integrity of the data. TeCmp and TeCnn together depict the functionality of the system at runtime.

The overall behavior of TeCnn is to control in a timely fashion the way TeCmp communicate with each other and provide detailed control over the data- and control-flow. Refer to the example of the IoT irrigation application where the system is composed of several TeCmp and TeCnn in Section VII.

IV. Component Comprehension's Challenges in Real-time Systems

We focus our attention on the role of time and modeling which are the most predominant factors when comprehending a real-time system through its source code. In general, real-time systems may involve different disciplines (*i.e.,* IoT, robotic automation), function typically under different real-time computing constraints, and are distributed in various environments. These underlying constraints include, but not limited to timing, liveness, safety, dependability requirements, and evolution of each discipline. Real-time systems are also composed of components that communicate with each other, and each component performs a set of dedicated functions under real-time computing constraints. In a component-based system, components interact with each other in their environment through well-defined interfaces and coordinate protocols by combining each individual component's functionality. Thus, the component-based paradigm entangles both components' computations and services with components' coordination, which turns collectively these autonomous components into a coherent software working application.

First, we focus our attention on the interaction that describes how TeCmp interact rather than focusing on the individual functionalities and services. Furthermore, in this work and in essence of implementing and automating our results, we are aiming at mapping the theory and properties of timed event transitions systems. In particular, timed event automata to TeCmp, an insight in supporting program comprehension for real-time systems. In addition, abstraction, modularity, and modeling are key factors that enable the development of reusable software. We propose a multitude number of layered abstraction views and models which mimic not only common modeling architectural designs, but also improving maintainability and promoting reusability. In the context of this paper, this high layer of abstraction consists of several constructs such as timed event components, ports, timed event connectors, configurations, and interfaces. Importantly, our focus is still on TeCmp and TeCnn. That is, we explicitly express TeCmp and TeCnn, two distinguished component classes, at the implementation level by formally modeling the functionality of TeCmp units and the interaction protocols of TeCnn as timed event finite automata.

The correctness of real-time and embedded systems depends not only on the logical correctness of the computation, but also on the time at which these computations occurred. Furthermore, the structural decomposition of such systems is embodied in their various components and relationship to each other. Thus, there is a need to promote a software space of design alternatives by putting these pieces together, namely a collection of application-specific interfaces, ports, timed event components, port-connectors, and a set of defined real-time constraints. More explicitly, interfaces describe services that TeCmp provide and services they require from other TeCmp, including their compliance with executions. Ports are the access points in TeCmp through interfaces and services. TeCmp can be atomic or composed of layered interactions between a collection of TeCmp that interact with each other providing new functionalities. TeCnn play a primary role in mediating interactions among TeCmp by providing architectural interaction using different techniques such as queries. Furthermore, they provide different type of services such as data transfer, communication protocols, and control transfer. Configurations are a set of associations between TeCmp and TeCnn.

We assume TeCnn can have at least one TeCmp coupled at each of its ports performing operation requests (*i.e.,* data and control). We define three types of interaction interfaces, *get-interface*, *put-interface*, and *syn-interface* where get-interfaces are required and put-interfaces are provided interfaces by TeCmp. However, there may be complicated synchronization constraints between two or more interfaces of a single TeCmp, then we complement TeCmp with a third type of interface that we refer to as *syn-interface*. Two TeCmp, $\mathcal{C}_1$ and $\mathcal{C}_2$, may interact synchronously through syn-interface. Figure 1 shows a timed component-based system with three timed event components, C_1, C_2, and C_3 which communicate through their respective ports, interfaces, and TConn.

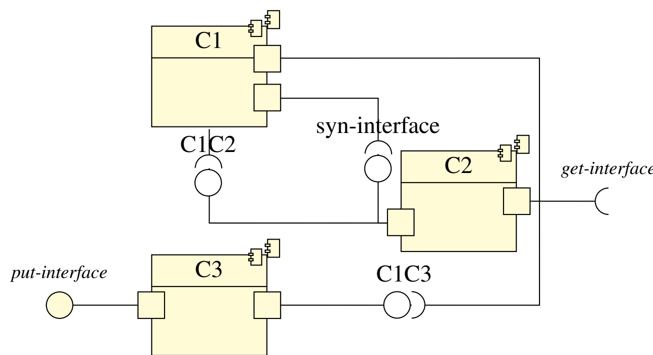

Figure 1. A component-based system with three composed TeCmp, $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ communicating via encapsulated ports/TeCnn, and interfaces.

Now, we can define the following relations between TeCmp. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two TeCmp, we define the following TeCmp relationships:

1) TeCmp *Inheritance*

We say two TeCmp, $\mathcal{C}_2$ and $\mathcal{C}_1$, have an inheritance relation, in terms of object-oriented classes, if $\mathcal{C}_2$ inherits all the properties of $\mathcal{C}_1$. In addition, $\mathcal{C}_1$ may have more interaction interfaces and all the inherited interaction interfaces of $\mathcal{C}_2$ work exactly the same way as those of $\mathcal{C}_1$.

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-750-4

2) TeCmp *Association*

We say two TeCmp, $\mathcal{C}_1$ and $\mathcal{C}_2$, have an association relation if they have at least one interaction interface.

3) TeCmp *Aggregation*

We say two TeCmp, $\mathcal{C}_1$ and $\mathcal{C}_2$, have an aggregation relation if $\mathcal{C}_1$ is a subset of $\mathcal{C}_2$. In addition, a single TeCmp can be aggregated by several TeCmp. The aggregated TeCmp has all the interaction interfaces of its TeCmp.

4) TeCmp *Composition*

A composition is the combination of two or more TeCmp at different levels of abstraction to achieve modularity and decomposition of TeCmp using various programming languages or composition tools as defined by the TeCmp infrastructure. Let $\mathcal{C}_1$, $\mathcal{C}_1'$, $\mathcal{C}_2$, $\mathcal{C}_2'$ be four TeCmp. Let the operators $\equiv$ and $\times$ be the equivalence and composition operators in the semantic context, respectively. Then, if $\mathcal{C}_1 \equiv \mathcal{C}_1'$ and $\mathcal{C}_2 \equiv \mathcal{C}_2'$ implies $\mathcal{C}_1 \times \mathcal{C}_2 \equiv \mathcal{C}_1' \times \mathcal{C}_2'$.

5) TeCmp *Encapsulation*

We say that TeCmp $\mathcal{C}_1$ exhibits functional encapsulation if $\mathcal{C}_1$ hides its details while exposing a well-defined interface through its ports. Furthermore, embedded TeCmp may occur at different levels of abstraction and could potentially foresee what we call recursive encapsulation, a fundamental scheme in comprehending programs. We say two TeCmp, $\mathcal{C}_1$ and $\mathcal{C}_2$, have an association relation if they have at least on interaction interface.

The terms association, aggregation, and composition are extended versions of the common terms used in conceptual modeling. In general, the definition of inheritance in this paper is defined in the context of object-oriented programming languages (including C#, C++ and Java). For instance, the inheritance could be considered as covariant, invariant and contravariant in C#.

Abstraction and modularity are key factors in timed component-based framework that enable the development of re-usable software. We start with various and rigorous levels of abstractions and structures that are refined at each stage of the development before mapping them to programming. For instance in timed event components and connectors (TeC&C) model, TeCmp architectural abstractions expose a high-level of the structure of the system, including TeCmp's logical abstractions. On the other hand, TeCnn data- and control-flow abstractions propose categorization spaces of data types and control the flow of imposed conditions. TeCnn communication and synchronization abstraction styles support protocols, and enforce synchronous and asynchronous requests. TeCmp and TeCnn timing abstractions and properties address several issues of real-time systems throughout modeling formalisms.

Component Comprehension's Abstraction

Abstraction can take many forms and dimensions to serve various purposes in software development. In the context of this work, we propose two different levels of abstraction. A horizontal abstraction that studies component comprehension at a very high level of abstraction, such as TeCmp's functionality, ports, interfaces, and TeCnn. However, details and refinements regarding low-level abstractions such as time structures, timed automata, data types, configuration protocols, data structures, and algorithms are performed on a vertical level. In fact, the integration of both horizontal and vertical abstractions reflect an orthogonality at the system and process models, respectively. Partitioning for the purpose of comprehension through various dimensions and abstractions can be found in literature [20] and [21]. Figure 2 views a prism rectangle box with special components, TeCmp, ports, TeCnn, interfaces and a "time event clock".

A prism rectangle box as shown in Figure 2 views special components, TeCmp, ports, TeCnn, interfaces and a "time event clock". Similarly, Figure 3 shows explicitly a series of comprehension views through a high-level horizontal and low-level vertical layers of abstraction. The former layer is composed of constructs such as TeCmp, interfaces, and ports. The latter is composed of constructs such as timed event automata, timed event signature, and source code.

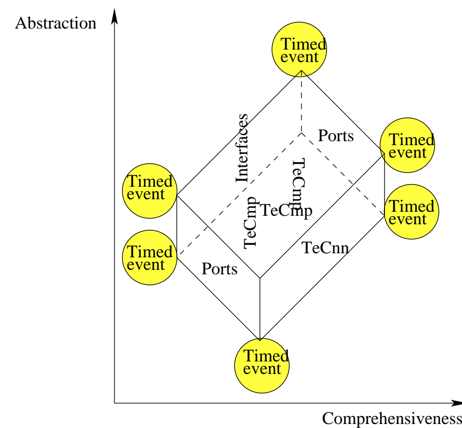

Figure 2. Comprehension dimensions through TeCmp, TeCnn, ports and interfaces.

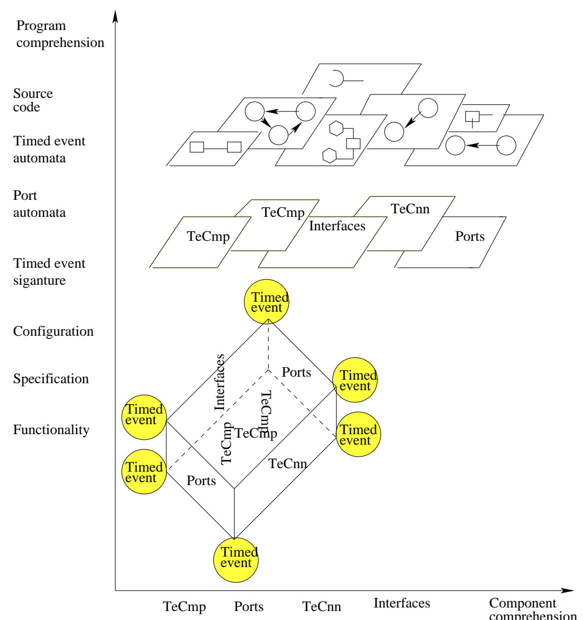

Figure 3. Comprehension dimensions through timed event program.

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-750-4

V. Timed Event Component and Connector Models

The coordination and interaction between TeCmp is fully delegated to a special class of component connector that we refer to as TeCnn. An expressive and intuitive way of visualizing TeCnn is to view such a special type of component as a "black box" with some "special code" and a "clock" that allows real-time coordination between the active software timed component entities. Clocks are used to justify timed transitions and sequences of events in such a model. Connectors are modeled as a relation between timed event component streams.

We leverage the time logic and dimension structures of [6] and [22] to describe real-time interactive or concurrent systems in this work. Importantly, we consider the time-dependent behavior of any TeCmp is an important aspect of the system's requirements, enforced by the component itself and coordinated by TeCnn. To develop a uniform timing framework, we consider the absolute time which could be modeled using a global clock.

Let $\mathcal{C}= \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\} \subseteq$ TeCmp be a finite set of timed event component instances where $|\mathcal{C}| = n$. Let $\Omega = (\mathcal{T}, \mathcal{E})$, where $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$ is a set of points in the time domain and $\mathcal{E} = \{e_1, e_2, \ldots, e_k\}$ is a set of events in the event domain. For convenient, we assume $|\mathcal{T}| = |\mathcal{E}| = k$. Let $\prec$ be a strict partial order precedence relation over $\mathcal{T}$. Let $\mathcal{C}_1(e_1, t_1)$, $\mathcal{C}_2(e_2, t_2)$, and $\mathcal{C}_3(e_3, t_3)$ indicate that $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ are being active on the occurrence of the event e_i at time t_i, respectively where $i = 1 \ldots, n$. We define the timed event dimension structure over TeCmp as a tuple in the form $\mathcal{C}(\mathcal{E}, \mathcal{T})$ that satisfies the following properties:

(*i*) For all $e \in \mathcal{E}$, if $\mathcal{C}_1(e, t_1) \prec \mathcal{C}_2(e, t_2)$ and $\mathcal{C}_2(e, t_2) \prec \mathcal{C}_3(e, t_3)$ then $\mathcal{C}_1(e, t_1) \prec \mathcal{C}_3(e, t_3)$.

(*ii*) For all $e \in \mathcal{E}$ and $t \in \mathcal{T}$, $\mathcal{C}_i(e_i, t_i) \not\prec \mathcal{C}_i(e_1, t_i)$, $i = 1, \ldots, n$.

(*iii*) For all $e \in \mathcal{E}$ and $t \in \mathcal{T}$, if $\mathcal{C}_1(e_1, t_1) \prec \mathcal{C}_2(e_2, t_2)$ then $\mathcal{C}_2(e_2, t_2) \not\prec \mathcal{C}_1(e_1, t_1)$.

(*vi*) For all $\mathcal{C}_i(e, t)$ and $\mathcal{C}_j(e, t)$, if $\mathcal{C}_i(e, t) \not\prec \mathcal{C}_j(e, t)$ then $\mathcal{C}_i$ and $\mathcal{C}_j$ are interpreted as being concurrent, for all $e \in \mathcal{E}$ and $t \in$, and where $i, j = 1, \ldots, n$.

The external view of the port model is based on the pipe-and-filter architectural style with consists of a set of *data* and *control port* groups. In addition and for various purpose, we assume there is one extra internal group ports that we refer to as *special ports*. The data port group is explicitly divided into input and output data ports. Similarly, the control port group is explicitly divided into input and output control ports. Both the data and control ports are provided by default for each port. However, other types of variables such as monitoring and controlling ports can be an intrinsic part of the internal port depending on the application domain. In the context of this paper, we define a port signature $\mathcal{S}$ as follows:

Definition 5.1: A timed event port signature is a quintuple $\mathcal{S}$ = ($Event$, $Type$, $Data$, $Control$, $Time$), where $Event = \{In, Out, Spec\}$ and In, Out, $Spec$ are the set of input, output, and special ports respectively; $Type$ is a finite set of type names, $Data$ and $Control$ are sets of data and control values, respectively. $Time$ is a set of point structure of time, modeled by a global clock. Moreover, $(In \cap Out \cap Spec) = \emptyset$, and the set of data and control values is disjoint.

In the rest of the paper and for clarity, the terms timed event port signature and port signature are interchangeable. Now, borrowing from the syntax and semantics of components and connectors views, [23] and [24], we formalize the structure of the timed event component and connector model (TeC&C) model by not focusing on the interfaces defined for the ports, but rather on the relation between the different pieces of the TeC&C model.

Definition 5.2: A timed event component and connector TeC&C model is a sextuple structure $\mathbb{CC}$ = ($\mathcal{C}$, $\widehat{\mathcal{C}}$, $\mathcal{P}$, $\mathcal{S}$, δ_p, δ_t) where

(*i*) $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\} \subseteq$ TeCmp is a finite set of timed event component instances where $|\mathcal{C}| = n$.

(*ii*) $\mathcal{P} = \{p_1, p_2, \ldots, p_m\}$ is a finite set port instances where $|\mathcal{P}| = m$

(*iii*) $\mathcal{S} = \{s_1, s_2, \ldots, s_m\} \subseteq$ port signatures is a finite set of port signature instances where $|\mathcal{S}|$ = $|\mathcal{P}| = m$.

(*iv*) $\widehat{\mathcal{C}} = \{\widehat{\mathcal{C}}_1, \widehat{\mathcal{C}}_2, \ldots, \widehat{\mathcal{C}}_q\} \subseteq$ TeCnn is a finite set of connector instances which are used to capture pathways of events (data transfer flow and control flow) between $\mathcal{C}_i$, $i = 1 \ldots n$. ($|\widehat{\mathcal{C}}| << |\mathcal{C}|$).

(*v*) δ_p: $\mathcal{C} \times \mathcal{P} \rightarrow \mathcal{C} \times \mathcal{P}$. That is, $\delta_p(\mathcal{C}_i,\ p_j) \subseteq \mathcal{P}$, for all $i = 1 \ldots n$ and $j = 1 \ldots m$.

(*vi*) δ_t: $\mathcal{P} \times \mathcal{S} \rightarrow \mathcal{P} \times \mathcal{S}$. That is, $\delta_t(p_j,\ s_j) \in$ ($\mathcal{P}$ $\times$ $\mathcal{S}$), for all $j = 1 \ldots m$.

The requirements of real-time systems must be able to accommodate real-time timing constraints and discrete/continuous behaviors, such as safety, resources limitations, predictability, and reliability. Thus, the designer must be equipped with modeling formalisms, formal analysis, techniques, and support tools throughout the development process of TeC&C. A well-established modeling formalism to support real-time systems is timed automata [12] that extend finite state automata where transitions are guarded with conditions based on clock variables. Each TeCmp consists of the component requirement specifications, implementations, and interfaces. Consequently, the development of the TeC&C model typically starts with requirements specification which should be written in some formal notations (*i.e.*, formal methods). Thus, it is important to develop a formal description of TeCmp, TeCnn, and TeC&C models for real-time systems. In the following sections, we focus on a series of methodology and automata-based formalisms that capture this collection of timed event interconnected components and connectors. Refer to the example of the IoT irrigation application where the system is composed of several TeCmp and TeCnn in Section VII.

VI. Automata-Based Component and Program Comprehension

In the program source comprehension, developers pursue their familiarization effort with TeCmp at various level of granularity, and try to gain an understanding of the program through preliminary evaluations, its structure, and through static and dynamic analysis (or by a combination of both). A drawback is that dynamic analysis can only provide a partial picture of the system based on the developers explorations of the program's behavior through the execution of the system. Static analysis focuses on the source code and extract important information from the program source. The

correctness of a real-time system depends not only on the correctness of the sequence of the events, but also on their time of occurrence. In the following section, we establish approaches and models to understand visualize, and navigate through the source code. First, developers focus on understanding the software as a whole (*i.e.*, component comprehension) avoiding program comprehension whenever possible. Second, developers focus on mental models and visualization during program comprehension inspection activities using timed event automata formalism for comprehending real-time source code and acquire run-time information.

In general, timed transition systems and in particular event transition systems have been extensively studied in the literature [13] [25] [26]. Moreover, both systems have been combined and used practically in the verification, testing, and development of real-time platforms where reliability, safety, and correctness depend to a large extent on the time features. Both time-driven and event-driven computing models are fundamentally important for real-time and embedded systems, given that such systems are reactive by nature. In a time-driven model, computations and actions are triggered by time, either periodically or in terms of deadlines by which computational activities must be completed. In a time-driven model, the state continuously keeps changing as time changes. Thus, the synchronous nature of time guarantees the deterministic behavior of the model. In contrast, in an event-driven model, computational activities or actions are triggered upon occurrences of asynchronous events. We combine time-driven and event-driven models into one unified hybrid system architecture, and propose a real-time model that we refer to as a *Timed Event Automaton* (TeAut). The state of the TeAut continuously changes as time changes and the occurrences of asynchronously generated events forces instantiated state transitions. In consequence, the correctness of real-time system's TeCmp depends not only on the correctness of the computational tasks in the system, but also on the time at which these computations are performed. Let $\mathcal{E}$ and $\mathcal{T}$ denote the event set and time base, respectively, The time domain $\mathcal{T}$ can be modeled as discrete, continuous, or over an interval $[t_l, t_u] \subseteq \mathcal{T}$, $t_l \leq t_u$. In this work, we consider continuous time systems, that is all the variables (*i.e.*, input, output, states) are defined over all possible values of time. In particular, we consider the time domain, $\mathcal{T}$, as the non-negative reals $\mathbb{R}_{\geq 0}$. A timed event $\omega = (e, t)$ over a finite set of events $\mathcal{E}$ and the time $t \in [0, \infty)$ denotes an event $e \in \mathcal{E}$ occurs at time t.

Let $\mathcal{X}$ be a set of finite clock variables (or clocks for short), the set $\Phi(\mathcal{X})$ of clock constraints ϕ over $\mathcal{X}$ is defined by the following grammar:

$$\phi := x \bowtie c \mid \phi_1 \wedge \phi_2 \mid \mathit{true} \mid \mathit{false}$$

where $c \in \mathcal{X}$, $c \in \mathbb{N}$ such that $c \geq 0$, $\bowtie \; \in \{<, \leq, =, >, \geq\}$, and $\wedge$ stands for the *and* logical operator. The precondition clock constraint $\phi \in \Phi$ specifies when the transition is enabled, and the postcondition set $\mathcal{X}_0 \in 2^{\mathcal{X}}$ gives the set of clocks to be reset to zero while all other clocks remain unchanged. A clock valuation represents the values of all clocks in $\mathcal{X}$ at a given snapshot in time.

Definition 6.1: Let $\mathcal{X}$ be the set of clock variables. A *clock valuation* over $\mathcal{X}$ is a function ν from $\mathcal{X}$ to $\mathbb{R}_{\geq 0}$ that maps every clock $x \in \mathcal{X}$ to a non-negative real number.

For $t \in \mathbb{R}_{\geq 0}$, the valuation $\nu + t$ is defined as $(\nu + t)(x) = \nu(x) + t$. For $\mathcal{X}' \subseteq \mathcal{X}$ the valuation is defined as $(\nu[\mathcal{X}' := 0])(x) = 0$ if $x \in \mathcal{X}'$ and $(\nu[\mathcal{X}' := 0])(x) = \nu(x)$ otherwise. We denote by $\mathbb{v} = (\nu_1, \ldots, \nu_n)$ a characteristic vector of clock valuations of the timed automata $\mathcal{A}$. In general, our work partially borrows the concept of time stamps of Leslie Lamport [19], but in particular it is grounded on the foundation of timed automata of Alur [12]. Without loss of generality, a state is defined as a pair $(q, \mathbb{v})$, where $q \in Q$ and $\mathbb{v}$ is a clock valuation at state q.
A time sequence t is a non-empty finite (or infinite) sequence of time values denoted by $t = t_1 t_2 \ldots t_n$ such that $t_i \in \mathbb{R}_{\geq 0}$ and all $t_i's$ satisfies the monotonicity and progressiveness conditions. That is, for all $1 \leq i \leq |t|$, $t_i \leq t_{i+1}$, and for each $t \in \mathbb{R}_{\geq 0}$ there exists t_i, $i \in \mathbb{N}$ such that $t_i < t_{i+1}$. If t is infinite then t_i is not bounded for all $i \geq 1$.

We define a finite set of timed events $\Omega = (e_1, t_1)(e_2, t_2) \ldots (e_n, t_n)$ over $\mathcal{E}$ and $\mathcal{T}$, formally denoted as

$$\Omega = \{(e, t)^* \mid e \in \mathcal{E} \cup \{\omega_\lambda\}, t \in \mathcal{T}\}$$

Define $\omega_\lambda = (\lambda, 0)$, where $\lambda \notin \mathcal{E}$ is the null time event to indicate no event has occurred.

Now, we abstract and simulate TeCmp and TeCnn in terms of timed event automata and timed event port-automata, respectively. A timed event automaton induces a timed event transition system. A timed event automaton (TeAut) is a structure defined as follows:

Definition 6.2: A timed event automaton (TeAut) is a sextuple $A = (\Omega, \mathcal{X}, Q, q_0, \Gamma, F)$, where (i) Ω is the finite set of timed events over $\mathcal{E} \times \mathcal{T}$, (ii) $\mathcal{X}$ is the set of clock variables; (iii) Q is the set of states; (iv) $q_0 \in Q$ is the initial state; (v) $\Gamma \subseteq \Omega \times Q \times \Phi(\mathcal{X}) \times 2^{\mathcal{X}} \times Q$ is a finite set of transitions; (vi) $F \subseteq Q$ is the set of accepting states.

The values of the clock variables increase monotonically with the passage of time. The next state of a timed event automaton depends on both the event symbol and the values of the clock constraints. In addition, each transition may reset some of the clocks. A transition can only be taken if the current clock values satisfy the time constraints and the event symbol.

Let A be a TeAut and $t \in \mathbb{R}_{\geq 0}$. Define a *timed event requirement specification* A as

$$\mathcal{R}(A) = \{\omega \in \Omega \;:\; \Gamma(q_0, \omega) \in F\}$$

Now, we define timed event port-automata over a single and global clock.

Definition 6.3: A timed event port-automaton (TePA) is a sextuple $\mathcal{A} = (\Omega, \mathcal{S}, Q, \mathcal{P}, \mathcal{X}, \delta)$ where (i) Ω is the set of timed events set; (ii) $\mathcal{S}$ is a port signature; (iii) Q is the set states; (iv) $Q_0 \subseteq Q$ is the set of starting states; (v) $\mathcal{P}$ is the set of all ports; the transition function (vi) $\delta \subseteq \Omega \times Q \times \mathcal{S} \times 2^{\mathcal{P}} \times \Phi(\mathcal{X}) \times Q$.

We extend each timed event port-automaton $\mathcal{A}$ with the powerful primitives of Reo [27] and [28] connectors, a paradigm for communication protocols and composition of software components. Each timed event connector TeCnn via its ports imposes a specific coordination on the active TeCmp, which in turn offer a set of services. The $sync(a, b, e, t)$ time event port-automaton models the Reo primitive that allows

synchronous activities on two ports a and b. Moreover, the synchronous nature of time guarantees the deterministic behavior of the port-automaton. In contrast, computational activities or actions are triggered upon occurrences of asynchronous events. The $lossy(a, b, e, t)$ is similar to the $sync$ primitive, in addition it can have activities through its end, a. The *xor* (a, b, c, e, t) primitive synchronizes a with either b or c. The *fifo* (a, b, e, t) models a buffer with a source port-automaton a and a sink port-automaton b, which are synchronously timed coordinated and asynchronously event triggered. Other operations can be performed on timed event port-automata, such as the product and composition. The desired coordination between two exclusive ports is given as timed event port-automaton. However, the coordination among all $\mathrm{T_{eCmp}}$, as shown in Figure 4, is modeled through individual ports.

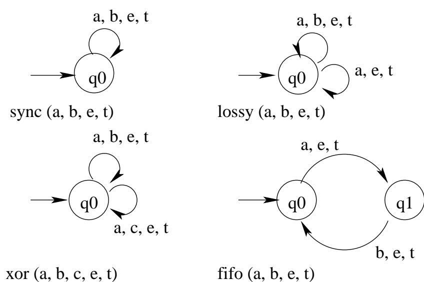

Figure 4. Examples of timed event port automata using Reo's primitives.

VII. IoT Case Study

We describe a case study that has been conducted and implemented on an IoT irrigation embedded system. Overall, the system regulates a water solenoid valve for controlling a drip irrigation system using Arduino and Raspberry Pi infrastructure. For the experiment, we selected and expand the recent IoT project of three graduate students as the basis of our case study by running a variety of experiments to test the proposed theoretical work. The experiment was tested on several events, such as moisture, temperature, and humidity. The system is able to deliver water to the plants based on the moisture of the soil, temperature, and humidity of the day which are obtained through DHT sensors. Importantly, we use a real-time clock that allows the system to set the start of the irrigation system based on the moisture and temperature levels. Furthermore, the system can also start and stop at the specified time intervals to control the water management. In the experiment, the IoT system is controlled by the real-time status of the soil moisture, atmospheric conditions, and on the real time clock to adjust the irrigation scheduling through time intervals. In this IoT-based system, a strong emphasis is put on timed event components of the system and empirical evaluations. The comprehensiveness at both component and program must be sufficiently understood by its developers on performing a broad spectrum of maintenance tasks.

In analogical mapping, our abstract model of study, timed event automata-based components, chas been mapped into the real-time target irrigation domain. That is, soil moisture, temperature, and humidity sensors send real data to the microcontroller, which is considered as the central $\mathrm{T_{eC\&C}}$ comprehension information gateway. The microcontroller can be monitored and operated via WiFi using a Web browser, or managed by the user through a mobile application. The $\mathrm{T_{eCmp}}$ sprinkler controller ensures uniform distribution of water to all part of the plant and it is monitored by the microcontroller. In addition, the $\mathrm{T_{eCmp}}$ sprinkler may be switched off and on once the soil moisture sensor has reached the appropriate threshold value. We may consider that DHT moisture, temperature, and humidity sensors are equipped with some ports communicating with various $\mathrm{T_{eCmp}}$. The coordination and interaction between various $\mathrm{T_{eCmp}}$ is fully delegated to a special class of component that we referred to as $\mathrm{T_{eCnn}}$. These connectors have no relevant role in the irrigation aspect, but mediate, coordinate, and control interactions among various $\mathrm{T_{eCmp}}$ photons. In addition, the data of sensors is displayed in a graphical format, analyzed and visualized by the end-user. This is considered as part of the multi-view learning approaches to perform program comprehension activities. The event domain $\mathcal{E}$ is a set of events in the irrigation domain. That is, $\mathcal{E}$ could be $\{moisture, temperature, wind, humidity, precipitation\}$. When the sensors report that the moisture, temperature, or humidity levels have fallen below the threshold level, the LED light glows, indicating that a timed irrigation event has to be initiate. In addition, the LED lights are also used for other purposes in the context of this irrigation project as summarized below in Figure 5.

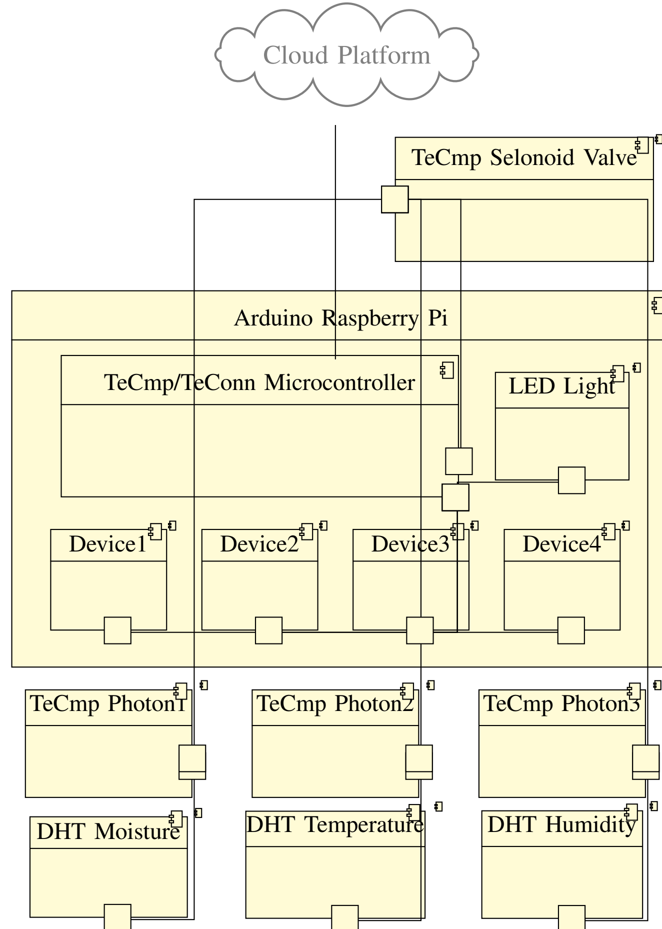

Figure 5. Cloud Control and flow of information in the IoT irrigation System: Soil moisture, temperature, humidity sensors send real-time data to the timed event microcontroller.

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-750-4

VIII. Conclusion

Our work revealed an apparent lack of foundations in the literature that relates to program comprehension for real-time and embedded systems. This is an area of research that has not received much attention and could be investigated in various directions. We investigated two timed orthogonal program comprehension paradigms, timed event component and program comprehension, which led to comprehending programs with a greater degree of structure, abstraction techniques, and architecture reconstruction, hence offered a series of potential effectiveness and enhancement in gaining a deeper understanding of program comprehension in real-time systems. First, we mainly rely on architectural levels and time dimensions which have been explicitly targeted in our work and how they are clearly manifested in a real-time systems's implementation. Second, we have examined the relationships between timed event component comprehension and timed event automata-based program comprehension. Such refinement and analysis from component levels to program levels comprehension is significantly represented in the source code. Furthermore, we have performed an empirical IoT irrigation case study in order to complement and provide a qualitative base and characterization of our approach to program comprehension and software evolution. In this work, we validated our theoretical framework on the IoT irrigation application. As a future work, we will investigate this program comprehension paradigm by applying it in various real-time and embedded systems of different domains.

References

[1] M. A. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present and Future," in Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05). Washington DC, DC, USA: IEEE Computer Society, 2005, pp. 181–191.

[2] J. Siegmund, "Program Comprehension: Past, Present, and future," in Proceedings of the 23th Internationa Conference on Software Analysis, Evolution, and Engineering (SANER '16). IEEE SANER, 2016, pp. 13–20.

[3] B. Yuan, V. Murali, and C. Jermaine, "Abridging source code," in Proceedings of the ACM on Programning Languages (OOPSLA 58:1). ACM New York, NY, USA: ACM, 2017, pp. 13–20.

[4] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton, "Tassal: Autofolding for Source Code Summarization," in Proceedings of the 38th International Conference on Software Engineering Companion, (ICSE 16). ACM New York, NY, USA: ACM, 2016, pp. 649–652.

[5] D. e. a. Lucia, "Labeling Source Code with Information Retrieval Methods: An Empirical Study," Empirical Software Engineering, vol. 19, No. 5, 2004, pp. 1383–1420.

[6] T. Ben-Nun, A. S. Jakobovit, and T. Hoefler, "Neural Code Comprehension: A Learnable Representation of Code Semantics," in Proceddings of the 32nd Conference on Neural Information Processing Systems (NeurIPS '18, Montreal, Canada, 2018, pp. 3589–3601.

[7] S. Xu, "A Cognitive Model for Program Comprehension," in Proceedings of the 3rd International Conference on Software Engineering Research, Management and Applications (ACIS '05), Montréal, Canada, 2005, pp. 392–398.

[8] T. Reenskaug and J. O. Coplien, "DCI as a new Foundation for Computer Programming," Software Engineering in Intelligent Systems, Springer, vol. 3, no. 5, 2009, pp. 1383–1420.

[9] J. Riling and S. P. S.P. Mudur, "3D Visualization Techniques to Support Slicing-based Program Comprehension," Computer & Graphics, vol. 29, No. 3, 2005, pp. 311–329.

[10] S. Letovsky, "Cognitive Process in Program Comprehension," Journal of Systems and Software, Elsevier, vol. 7, no. 4, 1998, pp. 325–339.

[11] T. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: a Study of Developer Work Habits," in Proceedings of the 28th international conference on Software engineering. ACM, 2006, pp. 492–501.

[12] R. Alur and A. A. Dill, "Theory of Timed Automata," Theoretical Computer Science, vol. 126, no. 2, 1994, pp. 183–235.

[13] A. Fellah, "Timed Event Systems and Automata," in Proceedings of the 13th IASTED International Conference on Control and Applications. Acta Press, 2011, pp. 730–739.

[14] A. Ahmad, P. Jamshidi, and K. Fawad Pahl Claus, "A pattern Language for the Evolution of Component-based Software Architectures," Electronic Communications of the EASST, vol. 59, 2014, pp. 1–31.

[15] O. Le Goaer, D. Tamzalit, M. Oussalah, and A.-D. Seriai, "Evolution Shelf: Reusing Evolution Expertise within Component-based Software Architectures," in Proceedings of the 32nd Annual IEEE International Computer Software and Applications. IEEE, 2008, pp. 311–318.

[16] H. Yin and H. Hansson, "Fighting CPS Complexity by Component-based Software Development of Multi-mode Systems," in Proceedings of the 32nd Annual IEEE International Computer Software and Applications, vol. 2, no. 4. Designs, 2018, pp. 1677–1718.

[17] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron, "A classification Framework for Software Component Models," IEEE Transactions on Software Engienering, vol. 37, 2011, pp. 593–615.

[18] J. Criado, D. Rodriguez-Gracia, L. Iribarne, and N. Padilla, "Toward the Adaptation of Component-based Architectures by Model Transformation: Behind Smart User Interfaces," Software Practice Exp., vol. 45, 2015, pp. 1677–1718.

[19] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill, "Lamport clocks: Verifying a Directory Cache-coherence Protocol," in Proceedings of the 10th Annual ACM symposium on Parallel Algorithms and Architectures. ACM, 1998, pp. 67–76.

[20] R. T. Mittermeir, A. Bollin, H. Pozewauning, and D. Rauner-Reithmayer, "Fighting CPS Complexity by Component-based Software Development of Multi-mode Systems," ACM SIGSOFT Software Engineering Notes, vol. 26, no. 3, 2001, pp. 95–102.

[21] M. Tomgren, D.-J. Chen, and I. Crnkovic, "Component-based vs Model-based Development: a Comparison in the Context of Vehicular Embedded Systems," in Proceedings of the 31st EUROMICRO Conference on Software Engineering, 2001, pp. 95–102.

[22] S. Yu, "The Time Dimension of Computation Models," Where Mathematics, Computer Science, Linguistics and Biology Meet, Springer, Dordrecht, 2001, pp. 161–172.

[23] S. Maoz, N. Pomerantz, and B. Rumpe, "Synthesis of Component and Connector Models from Crosscutting Views," ACM ESEC/SIGSOFT FSE, 2013, pp. 444–454.

[24] S. Maoz, N. Pomerantz, J. O. Ringert, and R. Shalom, "Why is my Component and Connector Views Specification Unsatisfiable?" in Proceedings ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems. ACM/IEEE, 2017, pp. 134–144.

[25] T. Brengos, "Behvioural Equivalences for Timed Systems," Logical Methods in Computer Science, vol. 15, no, 1, 2019, pp. 17:1–17:41.

[26] T. Henzinger, Z. Manna, and A. Pnueli, "Timed Transition Systems," in Proceedings of the Real-Time: Theory in Practice, 1991, pp. 226–251.

[27] A. Arbab Reo, "A Channel-Based Coordination Model for Component Fomposition," Mathematical Structures in Computer Science, vol. 14, 2004, pp. 329–366.

[28] F. Arbab, C. Baier, F. de Boer, and J. Rutte, "Models and temporal logical specifications for timed component connectors," Software and Systems Modeling, vol. 6, no. 3, 2007, pp. 59–82.