IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received June 21, 2016, accepted July 8, 2016, date of publication August 2, 2016, date of current version September 28, 2016.

Digital Object Identifier 10.1109/ACCESS.2016.2592381

MAPE-K Interfaces for Formal Modeling of Real-Time Self-Adaptive Multi-Agent Systems

AWAIS QASIM AND SYED ASAD RAZA KAZMI
Department of Computer Science, Government College University, Lahore 54000, Pakistan
Corresponding author: A. Qasim (awais@gcu.edu.pk)

ABSTRACT Formal modeling of multi-agent systems is an active area of research. The use of precise and unambiguous notation of formal methods is used to accurately describe and reason about the system under consideration at the design time. Multi-agent systems deployed in dynamic and unpredictable environment needs to have the ability of self-adaptation, making them adaptable to the failures. State of the art encourages the use of MAPE-K feedback loop for the provision of self-adaptation in any system. There is a dire need of formal vocabulary that can be used for the conceptual design of any real-time multi-agent system with self-adaptation. In this paper, we have proposed a set of predefined interfaces for the provision of self-adaptation in real-time multi-agent systems. The interfaces are based on monitor, analyze, plan, and execute phases of the MAPE-K feedback loop. We formally specify our interfaces using timed-communicating object-Z language. The complete framework is elaborated using a trivial case-study of conveyor belt system based on a real-time agent architecture.

INDEX TERMS Formal methods, self-adaptation, autonomic computing, multi-agent systems, real-time systems.

I. INTRODUCTION

Formal modelling means using some technique of formal methods to create a mathematical model of a system for the purpose of analyses and enhanced understanding. The use of formal methods provides an opportunity to analyze the system at the design time without worrying about the actual implementation details. In programming the term interface refers to an abstract type that defines behaviors as methods. In contrast to the inheritance, a class can implement as many interfaces as required to provide the desired functionality. Interfaces are especially useful at the design time to make classes inter-operable because rather than specifying a particular class, the type of the object to be exchanged can be defined in terms of its interface. Also in the development process, a mock implementation may be substituted for the testing purpose that is later replaced by the real code. An agent is a software that performs various actions continuously and separately on behalf of an individual or an organization in a specific environment. It is reserved by other processes and agents, but it can also learn from its experience during functioning in an environment over a longer period of time. An agent with restrictive timing constraints is called a Real-Time Agent (RTA). RTAs are expected to complete their task within the specified deadline or else they are of no use. Usually an off-line schedulability analysis is done before these RTAs are deployed to make sure that these agents will meet their deadlines. A multi-agent system with at least one real-time agent is called a Real-Time Multi-Agent system (RTMAS). Multi-agent systems paradigm has emerged as a useful technique to model real-time systems like mobile robots, online auction systems, intrusion detection systems, control processes, etc. Since it is not possible to build RTMASs with all the world knowledge incorporated into them they are expected to face unfamiliar situations. Self-adaptation endows a system with the ability of self-organization according to the changes in the user requirements and execution context. This ultimately leads to better fault tolerance in case of any failure without affecting the system's overall functionality. Ideally to an external observer/user of the system, the restructuring of the system as part of its self-adaptation should be as abstract as possible. We believe that the formal modelling of these systems with temporal deadlines and adaptation capability will help to better analyze and understand them. This work has focused on the formal modelling of real-time self-adaptive multi-agent systems, where the system's domain functionality is provided by the autonomous working agents.

Much work has been done in the past on the formal specification and verification of multi-agent systems and self-adaptive multi-agent systems. However according to our knowledge no work has been done for the formal modelling of self-adaptive real-time multi-agent systems. Time-extended models like timed automata have been proposed for the formal verification of real-time systems in [1] and [2], but not for multi-agent systems. In [3] a technique for controlling decentralized multi-agent systems using LTL specifications have been proposed. An Eclipse based CASE tool for the development and verification of multi-agent systems comprising heterogeneous agents has been discussed in [4]. They have used model checking approach for the verification of such systems. Reference [5] presented a prototyping approach for the formal modelling of multi-agent systems. Their framework described the key entities like organization, role and interaction using a language called OZS, which is a formal language combining Object-Z and state charts. Reference [6] has specified distributed multi-agent systems using Timed Communicating Object-Z (TCOZ) language, which is a combination of Object-Z and Timed CSP. The work of [7] has focused on handling fault-tolerance in multi-agent system's interaction using Event-B Patterns. Event-B is a formal technique used for the modelling of distributed systems for their analysis. In [8] Object-Z has been used to formally model the incremental development of multi-agent systems. Basically they introduced a notion of action refinement for agent's decision-making and inter-agent negotiations in Object-Z. In [9] hierarchical real-time systems have been formally modelled and verified using an extension of Timed CSP called Stateful Timed CSP. Mainly they solved the problem of verification with non-Zeroness assumption. Additionally many researchers have proposed several frameworks for constructing systems with self-adaptation ability. In [10] different stages of self-adaptation has been identified as MAPE-K conceptual framework. According to [11], feedback-control loop is an essential activity in any self-adaptive system. In [12] an architectural model has been presented for adaptivity, which divides change management, component control and goal management into three different levels. A software architecture-based approach has been recommended in [13]–[15] for change management and assessing the adaptation decisions. A Timed CSP based approach for the specification and verification of embedded distributed real-time systems has been proposed in [16], but not for RTMAS. Multi-agent systems have been formally specified and verified using modal *mu*-calculus and Timed-Arc Petri-nets in [17] and [18]. In [19] a framework for formal modelling of distributed self-adaptive systems has been proposed called FORMS, which provides different modelling elements and a set of relationships guiding the design of self-adaptive software systems. Reference [20] has formally modelled real-time systems using an extension of the Rebeca language. They used Structural Operational Semantics for modeling distributed systems with temporal constraints. In [21] a new language named STeC (an extension of process algebra) has been proposed for the formal specification of location-trigger real-time systems. In [22] a multi-agent system approach has been presented for the real-time operation of scheduling and demand management in microgrids. Reference [23] has presented a new model-checker named MCMAS for the formal verification of multi-agent systems. Their model checker can be used to verify the epistemic, strategic and temporal properties of interest for these multi-agent systems. Reference [24] has presented a new combined model checking approach for eliminating the problem of introducing new logic for the verification of different aspects of multi-agent systems like knowledge and time, knowledge and probability, real-time and knowledge, etc. This will help to reduce the problem of having different model checking tools targeting different aspects of multi-agent systems. Reference [25] has proposed timed automata based behavior specification and property specification templates based on MAPE-K feedback loop for the formal modeling of self-adaptive systems. However their templates are not directly applicable to the multi-agent system's domain. Reference [26] has comprehensively analyzed the self-adaptive software systems to tackle the problem of assuring that the system provides its domain functionality under uncertain conditions. Reference [27] proposed an analytical framework for the architectural reasoning of self-adaptive systems. Their framework extended the Architectural Reasoning Framework (eARF) to ensure that the design complies to the requirements for single product instance. It has been argued in [28] and [29] that the formal methods should be used for the automated verification of safety critical and real-time systems to ensure their correct functioning.

However up to our knowledge currently we do not have a precise vocabulary for the formal specification of key architectural characteristics of self-adaptive real-time multi-agent systems. Encompassing different perspectives of self-adaptation into the real-time multi-agent systems would help in the construction of such systems. The proposed MAPE-K interfaces provides a conceptual design for the system designer to integrate self-adaptation into any real-time multi-agent system. For complex systems conceptual design help to delegate responsibilities to different system entities without the concern for the implementation details. A clear distinction between the entities responsible for providing the system's domain functionality and entities responsible for the self-adaptation would help in validating the system against user requirements. In this paper we have proposed a set of predefined interfaces for the provision of self-adaptation in real-time multi-agent systems. The interfaces are based on Monitor, Analyze, Plan and Execute phases of the MAPE-K feedback loop. We formally specify our interfaces using Timed-Communicating Object-Z (TCOZ) language.

The rest of this paper is divided as follows. In section 2 some preliminaries for MAPE-K feedback loop, TCOZ language and ARTIS agent architecture are explained. Section 3 describes the proposed interfaces and their formal specification in TCOZ. In section 4 we demonstrate the application of our proposed interfaces using the conveyor belt example. Section 5 concludes the paper.

II. PRELIMINARIES

A few terminologies have been described in this section that will be used in the rest of the discussion for the specification of the problem under analysis.

A. MAPE-K FEEDBACK LOOP

A self-adaptive system typically consists of a managed system which provides the domain functionality and a feedback loop that deals with the architectural adaptations of the system. Architecture-based adaptation requires a system to interact with the environment, reason about its models based on the stimulus received and then adapt itself. The feedback loop is known as MAPE-K and it was proposed in [10]. The MAPE represents Monitor, Analyze, Plan, and Execute phase, whereas the K represents the models of the system, its environment and adaptation goals. We refer the reader to [10] for details concerning the MAPE-K feedback loop.

B. ARTIS AGENT ARCHITECTURE

ARTIS agent architecture was proposed in [30] and it is an extension of the blackboard model that has been modified to work in environments with hard temporal constraints. This agent guarantees that it will meet its temporal constraints by the use of an off-line schedulability analysis. Agents perception occurs through a set of sensors and the system's response is exhibited using a set of effectors. These perception and action processes are real-time in nature. The agent has two different categorization of processes namely reflex process and a deliberative process. Every ARTIS agent has a number of internal agents (In-agent) that provides the domain functionality. Every In-agent is designed to solve a particular problem. An In-agent is characterized as critical or acritical. A critical In-agent has a period and a deadline and the agent must perform its operations within those deadlines. In other words it provides the minimum system functionality. On the contrary acritical In-agent can utilize artificial intelligence techniques to better achieve the system goal. Every In-agent has two layers namely reflex layer and real-time deliberative layer. When a task arrives for execution, the In-agent checks the deadline if it can provide a response via a real-time deliberative layer. The real-time deliberative layer provides an improved response as compared to reflex layer, hence it needs more time. The reflex layer only provides a minimal quality response. For real-time environments most of the In-agents are critical in nature. Each In-agent has a set of beliefs comprising the domain knowledge relevant to it. Each ARTIS agent has a Control Module that controls the execution of all the In-agents that belongs to it. It is divided into two sub-modules namely the Reflex server (RS) and the Deliberative server (DS). Reflex server controls the execution of the tasks with critical temporal restrictions. Deliberative server controls the execution of the deliberative tasks.

C. TCOZ LANGUAGE

TCOZ is a formal specification language and was first proposed in [31]. It is a assimilation of Object-Z and Timed Communicating Sequential Processes (CSP). Object-Z incorporates object-oriented paradigm into the standard Z formal specification language. It provides prevailing features for recitation of complex data structures and their operations. Incorporation of object-oriented paradigm improves the clarity of formal specifications documents through enhanced structuring. However Object-Z lacks the operators for determining the duration of the operations and assumes that all the operations are atomic. Because of these limitations it is intricate to use Object-Z for real-time concurrent systems modelling. Timed CSP is a real-time extension of CSP notation and does not have significant modelling elements to encapsulate the state of a system. Hence both Timed CSP and Object-Z supplement each other in their perspicuity. The approach taken in TCOZ is to identify the operation schemas with CSP processes that perform only state update events. Non-terminating CSP processes are represented by active classes in TCOZ. It also provides special channel-based communications for inter-process communications.

III. PROPOSED FORMAL MAPE-K INTERFACES

An overview of the proposed MAPE-K Interfaces is provided in Figure 1. A TCOZ specification of the interfaces is provided along with.

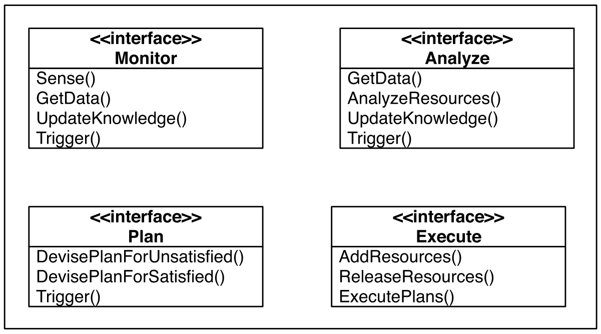


FIGURE 1. Proposed MAPE-K interfaces.

The *MonitorAgent* will continuously perceive the environment and after any pre-processing of data it will update the models and trigger the next i-e *AnalyzeAgent*. *aAgent* represents the *AnalyzeAgent* to whom this agent will notify in case an event of interest occurs requiring adaptation.

$$\begin{array}{l} \textit{MonitorAgent} \\ \hline aAgent : AnalyzeAgent \end{array}$$

$$\begin{array}{l} \textit{Sense}' \\ \hline senseAction : \mathbb{P}\, Subsystem \times \\ \mathbb{P}\, State \rightarrow \mathbb{P}\, State \\ subsystem? : Subsystem \\ \hline state' = senseAction(subsystem?, state) \end{array}$$

$$\begin{array}{l} \textit{GetData}' \\ \hline readAction : \mathbb{P}\, ReflectionModel \\ \times \mathbb{P}\, State \rightarrow \mathbb{P}\, State \\ \hline state' = readAction(rModel, state) \end{array}$$

$$\begin{array}{l} \textit{UpdateKnowledge}' \\ \hline writeAction : \mathbb{P}\, State \times \\ \mathbb{P}\, ReflectionModel \rightarrow \mathbb{P}\, ReflectionModel \\ \hline rModel' = writeAction(state, rModel) \end{array}$$

$$\begin{array}{l} \textit{Trigger}' \\ \hline triggerAction : \mathbb{P}\, State \times \\ AnalyzeAgent \rightarrow AnalyzeAgent \\ \hline aAgent' = triggerAction(state, aAgent) \end{array}$$

$$\begin{array}{l} Sense \mathrel{\widehat=} [subsystem? : Subsystem] \bullet Sense' \\ \rightarrow SKIP \\ GetData \mathrel{\widehat=} GetData' \rightarrow SKIP \\ UpdateKnowledge \mathrel{\widehat=} UpdateKnowledge' \rightarrow \\ SKIP \\ Trigger \mathrel{\widehat=} Trigger' \rightarrow SKIP \\ MAIN \mathrel{\widehat=} \mu\, T \bullet Sense;\ GetData; \\ UpdateKnowledge;\ Trigger;\ T \end{array}$$

The *AnalyzeAgent* will make decisions regarding whether the adaptations are needed or not. In case an adaptation is needed it will trigger the *PlanAgent*. *pAgent* represents the *PlanAgent* to whom this agent will notify to plan for the necessary adaptations. *requiredResources* represent the resources that this agent needs to complete its assigned tasks. *availableResources* are the resources that have been assigned to this agent. At anytime the attribute *rRequirement* represents the situation of the resources for this agent. Resource requirement can be divided into four classes, one in which the system does not require additional resources, second in which the system has more resources than it needs, third in which the system needs more resources and fourth in which it is not possible to get a predictable total of the system resources.

$$\begin{array}{l} ResourcesRequirement ::= SATISFIED | \\ OVERSATISFIED | UNSATISFIED | UNDETERMINED \end{array}$$

$$\begin{array}{l} \textit{AnalyzeAgent} \\ \hline pAgent : PlanAgent \\ requiredResources : \mathbb{P}\, Resource \\ availableResources : \mathbb{P}\, Resource \\ rRequirement : ResourcesRequirement \end{array}$$

$$\begin{array}{l} \textsc{Init} \\ \hline rRequirement = UNDETERMINED \end{array}$$

$$\begin{array}{l} \textit{GetData}' \\ \hline readAction : \mathbb{P}\, ReflectionModel \\ \times \mathbb{P}\, State \rightarrow \mathbb{P}\, State \\ \hline state' = readAction(rModel, state) \end{array}$$

$$\begin{array}{l} \textit{AnalyzeResources}' \\ \hline \Delta requiredResources \\ \Delta availableResources \end{array}$$

$$\begin{array}{l} \textit{UpdateKnowledge}' \\ \hline writeAction : \mathbb{P}\, State \times \\ \mathbb{P}\, ReflectionModel \rightarrow \mathbb{P}\, ReflectionModel \\ \hline rModel' = writeAction(state, rModel) \end{array}$$

$$\begin{array}{l} \textit{Trigger}' \\ \hline triggerAction : \mathbb{P}\, State \times PlanAgent \\ \rightarrow PlanAgent \\ \hline aAgent' = triggerAction(state, pAgent) \end{array}$$

$$\begin{array}{l} GetData \mathrel{\widehat=} GetData' \rightarrow SKIP \\ AnalyzeResources \mathrel{\widehat=} [requiredResources?, \\ availableResources : Resource] \bullet \\ ((AnalyzeResources' \wedge rRequirement \\ = SATISFIED) \\ \Box(AnalyzeResources' \wedge rRequirement \\ = UNDERSATISFIED)) \\ \Box(AnalyzeResources' \wedge rRequirement \\ = OVERSATISFIED)) \rightarrow SKIP \\ UpdateKnowledge \mathrel{\widehat=} UpdateKnowledge' \\ \rightarrow SKIP \\ Trigger \mathrel{\widehat=} Trigger' \rightarrow SKIP \\ MAIN \mathrel{\widehat=} \mu\, T \bullet GetData; \\ AnalyzeResources;\ UpdateKnowledge; \\ Trigger;\ T \end{array}$$

The *PlanAgent* will prepare a set of tasks/actions that are required for the adaptation and then it will trigger the *ExecuteAgent*. It basically plans two type of actions. In case the system resources are unsatisfied it creates plans to add the resources to the system. On the contrary if the system resources are over satisfied it creates plans to release the extra resources. *eAgent* represents the *ExecuteAgent* to whom this agent will notify to execute the actions required for the necessary adaptations. In case the resources are under-satisfied, a set of plan actions are devised in order to add the resources to the managed system. Similarly in case of over-satisfied, a set of plan actions are devised in order to release the extra resources of the managed system. We define a new type *Plan* which is a collection of tasks.

$Plan ::= \mathbb{P}\, Task$

PlanAgent

$eAgent : ExecuteAgent$
$rRequirement : ResourcesRequirement$

INIT
$rRequirement = UNDETERMINED$

DevisePlanForUnsatisfied′
$rRequirement? = UNSATISFIED$
$plans? : Plan$

$c!plans$

DevisePlanForOversatisfied′
$rRequirement? = OVERSATISFIED$
$plans? : Plan$

$c!plans$

Trigger′
$triggerAction : \mathbb{P}\, State \times ExecuteAgent \rightarrow ExecuteAgent$

$eAgent' = triggerAction(state, eAgent)$

$DevisePlanForUnsatisfied \mathrel{\widehat=} [rRequirement : ResourcesRequirement;\ plans? : Plan] \bullet (DevisePlanForUnsatisfied' \wedge rRequirement = UNSATISFIED) \bullet c?(plans) \rightarrow SKIP$
$DevisePlanForOversatisfied \mathrel{\widehat=} [rRequirement : ResourcesRequirement;\ plans? : Plan] \bullet (DevisePlanForOversatisfied' \wedge rRequirement = OVERSATISFIED) \bullet c?(plans) \rightarrow SKIP$
$Trigger \mathrel{\widehat=} Trigger' \rightarrow SKIP$
$MAIN \mathrel{\widehat=} \mu\, T \bullet (DevisePlanForUnsatisfied \,\square\, DevisePlanForOversatisfied);\ Trigger;\ T$

The *ExecuteAgent* is responsible for executing the adaptation actions of the generated plans. There are three phases in an execute behavior i-e *PreProcess*, *ExecutePlan* and *PostProcess*. In *PreProcess* the agent acquires all the resources that are required for the adaptation goals. Once all the pre-processing has been completed, the agent performs *ExecutePlan* to perform the necessary adaptations. After all the plans have been executed the agent performs *PostProcess* to release all the acquired resources.

ExecuteAgent

$InAgent$
$requiredResources : \mathbb{P}\, Resource$
$availableResources : \mathbb{P}\, Resource$
$rRequirement : ResourcesRequirement$

INIT
$rRequirement = UNDETERMINED$

AddResources′
$newResources?\ \mathbb{P}\, Resource$
$\Delta requiredResources$
$\Delta availableResources$
$\Delta alloc$

$availableResources' = availableResources \cup \{newResources\}$
$requiredResources' = requiredResources \setminus \{newResources\}$

ExecutePlans′
$plans? : Plan$
$execute : Plan \times \mathbb{P}\, State \times \mathbb{P}\, ReflectionModel \rightarrow \mathbb{P}\, ReflectionModel$

$plans \neq \varnothing$
$\forall (plan \in plans \wedge execute(plan, state, rModel))$

ReleaseResources′
$extraResources?\ \mathbb{P}\, Resource$
$\Delta availableResources$
$\Delta alloc$

$availableResources' = availableResources \setminus \{extraResources\}$

$PreProcess \mathrel{\widehat=} [newResources? : Resource] \bullet (AddResources' \wedge rRequirement = UNSATISFIED) \rightarrow SKIP$
$ExecutePlans \mathrel{\widehat=} [plans? : Plan] \bullet ExecutePlans' \rightarrow SKIP$
$PostProcess \mathrel{\widehat=} [extraResources? : Resource] \bullet (ReleaseResources' \wedge rRequirement = OVERSATISFIED) \rightarrow SKIP$
$MAIN \mathrel{\widehat=} \mu\, T \bullet PreProcess;\ ExecutePlans;\ PostProcess;\ T$

A *model* is a representation describing entities of interest in the physical or virtual world. Concrete models can have different types of representations. Models are defined as generic constant.

$$
\begin{array}{|l}
[\mathit{Model}[\mathit{Representation}]] \\
\mathit{rep} : \mathbb{P}\, \mathit{Representation} \\
\hline
\mathit{rep} \neq \varnothing
\end{array}
$$

A *domain model* contains representations of entities necessary for the provision of required features. Here *EnvironmentRepresentation* specifies the attributes of the environment. Domain model is defined as a passive class. The channel *c* is used for communication with any active class. The domain model here represents the K (Knowledge) part of the MAPE-K feedback loop.

$$
\begin{array}{|l}
\mathit{DomainModel} \\
\mathit{Model}[\mathit{EnvironmentRepresentation}] \\
\mathit{map} : \mathbb{P}\, \mathit{Attribute} \leftrightarrow \\
\mathit{EnvironmentRepresentation} \\
\mathit{envr} : \mathbf{chan} \\
\hline
\mathrm{dom}\, \mathit{map} \subseteq \{\mathit{attrs} : \mathbb{P}\, \mathit{Attribute} \\
\mid \mathit{attrs} \subseteq \mathit{envr}?.\mathit{attributes}\} \\
\mathrm{ran}\, \mathit{map} = \{r : \mathit{EnvironmentRepresentation} \\
\mid r \in \mathit{rep}\}
\end{array}
$$

IV. APPLICATION OF MAPE-K INTERFACES

In this section we explain the application of our proposed MAPKE-K interfaces. We have chosen ARTIS agent as described in section 2 for this purpose.

A. CONVEYOR BELT SYSTEM

In this section we will demonstrate the working of *conveyor belt system* as described in [30]. The system under consideration is a control system for a conveyor belt in a glazed tile factory. A *conveyor belt* moves *glazed tiles* from left to right at the configured speed. There is a robot arm at the start of the conveyor belt on the left side whose job is to place glazed tiles on the conveyor belt. The speed of the tiles movement is adjustable. An *optical sensor* equipped with an infrared system and a video camera will detect the faulty glazed tiles. A *robot arm* adjacent to the optical sensor will remove the faulty tiles. Another robot arm at the right side of the conveyor belt is responsible for picking the non-faulty glazed tiles. Analysis of the glazed tiles is the main goal of the system. Main features of the system are defined below.

- The *glazed tiles* are placed on the *conveyor belt* at a pre-defined speed. The *robot arm* responsible for this can be configured to change the number of tiles placed on the conveyor belt per unit time.
- The *optical sensor* will be responsible for analyzing the quality of the glazed tile passing in front of it within the temporal constraints.
- The *robot arm* will remove a faulty glazed tile from the conveyor belt. However if the deadline allows (system throughput is low) it will classify the faulty glazed tile into three different categories. These categories are first quality, second quality, or garbage.
- The remaining non-faulty *glazed tiles* are removed from the *conveyor belt* by a *robot arm.*
- The system throughput can be increased or decreased by changing the conveyor belt speed. This will require adaptive behavior from all the working components of the system.

The system will utilize three ARTIS agents namely *PlacingARTISAgent*, *WorkingARTISAgent* and *PickingARTISAgent*. The temporal restrictions of these ARTIS agents needs to be evaluated at the system design time. An off-line schedulability test will ensure that all the agents meet their deadlines. The PlacingARTISAgent will be responsible for placing tiles on the conveyor belt at a defined speed. It will contain only one In-agent namely *PlacingRobotArmInAgent* which will be in charge of placing the tiles on the conveyor belt. Its reflex action is to place the tiles on the conveyor belt at the defined speed. Its cognitive action is to try to increase the system throughput by coordinating with the In-agents of other ARTIS agents. The WorkingARTISAgent will be responsible for removing the faulty tiles from the conveyor belt and categorizing the faulty tiles. It will contain two In-agents namely *WorkingOpticalSensorInAgent* and *WorkingRobotArmInAgent*. The WorkingOpticalSensor InAgent will examine the tiles and store its visual pattern in a shared memory. The visual pattern can be in the form of an image or video. Its reflex action is to get a visual pattern of the tile in the form of an image based on the infrared. Its cognitive action is to get a visual pattern of the tile in the form of video by means of a video camera. The WorkingRobotArmInAgent will use the visual pattern obtained by the optical sensor In-agent and evaluate the tile's quality. Its reflex action is to detect the faulty tiles and remove them from the conveyor belt. Its cognitive action is to categorize the faulty tiles according to the first quality, second quality or garbage. The PickingARTISAgent will be responsible to pick the non-faulty tiles from the conveyor belt at a defined speed. It will contain only one In-agent namely *PickingRobotArmInAgent*. The PickingRobotArmInAgent will pick the tiles from the conveyor belt. Since this agent cannot increase the throughput itself like placing ARTIS agent so it will not have the cognitive action. Its reflex action is to pick tiles from the conveyor belt at a defined speed.

B. APPLICATION OF PROPOSED MAPE-K INTERFACES TO CONVEYOR BELT SYSTEM

Conveyor belt system explained in the previous section is an example of a real-time self-adaptive multi-agent system. The system is real-time because of the temporal restrictions for each agent to process the tiles and it is self-adaptive as each agent has to reorganize itself based on the system parameters like throughput. Hence a behavior feature of adaptivity is necessary for the *ARTIS agent* to allow the system to react

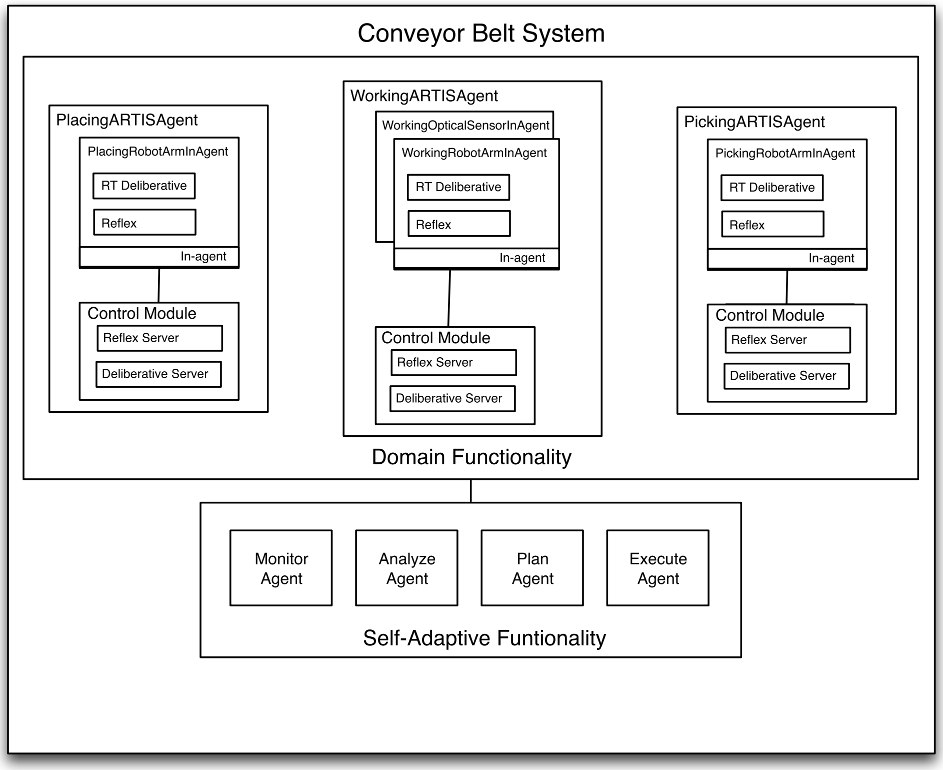


FIGURE 2. **Complete overview of the conveyor belt system.**

to the environmental changes. Figure-2 provides a complete overview of the conveyor belt system. For brevity we will not provide the TCOZ specification of the Monitor, Analyze, Plan and Execute agents again. We formally specify the conveyor belt system by individually specifying the entities in TCOZ specification. First we give the type definitions and the constant definitions.

$TileType ::= UN\_IDENTIFIED | FIRST\_QUALITY |$
$SECOND\_QUALITY | FAULTY |$
NON_FAULTY
$TileStatus ::= ON\_BELT | GRASPED |$
$OpticalSensorStatus ::= RECOGNIZE | IDLE$
$WorkingAgentStatus ::= ANALYZE | IDLE$
$RobotArmStatus ::= HOLDING | EMPTY$
$ConveyorBeltStatus ::= MOVING | STOPPED$
$SystemThroughput ::= LOW | HIGH$
$TaskDuration ::= MAX | MIN$

We define the conveyor belt system's attributes and processes that will be used further in the discussion. Conveyor belt processes represents the different ongoing activities and the communication between the agents.

conveyor_belt_attributes == {GlazedTile, PlacingARTISAgent, PlacingRobotArmInAgent, WorkingARTISAgent, WorkingOpticalSensorInAgent, WorkingRobotArmInAgent, PickingARTISAgent, PickingRobotArmInAgent, ConveyorBelt}

conveyor_belt_processes == {ReflexPlaceTile, DeliberativePlaceTile, ReflexRecognize, DeliberativeRecognize, ReflexAnalyze, DeliberativeAnalyze, RemoveFaultyTile, ReflexPickTile, ReflexPickTile, Move, Place, Stop, Start}

A conveyor belt environment is an environment with the conveyor belt attributes and processes.

ConveyorBeltEnvironment
Environment

$attributes \subseteq conveyor\_belt\_attributes$
$processes \subseteq conveyor\_belt\_processes$

We consider the following Conveyor belt environment representations.

$Tile, PlacingAA, PlacingRAI, WorkingAA,$
$WorkingOSI, WorkingRAI, PickingAA,$
$PickingRAI, CBelt, Ping,$
$Echo : EnvironmentRepresentation$

cb_environment_representations == {Tile, PlacingAA, PlacingRAI, WorkingAA, WorkingOSI, WorkingRAI, PickingAA, PickingRAI, CBelt, Ping, Echo}

Environment representations and attribute sets are mapped as

cb_attribute_representation_mapping == { GlazedTile $\mapsto$ *Tile, PlacingARTISAgent, PlacingRobotArmInAgent, WorkingARTISAgent, WorkingOpticalSensorInAgent,* *WorkingRobotArmInAgent*, *PickingARTISAgent*, *PickingRobotArmInAgent*, *ConveyorBelt }*

A conveyor belt model is defined as

ConveyorBeltModel

$Model[EnvironmentRepresentation]$
$mapping : \mathbb{P}\, Attribute \leftrightarrow EnvironmentRepresentation$
$c : \mathbf{chan}$

$reps \subseteq cb\_environment\_representations$
$\mathrm{dom}\, map \subseteq \{attrs : \mathbb{P}\, Attribute \mid attrs \subseteq c?.attributes\}$
$\mathrm{ran}\, map = \{r : EnvironmentRepresentation \mid r \in reps\}$

The specification below introduces the features and operations of a glazed tile. A glazed tile has a unique identifier position, status, type and identifier of the conveyor belt on which it is placed. There are two operations defined for a glazed tile, to change its position and to change its status. The id of a part cannot be changed. It is defined as a passive class.

GlazedTile

$id : \mathbb{N}$
$type : TileType$
$position : Point$
$status : TileStatus$
$cbId : ConveyorBelt$

INIT
$id \neq \varnothing$
$type = UN\_IDENTIFIED$
$cbId \neq \varnothing$

ChangePosition
$\Delta(position)$
$newPosition? : Point$

$position' = newPosition?$

ChangeStatus
$\Delta(status)$
$newStatus? : TileStatus$

$status' = newStatus?$

ChangeType
$\Delta(type)$
$newType? : TileType$

$type' = newType?$

Before specifying the agents we introduce an abstract type for the coordinating agents. Each agent has two type of roles.

$Role ::= master \mid slave$

The coordination protocol used by the agents is defined below.

MasterSlave
$role : Role$

Now we define the coordination model used by the agents. *Partners* represent the name of the agents with which this agent is currently communicating with. There can be only one partner of a slave, which is its master. For master, its partners are its slaves. *NeighborOrganizations* corresponds to the masters of the neighboring organizations.

OrganizationPartners
$partners : \mathbb{P}\, Id$
$neighborOrganizations : \mathbb{P}\, Id$

To represent the messages exchanged between the agents, we define a message type. It contains the identifier of the sender, the receiver and the actual message content.

MasterSlave
$to : Id$
$from : Id$
$content : Content$

The coordination mechanism used by the agents is defined below. The function *senderOrReceiver* means either one of the sender or the receiver of the channel.

DynamicAgentOrganizations
$cbsProtocol : MasterSlave$
$cbsModel : OrganizationPartners$

$\forall\, p : cbsModel.partners \bullet \exists\, l : \mathbf{chan} \bullet senderOrReceiver(l) = p \land$
$\forall\, org : cbsModel.neighborOrganizations \bullet \exists\, l : \mathbf{chan} \bullet senderOrReceiver(l) = org$

The PlacingARTISAgent will be responsible to place the tiles on the conveyor belt at a defined speed. It will contain only one In-agent namely robot arm In-agent.

PlacingARTISAgent

$ARTISAgent$
$id : Id$
$agent : PlacingRobotArmInAgent$
$cbCoordnationMechanism : DynamicAgentOrganizations$

INIT
$distance = MIN\_DISTANCE$

Each PlacingRobotArmInAgent is in-charge of placing the tiles on the conveyor belt one by one. The attribute *position* represents the position of the RobotArm. Its reflex action is to place the tiles on the conveyor belt at a defined speed.

Its deliberative action is to try to increase the system throughput by coordinating with the In-agents of other ARTIS agents. In operations *DeliberativePlaceTile* and *ReflexPlaceTile* the attributes δ and *throughput* are inversely proportional to each other. When throughput will be high the time taken to place a glazed tile on the conveyor belt will be minimum and vice versa.

PlacingRobotArmInAgent

$InAgent$
$id : Id$
$belt : ConveyorBelt$
$cbCoordnationMechanism :$
$DynamicAgentOrganizations$
$position : \mathbb{N}$
$throughput : SystemThroughput$
$status : WorkingAgentStatus$
$armstatus : RobotArmStatus$

ReflexPlaceTile
$tile? : GlazedTile$
$\delta = MAX$
$throughput = HIGH$

DeliberativePlaceTile
$tile? : GlazedTile$
$\delta = MIN$
$throughput = HIGH$

INIT
$status = IDLE$

ChangeStatus
$\Delta(status)$
$status' = if\ status = ANALYZE$
$then\ IDLE$
$else\ if\ status = IDLE$
$then\ ANALYZE$

ChangeSpeed
$\Delta(speed)$
$newSpeed? : \mathbb{N}$
$speed' = newSpeed?$

$ReflexPlaceTile \mathrel{\widehat=} [armstatus = HOLDING$
$\wedge\ status = IDLE] \wedge [tile? :$
$Glazed\_Tile \mid tile.position = position$
$\wedge\ tile?.status = GRASPED] \bullet$
$ChangeStatus \wedge tile?.ChangePosition$
$[old\_position/new\_position] \wedge$
$[newst? : TileStatus \mid newst? = ON\_BELT]$
$\bullet\ tile?.ChangeStatus \bullet \delta = MAX$
$\rightarrow SKIP$
$DeliberativePlaceTile \mathrel{\widehat=} [armstatus =$
$HOLDING \wedge status = IDLE] \wedge$
$[tile? : Glazed\_Tile \mid tile.position =$
$position \wedge tile?.status = GRASPED] \bullet$
$ChangeStatus \wedge tile?.ChangePosition$
$[old\_position/new\_position] \wedge$
$[newst? : TileStatus \mid newst? = ON\_BELT]$
$\bullet\ tile?.ChangeStatus \bullet \delta = MIN$
$\rightarrow SKIP$
$MAIN \mathrel{\widehat=} \mu T \bullet (ReflexPlaceTile$
$\Box DeliberativePlaceTile);\ T$

The WorkingARTISAgent will be responsible for removing the faulty tiles from the conveyor belt and categorizing the faulty tiles. It will contain two In-agents namely *WorkingOpticalSensorInAgent* and *WorkingRobotArmInAgent*.

WorkingARTISAgent

$ARTISAgent$
$id : Id$
$agent : WorkingRobotArmInAgent$
$agent : WorkingOpticalSensorInAgent$
$cbCoordnationMechanism :$
$DynamicAgentOrganizations$

INIT
$distance = MIN\_DISTANCE$

The WorkingOpticalSensorInAgent will examine the tiles and store its visual pattern in a shared memory. The visual pattern can be in the form of an image or video. Its reflex action is to get a visual pattern of the tile in the form of an image based on infrared. Its cognitive action is to get a visual pattern of the tile in the form of video by means of a video camera. Its attribute *status* is used to indicate whether the agent is currently analyzing a tile or is idle. Initially, it is idle but when it is analyzing a tile, its status changes from IDLE to RECOGNIZE.

WorkingOpticalSensorInAgent

$InAgent$
$id : Id$
$status : OpticalSensorStatus$
$c : \mathbf{chan}$
$position : \mathbb{N}$
$cbCoordnationMechanism :$
$DynamicAgentOrganizations$

INIT
$status = IDLE$

ReflexRecognize'
$\Delta(status)$
$tile? : GlazedTile$
$image! : IMAGE$
$tile?.position = self.position$
$status = IDLE$
$status' = RECOGNIZE$

DeliberativeRecognize′

$\Delta(status)$
$tile? : GlazedTile$
$video! : VIDEO$

$tile?.position = self.position$
$status = IDLE$
$status' = RECOGNIZE$

Idle

$\Delta(status)$
$newStatus? : OpticalSensorStatus$

$status = RECOGNIZE$
$status' = IDLE$

$ReflexRecognize \mathrel{\widehat=} [tile? : GlazedTile\ ;\ image! : IMAGE] \bullet ReflexRecognize' \rightarrow c!(image) \rightarrow SKIP$
$DeliberativeRecognize \mathrel{\widehat=} [tile? : GlazedTile;\ video! : VIDEO] \bullet DeliberativeRecognize' \rightarrow c!(video) \rightarrow SKIP$
$MAIN \mathrel{\widehat=} \mu\, T \bullet (ReflexRecognize \ \Box\ DeliberativeRecognize);\ T$

The WorkingRobotArmInAgent will use the visual pattern obtained by the WorkingOpticalSensorInAgent and evaluate the tile's quality. Its reflex action is to detect the faulty tiles and remove them from the conveyor belt. Its cognitive action is to categorize the faulty tiles according to the first quality, second quality or garbage.

WorkingRobotArmInAgent

$InAgent$
$id : Id$
$status : WorkingRobotArmStatus$
$position : \mathbb{N}$
$c : \textbf{chan}$
$cbCoordnationMechanism : DynamicAgentOrganizations$

INIT

$status = IDLE$

DeliberativeAnalyze′

$\Delta(status)$
$video? : VIDEO$
$type! : TileType$

$status = IDLE$
$status' = ANALYZE$
$type! = tile?.type$

ReflexAnalyze′

$\Delta(status)$
$image? : IMAGE$
$type! : TileType$

$status = IDLE$
$status' = ANALYZE$
$type! = tile?.type$

ChangeStatus

$\Delta(status)$

$status' = if\ status = ANALYZE\ then\ IDLE\ else\ if\ status = IDLE\ then\ ANALYZE$

Idle

$\Delta(status)$
$newStatus? : WorkingRobotArmStatus$

$status = ANALYZE$
$status' = IDLE$

$ReflexAnalyze \mathrel{\widehat=} [image? : IMAGE;\ type! : TileType] \bullet ReflexAnalyze' \rightarrow c!(type) \rightarrow SKIP$
$DeliberativeAnalyze \mathrel{\widehat=} [video? : VIDEO;\ type! : TileType] \bullet DeliberativeAnalyze' \rightarrow c!(type) \rightarrow SKIP$
$RemoveFaultyTile \mathrel{\widehat=} [status = IDLE] \wedge [tile? : Glazed\_Tile \mid tile?.position \neq position \wedge tile?.status = ON\_BELT] \bullet ChangeStatus \wedge tile?.ChangePosition[position/new?] \wedge [newst? : TileStatus \mid newst? = GRASPED] \bullet tile?.ChangeStatus \rightarrow SKIP$
$MAIN \mathrel{\widehat=} \mu\, T \bullet (ReflexAnalyze \ \Box\ DeliberativeAnalyze \ \Box\ RemoveFaultyTile);\ T$

The PickingARTISAgent will be responsible to pick the non-faulty tiles from the conveyor belt at a defined speed. It will contain only one In-agent namely *PickingRobotArmInAgent*.

PickingARTISAgent

$ARTISAgent$
$id : Id$
$agent : PickingRobotArmInAgent$
$cbCoordnationMechanism : DynamicAgentOrganizations$

INIT

$distance = MINDISTANCE$

The PickingRobotArmInAgent is in charge of picking the tiles from the conveyor belt. Since this agent cannot increase the throughput like placing agent so it will not have the cognitive action. Its reflex action is to pick the tiles from the conveyor belt at a defined speed.

PickingRobotArmInAgent

$InAgent$
$id : Id$
$belt : ConveyorBelt$
$position : \mathbb{N}$
$throughput : SystemThroughput$
$status : WorkingAgentStatus$
$armstatus : RobotArmStatus$
$cbCoordnationMechanism :$
$DynamicAgentOrganizations$

DeliberativePickTile
$tile? : GlazedTile$

$\delta = MIN$
$throughput = HIGH$

INIT
$status = IDLE$

ReflexPickTile
$tile? : GlazedTile$

$\delta = MAX$
$throughput = HIGH$

ChangeStatus
$\Delta(status)$

$status' = if\ status = ANALYZE$
$then\ IDLE$
$else\ if\ status = IDLE$
$then\ ANALYZE$

ChangeSpeed
$\Delta(speed)$
$newSpeed? : \mathbb{N}$

$speed' = newSpeed?$

$DeliberativePickTile \mathrel{\widehat=} [status = EMPTY]$
$\wedge\ [tile? : Glazed\_Tile \mid tile?.position$
$\neq position \wedge tile?.status = ON\_BELT] \bullet$
$ChangeStatus \wedge tile?.ChangePosition$
$[position/new?] \wedge [newst? : TileStatus$
$\mid newStatus? = GRASPED] \bullet tile?$
$.ChangeStatus \bullet \delta = MAX \rightarrow SKIP$
$ReflexPickTile \mathrel{\widehat=} [status = EMPTY] \wedge$
$[tile? : Glazed\_Tile \mid tile?.position$
$\neq position \wedge tile?.status = ON\_BELT] \bullet$
$ChangeStatus \wedge tile?.ChangePosition$
$[position/new?] \wedge [newst? : TileStatus$
$\mid newStatus? = GRASPED] \bullet tile?$
$.ChangeStatus \bullet \delta = MIN \rightarrow SKIP$
$MAIN \mathrel{\widehat=} \mu\, T \bullet (ReflexPickTile$
$\Box DeliberativePickTile);\ T$

Every conveyor belt is of finite length and has a distinct start and end point. The status of the conveyor belt indicates whether it is moving or stopped. Its state invariant ensures that no two glazed tiles on the conveyor belt occupy the same position. Also it ensures that the position of every glazed tile on the conveyor belt is within its region and their status is ON BELT. There are four operations defined for a conveyor belt i-e *Start*, *Stop*, *Move* and *Place*. The operations Start and Stop denote the status when the conveyor belt is started and is stopped respectively. Move operation specifies the constraints to move a conveyor belt through a distance *dist* and asserts that the status of the conveyor belt should be MOVING when it is invoked. The Move operation collectively moves all glazed tiles on the belt. Due to the movement, some glazed tiles may accidentally go out of the conveyor belt (indicated by out! in Move). Finally, the Place operation defines the constraints for placing a glazed tile on the conveyor belt. It asserts that the id of the new glazed tile should not match with the id of any other glazed tile on the conveyor belt and every new glazed tile is placed at the start of the conveyor belt.

ConveyorBelt

$startofbelt : \mathbb{N}$
$endofbelt : \mathbb{N}$
$tiles : iseq\ GlazedTile$
$status : ConveyorBeltStatus$

INIT
$tiles = \langle\rangle$
$status = STOPPED$
$\forall i,j : 1..\#tiles \bullet$
$(i \neq j \Rightarrow tile(i).id \neq tiles(j).id \wedge$
$tile(i).position \neq tiles(j).position) \wedge$
$tiles(i).status = ONBELT \wedge$
$IsOnBelt(tiles(i).position)$
$IsOnBelt(start\_of\_belt)$
$\wedge\ IsOnBelt(end\_of\_belt)$
$start\_of\_belt \neq end\_of\_belt$

Move'
$\Delta(tiles)$
$distance? : \mathbb{R}$
$outofbelt! : iseq\ GlazedTile$

$status = MOVING$
$tiles = tiles' \frown outofbelt!$
$\forall i : 1..\#tiles' \bullet (\exists_1 j : 1..\#tiles$
$\mid tiles'(i).id = tiles(j).id \bullet tiles'(i).position$

$= tiles(j).position + distance?$
$\wedge\ IsOnBelt(tiles'(i).position))$
$\forall\, tile : GlazedTile \mid tile \in ran\ outofbelt!$
$\bullet\ \neg(IsOnBelt(tile.position))$

Start′

$\Delta(status)$

$status = STOPPED$
$status' = MOVING$

Stop′

$\Delta(status)$

$status = MOVING$
$status' = STOPPED$

Place′

$\Delta(tiles)$
$newTile? : GlazedTile$

$newTile?.id \notin \{tile : GlazedTile \mid$
$tile \in tiles \bullet tile.id\}$
$tiles' = \langle(\mu\, tile : Glazed\_Tile \mid$
$tile.id = newTile?.id \wedge tile.status$
$= ONBELT \wedge tile.position$
$= start\_of\_belt)\rangle \frown tiles$

$Move \mathrel{\widehat=} [distance? : \mathbb{R};\ outofbelt! :$
$iseq\ GlazedTile] \bullet Move' \to SKIP$
$Place \mathrel{\widehat=} [newTile? : GlazedTile] \bullet$
$Place' \to SKIP$
$Start \mathrel{\widehat=} Start' \to SKIP$
$Stop \mathrel{\widehat=} Stop' \to SKIP$
$MAIN \mathrel{\widehat=} \mu\, T \bullet (Move \Box Place \Box Start$
$\Box Stop);\ T$

V. CONCLUSION

In this paper we have proposed a set of interfaces for the formal modelling of self-adaptive real-time multi-agent systems. We proposed four set of interfaces, each targeting four phases of the MAPE-K feedback loop namely Monitor, Analyze, Plan and Execute. The multi-agent systems paradigm has been in use for the ubiquitous and pervasive environments. The dynamism in the execution of such systems has led to the development of self-adaptive systems. Though there exists many different frameworks for self-adaptive multi-agent systems but according to our knowledge no work has been done for the formal modelling of real-time multi-agent systems with the ability of self-adaptation. Hence there is a dire need of formal vocabulary that can be used for the conceptual design of any real-time multi-agent system with self-adaptation. We used TCOZ for the formal specification of our proposed interfaces and the major advantage was that we can utilize the active class concept of TCOZ to express the non-terminating behavior of the autonomous agents. The provision of communication channels in TCOZ greatly simplified the class definitions and class referencing leading to their enhanced modularity. For future work we intend to overcome the limitations of our research and provide detailed specifications for managed system components and self-adaptive units that will make a complete self-adaptive system. We also intend to work on the issues of communication between multiple self-adaptive systems using diverse agent platforms. Specifically issues related to the agent communication languages for diverse agents with self-adaptive ability.

REFERENCES

[1] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.
[2] R. Alur, A. Itai, R. P. Kurshan, and M. Yannakakis, "Timing verification by successive approximation," *Inf. Comput.*, vol. 118, no. 1, pp. 142–157, 1995.
[3] I. Filippidis, D. V. Dimarogonas, and K. J. Kyriakopoulos, "Decentralized multi-agent control from local LTL specifications," in *Proc. IEEE 51st Annu. Conf. Decision Control (CDC)*, Dec. 2012, pp. 6235–6240.
[4] E. Garcia, A. Giret, and V. Botti, "A model-driven CASE tool for developing and verifying regulated open MAS," *Sci. Comput. Program.*, vol. 78, no. 6, pp. 695–704, 2013.
[5] V. Hilaire, P. Gruer, A. Koukam, and O. Simonin, "Formal driven prototyping approach for multiagent systems," *Int. J. Agent-Oriented Softw. Eng.*, vol. 2, no. 2, pp. 246–266, 2008.
[6] T. Miller and P. McBurney, "Multi-agent system specification using TCOZ," in *Multiagent System Technologies*. Berlin, Germany: Springer, 2005, pp. 216–221.
[7] E. Ball and M. Butler, "Event-B patterns for specifying fault-tolerance in multi-agent interaction," in *Methods, Models and Tools for Fault Tolerance*. Berlin, Germany: Springer, 2009, pp. 104–129.
[8] G. Smith and K. Winter, "Incremental development of multi-agent systems in Object-Z," in *Proc. 35th Annu. IEEE Softw. Eng. Workshop (SEW)*, Oct. 2012, pp. 120–129.
[9] J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and É. André, "Modeling and verifying hierarchical real-time systems using stateful timed CSP," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, p. 3, 2013.
[10] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.
[11] M. Shaw, "Beyond objects: A software design paradigm based on process control," *ACM SIGSOFT Softw. Eng. Notes*, vol. 20, no. 1, pp. 27–38, 1995.
[12] J. Kramer and J. Magee, "Self-managed systems: An architectural challenge," in *Proc. Future Softw. Eng. (FOSE)*, 2007, pp. 259–268.
[13] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proc. 20th Int. Conf. Softw. Eng.*, 1998, pp. 177–186.
[14] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.
[15] K. Geihs, R. Reichle, M. Wagner, and M. U. Khan, "Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments," in *Software Engineering for Self-Adaptive Systems*. Berlin, Germany: Springer, 2009, pp. 146–163.
[16] B. Bartels and S. Glesner, "Verification of distributed embedded real-time systems and their low-level implementations using timed CSP," in *Proc. 18th Asia Pacifi Softw. Eng. Conf. (APSEC)*, 2011, pp. 195–202.
[17] A. Qasim, S. A. R. Kazmi, and I. Fakhir, "Executable semantics for the formal specification and verification of E-agents," *Indian J. Sci. Technol.*, vol. 8, no. 16, pp. 1–8, 2015.
[18] A. Qasim, S. A. R. Kazmi, and I. Fakhir, "Formal specification and verification of real-time multi-agent systems using timed-arc petri nets," *Adv. Elect. Comput. Eng.*, vol. 15, no. 3, pp. 73–78, 2015.
[19] D. Weyns, S. Malek, and J. Andersson, "FORMS: Unifying reference model for formal specification of distributed self-adaptive systems," *ACM Trans. Auto. Adapt. Syst.*, vol. 7, no. 1, p. 8, 2012.
[20] A. H. Reynisson *et al.*, "Modelling and simulation of asynchronous real-time systems using timed Rebeca," *Sci. Comput. Program.*, vol. 89, pp. 41–68, Sep. 2014.

[21] Y. Chen, "STeC: A location-triggered specification language for real-time systems," in *Proc. 15th IEEE Int. Symp. Object/Compon./Service-Oriented Real-Time Distrib. Comput. Workshops (ISORCW)*, Apr. 2012, pp. 1–6.

[22] T. Logenthiran, D. Srinivasan, A. M. Khambadkone, and H. N. Aung, "Multiagent system for real-time operation of a microgrid in real-time digital simulator," *IEEE Trans. Smart Grid*, vol. 3, no. 2, pp. 925–933, Jun. 2012.

[23] A. Lomuscio, H. Qu, and F. Raimondi, "MCMAS: An open-source model checker for the verification of multi-agent systems," *Int. J. Softw. Tools Technol. Transf.*, pp. 1–22, Apr. 2015. [Online]. Available: http://link.springer.com/journal/10009/onlineFirst/page/2

[24] S. Konur, M. Fisher, and S. Schewe, "Combined model checking for temporal, probabilistic, and real-time logics," *Theor. Comput. Sci.*, vol. 503, pp. 61–88, Sep. 2013.

[25] D. Gil De La Iglesia and D. Weyns, "MAPE-K formal templates to rigorously design behaviors for self-adaptive systems," *ACM Trans. Auto. Adapt. Syst.*, vol. 10, no. 3, p. 15, 2015.

[26] D. Weyns *et al.*, "Perpetual assurances in self-adaptive systems," in *Proc. Assurances Self-Adapt. Syst., Dagstuhl Seminar*, 2014, p. 13511.

[27] N. Abbas, J. Andersson, M. U. Iftikhar, and D. Weyns, "Rigorous architectural reasoning for self-adaptive software systems," in *Proc. 1st Workshop Qualitative Reason. About Softw. Archit.*, 2016, pp. 1–8.

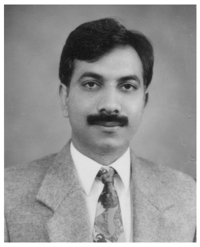
[28] I. Schaefer and R. Hahnle, "Formal methods in software product line engineering," *Computer*, vol. 44, no. 2, pp. 82–85, 2011.

[29] D. Gil De La Iglesia, "A formal approach for designing distributed self-adaptive systems," Ph.D. dissertation, Dept. Comput. Sci., Linnaeus Univ., Växjö, Sweden, 2014.

[30] V. Botti, C. Carrascosa, V. Julián, and J. Soler, "Modelling agents in hard real-time environments," in *Multi-Agent System Engineering*. Berlin, Germany: Springer, 1999, pp. 63–76.

[31] B. Mahony and J. S. Dong, "Timed communicating object Z," *IEEE Trans. Softw. Eng.*, vol. 26, no. 2, pp. 150–177, Feb. 2000.

AWAIS QASIM received the B.S. degree in computer science from the Punjab University College of Information Technology, Lahore, Pakistan, in 2009, and the M.S. degree in computer science from the Lahore University of Management Sciences, Lahore, in 2011. He is currently pursuing the Ph.D. degree in computer science with the Computer Science Department, Government College University. He was a Software Engineer in Industry and developed a number of iPhone and Android applications. He joined the Computer Science Department, Government College University, in 2012 as a Lecturer. He has authored four research papers in peer-reviewed ISI indexed journals. His research interests include model checking, multi-agent systems, real-time systems, and self-adaptive systems.



SYED ASAD RAZA KAZMI received the M.S. degree in computer science from Sir Syed Engineering University, Karachi, Pakistan, and the Ph.D. degree in computer software and theory from the State Key Laboratory, Computer Science Institute of Software Chinese, Academy of Sciences, Beijing, China, in 2008. He is currently an Assistant Professor/In-charge with the Computer Science Department, Government College University, Lahore. He conducts research in the area of formal methods, specifically using techniques, such as LTL, CTL, and modal μ-calculus to specify the properties of reactive and concurrent systems for the verification of properties, such as safety, liveness, and deadlock. He was also involved in the area of formal modeling of real-time multi-agent systems, model checking, compositional reasoning, and fixed point theory, and has written over ten peer-reviewed ISI indexed journals.