Tokenit: Designing State-driven Embedded Systems Through Tokenized Transitions

Amir Taherkordi†‡, Christian Johansen†, Frank Eliassen†, and Kay Römer§

University of Oslo, Norway
{amirhost,cristi,frank}@ifi.uio.no

TU Graz, Austria
roemer@tugraz.at
Sonitor Technologies, Oslo, Norway
amir@sonitorn.com

†Department of Informatics
‡Institute for Technical Informatics
§R&D Department

Abstract—The development of resource-constrained embedded systems that are naturally state-driven is still a challenging issue, especially in industrial applications—developed on a bare-bone style runtime system with basic programming features. This is because of the complexity of state-driven design in embedded applications, such as parallel and complicated event-based activity flows, and complicated constraints for transitioning between program states. State machines are considered a systematic approach for such needs. However, existing approaches, in this area, either do not satisfactorily address the above complexity aspects, or force the developer to write code intermingling state handling logic with the functional code. To tackle these issues, we propose Tokenit, a state machine-based development framework for resource-constrained embedded systems. Using Tokenit, the programmer models the application as a set of parallel processes, where each process consists of sequenced activities with state constraints such as delayed transitions or interdependency between the states of parallel processes. Tokenit, then, processes the obtained model and associates a token to each sequential flow of activities, synthesizing them and executing state transitions according to the constraints expressed in the Tokenit model. The evaluation results show that Tokenit reduces significantly the complexity of state-driven programming in embedded systems at an acceptable memory cost and with no extra processing overhead.

I. INTRODUCTION

Programmers are being offered more and more mature programming models that fit better to the requirements of resource-constrained embedded systems [1], [2], [3], [4]. However, development of today’s embedded applications is still a challenging issue due to the sophisticated state-driven logic they introduce, such as in reliable data transport services [5] and WSN-based tracking applications [6], [7]. Moreover, a common practice in industry is to realize WSN applications with a flat software architecture in which network and application functions are implemented at the same level over a Hardware Abstraction Layer (HAL) [8], using native programming languages such as C. This gives full control over the different parts of the system from configuring and optimizing low-level radio system to implementing high-level application code. However, such applications often contain complex state-driven processes, making the programs over HAL complicated and difficult to implement, read and verify, as witnessed in, e.g., the real-time patient tracking application Sonitor Sense [9].

In addressing this challenge, a careful consideration should be given to the flow of control and operations with complex event-driven interactions and consequent state changes [10], [11]. State machines are natural for describing reactive processing and control-oriented embedded systems. However, a resource-efficient state machine design may need to intertwine the functional code with state handling code, making it difficult to trace, implement and maintain states and the associated constraints (e.g., transitions). Research efforts have been made to address this by introducing activities as a function of both the event and the program state, like in the Object State Model (OSM) [10], based on StateCharts and Esterel [12], [13].

While existing solutions address this concern with new state-based programming abstractions, they do not satisfactorily address some important complexity and design concerns.

First, they come with a high degree of complexity in state-driven programming, caused by, e.g., parallelism, and conditional and event-based transitions. In the case of parallelism, embedded applications typically involve parallel execution of chains of activities: from dealing with various network-level interaction logics (e.g., data dissemination) to processing repetitive activities and system events (e.g., timing and sensor events). Second, most state machine-based development approaches for WSNs have led to completely new programming abstractions [10] with additional memory, processing and learning overhead. This is caused by the need to close the semantic gap between the definition of state machines and event-driven programming models in WSNs [2], [1]. Third, existing work does not exploit the meta information that can be provided by state machines, e.g., to observe the behaviour of an activity or an activity flow and verify the system functionality.

To tackle these concerns, we propose Tokenit, a generic modeling and programming framework to ease state-driven programming in resource-constrained embedded systems. The modeling formalism, in Tokenit, is based on finite state automata and incorporates timing notions and parallel execution from timed automata [14] and Harel’s StateCharts [12], respectively. Moreover, it introduces complementary notions and concepts (e.g., repetition and variable sharing) to meet the specific modeling requirements of embedded systems. Then, we enhance Tokenit with a novel programming approach to reduce the complexity of state-driven programming, based on the notion of tokens. Unlike in traditional token-based approaches, in this work, a token is an entity with a set of pre-defined attributes and behaviours, associated to an activity flow (i.e., a transition path). A token synthesizes the activities of a path and implements state semantics of the path such as conditions, variable sharing, and events (e.g., timer events). Moreover, associating one token to one path allows better handling of parallelism in concurrent activity flows.

Tokenit offers a model processor to generate token-based state handling code and embed it into the actual application code (i.e., activities). Our programming abstraction allows more flexibility in state-driven programming and avoids intermingling state handling code with the functional code. Tokenit features a runtime system to host and monitor activity flows. The latter refers to token instrumentation for observing or verifying activity flows, e.g., the timing overhead of a packet processing activity flow. The runtime system can be hosted by typical event-driven operating systems and allows many concurrent activities to be serviced on a single stack. Our evaluation shows that Tokenit reduces efficiently the programming complexity and offers significantly reduced programming effort compared to existing approaches (about 40% less lines of code and 16% reduction of Cyclomatic complexity [15] for implementing the Deluge protocol), with virtually no processing overhead and acceptable memory cost (e.g., 11% for implementing Deluge).
The rest of the paper is organized as follows. In Section II, we present two use case scenarios motivating our approach. Section III introduces the modeling principles of TOKENIT, while the token-based design approach is presented in Section IV. Implementation details and evaluation results are discussed in Sections V and VI, respectively. We present related work in Section VII and conclude in Section VIII.

II. MOTIVATING USE CASES

In this section, we study two use cases (application-level and service-level) to motivate the need for a state-driven approach addressing the discussed design concerns.

Application-level Use Case. Tracking applications require dealing with various states of sensor nodes (e.g., moving, stationary, and location data propagation) over time. Real-time patient tracking is one example in this domain which is often designed in a state-driven manner. The simplified example we describe in this section is taken from the considerable field experience acquired in [9]. This industrial application is described as a set of parallel processes over a HAL, where each process is labelled with a state and contains a set of sequenced activities with eventing constraints. Figure 1 depicts the main processes involved in the tracking scenario, including initialization, body temperature monitoring and reporting (P1), motion processing (P2), and listening to gateway units (P3). All these activity flows should be maintained in parallel and they may depend on the status of each other under some circumstances—a non-trivial programming task. For instance, P2 has to carefully schedule the sending, listening with different time delays ($d_2, d_3$) and resending activities ($d_4$) for energy saving reasons. At the same time, P3 may interfere with other processes to update configuration parameters such as frequency for reading temperature ($d_5$) or adjusting listening windows.

Service-level Use Case. A typical network service for WSNs is the dissemination of large data objects for different purposes such as updating sensor software. Deluge is perhaps the most popular protocol in this context—an epidemic protocol operating as a state machine where each node follows a set of local rules for quick and reliable dissemination of large data objects to many nodes over a multihop wireless network [16]. We focus on the state management aspect of Deluge in this paper, while its detailed description is available in [16]. Figure 2 demonstrates Deluge’s main activities and their relations in both the data sender node and the receiver. Both of these nodes share a set of activities under the so-called Maintenance state to propagate the data object’s summary and profile after a time interval of $d_{11}$, computed dynamically. In the sender, a parallel activity, called Transmit, receives data packet requests and transmits (after $d_7$ ms) all requested packets for a given page every $d_{7}$ ms. On the receiver side, the protocol should process the meta data (profile and summary) and data packets simultaneously, then proceed to sending page requests to the sender after a delay computed in each round ($d_3$, $d_4$, or $d_5$). The SendRequest activity should also be repeated up to a threshold every $d_6$ ms.

Common Design Challenges. Modeling the above scenarios needs a design approach that can abstract parallel activity flows, scheduling, delays, order of activities, conditional transitions, and events—a non-trivial design problem. For example, controlling and listening in the first scenario include complex parallel processing of different activities and parametric delays for repeating an activity, respectively. Similarly, in the Deluge protocol, some activity transitions may take place in the middle of a running activity based on a condition (e.g., Transmit and SendRequest). Moreover, as seen in both use cases, delay on transitions is a central design element. This raises the need for a principled modeling and programming approach that can mitigate the complexity of developing such systems, while avoiding additional programming effort and high resource overhead. However, to the best of our knowledge, no existing state machine-based development approach has addressed these issues for applications on resource-constrained systems so far.

III. MODELING SCOPE: CONCEPTS AND EXTENSIONS

In this section, we describe the modeling scope of TOKENIT, including the relevant concepts adopted from state machines and the extensions proposed to meet the special requirements of embedded systems. The TOKENIT model is based on finite state automata [17], incorporating timing notions from timed automata [14], and parallel execution notions from alternating automata [18] and Harel’s StateCharts [12]. In TOKENIT, we use the term activity, in addition to state. An activity indicates the operation being performed within a state, e.g., the Listening activity in the patient tracking use case listens to incoming data traffic while the sensor node is in the Listening state.

A key aspect of the TOKENIT model is events. In this work, we categorize events as: non-timer events and timer events. The former refers to asynchronous events corresponding to specific actions leading to state changes in an asynchronous way, e.g., motion events. In order to detect state changes made by asynchronous events, non-timing events in TOKENIT appear only on origin transitions into start activities, e.g., in Figure 1, a motion event triggers the MotionDetection activity. Start activities have the same meaning as start states in automata, called origin activities in this paper. The second type of events appears on transitions to capture delays. Timer events are the predominant event type for many WSN applications—central to the design of TOKENIT. The intuition for transition delays is that the transition must be taken after the specified delay, e.g., the transition $A_1 \rightarrow A_2$ occurs automatically after timeout $t$, triggering the execution of activity $A_2$.

Execution of an activity may need to retain a set of activity variables, with either local or global scopes. The former includes all values returned by an activity for the use in the next activity, while the latter is the set of variables globally accessible by all activities. A transition can be executed only if the condition is satisfied. Conditions can be seen as simple boolean combinations of tests on variables, or more complex expressions. The termination of an activity may trigger several new activities (maybe in parallel). For parallel executions, we adopt the notions from alternating automata called conjunctive transitions, i.e., a transition may have multiple target activities. However, each target activity may have different local variables, conditions, and delays. There is no non-deterministic choice in TOKENIT, however parallel transitions and conditions allow modeling conditional choices.
Definition 3.1: A Tokenit model \((A, V, C, \delta, \Lambda_0)\) consists of: \(A\), a finite set of activities, \(V\), a finite set of variables, \(C\), a finite set of conditions, \(\delta: A \rightarrow \mathcal{P}(\mathcal{C} \times D^* \times \mathcal{P}(V) \times A)\), the transition function, \(D\) denotes duration\(^2\), and \(\Lambda_0 \subseteq A\) the origin activities.

Transitions can be specified using the notation:
\[
A_{\text{control}} = \{[\text{Temp.}, \text{Sensor System Software}] \rightarrow A_{\text{readTemp}}\}
\]
where any of the conditions, delays, or set of variables can have various default values like \(\text{true}, 0\), and \(\emptyset\), respectively.

In the following, we do not display them on transitions to avoid cluttering the model. The transition \(\text{HandleRequest} \rightarrow (d_7, \text{Transmit})\) includes a delay, specifying that we need to wait for \(d_7\) ms before starting \(\text{Transmit}\). In the above transition, the condition is \(\text{true}\) and there are no local variables.

Transitions represent a high-level execution flow of major software operations (i.e., activities). Parallel transitions enable parallel execution of activities when the function \(\delta\) returns more than one activity with satisfied conditions. In Figure 2, the transition \(\text{Maintain} \rightarrow \{d_2, \text{AdvSummary}\}\), \((d_2, \text{SendProfile})\) denotes that if the execution of \(\text{Maintain}\) terminates, two new activities should be activated simultaneously, but after a delay of \(d_2\). As an example of conditional transitions, in the Deluge protocol, advertising the summary is performed only if fewer than \(k\) advertisements have been received. Therefore, we need to identify the condition \(\text{adv} < k\) associated to the activation of \(\text{AdvSummary}\) which will test the variable \(\text{adv}\).

Moreover, the timing feature of Tokenit introduces the concept of parametric repetition. This allows the developer to abstract operations performed repetitively and maybe with different delay times in each stage of execution, e.g., the sensor node may make three consecutive attempts with different time windows to listen to an incoming data packet. For this, a transition can be labeled with a sequence of executions \(T = [d_1, d_2, \ldots, d_n]\), where \(d_i\) denotes the waiting time for the execution of the activity, after the execution at time \(d_{i-1}\). In Figure 1, we have \(A_{\text{listen}} \rightarrow \{d_2, d_3\}, A_{\text{listen}}\), so that we allow two listening windows of different time intervals.

IV. TOKENIT DESIGN

The main goal of Tokenit is to provide a design and programming framework that reduces the complexity of developing state-driven embedded applications in a resource-efficient manner. In particular, the design of Tokenit is aimed to address the specific concerns presented earlier in this paper. Moreover, the primary challenges for state machine-based design of tiny devices should be taken into account. These include the increased complexity of programs with nested conditions, the lack of global visibility of states and transitions in the code, and violating the formal semantics of state definition in the code for the sake of resource efficiency or the event-driven nature of programming models. Therefore, as a state-driven model for such systems, Tokenit should have acceptable resource overhead, adhere to the primitives of state-based programming, and avoid substantial restructuring of the programming abstraction when states come into the scenario.

A. Overview

A sensor application, in the context of this work, consists of a set of activities implemented by the programmer. Activities in our model intuitively refer to units of functionality, therefore not only including the state of the program, but also specifying an activity being performed in that state.

Figure 3 illustrates an overview of our approach. The Tokenit model of the target application is described by the programmer, including description of activity flows, transitions between activities, and constraints such as delays. The Tokenit Model Processor first processes the model description, then generates and embeds so-called housekeeping code (HOC)\(^3\) into each individual activity. This code basically links the activities of a flow based on the constraints defined by the programmer for each transition, e.g., a delayed or a conditional transition (An example of generated HOC is given in Subsection V-B). This is achieved by allocating a token object to each transition path, e.g., Tokenit to \(A_1 \rightarrow A_2 \rightarrow A_4 \rightarrow A_4\). This object features a set of behaviours and properties to allow token-based state transitions in collaboration with HOC. In particular, HOC connects activities based on the transition constraints, while tokens are used to execute transition paths and respective constraints. Activities, including the generated HOC and tokens, reside on Tokenit Runtime, which is a container for model initialization, activity scheduling, state handling and monitoring activity flows. In the rest of this section, we discuss these concepts in detail.

B. Activity Flows Analysis

The key design element in our approach is to extract all parallel activity flows (i.e., transition paths) of the target application (e.g., \(P_1, P_2\) and \(P_3\) in the patient tracking application) and associate a token to each one. This means that there is a one-to-one association between activity flows and generated tokens. Therefore, we first need to discuss how the activity flows are defined and extracted in Tokenit.

1) Transition Paths: Initially, we need to know all possible transition paths in the application as described by the model. A transition path can be initiated from two different \(\text{origin}\) activities: i) created by the application, e.g., main-like functions, and ii) triggered by the system, e.g., interrupt-like functions. For example, in the patient tracking use case, \(P_1\) falls in the first category, while \(P_2\) and \(P_3\) are triggered by a system event. The Tokenit model proposes a directed call graph which is traversable by Model Processor and allows it to find all possible paths from the origin activities. Figure 4 depicts an abstract call graph with two origin activities. Based on this, the following transition paths can be extracted:

\[
\begin{align*}
\text{Path1: } & A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_6 \rightarrow A_2 \\
\text{Path2: } & A_1 \rightarrow A_3 \rightarrow A_7 \rightarrow A_3 \\
\text{Path3: } & A_1 \rightarrow A_4 (\rightarrow A_4) \ast \\
\text{Path4: } & A_4 \rightarrow A_{10} \\
\end{align*}
\]

Fig. 3: Overview of Tokenit-based design and development

Fig. 4: Discovering all possible transition paths in a call graph

In the context of this paper, transition paths are acquired in a stepwise manner based on the number of incoming (\(\lambda_i\)) and outgoing transitions (\(\lambda_o\)) in each activity. For instance, the execution of \(A_4\) will yield two possible transitions, while \(\lambda_i(A_4) = 1\), resulting in new transition path \(A_1 \rightarrow A_4\). This technique is a natural way for discovering activity flows, and more importantly it avoids overlapping paths or cyclic paths. For example, \(\text{Path2}\) is always defined as the one listed above, not \(A_1 \rightarrow A_4 \rightarrow A_4 \rightarrow A_1\). In the next subsection, we demonstrate how the Tokenit code generation framework finds the possible paths of a Tokenit model based on this technique and generates a token for each individual path.

\(^2\)We use the Kleene * notation to denote words over the time domain \(D\), i.e., sequences of time values.
2) Token Generation: Based on the concept of token, the key question is how are the tokens created and associated with the activities of transition paths? To reduce complexity and resource overhead, the ultimate goal is to create the minimal number of tokens maintaining the states of running activities. Intuitively, a transition path is the largest part of a model that can share a token. Therefore, the number of transition paths is an indication of the number of required tokens.

Algorithm 1 shows how Model Processor generates tokens from the directed graph description of the target TOKENIT model. The main idea behind the algorithm is to traverse the graph using the Depth First Search (DFS) algorithm and generate new tokens based on the value of \( \lambda_i \) and \( \lambda_o \) in each node of the graph. For each origin node in \( OV \), a new token is generated, e.g., \( A_1 \) and \( A_3 \) in Figure 4. If, in a node, the number of outgoing transitions is greater than the number of incoming transitions (i.e., \( \lambda_o > \lambda_i \)), we generate \((\lambda_o - \lambda_i)\) new tokens (line 8). Otherwise, the current number of tokens is sufficient. As an example, in Figure 4 activity \( A_1 \) has one incoming transition and three outgoing transitions; two new tokens should therefore be generated. For the sample model in Figure 4, the algorithm generates five tokens, equal to the number of identified transition paths.

Algorithm 1 Generate required number of tokens

Input: \( G = (V,E) \) directed graph of the TOKENIT model

Output: \( T \) generated tokens set

1. \( OV \leftarrow \text{ORIGIN\_VERTEXES}(G) \)
2. \( T \leftarrow \emptyset \)
3. for each \( v \) in \( OV \) do
4. \( \text{ADDTOKEN}(v) \)
5. for each \( d_v \in \text{DFS-VISIT}(v) \) do
6. if \( \lambda_o(d_v) > \lambda_i(d_v) \) then
7. for \( i \leftarrow \lambda_i(d_v) \) to \( \lambda_o(d_v) \) do
8. \( \text{ADDTOKEN}(d_v) \)
9. end for
10. end if
11. end for
12. end for
13. procedure \( \text{ADDTOKEN}(\text{Vertex } v) \)
14. initialize new Token \( t \) with \( v \) as start vertex
15. \( T \leftarrow T \cup \{t\} \)
16. end procedure

3) Token Structure: The primary goal for introducing the concept of token is to consolidate the state handling details, including the static information (e.g., activity attributes) and the dynamic behaviour (e.g., activity functions). Figure 5 shows a high-level description of the token structure, including attributes and behaviours.

The basic idea is to exploit the attributes carried by a token and dynamically manipulate their values at runtime in order to traverse transition paths according to the state logic defined in the TOKENIT model. When an activity is performed successfully, the associated token will be updated with attribute values that allow the TOKENIT runtime system to proceed to the invocation of next activity identified in the model. Initialization and updating of the attributes of a token are carried out by the generated HOC. \( \text{delayToNextActivity} \) is the key attribute of a token, specifying the possible time delay before proceeding to the next activity. Additionally, the running activity can provide some input data to the next activity in the transition path through \( \text{toNextActivity} \).

Tokens can also include dynamic behaviours that enhance the runtime system with better control of token-based execution flows. For instance, when we want to stop the execution process along a transition path (e.g., stopping radio data propagation when a packet is received), the \( \text{stop} \) function of the associated token should be called. Likewise, the execution of a token can be repeated in order to re-invoke the latest executed activity (e.g., \( A_3 \) in Figure 4). Finally, \( \text{moveToNext} \) is another type of token behaviour, allowing conditional transitions, discussed later in this section.

C. Tokenized Transitions

From the process execution viewpoint, the activities of a model can be concurrently active, either being processed or waiting to be processed. This section explores the design aspects of handling and scheduling transition paths by TOKENIT's runtime system and discusses how tokens are exploited to maintain the execution of activities along the transition paths.

As mentioned in Section III, transitions in TOKENIT can come in two forms: non-delayed and delayed transitions. Let us consider the former and assume that no delayed transition appears in the given model. The main data structure that keeps track of the current tokens is a circular queue of size \( N \), where \( N \) denotes the total number of tokens required to traverse the transition paths. The main steps taken by Immediate Activity Scheduler (IAS) are as follows:

1) initialize tokens \( \{t_0,t_1,t_2,...,t_N\} \) with default values;
2) initialize the waiting queue \( Q \) with \( t_0 \) at the head of queue;
3) dequeue \( t_i \) from \( Q \);
4) invoke the activity \( A_m \) assigned to \( t_i \);
5) enqueue all tokens \( \{t_k,t_j,...\} \) outgoing from \( A_m \);
6) store the value of local variables \( \{v_0,v_1,v_2,...,v_N\} \) in \( t_i \), which are required for the execution of the next activity along the transition path;
7) return to 3.

The main idea is to invoke, in each step, the activity assigned to a token and schedule the outgoing tokens for later execution based on some pre-defined temporal ordering strategy, e.g., FIFO. In step 5, the list of consequent tokens \( \{t_k,t_j,...\} \) after the execution of \( A_m \) is provided by HOC and added to the activity code (cf. Section V-B). Figure 6 presents part of a TOKENIT model and the first four steps of token-based activity invocation. \( A_1 \) is the main function of the application with token \( t_0 \) pointing to this activity. When \( A_1 \) is executed successfully, \( t_0 \) will be removed from the waiting queue and the child tokens \( \{t_0,t_1,t_2\} \) are enqueued for execution and so on.

![Fig. 5: Description of token data structure](image-url)

![Fig. 6: Token scheduling strategy for non-delayed transitions](image-url)
by the underlying operating system. In our implementation, we develop the transition handling runtime system over the Contiki operating system [20], thereby TOKENIT runtime exploits the ctimer module of Contiki.

1) Repetition and Termination: Repetition of a transition and termination of a transition path are two additional capabilities that should be supported as part of the behaviour of a token. Each token can include standard functions repeat and stop, whose implementations are token-specific and identify the actions that should be taken for repeating an activity or stopping a transition path. Whenever an activity has to be repeated (perhaps with some different parameters like delay time), the associated token can be called for repetition (by the generated HOC). Although repetition can be seen as a self-loop transition, we introduce this notion as it simplifies state handling by preserving the current state of a token and changing only transition-related attributes such as delay.

Figure 7 illustrates part of the TOKENIT model for the patient tracking application, reflecting both above concepts. The model includes two origins: the main application function (Conf&Init), and the second triggered whenever a movement event is detected or the node switches to the inactive mode (after movement). On the right hand side, another process is initiated for reporting the body temperature (if beyond a threshold) every $d_1$ seconds. As shown, we need tokens $t_0, t_1, t_2$ respectively to report temperature readings, repeat temperature checking, and handle the motion detection process and sending location-related data to the gateway. As a use case for transition termination, the process of packet sending and listening should be stopped whenever the motion sensor does not detect any movements (i.e., inactive mode). To do this, HOC should simply call the stop behaviour of $t_2$, which in turn stops the execution of the current activity assigned to $t_2$ (either Sending or Listening).

![Figure 7: Part of TOKENIT model for patient tracking application](image)

2) Conditional Transitions: One form of conditional transition appears inside activities as internal conditions, handled locally by the associated activity to decide, e.g., which transition path to choose based on some criteria. The more complicated case occurs when two different transition paths are dependent on each other, while they do not directly share a transition—cross-path conditions. As an example, in Figure 7, TempReporting can be allowed to send data, but it should first ensure that $t_2$ is not serving the Listening activity. Based on the token structure in Figure 5, this feature is provided through the currentStatus attribute and the moveToNext behaviour accordingly. On each transition, the runtime system calls the moveToNext function of the token and proceeds along the path if it returns true. Note that the body of this function can be either generated automatically by the model processor from the description of a condition in the model, or provided by the developer for more complicated cases, e.g., moveToNext for $t_0$ in the aforementioned application is (provided by the programmer):

```java
boolean moveToNext() {
    if (tokens[2].currentStatus == LISTENING)
        return false;
    else return true;
}
```

In general, supporting conditional transitions is rather beneficial when two or more transition paths are competing to access or manipulate a shared resource such as the radio. In addition, the initiation or continuation of a path might depend on the completion of another path, e.g., when a network configuration packet is received by the node, other processes may require the new configuration to be applied and then resuming their execution.

D. Discussion

The token-driven approach of TOKENIT may raise issues with respect to activity scheduling and execution which we investigate and discuss in the following.

Event abstraction by TOKENIT. Even though this concern is primarily addressed by the operating system, we need to clarify how TOKENIT’s abstractions are linked to events. As discussed in Section 3, TOKENIT distinguishes timer events from non-timer events. The latter event type is modeled through origin activities, including both user-defined and system-level events. In many applications of WSNs, timer events play the main role for correlating and ordering between the activities of a process [21], thereby we can mitigate the complexity of defining other event types on inner transitions of the model by introducing them through transitions to origin activities.

Token overriding due to system events. When a transition path is initiated by a system event (e.g., network packet), it is likely that, in the middle of processing a path, a new event arrives and requests for re-initiation of that path and the token. For example, in Deluge we may face this situation when a data packet arrives while the previous one is being processed. Such cases are considered as conditional transitions, in which we need to check the status of the associated token and take the appropriate actions based on the logic of the target application, e.g., terminating the running transition path or discarding the new events. Otherwise, to allow multiple tokens for one path, the programmer has control over how many parallel executions of a path are possible by defining the number of tokens available for each path (in addition to the minimal number of tokens computed by Algorithm 1).

V. IMPLEMENTATION

The implementation of TOKENIT involves two complementary components: 1) the TOKENIT-based modeling of applications and generating the code for activity and transition management, and 2) the runtime system hosting activities and handling the transitions among activities. Prior to discussing the implementation, we clarify the scope of the aforementioned components in the context of this paper.

Modeling and code generation. A TOKENIT model is described in XML along with well-defined semantics for activities, transitions and constraints like delays between transitions. The model processing and code generation component follows the so-called housekeeping code (HOC) generation model [19]. In this model, the programmer must explicitly provide all the application-specific code such as the code of activities. The role of HOC is to glue the various activities and constraints together to ensure proper execution of a TOKENIT model, e.g., creating required tokens and handling transitions and delays. We adopt this approach for the following reasons. First, based on our observations in sensor applications, the state transition decisions are not necessarily made at the end of the execution of an activity, rather it may transit to other activities in the middle of the code based on a condition or input data values. Second, transitions are often parametrized with respect to delay time or activity variables ($\nu$), computed and changed dynamically at runtime.

Runtime system and target platform. The main role of the runtime system is to provide an execution container for activities by invoking them, handling transitions among them, and taking care of constraints such as timing, conditional transitions, etc. All these are achieved with the help of the token management part of the container. Obviously, the implemen-
tation of the container is OS-specific, however the modeling concept itself and the token-based technique are generic. Our target for implementing the container is the Contiki OS [20]. The reason for this choice is that the process management system and event handling model of Contiki [1] allow better observation on the behaviour of the container and evaluation of its overhead and efficiency.

Overview of implementation. Figure 8 illustrates the overall scope of the implementation, where, on the left side, the programmer needs to develop only activities and prepare the XML description of the target TOKENIT model. Model Processor will parse the model description and the input source code. Then, it generates HOC, including global code (e.g., token creation) and local code (e.g., specific to an activity). The former is added to the global scope of the input source code, while the latter is added to the body of the respective activity. Next, the programmer needs to review the local code, places it in the appropriate location in the activity code, and makes the small required modifications (e.g., time for delayed transitions) to weave state-related code into the functional code. Finally, the code is ready for deployment on TOKENIT Runtime.

![Fig. 8: Overview of TOKENIT-based development](image)

In the rest of this section, we further explore the implementation components. We first revisit Deluge, then we show how the different components of the implementation are exploited to model and implement this use case.

A. Deluge: A Case for TOKENIT

As mentioned, the Deluge protocol operates as a state machine where each sensor node maintains a local state machine to disseminate large data objects to many nodes. We consider Deluge’s design from a different viewpoint: as a set of well-defined and coarse-grained activities that communicate with each other in order to fulfill the main goal of this protocol.

Figure 9 depicts the activities of Deluge, transitions, and delays on the transitions. There are two origins in the model: Deluge-origin initiated by the deluge protocol itself and OS-origin triggered whenever a Deluge data packet is received by the radio system. Within the Maintain activity, the protocol continuously (every \(d_1\) seconds) checks inconsistency among neighbouring nodes and also transits to the SendProfile and AdvertiseSummary activities after \(d_2\) seconds calculated by the same activity. ActivityDispatch listens to different Deluge commands sent by other nodes and routes the received packet to the appropriate activity. For example, if a profile packet is received, HandleProfile should first allocate and initialize the memory required for storing the object data file, then transits to SendRequest after \(d_3\) ms for receiving the data object’s pages.

It should be noted that the illustrated model for Deluge reflects only the activities and the associated transitions. Other concerns such as conditions on transitions are not shown in the figure. We clarify some of them later in this section when exemplifying the implementation details.

B. From Model to Code

The XML description of TOKENIT is inspired from SCXML specifications [22] and includes the following data elements: activity definitions, transitions along with timing constraints, and conditions. The following figure presents an excerpt of the XML description of the TOKENIT model for Deluge. Maintain is defined as an initial activity, containing three transitions as depicted in Figure 9. The variable tag shows the activity variables that should be carried by the transition to the target activity, e.g., for transitioning from Maintain to SendProfile, delugeObject should be transmitted as well. To identify an activity as an origin activity, the initial attribute of the activity should be set to true like ActivityDispatch. The value of duration for delayed transitions can be either a constant value, a parameter, or an expression, for example \(d_2\) can be replaced with \(\text{rand}\_\text{CLOCK\_SECOND}\), where both variables are already defined in the programmer’s code for Deluge. TOKENIT relies on compile-time type checking to perform matching and error-checking between the variable names as strings in the XML model and the generated C code.

![Fig. 9: The TOKENIT model of the Deluge protocol](image)

![Fig. 10: An excerpt of TOKENIT model description for Deluge](image)

In the next step, these artefacts (the model description document and the input source code) are given to Model Processor implemented as a Java tool. It first analyses the XML description of the given model against design issues such as orphan nodes (never accessible by any origins in the graph), and duplicated transitions. Then, it creates the directed disconnected graph (DDG) of the model using the Depth First Search (DFS) algorithm. The first visited vertex is the activity defined as initial in the main tag of the XML file. The other initial activities (e.g., ActivityDispatch) are also visited later in order to scan all nodes and create the target model’s DDG. The resulting graph serves as a basis for token management and code generation.

Token generation. The token generation algorithm starts by
scanning DDG, vertex by vertex. For each current vertex, if the number of outgoing edges is greater than the number of incoming edges, we add a new token to the current set of tokens and consider the current vertex as the initiator of the generated token. At the end of the token generation phase, the number of tokens to be generated, as well as their initiator activities, are determined.

**Code generation.** Model Processor is in charge of generating the state-related code and synthesizing it with the input source code. The scope of the generated code is either global or local. The former is the code added to the global scope of the input code, creating tokens, initializing them, and including the supplementary functions and entities such as timers for tokens (cf. Figure 11.a) and global variables. The local code is concerned with the activity-specific code such as local activity variables, transitions and the associated tokens, appended to the end of activity code block (cf. Figure 11.b). Local variables are added to the local scope of the activity code as static variables (rather than on the stack), enabling tokens to make local variables accessible to the next activity of a path. The key statement, generated as local code, is `transit(tokenId, &tokenQueue, delayTime, delugeObject, ...),` which handles transitions. The local code is generated using DDG of activities which contains the detailed information about tokens, their initiator activities, and transitions (i.e., edges of DDG). To locate activities and append the corresponding local code, Model Processor parses the input code and finds the definition of activities (defined in the model) within the input source code.

To this end, we used the ANTLR parser generator—a popular tool for parsing the standard ANSI C source code [23].

```
#define defineTokenSize

struct token { 
    int tokenSize[2]; 
    uint32_t tokenQueue[tokenSize];
    static struct ctimer state_timer0;
    static struct ctimer state_timer1;
};

void initActivities() {
    for(i=0; i<tokenSize; ++i) {
        tokenQueue[i]=1;
        tokens[i].delay=0;
        ...
    }
    tokens[0].myTimer=state_timer0;
    ...
}

static void HandleProfile(struct deluge_msg_profile *msg); 
...
/body of HandleProfile activity 
/* Tokenit local code */
transit(2, &SendRequest, delayTime, delugeObject);
```

Fig. 11: An excerpt of (a) global and (b) local code generated for Deluge.

**Code modification and completion.** Having the generated code, the programmer should review the housekeeping code and ensure that everything is in place in accordance with the model description. Importantly, conditional transitions should be completed by the programmer. For this case, we highlight a concrete example from the Deluge protocol. Part of the logic in data dissemination is that whenever the system is performing any of the following transitions, it should ensure that the other transitions in this set are not in progress.

\[
\text{HandleProfile} \rightarrow \text{SendRequest}, tokenId = 2 \\
\text{HandleSummary} \rightarrow \text{SendRequest}, tokenId = 3 \\
\text{HandlePacket} \rightarrow \text{SendRequest}, tokenId = 4 \\
\text{SendRequest} \rightarrow \text{SendRequest}, tokenId = 2V3V4
\]

To this end, the programmer needs to develop a common moveToNext234 behaviour for all of them which evaluates the status of tokens associated to them:

```c
void moveToNext234() {
    if((tokens[2].state!=TOKEN_WAITING_PROCESS) &&
       (tokens[3].state!=TOKEN_WAITING_PROCESS) &&
       (tokens[4].state!=TOKEN_WAITING_PROCESS))
       return true;
}
```

In this way, when any of these transitions is scheduled for execution, moveToNext234 should be evaluated first.

**C. Runtime System**

The main component of the runtime system is the transition scheduler which handles both immediate and delayed transitions. According to the design choice presented in Section IV-C, the scheduler maintains two separate waiting queues in IAS and DAS modules. The main scheduler function is implemented as a Contiki `prootentry`, which is periodically polled by the Contiki runtime system in order to invoke the activities in the waiting queues. Whereas the runtime system possesses its own queuing system for IAS, the Contiki’s timer libraries (i.e., ctimer module for invoking scheduled tasks) are utilized for implementing DAS.

Token management is the other component of the runtime system. As indicated in the sample code of Figure 11.a, the memory allocation model to tokens is static and all required tokens are available in the memory before initialization. Once Contiki boots, this component initializes the tokens with default attribute values and behaviour functions. During the application execution, its main responsibility is to deal with the behavioural aspects of tokens such as starting and stopping, as well as to evaluate conditional transitions. Concerning the latter, the runtime system, on each transition, invokes the moveToNext function of the current token and proceeds based on the invocation result.

Additionally, it is worthwhile to highlight the other aspect of the runtime system that introduces context for a running activity. During the execution of the activity, it is very likely that, e.g., the id or status of its current token is needed. Such information about tokens (i.e., available during the execution of an activity) is referred to as context. The runtime system provides this through `getCurContext()`, which returns a struct containing the relevant context elements such as current token id. For example, in the `SendRequest` activity of Deluge, the id of the current token is required in order to be able to repeat the execution of this activity. This is achieved by the following code (generated and appended to the body of `SendRequest`) referring to the context service of the runtime system:

```
curTokenId = getCurContext()->tokenId;
(tokens[curTokenId].repeatToken) (curTokenId);
```

**D. Discussion**

As discussed earlier in this section, the generated HOC might need additional modifications by the programmer for, e.g., fixing variable time values on transitions. This seems to be contrary to the code generation principle of Model-Driven Development (MDD) approaches in which the generated code is generally not altered.

Indeed, the aim of MDD is to have a compiler generating the implementation code automatically and fully from a model description. However, in resource-limited platforms, it would not be easy to generate all required code automatically from the model because of the tight and ad-hoc couplings between different functions of the system, making developers reluctant to adopt MDD approaches. In TOKENIT, we adopt an intermediate solution, i.e., a modeling and programming framework which adds a negligible resource overhead and allows the developer to modify the code and provide complementary information for the specific parts of the application that TOKENIT does not handle. In particular, the main required modifications are concerned with parametric and varying state constraints, which are not known at design time. As part of our future work, we aim to address this issue and investigate how to minimize the input required from the developer.
VI. EVALUATION

As a development solution for resource-constrained systems, the main evaluation concern is to ensure that the overhead of the programming constructs and the TOKENIT runtime system is acceptable in terms of resource usage. The other goal is to investigate the reduced programming effort, as well as the potential features of TOKENIT with respect to the meta-information it provides on tokens and transition paths, such as lifetime of a path. This can be useful when evaluating end-to-end performance of different activities of an application.

As mentioned before, we adopt Contiki as our OS platform to assess the TOKENIT model. Contiki is being increasingly used in both academia and industrial applications [24] in a wide range of embedded systems. Our hardware platform is the TelosB mote equipped with a 16-bit TI MSP430 MCU with 48KB ROM and 10KB RAM. Moreover, to further evaluate the above performance metrics, we focus on the TOKENIT-based implementation of the Deluge middleware and compare its performance with the Contiki-based implementation.

A. Memory Footprint

High memory overhead is often the main reason behind avoiding new programming abstractions in resource-limited systems. TOKENIT’s design gives a particular attention to this issue. The model processing component of TOKENIT avoids dynamic memory allocation by knowing the memory requirements (e.g., number of tokens) at design time. Moreover, designing a lightweight queuing system for non-delayed transitions and leveraging the operating system’s facilities for delayed transitions can largely reduce the memory overhead.

The memory footprint of TOKENIT is categorized into minimum overhead and dynamic overhead. The former is paid once and for all, regardless of the amount of memory needed for the target application, while the latter depends on the number of transition paths and activities. Table I shows the minimum memory requirements of TOKENIT, which turns out to be reasonable with respect to both code and data memory. As Contiki consumes roughly 24 Kbytes (without uIP support) of both these memories, TOKENIT induces a low memory overhead.

The dynamic memory overhead is presented in Table II. Each individual transition path requires a new token which occupies 24 bytes of data memory space, besides additional byte for maintaining the activity scheduling queue (cf. Section IV-C). We believe that this overhead is acceptable since the total number of transition paths is limited for a typical application. For instance, Deluge as a medium size application (3.5 KB) includes six transition paths, thereby it needs six tokens. In a larger system, if ten tokens are generated on average, the additional required data memory will be only 250 bytes (10×25 bytes), resulting in 2.4% data memory overhead on the TelosB mote. The other dynamic overhead is concerned with application-specific behaviour code, which depends on the complexity of a given behaviour. For example, in Deluge, the implementation of the conditional behaviour `canMoveToken234()` requires 26 bytes of ROM. Finally, it should be noted that the transition-related invocations (both delayed and non-delayed ones) do not impose additional memory overhead since they are replaced with the normal function calls which take the same size of memory space. For example, a delayed transition call consumes 22 to 24 bytes of code memory, which is equal to calling the `ctimer_set` function of Contiki.

We have also measured the overall memory overhead of the TOKENIT-based implementation of Deluge, shown in Table III. The first row summarizes the memory footprint of Deluge when implemented based on Contiki’s programming model. The second row shows the cost of TOKENIT-based implementation of Deluge, where we add the minimum 460 bytes memory overhead of the TOKENIT runtime as part of the underlying operating system. Deluge, in this case, needs additional 416 bytes (3920-3504) of memory, resulting in 11% additional overhead which we believe is acceptable. The large portion of this overhead is due to creating tokens and associated timer modules.

B. Processing Overhead

There are two potential points of processing overhead that should be investigated: the scheduling mechanism of the runtime system, and the additional cost for the invocation of activities on transitions. The former is basically dominated by the queue handling the non-delayed activities (delayed activities rely on the operating system’s scheduling module). The latter is dynamic and determined based on the number of activity calls that occurred during the lifetime of the target application. Table IV reports the fine-grained processing cost of these operations in terms of CPU cycles. For instance, invocation of a delayed activity requires additional 63 cycles, while token-based processing of a transition and invoking the next activity consume 80 cycles. Considering the processing speed of typical low-power microcontrollers, we believe that these costs are acceptable and do not lead to significant prolongation of application execution time.

To further investigate the processing overhead on invocations, we return to the Deluge middleware and study the total propagation time for different object sizes when it is implemented based on the TOKENIT model. Figure 12 shows the results we obtained in comparison with the event-driven model of the implementation on Contiki. The size of data object files, in our experiment, is chosen based on experiments in [16], ranging between 128 bytes to 2048 bytes. Overall, the graph shows that TOKENIT does not impose any additional processing overhead and interestingly it outperforms the Contiki-based version for some object files. In particular, receiving a 512 bytes data file takes 24.12 seconds, while, in the original implementation, this data file is received and processed by the receiver node in 27.47 seconds. This result is probably caused by the token processing data file advertisement messages. In particular, when this token is processing a message, it discards other incoming advertisement messages upon arrival, while in the Contiki-based implementation this occurs a bit later by checking the associated timer.

C. Programming Effort

As discussed, TOKENIT comes with a modeling solution to help the programmer reduce the programming effort required to develop state-driven scenarios. This not only refers to
measuring the lines of code (LOC) and code complexity, but includes the qualitative evaluation of user’s experience in TOKENIT-based development. However, the latter typically needs extensive user study and empirical methods for assessment. In this paper, we therefore focus on two metrics for evaluating the programming effort: LOC and the complexity of programming based on Mc Cabe’s cyclomatic complexity method [15]. We report the evaluation results for TOKENIT-based implementation of three different applications.

The applications are chosen based on various complexity aspects of state transition models such as the number of parallel transition paths (i.e., the number of tokens) and the number of activities involved in each transition path (i.e., from short to long paths). In addition to Deluge with an average level of complexity, we select EnviroTrack [11] as an application with several parallel and short transition paths. EnviroTrack is a framework that supports tracking of mobile targets with a WSN. A sensor node, in this framework, can be in seven states. The major states include: free, follower, member, and leader. Base on the type of data message received by a sensor node from its neighbors (e.g., join, leave, recruit, etc.), the node will take some pre-defined actions and may transit to another state. For example, if a node is free, it will become a follower if it receives a recruit message. Considering each individual state as a token, TOKENIT will generate seven tokens with one activity for each to select the right actions and the next state.

In addition to these, we developed an exemplary Multi-Purpose Service framework (MPS) which contains five tokens and 12 activities to constantly process five parallel transition paths for reading sensor data (i.e. reading temperature and humidity), propagating heartbeat messages, and listening and processing network data. We chose and designed this service in order to find the highest efficiency achieved with respect to programming effort, while the first two use cases show our observations in real scenarios with different degrees of complexity.

Table V shows the results we obtained using TOKENIT to generate and develop state handling code for the above use cases. In the Deluge case, the programmer is required to develop the code for conditional transitions and initialization of tokens, while the rest will be generated by TOKENIT. This leads to a modest reduction in programming effort compared to the LOC of full Deluge (470 lines). In use cases such as Deluge, the main programming benefit is in simplifying the design, monitoring and verification of parallel states (cf. next subsection). In contrast, 28% (328/95) of EnviroTrack code is allocated to state management, and we observe a noticeable reduction in LOC for this application. Finally, TOKENIT yields a significantly reduced effort for MPS with long transition paths and the large number of LOC for state handling.

Finally, we focus on measuring the complexity of developing the above use cases with TOKENIT’s approach. To this end, we use the Mc Cabe’s cyclomatic complexity method [15] for measuring quantitatively the complexity. This technique measures all the linearly independent paths of a program’s source code and the number of branches in the execution flow by creating a control flow graph. The Cyclomatic Complexity Number (CCN) value typically ranges from 1 to 10 indicating low and high complexity, respectively. Table VI shows the result of analyzing the reference use cases in terms of CCN. As illustrated, TOKENIT reduces efficiently the development complexity in all of them, with a better result for EnviroTrack as it includes several state changes and transitions.

D. TOKENIT-based Service Instrumentation

The way we adopted to design TOKENIT and the one-to-one association between generated tokens and transition paths allow us to monitor and verify the paths. This can be achieved by instrumenting tokens with attributes that are relevant to monitoring or verification of the target service. Then, the TOKENIT runtime system will be able to provide real-time information about those attributes. To highlight this aspect of TOKENIT, we focus on the part of Deluge’s model in the receiver node that listens to network messages and proceeds towards one of the following transition paths: Handle Profile, Handle Summary, and Handle Packet. The goal is to instrument Deluge’s tokens with token creation and termination timestamps in order to find out if the processing time of each token is in accordance with the rules described in the protocol.

Figure 13 shows the results of monitoring the selected tokens for different data file sizes, e.g., the lifetime of Handle Profile is 41% of total time required to receive all pages of a 512 bytes data file. As the file size becomes larger, this token’s lifetime is reduced as well to 14%. This occurs because this token is activated only once for receiving an updated data profile from the sender and performing Send Request within the same round. Thus, for larger files, the lifetime of this token will be reduced accordingly. Our observations on further two tokens also confirm the rule defined for the maximum number of requests (λ parameter in rule R.2 of Deluge) sent from the receiver in each round. Specifically, both tokens 3 and 4 contribute sequentially in triggering the Send Request activity. Given that λ = 2 and each data page is 256 bytes, for receiving a 1024 bytes file, tokens 2 and 4 will call Send Request once, while token 3 will perform this twice.

![Fig. 13: Results of TOKENIT-based monitoring of main tokens in Deluge](image-url)
VII. RELATED WORK

From the modeling viewpoint, TOKENIT adopts concepts from state machine formalisms that are relevant for modeling resource-limited platforms, e.g., delayed transitions, parallelism and conjunctive transitions. In TOKENIT, these concepts are further extended with notions such as repetition and variable sharing between activities and conditions, all modeling requirements of such platforms. Beyond that, TOKENIT is specially focused on how to map a state machine, empowered by the above semantics, to efficient design choices and programming abstractions for embedded systems.

OSM [10] is perhaps the most relevant work both for modeling and development of state-oriented embedded systems. Kasten et al. introduced OSM to allow developers to describe their applications as OSM code, which is, at a later stage, compiled into native code through the OSM compiler. OSM is implemented on top of Esterel [13]—a synchronous language for the specification of reactive systems. The basic idea behind OSM is to extend the event paradigm with states and transitions, making actions a function of both the event and the program state. OSM borrows the concepts of hierarchical and parallel composition, and concurrent events from StateCharts and SyncCharts, respectively. It also introduces state attributes to share information among actions. State machines with Datapath (FSMD) [25] had earlier introduced similar attributes, reducing the number of states that should be declared explicitly, but attributes have global scopes (similar to TOKENIT). In OSM state attributes are local and bound to a state hierarchy.

Besides similarities between TOKENIT and OSM, TOKENIT proposes a more flexible modeling solution which allows sharing different types of variables among parallel activity flows, in-activity state transition and condition definition. The main differences of TOKENIT and OSM lie in the development and code generation mechanism, where OSM proposes a completely new programming model. TOKENIT is aimed at introducing minimal additional abstractions as we have witnessed several cases in which the programmer needs more flexibility in manipulating states and conditions. It should also be noted that OSM’s language introduces the same order of programming effort as TOKENIT does. For instance in the EnviroTrack application, the total LOC of OSM is 56 which is close to the 54 LOC of TOKENIT for this application (Table V).

Further work has been done to apply state machine formalisms to model sensor systems. In [26] and [27], techniques are proposed to optimize programming and formulate interaction between TinyOS components respectively, using StateCharts [12]. However, state diagrams in these approaches model the different states of a single object in the system and do not address timing constraints. A number of frameworks have been proposed to apply standard software modeling techniques to describe the logic of WSN applications. Some initiatives have been taken to employ behavioural UML diagrams, such as Activity diagram and UML StateCharts, for visualizing and implementing the software as a set of activities. In [28], UML Activity Diagrams are extended to introduce control structures in the execution flow of software deployed on sensor nodes. Glombitza et al. in [29] propose using state machines to orchestrate Web services and control flows on sensor nodes. However, the bridge between models and the detailed behavioural and state-based aspects of application logic still remains unsolved in the above approaches.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated TOKENIT, a novel modeling and programming approach for reducing the complexity of developing resource-limited embedded applications that are naturally state-driven. The key design element of TOKENIT is the notion of tokens, separating the state-related concerns from the actual application activities, synthesizing activities, and executing activity flows according to the constraints in the target TOKENIT model. This approach efficiently exploits existing event-driven programming models and reduces the complexity and programming effort for implementing the state handling code. We have shown the feasibility and performance of TOKENIT for a number of use cases in particular for Deluge—a protocol for reliable propagation of large data files over a multi-hop wireless network. Further consideration to TOKENIT-based verification of embedded systems is part of our future plan. In addition, we aim to apply this approach in other embedded platforms such as cyber-physical systems which include safety-critical control flows and complicated state transitions.

REFERENCES

[14] E. Yoneki and J. Bacon, “Unified semantics for event correlation over a multi-hop wireless network. Further consideration to TOKENIT-based verification of embedded systems is part of our future plan. In addition, we aim to apply this approach in other embedded platforms such as cyber-physical systems which include safety-critical control flows and complicated state transitions.