

SLIME: Structural Low-Level Implementation Model Extraction

Christian Hakert (TU Dortmund University, Germany), Mario Günzel (MPI-SWS, Germany),

Jian-Jia Chen (RWTH Aachen University, Germany),

Silviu S. Craciunas (NXP Semiconductors & Technical University of Denmark)

Abstract—We present SLIME, a framework to automatically extract abstract task models (dedicated for high-level timing analysis) from a given implementation (as a compiled binary file). We position SLIME as an automated mechanic approach, being employed in the process of integrating or implementing real-time applications in a real-time operating system (RTOS). SLIME consists of three components: 1) control flow graph (CFG) extraction and semantic enrichment 2) modeling of an abstract task machine from the CFG as an abstracted task model, and 3) runtime tracing of the implementation for model validation and time measurement on a target platform. To the best of our knowledge, SLIME is the first framework to extract the abstracted real-time task models from binary programs in the context of real-time operating systems. The components of SLIME are implemented and evaluated for real-time applications on FreeRTOS and RTEMS, two widely used RTOSes. We empirically evaluate SLIME and show that the abstracted task models of all tested applications are correctly extracted while the error rates for runtime tracing remain at most at $\approx 1\%$.

Index Terms—extraction of task models, reverse engineering, schedulability

I. INTRODUCTION

REAL-TIME tasks are commonly presented in an abstract task model with some specific parameters, such as periods and worst-case execution times (WCETs). However, obtaining the corresponding abstract model of an actual implementation of real-time applications is often non-trivial. Furthermore, in reality, the implementation of a real-time task is an engineering process, which may result in complex code, including branches or (busy) waiting, making the relation to the abstract model a difficult process. This mismatch results in discrepancy of abstracted real-time task models and actual implementations.

In the most extreme case, there is only an implementation available, without any specification of the abstracted task model. In a less extreme case, there is a task model and an implementation, but there is a structural mismatch between them. The mismatch, for instance, could be due to additional tasks in the implementation, that are not considered in the model, or due to self-suspending behavior of certain tasks, that is not properly modeled. Even if there is no structural mismatch between the abstracted model and the implementation, there can still be quantitative mismatches, e.g. an incorrect period of a task or unknown execution time of certain code blocks or an incorrect self-suspension interval of a task. In order to address such model mismatches, a *mechanized way of lifting a given implementation into an abstracted structural and quantitative real-time task model* is required.

Deriving the WCET of a real-time task from the machine code has been widely studied in the literature, c.f. the survey by Wilhelm et al. [1]. Bound-T [2] provides a toolchain to pursue detailed timing analysis of periodic and sporadic real-time tasks directly from a binary program. The framework by Chen et al. [3] takes the program's compiled binary and operates only at runtime, using dynamic trace-based feedback to detect slowdowns and progress in multicore systems. Beyond WCET derivation, construction of semantically enriched state transition models based on static program analysis has been addressed with different focuses. Dietrich et al. [4] develop an integrated approach for automated worst-case response time (WCRT) analysis across multiple threads of execution in real-time operating systems, by extending the classic implicit path enumeration technique (IPET) over the whole system. Dietrich et al. [5], [6] further incorporate kernel semantics into control flow analysis in order to create tailored operating system kernels. Kässens et al. [7] present an approach for static analysis of cross-domain interactions in hypervisor-based real-time systems, which enables cross-domain WCRT analysis.

In the scope of real-time model extraction from runtime observations, earlier results target at event tracing and monitoring, with a research focus on reducing the tracing overhead, e.g., [8]–[12]. Recently, the LiME framework developed by Brandenburg et al. [13] and the extension to intra-thread analysis [14] provides a flame to ignite the derivation of the required parameters of a task set implementation. LiME runs implemented applications on Linux and extracts an abstracted task model by observing runtime scheduling behavior. Berger et al. [15] realize event tracing in FreeRTOS in order to model system interactions and derive tight WCRT analysis. In the specific direction of real-time model extraction on compiled application binaries, Chen et al. [3] realize automatic extraction of an abstracted model from the actual application implementation as a compiled binary in order to realize progress checking. To the best of our knowledge, LiME is the only existing framework which automatically extracts a semantically enriched real-time model from an application provided as a compiled binary.

Our contributions: In this paper, we present **SLIME**, which automatically extracts abstracted task models, dedicated for high-level analysis, from a given implementation, as a compiled binary file. SLIME is dedicated for real-time applications, that are implemented in a real-time operating system (RTOS). *The input to SLIME is the compiled binary of an RTOS application, while the output is a parametrized abstracted task model, that can be directly fed into, e.g., schedu-*

ability tests. To the best of our knowledge, **SLIME** is the first approach to utilize binary reverse engineering to extract real-time task models. The components of SLIME are implemented and evaluated on FreeRTOS (<https://www.freertos.org/>) and RTEMS (<https://www.rtems.org/>), two widely used RTOSes, in this paper. We make the following technical contributions:

- **Use Cases:** As a proof of concept of the first model extractor for RTOS, SLIME has been developed with certain limitations and assumptions imposed on the application implementation, creating a trade-off between design flexibility and analyzability. We provide concrete use cases and assumptions in Section III and system model in Section IV, supported by limitations and future work in Section XI.
- **SLIME Implementation:** SLIME consists of multiple components in order to extract an abstracted real-time task model with tested schedulability from a provided implementation binary with an overview in Section V. The components are
 - 1) control flow graph (CFG) extraction and semantic enrichment in Section VI,
 - 2) modeling of an abstracted task machine (ATM) from the CFG as an abstracted task model in Section VII, and
 - 3) runtime tracing of the implementation for model validation and time measurement on a target platform in Section VIII, which can serve as a basis for estimating the measurement-based WCET.
- **Task Set Analysis:** We equip SLIME with exporting functionality of the extracted task set model for widely used task set analyses. As a proof of concept, we showcase the application of schedulability tests, especially including the analysis of self-suspension task behavior in Section VII-E.
- **Evaluation:** We empirically evaluate SLIME and demonstrate the resulting model accuracy in Section IX. All tested applications lead to a correctly extracted model while the error rates for runtime tracing remain at most at $\approx 1\%$.

II. S.O.T.A. MODEL EXTRACTION

In this section, we provide an illustrative demo application and apply state-of-the-art model extractors, including LiME [13], LiME-IT [14], and Bound-T [2], to this application. Specifically, we compare those approaches and SLIME in terms of required inputs, derived outputs and practical applicability throughout this section. We further provide a preview of the capabilities of SLIME for the given application.

A. Demo Application

We construct a simple real-time demo application, consisting of three tasks, namely `sensor`, `processor` and `actuator`, in order to make the comparison easy to understand. The (lowest-priority) sensor task reads data from a sensor and uses a shared queue to send data samples to the processor task. Reading the sensor requires triggering the sensor, suspending the sensor task and then reading the actual value. The (medium-priority) processor task waits for a data sample to arrive in the shared queue, performs arithmetic on the sample, and writes the result to a global shared (memory) object. The (highest-priority) actuator task reads the shared

object asynchronously. When the value of the shared object has changed since the last job of the actuator task, the task writes the data to the actuator. This requires writing the data, suspending and committing the data in the actuator afterwards.

The periods of all the tasks are set to 200ms. The sensor task suspends for 10ms. Also, the actuator task suspends for 20ms, in case it writes the actuator value. We abstract the hardware interactions in the demo application with simulated functions.

B. Application Implementations

We implement the reference version of the demo application in FreeRTOS. Specifically, a periodic task in FreeRTOS is implemented by calling a function, implemented with an infinite loop and a `vTaskDelayUntil` statement. Self-suspension is realized with the `vTaskDelay` function. The reference implementation, which directly models an input to SLIME, is implemented for ESP32-C6 from Espressif.

In order to allow Bound-T [2] to analyze the demo application, several modifications need to be performed on the reference implementation. To the best of our knowledge, Bound-T has not been further developed since 2010. Since Bound-T is not specifically designed to be aware of the used RTOS, some modifications and assertions of the application and RTOS are needed to make Bound-T able to return its analysis. We omit those details in the paper but will provide comprehensive discussions in the artifact evaluation.

For LiME [13] and LiME-IT [14], we need to convert the demo application into a Linux application, since both tools require Linux event tracing mechanisms to work. For LiME, the three tasks are implemented as pthreads. Periodic delay, as well as self-suspension, is realized with the `clock_nanosleep` function. For the shared queue, we use a POSIX message queue. For LiME-IT, we only create one pthread, since all tasks share the same period. We let this thread call each task body as a subsequent function call and implement periodicity with `clock_nanosleep`. Since the execution of the sensor and the processor task is guaranteed to be sequentiality for the LiME-IT implementation, we replace the message queue by a simple shared memory object.

C. Input Specifications

For LiME and LiME-IT, the input is provided by the compiled Linux version of the demo application. A 60-second event trace is collected as the input for the model extraction of LiME (as well as one trace for LiME-IT).

Bound-T is not designed to analyze an RTOS and gets stuck in kernel internals during analysis. To ensure that Bound-T can deliver results to analyze our demo application, we have to provide application- and OS-specific assertions as part of the input. Beyond that, Bound-T requires a specification of the analyzed task-set and shared resources. Figure 1a illustrates an excerpt of the relevant lines of this specification.

D. Outputs

Figure 1b illustrates the corresponding excerpt of the output of Bound-T. It can be seen, that Bound-T does detect the

```

protected sensor_data_queue
type resource
entry queue_send_resource
entry queue_receive_resource
end sensor_data_queue

thread task_sensor
type cyclic
root sensor_activation_body
end task_sensor
[...]

```

```

protected sensor_data_queue
type resource
entry queue_send_resource
wcet 10023 , 3 , 2
entry queue_receive_resource
wcet 10020 , 4 , 2
end sensor_data_queue

thread task_sensor
type cyclic
[...]

```

(a) Bound-T Task and Protected Object Specification

(b) Bound-T Task Result

Fig. 1: Specification and Output of Bound-T

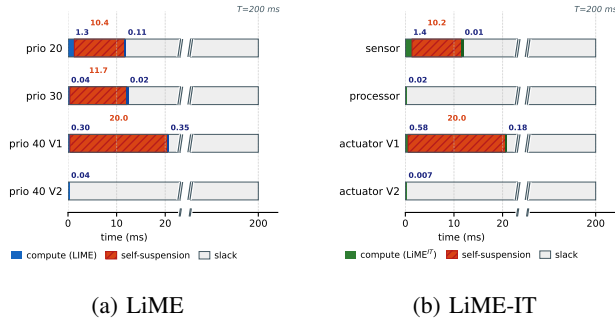


Fig. 2: LiME and LiME-IT Result Illustration

interaction with the shared queue in the sensor task and analyzes the WCET before and after this operation separately. Self-suspension is also not detected by Bound-T.

Figure 2 shows the result of a graphical illustration (implemented by us for this comparison) of the extracted model of LiME and LiME-IT. Each bar shows one detected task. The colors inside the bars show the segments of execution, suspension and slack. The width of the bars shows the detected period. With both LiME and LiME-IT, the information of self-suspension intervals can be extracted.

E. Capabilities of SLIME

SLIME extracts a full task-set model without any prior specification, like LiME and LiME-IT, but on the compiled application binary, like Bound-T. Hence, SLIME extracts the implemented task-set model and does not face the risk of identifying based on event traces. Furthermore, SLIME supports extracting RTOS semantics and therefore identifies resource allocation and self-suspension from the implementation.

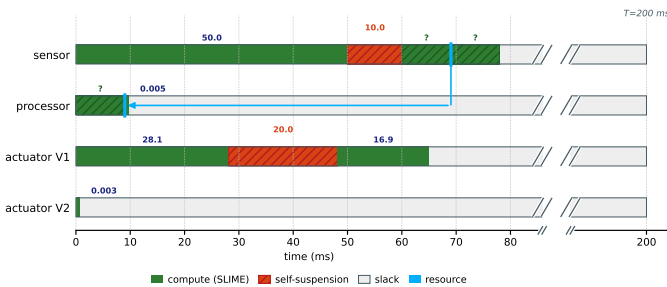


Fig. 3: SLIME Result Illustration

	Bound-T	LiME	LiME-IT	SLIME
Task-Set Detection	✗	(✓)	(✓)	✓
Binary Lifting	(✓)	✗	✗	✓
Self-Suspension	✓	✓	✓	✓
Synchronization	✓	✗	✗	✓
RTOS Applications	(✓)	✗	✗	✓
Runtime Information	✗	✓	✓	✓
High-Level Result	✗	✓	✓	✓
Sporadic Tasks	✗	✓	✓	(✓)

TABLE I: Comparison of Model Extraction Tools¹

For comparability, we render the result of SLIME on the demo application in a similar output as the result of LiME and LiME-IT in Figure 3. As SLIME directly extracts the model of the task set from the binary code of the demo application, it does not have any information about the execution times of the tasks. To overcome this, either a WCET analyzer has to be applied together with SLIME or fine-grained time measurements of the task set should be applied. Here, in Figure 3 we present the results from time measurements. We denote such parts of the model in green and strike through.

We note that the reported results in Fig. 2 for LiME and LiME-IT are on Linux on a Desktop PC and the results in Fig. 3 are from FreeRTOS on ESP32-C6. Therefore, their execution times are different.

F. Summary of State of the Art and SLIME

We provide a summary in Table I. All methods are capable of producing some forms of high-level representation, that can be used for theoretical schedulability analysis.

Bound-T does not detect implemented tasks, but rather needs them to be declared in a specification file. Due to the discontinued maintenance, Bound-T has limited supports regarding automated binary lifting and operating on RTOS applications as input.

LiME [13] and LiME-IT [14] can detect tasks sets automatically via tracing. In our demo application, the detected self-suspension interval length is larger than that reported by SLIME. Furthermore, LiME and LiME-IT are not designed to detect resource sharing via synchronizations (e.g., mutex locks). As static WCET analysis is out of the scope of SLIME, LiME, and LiME-IT, they all utilize the runtime trace for analyzing WCETs. But this can be integrated with existing WCET tools. Although LiME, LiME-IT, and SLIME all strive for automatic extraction of task parameters from a given implementation, LiME/LiME-IT and SLIME operate on entirely different input scenarios and apply different methodologies.

- The input to LiME and LiME-IT is a scheduler trace from a Linux application, whilst the input to SLIME is the binary implementation of real-time applications and the system calls from a real-time operating system.
- LiME and LiME-IT extract the task set model by continuously refining a model to explain the observed system events, which might not fully reveal the implemented

¹We put task-set detection for LiME and LiME-IT in braces, since the task set is inferred on a runtime trace and may not match the implementation. We put binary lifting and RTOS support for Bound-T in braces, since Bound-T is discontinued and does not support modern CPU architectures or RTOSes. We put sporadic tasks for SLIME in braces, since the existence of a sporadic task is detected in the binary, but can only be well parametrized with runtime information.

task set model due to incomplete coverage. SLIME directly extracts the implemented task set model from the application binary and enables the possibility to enrich the extracted model with time measurements of the implementation.

We note that LiME/LiME-IT and SLIME are orthogonal approaches to distinct scenarios.

III. USER STORY AND SYSTEM MODEL

We position SLIME as an automated mechanic approach, being employed in the process of integrating or implementing real-time applications in a real-time operating system. SLIME assumes a compiled binary image of the application as its input, which includes debug information in order to identify function calls by their name², and the specification of the underlying RTOS. Although there have been research activities to extract semantics and semantically equivalent implementations [16] and visualization of generic information in the binary file [17], to the best of our knowledge, there is no effort of reverse engineering in the literature to extract the abstracted real-time task models from a binary program.

We do not position SLIME as a method to unblind hidden or secret information in binaries, such as protected Intellectual Properties (IPs). As a proof of concept, we implement and validate SLIME using FreeRTOS and RTEMS. Section III-A presents the user story of SLIME. We further present the imposed requirements and limitations on the features for implementation and compilation in Section III-B.

A. User Story

SLIME can be employed at any stage during the development process and used for (at least) these two use cases:

Case 1 - **Early Validation**: If the SLIME user designs the implementation to match a certain task set specification, SLIME can be used to check if the implementation actually matches the specification. The SLIME user can get a list of implemented tasks, together with task-specific parameters and execution time measurements, enabling schedulability tests and response time analyses based on the resulting abstracted task parameters.

Case 2 - **System Integration**: If the SLIME user is a system integrator who has no knowledge about the underlying task model for the implementation to be integrated, SLIME can provide the abstracted task model which can be used to analyze the schedulability of the integrated application. Since the extracted model from SLIME potentially can be inaccurate, the user can further record traces of the implementation and analyze the accuracy of the model to match the recorded traces. This automatically enriches the extracted model with time measurements of the implementation.

The concept of SLIME is also useful for industrial applications. In the automotive domain, the emergence of the Software-Defined Vehicle (SDV) paradigm in which

safety-critical functions are increasingly implemented in software [18] and consolidated on shared computing platforms, the integration process becomes highly complex. Software suppliers often deliver functions without explicit timing information such as task activation rates, suspensions, or communication dependencies. As an example, MotionWise, which is deployed in over two million vehicles, includes in its integration workflow [19] a phase of measuring and characterizing software functions before integrating components into a time-triggered schedule to ensure deterministic behavior.

B. Prerequisites

We assume that function calls can be identified unambiguously in the binary program. *In practice, this means to disable function inlining in the compiler and to not use pointer indirect function calls in the implementation.* Throughout this paper, this is pragmatically realized by disabling any compiler optimization. Here are technical details that are needed for SLIME to successfully extract the abstracted task models:

- **Analysis Scope**: SLIME uses the binary reverse engineering tool Ghidra³ to extract the CFG, as well as function call parameters from the provided binary. In some cases, as for instance for FreeRTOS, the binary file includes the entire RTOS implementation, making the function call parameter extraction overly time consuming. Therefore, SLIME offers the user an option to specify a limited analysis scope in the form of an include list of debug symbols for the analysis. Such a list can be created by exporting debug symbols from the compiled user application files automatically.
- **Design Conventions**: SLIME extracts the implemented task model by identifying and analyzing system calls statically from the application to the RTOS. Consequently, SLIME can only extract a model correctly, if these system calls are used in a **statically unambiguous, convention conform** manner. In addition to the imposed limitations on compiler optimization and implementation features (no compiler optimization and indirect calls), one example of static ambiguity, that cannot be handled by SLIME, would be task creation within a loop with statically unknown bounds.

IV. FUNDAMENTALS

We first present the real-time task models considered in SLIME in Section IV-A and the interfaces for such real-time task models in RTOSes, especially FreeRTOS and RTEMS in Section IV-B.

A. Abstracted Task Models

SLIME aims to lift a concrete implementation of an application to an abstracted real-time task model, which can be analyzed for its schedulability and other guarantees afterwards. We focus on *periodic task models with self-suspension and blocking due to resource sharing*, as follows:

- We consider **periodic** real-time tasks. A periodic task releases its jobs (task instances) *periodically* with a fixed period, i.e., the Liu-and-Layland release pattern [20].

²If no debug symbols are available, SLIME can still operate if the names of function calls are provided by other means, such as, manual user input or linking tables. If function call names cannot be resolved, SLIME still extracts a model, which then, however, becomes imprecise and abstract.

³<https://github.com/NationalSecurityAgency/ghidra>

- We consider **self-suspension** [21], in which a job can suspend its execution and resume it later on. Our abstraction of self-suspension behavior is to characterize the sequence of execution and suspension intervals of a real-time job. This notion is similar with the classical segmented self-suspension task model, with a slight difference. The resulting abstraction of SLIME is a set of sequences of execution and suspension intervals of a periodic real-time task (denoted as bag of segments (BOS) in LiME [13]), which can be further converted into the classical segmented self-suspension task model.
- We also account for **resource sharing**, protected by semaphores [22], [23]. A classic example of this requirement is having an I/O device, which can only be used by one task at a time and therefore is protected by a mutex. Considering only the impact on the possible progression of code flow, mutexes, semaphores and blocking queues can all be equivalently modeled with the semantics of a semaphore. SLIME therefore includes semaphores as resource representation in the task model. We map all other instances (mutexes and queues) to the semantically equivalent semaphore during model extraction.

B. RTOS Programming Interfaces

The concept of SLIME can be implemented in general RTOSes. SLIME is currently realized as a proof of concept to work on FreeRTOS and RTEMS. We hence give an overview of the specific case of FreeRTOS and RTEMS in this section.

First, RTOSes offer the creation of tasks to allow implementing concurrent behavior. The code of a task is wrapped in a function, where the function is specified in the task creation process to serve as a body for the created task. While for FreeRTOS, the `xTaskCreate` function provides the interface for task creation, including specifying the task body function, associated scheduling period, and other parameters, RTEMS splits task creation into two functions, namely `rtems_task_create` and `rtems_task_start`.

Periodic tasks are realized in FreeRTOS and RTEMS by creating a loop in the task body, which realizes a periodic delay at the end of each loop iteration. In order to implement a periodic delay, FreeRTOS and RTEMS offer an interface to ensure strictly periodic timing. For FreeRTOS, this function is the `vTaskDelayUntil` function, while RTEMS provides it under `rtems_rate_monotonic_period`. Both functions require specifying the period and an object that stores the previous job release in order to maintain exact periodicity.

Self-suspension is realized through another interface, which lets the task execution suspend for at least a specified period. FreeRTOS realizes this as the `vTaskDelay` function, while RTEMS realizes the behavior as the `rtems_task_wake_after` function. Shared resources are realized by the FreeRTOS and RTEMS programming interfaces as semaphores. Semaphores are objects that must be created and deleted and can be used by their corresponding primitives. For FreeRTOS `xSemaphoreTake` and `xSemaphoreGive` are the corresponding primitives. For RTEMS, `sem_wait` and `sem_post` are the corresponding

interfaces. Mutexes and queues both have their own set of operations in FreeRTOS and RTEMS, which, however, leads to an equivalent behavior in terms of possible code flow progression and therefore is considered the same as the semaphore primitives in the following.

V. SLIME OVERVIEW

SLIME is constructed as three core components, namely SLIME CFG (control flow graph), SLIME ATM (abstract task machine) and SLIME Trace, that are further detailed in the following sections. The user application is compiled through the toolchain of the RTOS, resulting in a compiled binary file (firmware). This binary is analyzed by SLIME CFG, where the RTOS specific function calls are identified and a semantically enriched CFG is derived. This CFG then is transformed into an internal model in SLIME ATM, that models the task set and code flow of tasks in an abstract manner. From this model, data can be directly exported, that is required to run schedulability tests (e.g. tasks with their periods or execution suspension paths). Third, SLIME contains a Tracer component, that is compiled together with the user application. SLIME Trace produces an event trace of the application execution, that can be picked up by SLIME ATM and be validated to match the extracted task model.

VI. SLIME CFG EXTRACTION AND ANNOTATION

SLIME CFG takes as input only the compiled binary file of the user application and optionally a list of symbols to limit the analysis scope to. SLIME CFG exports a list of tasks with their priority and period as result. SLIME CFG furthermore exports semantically enriched CFGs for each task, that contain extracted information about relevant RTOS function calls. SLIME CFG bases on the binary reverse engineering tool Ghidra, which is executed with the provided binary implementation as an input.

A. Function Call Decompilation

First, a standard CFG is exported from the entire provided binary file, limited to the optionally provided analysis scope. Afterwards, we use Ghidra to enumerate all existing function calls in the provided binary and derive their decompilation. We match the function call name with the API of the RTOS and identify function calls to relevant system calls. We consider system calls for task creation, resource allocation, resource deallocation, periodic delay and sporadic delay (self suspension) as relevant. From the decompiled function call, we additionally derive the set of parameters for the function call. Matching these with the parameter specifications in the RTOS API, we create the following function call semantics:

Task creation: The entrypoint to a task is passed as a function pointer to either the task creation function or a task activation function. Additionally, the initial task priority is passed as an integer parameter.

Resource Sharing: Resource management functions, such as semaphore, mutex or queue operations, get the handle to the resource object passed as an argument.

Periodic delay: The period of the delay is passed as an integer

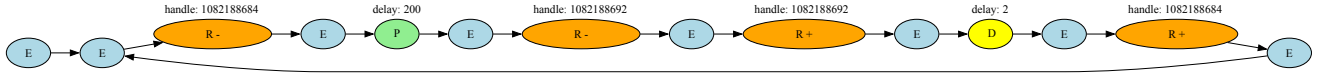


Fig. 4: SLIME ATM for an Example Task.

parameter.

Sporadic delay: Similar to the periodic delay, the minimal delay time is passed as an integer parameter.

For the described parameters, we note, that depending on the programming constructs used in the application, Ghidra potentially cannot decompile a static value assignment. In this case, the further application of SLIME to the corresponding sub components of the application is limited or impossible. In practice, however, task entry points, periods, and delay times tend to be statically assigned.

As a result of the CFG extraction, we derive a CFG (i.e. a directed graph) with a set of normal basic blocks, i.e., vertices, (V_n) and a set of semantically enriched basic blocks, i.e., vertices, (V_e). We describe the extracted semantically enriched CFG for task i by:

$$CFG_i = (V_n \cup V_e, E) \quad (1)$$

B. Task Set Extraction

Next, we extract the different tasks together with their priorities and periods. Based on the decompiled function calls, we analyze task creation calls and start a recursive search at the entry point of the application (main function). When we find a task creation call in the main function, we add the created task together with the extracted priority and the task entry point to the intermediate task set. We continue searching for task creation calls also starting at the task entry point. Consequently, we extract all created tasks, where the task creation call is reachable. By this, SLIME exports the structural abstract task set model, i.e. a list of tasks and priorities. For each extracted task in the task set, we extract the function-level CFG for the task entrypoint. Since the task body function may invoke further function calls, the following procedure is applied: We check if the call is a call to one of the decompiled relevant functions and append a special annotated node to the CFG in that case. If the function call is not a call to a relevant system call, we recursively create function-level CFGs for the function call, which are nested into the CFG.

In order to extend the exported list of tasks by periods, we check each task in the extracted task set for the existence of periodic delay calls. If a task contains exactly one periodic delay call, we annotate this task to be a periodic task. and annotate the task periodicity in the task set. Otherwise, the task does not match the underlying task model of SLIME.

VII. SLIME ABSTRACT TASK MACHINE (ATM)

The extracted CFGs contain basic blocks (which may have dedicated semantics) and edges between the basic blocks. This does not yet express the semantics of suspension, periodic delay and resource allocation in the model itself. In addition, the extracted CFGs not only cover the dedicated RTOS semantic behavior, but also contain all basic blocks of the application implementation. Hence, we introduce an abstracted model, that

centers around the RTOS semantic behavior. Specifically, we translate the extracted CFGs into an abstract task machine (ATM), which is a generalized model, consisting of the internal task behavior regarding execution, suspension and resource usage in an abstracted manner.

The ATM formally is a network of *integer-variable-extended timed-automata* [24], [25]. Timed-automata are automata models, where transitions between locations (states) do not only emit a symbol from the alphabet, but also can carry conditions (so-called *guards*) on a set of uniformly increasing clocks. In addition, there can be global invariants on the clocks. With the extension of integer variables, each guard can further include conditions on these variables. Each transition can update a subset of integer variables to a new value. In a network of such automata, there can exist globally shared clocks and integer variables across different automata instances.

The core intuitive concept of the ATM in this paper is to simplify the extracted CFG and augment the model by the RTOS function call semantics. Each basic block (i.e., vertex) is represented by a location of the timed-automaton. Each task becomes one automaton, while the entire task-set becomes a network. For the locations stemming from V_n , no guards are needed. For the locations stemming from V_e , we need to differentiate between suspension (sporadic delay), periodic delay and resource sharing. For periodic delay, we create a location, that guards leaving to all following states by strict periodicity on a globally shared clock x in the network. We use a local integer variable n_l for the created location l to count the job ID, i.e., counting how often the periodic delay location is visited. The guard then ensures that the location is left not before the release time of the next job and not after the release time of the job after the next job. For self-suspension, a local integer variable t_l records the time when the task enters the suspension state. The guard then ensures the suspension state is only left when at least the suspension delay has passed. For resource sharing, we use globally shared integer variables to count the number of elements in the shared resource. Guards then ensure that a resource-allocation locations can only be left, when there exists at least one element in the resource.

This concept of the ATM design aims for two central purposes. First, the ATM models any progression through the CFG with respect to progressing system time. ATM can serve as inputs for schedulability tests, since, for instance, the sequence of execution and suspension of a task can directly be exported by enumerating distinct paths through the ATM. Second, as the ATM models all possible executions through the CFG with respect to system time in the semantics of timed-automata, a runtime trace of the system can be directly validated against the ATM. By assigning observable runtime events (execution, suspension, periodic delay, resource sharing) as the alphabet of the automaton and emit them together with their corresponding states, a runtime trace is either an accepted

word of the automata network (valid trace for the model), or is not an acceptable word (trace and model mismatch). We further extend the alphabet by the binary addresses of execution, covered by each location / basic block.

In the remainder of this section, we show how we convert the extracted CFG to the ATM (Section VII-A), focusing on simplification steps. Then, we provide a full formal definition of the ATM (Section VII-B). We explain how we derive an efficient implementation of the ATM in Section VII-C and how we extract the relevant task behavior subsequently in Section VII-D. Last, we show how inputs to schedulability tests can be derived from the ATM (Section VII-E).

A. CFG of A Real-Time Task to ATM

We now show how to transform the extracted annotated CFG_i of a real-time task into an ATM. For simplicity, we remove subscription i in this subsection. This procedure requires 1) simplification of basic blocks (vertices) in V_n and 2) extraction of parameters for basic blocks (vertices) in V_e . Before that, we preprocess the CFG and split each vertex in V_e into a vertex in V_n , that has all incoming edges and only has an outgoing edge to the original vertex. By this, we ensure that RTOS specific function calls are separated into the instructions preparing the function call and the pure function call itself.

1) *CFG Simplification*: The ATM is dedicated to model the RTOS relevant semantics of the analyzed application. In the CFG, this information is represented solely by the basic blocks in V_e . Consequently, we can simplify the basic blocks in V_n before mapping a location to each basic block in order to reduce the complexity of the ATM. We conduct this simplification by merging all maximal connected components of basic blocks that are internal in V_n , i.e. a vertex is merged with each adjacent vertex, as long as they are in V_n and not adjacent to any vertex in V_e .

Each basic block in a CFG contains a set of binary addresses, covering the instructions of the basic block. When merging vertices in V_n , we also merge the corresponding set of binary addresses. Given the entire address space of the application as $\mathbb{A}$, we denote the associated binary addresses of a (potentially merged) vertex $v \in V_n$ by:

$$code(v) \subseteq \mathbb{A} \quad (2)$$

2) *CFG Parameter Extraction*: When creating one location per basic block (vertex) after the aforementioned simplification procedure, we need to extract semantic parameters from the CFG to implement the RTOS relevant semantics in the ATM model. We first assign a semantic class to each vertex:

$$class : V_n \cup V_e \rightarrow \{E, D, P, R^+, R^-\} \quad (3)$$

All vertices in V_n are assigned to the semantic class E (Execution). Depending on their identified function call in the CFG, the other vertices are correspondingly assigned to the semantic class of D (self-suspension), P (periodic delay) or R^+, R^- (resource sharing). We furthermore denote all vertices of a semantic class x by:

$$class^{-1}(x) = \{v \in V_n \cup V_e \mid class(v) = x\} \quad (4)$$

For all vertices of the self-suspension class (D), we extract the suspension time from the CFG:

$$delay : class^{-1}(D) \rightarrow \mathbb{N}_{>0} \quad (5)$$

For all vertices of the periodic delay class (P), we extract the period from the CFG:

$$delay : class^{-1}(P) \rightarrow \mathbb{N}_{>0} \quad (6)$$

For all vertices of the resource sharing classes (R^+, R^-), we extract the unique resource handle from the CFG:

$$handle : class^{-1}(R^+) \cup class^{-1}(R^-) \rightarrow H \quad (7)$$

with H denoting the set of all resource handles of the application.

Finally, for building transitions between locations in the ATM, we do not solely resemble the edges of the CFG, but also allow indirect connections in the CFG. This means, the ATM is allowed to transition between any two locations that stem from V_e with visiting at least one location stemming from V_n in between.

To deal with the fact that an execution trace (being an emitted word of the ATM) may not contain all visited basic blocks, we also model arbitrary skipping of other locations stemming from V_n that may be on the path through the CFG. Towards this, we extract the property $Z(v_1, v_2)$ for any $v_1, v_2 \in V_n \cup V_e$ from the CFG G_i as follows:

$$Z(v_1, v_2) = \begin{cases} \text{True} & \text{if either } v_1 \text{ or } v_2 \text{ is in } V_n, \text{ and} \\ & \exists \text{ a path from } v_1 \text{ to } v_2 \text{ in the CFG s.t. the} \\ & \text{vertices in the path except } v_1 \text{ and } v_2 \text{ are all in } V_n \\ \text{False} & \text{otherwise} \end{cases} \quad (8)$$

This property returns true for all pairs of vertices that lie on a path in the CFG with skipping arbitrary normal basic blocks from V_n on the path.

B. ATM as Timed-Automata Network

We define the abstracted model of k real-time tasks as a network of integer-variable-extended timed automata [24], [25]:

$$\mathcal{N} = (\{\mathcal{A}_0, \mathcal{A}_1, \dots, \mathcal{A}_k\}, x, G) \quad (9)$$

x denotes a global shared clock and G a set of shared integer variables, being used for the modeling of shared resources. Each ATM $\mathcal{A}_i$ is then defined as follows:

$$\mathcal{A}_i = (L_i, l_{0,i}, \Sigma_i, W_i, I_i, \Delta_i, class_i, code_i, delay_i, handle_i) \quad (10)$$

with a set of locations L_i , an initial location $l_{0,i}$, alphabet Σ_i , local integer variables W_i , invariants I_i , transition relation Δ_i and four helper functions ($class_i, code_i, delay_i, handle_i$), importing information from the extracted CFG_i . For each vertex v in the (simplified) CFG_i , one corresponding location l is created. The four helper functions are copied from the CFG parameter extraction (Section VII-A2). They take an automaton location as an input and return the same value as the functions for the corresponding vertex defined from Eq. (2)

Transition templates from $l_i \in L_i$ to $l'_i \in L_i$:				
$class_i(l_i)$	$class_i(l'_i)$	$(et, a) \in \Sigma_i$	Guard	Update
E	E	$et = e, a \in code_i(l'_i)$	$\top$	$\phi_i = \min(\phi_i, x)$
E	P	$et = p$	$\top$	-
E	D	$et = d$	$\top$	$t_{l'_i} := x$
E	R^+	$et = r^+$	$\top$	-
E	R^-	$et = r^-$	$\top$	-
P	E	$et = e, a \in code_i(l'_i)$	$\phi_i + JID_{l_i} \cdot delay_i(l_i) \leq x < \phi_i + (JID_{l_i} + 1) \cdot delay_i(l_i)$	$JID_{l_i} + 1$
D	E	$et = e, a \in code_i(l'_i)$	$x \geq t_{l_i} + delay_i(l_i)$	-
R^+	E	$et = e, a \in code_i(l'_i)$	$\top$	$res_{handle_i(l_i)} + 1$
R^-	E	$et = e, a \in code_i(l'_i)$	$res_{handle_i(l_i)} \geq 1$	$res_{handle_i(l_i)} - 1$

TABLE II: ATM Transition Templates

to Eq. (7). The global integer variables contain one element counter res_h for each shared resource:

$$G = \{res_h \in \mathbb{N} | h \in H\} \quad (11)$$

Local variables encode the job ID JID_l for periodic delay and the entering time t_l for suspension delay. In addition, we maintain a local integer variable, modeling the phase of a periodic task ϕ_i :

$$W_i = \{\phi_i\} \cup \{JID_l, \in \mathbb{N} | l \in L_i, class_i(l) = P\} \cup \{t_l \in \mathbb{N} | l \in L_i, class_i(l) = D\} \quad (12)$$

We initialize all job IDs with $JID_l = 1$, the phase with $\phi_i = \infty$ and all suspension entering times with $t_l = 0$.

The timed-automaton does not require any invariant. Therefore,

$$I_i(l) = \top, \forall l \in L_i \quad (13)$$

The global clock of the automata network x advances uniformly and is never reset. It resembles the shared tick counter in the RTOS. For defining the transition of the ATM, we import the Z predicate from the extracted CFG, as described in Eq. (8). Each two locations, where Z returns true, have a transition in the ATM.

The actual task-set RTOS semantics is encoded in the transition relation Δ_i . The detailed transition rules (emitted symbol, guard and variable update) from a location $l_i \in L_i$ to $l'_i \in L_i$ are presented as transition templates in Table II. We note that the guard and update expression of each transition can use global clocks, global integer variables from G and local integer variables from W_i . By construction of the Z property, a transition always starts or ends at a location with the semantic class E , which is furthermore ensured by the splitting of RTOS specific function call in the CFG. Therefore, we only consider transition templates in the following, that start or end with an E type location.

In Table II, a transition starting from an E -type location never has any guards, since there is no RTOS-specific semantics to be modeled during normal execution. The phase of each task is assigned to the clock value of the first execution location of a task. Due to the only increasing clock x , ϕ_i never is overwritten afterward. When entering a self-suspension (D -type location), a copy of the global clock is stored in the corresponding variable to ensure the suspension location is not left before the elapsed suspension time starting from the current point in time. Leaving a P -type location is guarded by ensuring the global clock is not earlier than the release

time of the next job and not later than the release of the job after the next job. As an update, only the job ID is increased, maintaining strict periodicity. Leaving a D -type location is guarded by the global clock being later than the entry time of the location plus the suspension time. Resource allocation and deallocation locations update the corresponding resource element counter as an update action. Allocating a resource is guarded by ensuring there is at least one element to allocate in the shared resource. Since only E -type locations have an associated binary address (*code*), any address is accepted for the emitted symbol when not transiting to an E -type location.

C. Efficient Implementation

Although being simplified, exporting the ATM to a vanilla timed-automata model (e.g. to be imported to UPPAAL [24]) would cause a rather large model definition, since each transition between locations must contain all variations of executed binary addresses as emitted symbols in between. Since we do not require the full feature set of timed-automata analysis in this paper, we derive a more efficient implementation, where simplified transitions between locations are kept, and the Z property is evaluated at runtime, when required.

Figure 4 provides a graphic illustration of an example of the efficient implemented simplified ATM. Semantic classes of locations are indicated by colors, relevant properties (delay times and resource handles) are annotated at the locations, without stating vanilla timed-automata transitions.

D. Task Behavior Extraction

While the previous description of the ATM only accounts for modeling possible executions of a task correctly and checking a concrete instance against the model, we describe how to extract the task behavior from the ATM in such a way, that it can be used as an input for schedulability tests.

Towards this, execution locations (E type) in the ATM have to be further annotated by their required execution time by either integration static WCET analysis into SLIME or by taking fine-grained time measurements, to be described in Section VIII-C later in this paper. Especially together with such execution time information, the timing behavior from each task can be extracted from the ATM. The timing behavior then includes combinations of execution time, suspension intervals and blocking due to resource sharing. We show in the following, how we derive this information in a format, that can be directly used for existing real-time task analyses.

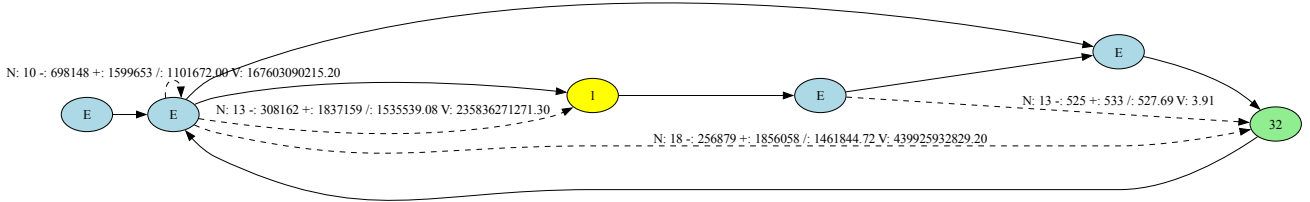


Fig. 5: ATM with Suspension and Execution Paths - Time Measurements by Dashed Arrows

The resulting format consists of multiple variants, called paths, of executing a single job, where each path is a sequence of symbols and numbers. The symbol describes the task state (e.g. executing or suspending), while the number describes the associated parameter (e.g. execution time, suspension time, resource handle or periodic delay). The following is an example of the exported format.

```
Path 1: E, 639, R-, 1082187864, E, 173624, S, 1,
E, 784776, S, 1, E, 469, R+, 1082187864, E, 246, D, 19
Path 2: E, 639, R-, 1082187864, E, 173624, S, 1,
E, 784776, R+, 1082187864, E, 246, D, 19
```

In detail, in this section, we convert a simplified ATM into *all distinct possible execution paths*. An execution path contains a sequence of execution events, each followed by a non execution event, i.e., periodic or sporadic delay or resource management. The simplest form of a periodic task has exactly one execution path in this format, which consists of exactly one execution event, followed by one periodic delay event. The delay event then is annotated with the periodicity of the task, the execution event with the corresponding WCET, which may be the result of an estimation from time measurements. If the task implementation uses suspension, sporadic delay events are furthermore contained in the paths. The paths then consist of sequences of execution and suspension events, each annotated with the execution time and suspension time. If suspension or resource management does not occur in the same sequence in every job of the task, multiple possible execution paths are exported.

In order to convert an ATM into these paths, the sub ATM for repetitive job execution is extracted. This is based on the efficient and simplified implementation, as illustrated in Figure 4. Additional ATM states, as for instance initialization code of the task, which is only run once before the execution of the first job, are discarded. In order to achieve this, we search for a periodic delay event in the ATM and set the starting point for each job to the immediate follow-up location after the periodic delay event. Then, all distinct paths starting from this point and ending in the periodic delay event are traversed and exported. Consecutive execution states are merged into one execution state for the exported path.

Consider, for instance, the example task in Figure 5, where delay information is now displayed as a pure number of ticks inside the corresponding (yellow and green) states. The periodic delay event of 32 ticks is directly followed by the second left-most execution event, which is set as the starting point of each job. The left-most execution event is excluded from the execution paths, since this is a one-time initialization

event. This task further includes a suspension event of 1 tick, which must not be executed in every instance. The ATM furthermore contains execution time measurements, which are indicated by dashed arrows. Since multiple measurements may exist for each transition, the number of samples, mean, min and max is listed in the ATM. Each periodic job of this task can have two distinct execution paths, one with suspension, one without suspension. SLIME ATM exports these paths automatically as follows (including the max of the measured execution time as an estimation for the WCET):

```
Path 1: E, 1856058, D, 32
Path 2: E, 1837159, S, 1, E, 533, D, 32
```

Both paths start with one execution event and end with the periodic job delay of 32 ticks. While the first path only consists of the execution event, the second path follows with the suspension event of 1 tick and another subsequent execution event. It has to be noted that the time for the suspension and delay events is a multiple of system ticks (100Hz in this case), while the measured execution time for execution events is a multiple of CPU clock (160 MHz) cycles.

E. Applying Analyses for the Abstracted Model

Given the extracted task behavior in the form of paths (cf. Section VII-D), this section discusses how the schedulability of the extracted tasks can be tested.

First, we consider the case **without self-suspension or shared resources**. In that case, there is only one path and it contains one E and one P entry. This is because sequences of execution events (E entries) are merged, and each periodic task has exactly one periodic delay event (P). By definition, the E entry relates to the estimated worst-case execution time (WCET) and the P entry relates to the task period. Using the worst-case execution time and the task period, classical results can be utilized to test for schedulability under Fixed-Priority (FP) [20], [26] or Earliest-Deadline-First (EDF) [20], [27] scheduling.

If tasks can access **shared resources** exclusively, they are modeled by mutexes / semaphores. That is, the semaphore is initialized with a counter value of 1, and the counter is reduced when the shared resource is accessed and increased when the shared resource is released. Specifically, there can be multiple paths, and R^- and R^+ only occur pairwise within the same path. The time between the R^- and the R^+ events corresponds to a section with access to the shared resource. To the best of our knowledge, there is no analysis that allows for multiple paths with different access to shared resources. Hence, one single path that provides upper bounds for each

individual segment of all the paths has to be determined to derive blocking times. Afterwards, typical analysis of the blocking time can be applied [28]. Such analysis could further include the employment of a resource sharing protocol, if one is realized in the user application. It should be noted, that the ATM does not depend on any assumptions about employed resource sharing mechanisms. The ATM solely models the sequence of code execution and resource allocation operations, which is oblivious to any resource sharing protocol. Please note that RTEMS also allows the usage of the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP), while FreeRTOS has only a simplified version of PIP implemented. A detailed description and schedulability analysis for such protocols can be found, for example, here: [23]

When **self-suspension** behavior is allowed, then each path is an alternating sequence of execution segments (E entries) and suspension segments (S entries). Since multiple paths define potentially multiple segmentation structures, this model relates to the “*Multiple paths*” hybrid self-suspension model introduced by von der Brüggen et al. [29]. Although, to the best of our knowledge, analyses for the hybrid self-suspension model are limited to [29], hybrid self-suspending tasks can be translated into the classical segmented self-suspending tasks by aggregating the maximal value of each individual segment. Similarly, they can be translated into the classical dynamic self-suspension model, by determining the maximal total execution time (i.e., sum of E entries within one path) and maximal suspension time (i.e., sum of S entries within one path) among all different paths. After reformulating the self-suspending tasks into the suitable model, the schedulability can be checked by utilizing the evaluation framework [30]. For further details on the modelling and analysis of self-suspending tasks, we refer to the review papers [21], [31].

If access to **shared resources** as well as **self-suspension** behavior is detected, the literature does not offer many opportunities for analysis. Specifically, some papers allow suspension only while the shared resource is being held [32], or translate the blocking induced by the access to shared resources into self-suspension. However, to the best of our knowledge, the only work that discusses resource sharing protocols for self-suspending tasks [33] is limited to the Stack Resource Policy (SRP), which is not available in FreeRTOS or RTEMS. Furthermore, whether converting suspension to additional execution (so-called suspension-oblivious analysis [21]) for the analysis is safe in the presence of shared resources remains an open question and is out of scope for this work.

VIII. SLIME RUNTIME TRACING

In addition to extracting a theoretical, analyzable task model in Sections VI and VII, the runtime behavior of the executing implementation is investigated for the third component of SLIME, namely SLIME Trace. Our approach is based on two objectives. The first, intuitive objective is to acquire fine-grained execution time measurements, which can serve as a basis to measurement-based WCET estimation. The second objective is to validate the runtime behavior of the

implementation against the abstracted model to detect potential inaccuracies and mismatches, that stem from the layers of abstraction or incorrect implementations. An example for such a mismatch, would be a task that violates the periodicity by overrunning the time budget. The extracted model implicitly assumes tasks to stick within their time budget, and, hence, a violation can be detected as a runtime-model mismatch.

To achieve both objectives, precise runtime traces need to be captured. In order to verify all location transitions in the ATM against a runtime trace, all delay and resource allocation events need to be captured. In order to verify execution locations in the ATM, the address of code being executed needs to be traced. Since the design of the ATM does not require a full execution traces of addresses as an input to execution state transition (execution states can be skipped if they are connected in the CFG), sampling executed addresses at runtime is sufficient, if at least one sample is recorded for every location change of the ATM between execution and non execution location. Last, all trace-points need to be associated with time-points in order to check their acceptance by the time semantics of the ATM and get fine-grained execution time measurements for ATM transitions. This section details the recording of the required runtime traces and their processing.

A. Stack and Event Tracing

In order to extract the current execution address of a task, we have to account for the fact, that a task may call a shared library during execution. Hence, the program counter cannot serve as a reliable source of the current position of execution within the task context. Instead, the call hierarchy of the investigated task has to be tracked back until a code position within the task implementation. To achieve this, we implement a stack-based backtrack mechanism. By using base pointers, which are placed by the compiler for every stack frame, we can jump back through the stack of function calls and trace the return addresses. To avoid runtime processing overhead during tracing, we design the trace mechanism to store the return addresses of a fixed number of stack frames at each trace point.

In order to sufficiently sample execution states such that there exists at least one sample for each ATM location change between execution and non execution locations, we use event traces, where all calls of the relevant RTOS system calls for delay and resource allocation are reliably traced. Generally, this is achieved by SLIME Trace due to small code portions being inserted into the corresponding system call routines, which then trigger recording a trace point. We showcase how this tracing works in FreeRTOS in the following. FreeRTOS provides a well usable tracing ecosystem, that allows for easy code injection into the corresponding kernel functions. Due to this, the code injection is realized by a single header file that has to be included in the system configuration file `FreeRTOSConfig.h`. In addition, the tracer consists of a C file, that has to be compiled together with the target application. We use the following FreeRTOS trace macros in order to get notifications about the corresponding set of events:

- **Task Switched In:** Context Switch to a new task context.

- **Task Switched Out:** Context Switch from an old task.
- **Delay Until:** Periodic delay function.
- **Delay:** Sporadic delay function (suspension).
- **Task Create:** A new task is created.
- **Queue Send:** An element is added to a queue or semaphore.
- **Queue Receive:** The counterpart of an element being removed from a queue.

Please note that FreeRTOS realizes semaphores and mutexes by creating dummy queues with an element size of 0. Hence, tracing queue events also captures the usage of semaphores and mutexes. Each of these macros is used to allocate a new trace point. For each trace point, we store the following information: Task Handle, Current System Tick, CPU Cycle, Event Type, Stack Trace. The minimal size for each trace point is therefore bound to 40 bytes. On top of this, 4 bytes for each backtracked stack address need to be stored. As a result of experimental testing, we store 8 backtracked addresses, leading to a total trace point size of 72 bytes. The trace points are collected in a memory buffer and exported after the user application is stopped.

B. ATM Validation

After recording the trace, a postprocessing step reads the return addresses of the stack backtrace and checks them against the address space of the application. The first return address in the trace within the task implementation scope is used for further processing. Then, the trace file is processed element by element and fed to the corresponding ATM. The corresponding ATM is determined by the task handle, which is part of every trace element. Every trace point then triggers a location transition (including self transitions in execution locations) in the ATM. When no accepted transition in the ATM exists, the recorded trace element could not be validated against the ATM, and there exists a model mismatch. In total, different types of such mismatches and trace processing failures can appear: .

- **Task handle not known:** Whenever a trace line shows a task executing, that cannot be assigned to any ATM. In FreeRTOS, for example, the main function and the idle task are instances causing this mismatch.
- **Created task not known:** When a trace line indicates the creation of a new task, but the entry point is not covered by any ATM.
- **Backtrace failure:** Whenever the stack backtrace does not contain any return address within the application scope. Typically, this error also occurs when interrupts trigger the call of system functions, or the call depth is too large.
- **Failed transition:** When a trace line is applied to an ATM, but no accepted transition is possible. This indicates a model mismatch between the ATM and the runtime behavior.

Whenever an error occurs during the application of a trace point to the ATM, it is reported. In addition, the ATM is reset to the corresponding execution location of the failure causing trace point.

C. Time Measurements

Due to the recorded time events (CPU cycles and system tick) in the trace file, we further associate the performed

ATM transitions for each trace point with time measurements. Naturally, multiple executions of a task lead to multiple time measurements. The measurements can be used as a basis for measurement-based WCET estimation when doing schedulability analysis. For the cases in this paper, we take the allover maximum measured execution time when exporting the execution paths from an ATM as an indicator of the WCET. Safety margins or other mechanisms can be added here to make the estimation of WCETs more reliable. Estimation of WCET from time measurements, however, is not the scope of this paper.

IX. EVALUATION

Empirical evaluation of SLIME naturally spawns two perspectives, namely, evaluation of the model extraction from a binary and of the runtime tracing and model validation. Assessment of the model extraction relates to whether SLIME can detect implemented tasks, associated with their task parameters, correctly and completely in a given implementation and derive a structural theoretical model. Evaluation of the runtime tracing relates to the time measurements and accuracy of runtime traces when being validated against the derived model by SLIME. We conduct experiments for the model extraction in Section IX-B1 and for the runtime tracing in Section IX-B2 and Section IX-B3.

For all following evaluation experiments, we use the ESP32-C6 DevKitC-1 as a target system. This system contains a RISC-V CPU core with a clock frequency of 160 MHz, 320 kB ROM, and 512 kB RAM. The device comes with a FreeRTOS port from the manufacturer, which we use to implement all experiments in. The system tick frequency of FreeRTOS is set to 100 Hz. We note that due to the rather small ROM and RAM capacity, the available space for runtime traces is rather limited. We therefore cap the experiments to run for several seconds and store at most 7000 trace points ($\approx 0.5\text{MB}$).

A. Studied Applications

Although there is wide public availability of code examples for FreeRTOS and RTEMS, to the best of our knowledge, none of them are provided together with their underlying task model to validate SLIME. For this evaluation, we need code examples, which do not require special hardware and special settings, and at the same time are well-documented enough to reliably know the underlying theoretical task model. Especially for FreeRTOS, most publicly available code does not mention a reliable underlying task model. Within the scope of this paper, we studied several publicly available code projects to reach this conclusion.

Consequently, we center this evaluation around three different types of applications. First, we utilize the FreeRTOS demo applications⁴ as a set of unknown applications, where we test the SLIME model extraction on. We filter out all demo applications that specifically show off hardware interaction, such as

⁴<https://github.com/FreeRTOS/FreeRTOS/tree/main/FreeRTOS/Demo/Common>

serial communication and different memory types⁵. We keep 18 out of 34 applications. It has to be noted that none of the demo examples implements classic periodic or sporadic tasks and no task sets with more than 4 tasks. Therefore, the demo applications only partially represent the target scenario of SLIME. We therefore use as a second source of applications, created by generative AI, namely GPT-5 within ChatGPT. We specify a textual definition of the underlying model and ask ChatGPT to generate a corresponding implementation. To produce implementations based on a textual definition, we use prompts like the following: We vary the number of tasks, as well as the inclusion of resource sharing and self-suspension, in different prompts. Both, the demo applications and the AI generated applications are used to test the model extraction from SLIME (Section IX-B1).

In order to evaluate the tracing component of SLIME, we use a third source of applications, which are synthetically generated (Section IX-B2, Section IX-B3). Synthetically generated applications are primarily impractical for testing the model extraction of SLIME, since the feature set of the generator is limited to the concepts that SLIME is designed for. However, we utilize the generated tasks for assessing the quality of time measurements and the occurrence of tracing errors.

In greater detail, for the task implementation generation, we sample a random number of tasks between 5 and 15. For each task, a random period is sampled between 100 and 500 ms. For the body of each task, we generate a random execution sequence of nested if-else block. Each if statement checks a random number for an arbitrary threshold. Before each if statement, we add a processing loop to the code, consuming execution time. The loop spans for a random number of iterations between 50000 and 80000. Each iteration increments a global variable by one. Right after the loop, we place a self-suspension call with a random probability of 50%. If self-suspension is put, the suspension delay is randomly sampled between 10 and 15 ms. With another 50% probability, before each if statement, we finish the job execution. The maximum nesting level of if-else statements for each task implementation is 3. The main function of a task suspends for 10 seconds and deletes all tasks afterward. The main function, in addition, activates SLIME runtime tracing and exports the trace after all tasks have been deleted.

B. Results

1) *Model Extraction*: For the case of FreeRTOS demo applications, **all tasks of all applications are correctly detected without failures**. For the AI generated applications, **all implementations are detected by SLIME exactly as stated by ChatGPT**.

2) *Runtime Tracing*: As a first result of running SLIME on the synthetically generated tasks, it has to be stated that SLIME statically extracts the model of the tasks and their

parameters correctly in every case of generated implementations. Second, the accuracy of the recorded runtime traces with regard to the extracted ATM is analyzed. Towards this, we evaluate the occurrence of trace processing failures as described in Section VIII-B and report the amount of failures in percent. In total, we generate 1000 independent task sets, run SLIME CFG, SLIME ATM, and SLIME Trace for each of them. We capture and report the different failure types. For graphical illustration, we sort the independently generated task sets for better visibility by the amount of caused trace processing failures. The corresponding results are shown in Figure 6a, Figure 6b and Figure 6c for the amount of not known task handles, failed backtraces and failed state transitions, respectively. Figure 6d analyzes the measured execution time of the generated task sets, which is discussed in Section IX-B3. For the case of unknown task handles (Figure 6a), a high ratio of up to $\approx 40\%$ of all events can be observed. However, as a second line in the plot, we denote the number of task handle failure events, that are caused by the idle task and the main function as explained in Section VIII-B, which makes almost all the observed task handle failures. Considering that switching to the idle task causes one task handle failure event also explains the high ratio of these failures. For the number of failed backtrace events and failed state transitions, the ratio stays below 1% in all cases, while for many experiments no state transitions fail at all. Each backtrace failure also likely causes a state transition failure.

3) *Time Measurements*: SLIME Trace delivers time measurements for the execution of fragments of the task. These time measurements can be used as a basis to estimate task WCETs. In order to investigate the quality of the recorded execution times, we compare the measurements to the amount of generated work in the synthetic tasks. Execution time is spent in the synthetic tasks by a loop increasing a variable for a certain amount of iterations. We build the fraction between the measured execution time for each execution block of a job and the corresponding amount of loop iterations and illustrate the results in Figure 6d. The target hardware has no main memory caches and no out-of-order execution. Hence, ideally, this fraction would be a constant across all experiments and therefore would highlight highly reliable time measurements. It can be observed that the ratio between generated work and measured execution time across all independently generated task implementations stays within $\approx 2-15$ cycles per iteration.

We repeat the same experiment but let a pair of two tasks allocate a shared resource at the start of each job and release it at the end of a job. By this, we evaluate if resource allocation and deallocation are correctly captured. As one task can be blocked by the entire execution of another task, resulting in very long blocking time, we limit the generated number of tasks in this experiment from 4 to 10, while leaving all parameters untouched. This experiment yields the same results as the experiment without resource sharing: All tasks and paths are entirely correctly detected, almost all task handle failures stem from the idle task and the main function and the validation error rate stays below 1%. The relation between measured execution time and generated work stays below 10 in this experiment. Due to space limitation, we omit the graphical

⁵Remaining demo applications after filtering are the following: AbortDelay, BlockQ, blocktim, countsem, death, EventGroupsDemo, flop, GenQTest, IntSemTest, MessageBufferAMP, QPeek, QueueOverwrite, QueueSetPolling, recmutex, sp_flop, StaticAllocation, StreamBufferInterrupt

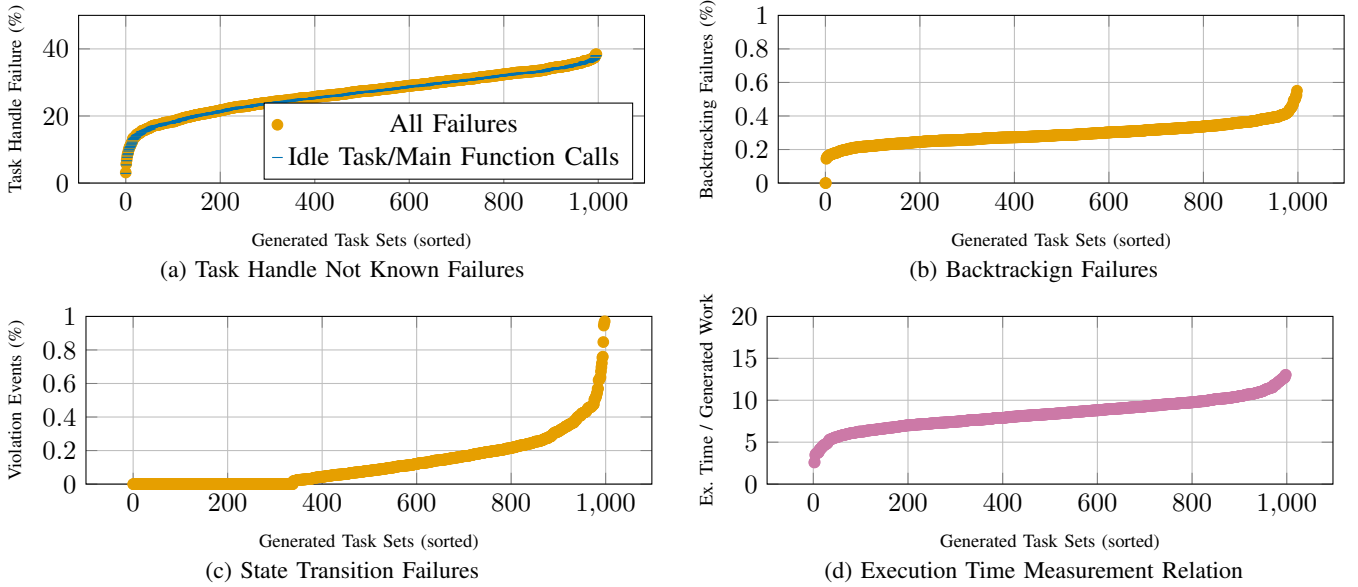


Fig. 6: Synthetic Suspension Task Results of 1000 Task Sets

illustration of the experimental results for this scenario.

X. CONCLUSION

In this paper, we address the fundamental problem of divergences between abstracted real-time task models and concrete implementations of corresponding tasks sets. This is relevant when 1) a model is not known at all and needs to be acquired from a provided implementation and 2) it has to be ensured that an implementation matches an assumed model. Orthogonal to existing dynamic fitting approaches for Linux [13], we present a layered methodology, that structurally converts a provided RTOS application low-level binary into a high-level task model. The derived task model resembles the widely used periodic task model and extends it by further considering self-suspension and resource sharing. In addition to model lifting, SLIME also provides runtime support, that verifies an execution trace against the lifted model and automatically derives execution time measurements. Our experiments show that all structural properties are correctly extracted from all tested implementations. The model accuracy compared to the runtime behavior shows less than 1% errors.

XI. LIMITATIONS AND FUTURE WORK

As a proof of concept, SLIME can support static model extraction of the classical periodic real-time task model as well as the sporadic real-time task model [34]. As a concrete case study, we present in this paper how to handle periodic self-suspending tasks. Although extracting an abstract model statically from the binary implementation, comprehensive task models must contain runtime information, such as release jitter and arrival curves, which is only supported to a limited degree by SLIME. SLIME poses as an orthogonal approach to LiME and LiME-IT [13], [14], which are well capable of extracting required runtime information in a comprehensive manner. In future work, such orthogonal approaches can be well combined to complement extraction of very comprehensive task models.

For periodic self-suspending tasks, as presented in Section IV-A, SLIME is able to extract a set of sequences of execution and suspension intervals for a periodic task. Although the extracted information can be used to establish a dynamic self-suspending task model as demonstrated in Section VII-E, the resulting schedulability analysis is overly pessimistic. To the best of our knowledge, schedulability analysis for self-suspending tasks with a set of sequences of execution and suspension intervals has never been explored in the literature.

In the current state of SLIME, there is a need for information to resolve function call names by debug symbols. In future work, the robustness of SLIME for the lack of such information will be analyzed.

ACKNOWLEDGMENT

According to the submission guideline to disclose the usage of AI tools, we used generative AI, namely GPT-5 within ChatGPT, to produce implementations of real-time tasks. Furthermore, we used Claude Opus 4.6 and Sonnet 4.6 to bootstrap the comparison to state-of-the-art tools (docker environments with dependencies and comparable result rendering).

This research is part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170). This paper has further been supported by Deutsche Forschungsgemeinschaft (DFG), as part of the project ARTS-NVM (502308721) and PEACH (569077889).

We thank Björn Brandenburg and Bite Ye (authors of LiME [13] and LiME-IT [14]) for their support in setting up the correct environment for the state-of-the-art comparison as well as their feedbacks to improve the presentation of our paper.

REFERENCES

- [1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case

- execution-time problem - overview of methods and survey of tools,” *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, 2008.
- [2] N. Holsti, T. Långbacka, and S. Saarinen, “Using bound-t in hrt mode.”
 - [3] W. Chen, I. Izhbirdeev, D. Hoornaert, S. Roozkhosh, P. Carpanedo, S. Sharma, and R. Mancuso, “Low-Overhead Online Assessment of Timely Progress as a System Commodity,” in *35th Euromicro Conference on Real-Time Systems (ECRTS)*, 2023, pp. 13:1–13:26.
 - [4] C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann, “Syswct: Whole-system response-time analysis for fixed-priority real-time systems (outstanding paper),” in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017.
 - [5] C. Dietrich, M. Hoffmann, and D. Lohmann, “Cross-kernel control-flow-graph analysis for event-driven real-time systems,” *ACM SIGPLAN Notices*, vol. 50, no. 5, pp. 1–10, 2015.
 - [6] —, “Global optimization of fixed-priority real-time systems by rtos-aware control-flow analysis,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 2, pp. 1–25, 2017.
 - [7] A. Kässens, M. Burg, and D. Lohmann, “Hypersse: Cross-domain static analysis of partitioned real-time hypervisor systems,” in *Euromicro Conference on Real-Time Systems, ECRTS*, ser. LIPIcs, A. Kritikakou, Ed., vol. 375, 2026, pp. 23:1–23:27. [Online]. Available: <https://doi.org/10.4230/LIPIcs.ECRTS.2026.23>
 - [8] M. Desnoyers and M. R. Dagenais, “The ltnng tracer: A low impact performance and behavior monitor for gnu/linux,” in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.
 - [9] B. Brandenburg and J. Anderson, “Feather-trace: A lightweight event tracing toolkit,” in *Proceedings of the third international workshop on operating systems platforms for embedded real-time applications*, 2007.
 - [10] D. Kim, J. Eom, and C. Park, “L4oprof: a performance-monitoring-unit-based software-profiling framework for the l4 microkernel,” *ACM SIGOPS Operating Systems Review*, vol. 41, no. 4, pp. 69–76, 2007.
 - [11] M. Pohlack, B. Döbel, and A. Lackorzynski, “Towards runtime monitoring in real-time systems,” in *Proceedings of the Eighth Real-Time Linux Workshop*, 2006.
 - [12] J. Huselius and J. Andersson, “Model synthesis for real-time systems,” in *Ninth European Conference on Software Maintenance and Reengineering*, 2005, pp. 52–60.
 - [13] B. B. Brandenburg, C. Courtaud, F. Marković, and B. Ye, “Lime: The linux real-time task model extractor,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2025, pp. 255–269.
 - [14] B. Ye, F. Markovic, and B. B. Brandenburg, “Framework-agnostic model inference for intra-thread real-time tasks.”
 - [15] A. Berger, S. Schuster, P. Wägemann, and P. Ulbrich, “Dynamic fuzzing-based whole-system timing analysis,” in *2025 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2025, pp. 420–433.
 - [16] V. Chipounov and G. Candea, “Reverse engineering of binary device drivers with revnic,” in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 167–180.
 - [17] G. Conti, E. Dean, M. Sinda, and B. Sangster, “Visual reverse engineering of binary and data files,” in *International Workshop on Visualization for Computer Security*. Springer, 2008.
 - [18] G. Niedrist, “Deterministic architecture and middleware for domain control units and simplified integration process applied to ADAS,” in *Fahrerassistenzsysteme 2016*. Springer Fachmedien Wiesbaden, 2018.
 - [19] TTTech Auto, “Motionwise global scheduler: Tackling the challenges of modern sdvs,” <https://www.tttech-auto.com/knowledge-platform/motionwise-global-scheduler-tackling-challenges-modern-sdvs>, Sep. 2024, accessed: 2026-03-27.
 - [20] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.
 - [21] J. Chen, G. Nelissen, W. Huang, M. Yang, B. B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. C. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen, “Many suspensions, many problems: a review of self-suspending tasks in real-time systems,” *Real Time Syst.*, vol. 55, no. 1, pp. 144–207, 2019.
 - [22] E. W. Dijkstra, “Cooperating sequential processes,” in *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer, 2002, pp. 65–138.
 - [23] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority inheritance protocols: An approach to real-time synchronization,” *IEEE Transactions on computers*, vol. 39, no. 9, pp. 1175–1185, 2002.
 - [24] G. Behrmann, A. David, and K. G. Larsen, “A tutorial on uppaal,” *Formal methods for the design of real-time systems*, pp. 200–236, 2004.
 - [25] K. G. Larsen, P. Pettersson, and W. Yi, “Uppaal in a nutshell,” *International journal on software tools for technology transfer*, 1997.
 - [26] M. Joseph and P. K. Pandya, “Finding response times in a real-time system,” *Comput. J.*, vol. 29, no. 5, pp. 390–395, 1986.
 - [27] S. K. Baruah, A. K. Mok, and L. E. Rosier, “Preemptively scheduling hard-real-time sporadic tasks on one processor,” in *RTSS*. IEEE Computer Society, 1990, pp. 182–190.
 - [28] G. C. Buttazzo, *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications, 4th Edition*. Springer, 2024.
 - [29] G. von der Brüggen, W. Huang, and J. Chen, “Hybrid self-suspension models in real-time embedded systems,” in *RTCSA*. IEEE Computer Society, 2017, pp. 1–9.
 - [30] M. Günzel, H. Teper, K. Chen, G. von der Brüggen, and J. Chen, “Work-in-progress: Evaluation framework for self-suspending schedulability tests,” in *RTSS*. IEEE, 2021, pp. 532–535.
 - [31] J. Chen, G. von der Brüggen, W. Huang, and C. Liu, “State of the art for scheduling and analyzing self-suspending sporadic real-time tasks,” in *International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, 2017, pp. 1–10.
 - [32] P. Patel, I. Baek, H. Kim, and R. Rajkumar, “Analytical enhancements and practical insights for MPCP with self-suspensions,” in *RTAS*. IEEE Computer Society, 2018, pp. 177–189.
 - [33] G. Nelissen and A. Biondi, “The SRP resource sharing protocol for self-suspending tasks,” in *RTSS*, 2018, pp. 361–372.
 - [34] A. K.-L. Mok, “Fundamental design problems of distributed systems for the hard-real-time environment,” Ph.D. dissertation, Massachusetts Institute of Technology, 1983.



Christian Hakert is a postdoctoral researcher at TU Dortmund University, Germany. His research focuses on emerging technologies, system software management, resource efficient AI and on real-time systems, as well as real-time operating systems. He received his Ph.D. degree in 2024 and was awarded with the outstanding dissertation award of TU Dortmund University in 2024.



Mario Günzel is a postdoctoral researcher at the Max Planck Institute for Software Systems (MPI-SWS). His research interest is in the area of embedded and distributed real-time systems, with a focus on self-suspending tasks and end-to-end latency of cause-effect chains. He received three best paper awards (ECRTS’23, EMSOFT’24, RTSS’25), one outstanding paper award (RTSS’24), and the ACM SIGBED Paul Caspi Memorial Dissertation Award 2024–2025.



Silviu S. Craciunas is a Senior Principal Scientist at NXP Semiconductors, working primarily in the automotive domain, and a part-time Associate Professor at the Technical University of Denmark. His research focuses on Time-Sensitive Networking, edge AI, and safety-critical distributed systems, with recent work on scalable scheduling methods for multi-core and multi-SoC platforms. He has received multiple best and outstanding paper awards at venues including RTNS, ETFA, ECRTS, and RTAS.



Jian-Jia Chen is Chair Professor of Cyber-Physical Systems (CPS) at Faculty of Computer Science in RWTH Aachen University in Germany. He was Chair Professor of Design Automation for Embedded Systems in TU Dortmund in 2014–2026. His research interests include real-time systems, embedded systems, cyber-physical systems, and distributed computing. He received the European Research Council (ERC) Consolidator Award in 2019.