SafeMC: A system for the design and evaluation of mode-change protocols

Tianyang Chen Linh Thi Xuan Phan
University of Pennsylvania

***Abstract*—Real-time systems with multiple modes require mode-change protocols (MCPs) to ensure safety during mode transitions. A variety of MCPs are available in the literature; however, it can be difficult to tell which of them is the most suitable for a given application. This is because 1) existing work often evaluates MCPs analytically, without considering platform-specific overheads in a real deployment; 2) experimental evaluations, where available, tend to make very different choices in run-time environments and workloads, which hinders a direct comparison; and 3) practical applications often require at least some customization, which can completely invalidate the analysis and/or experimental evaluation of the underlying MCP.**

In this paper, we take a first step towards more principled comparisons. We identify a set of fundamental primitives that most MCPs tend to be built on, and we show that a variety of existing MCPs can be formulated by composing these primitives in different ways. We then present SafeMC, a system for specifying and evaluating current and new MCPs. SafeMC provides an easy-to-use specification language, a library of existing MCPs that can be customized by the user, as well as several tools for test generation, automatic evaluation, tracing, and performance analysis. To demonstrate the utlity of SafeMC, we use it to compare the performance of five classical MCPs in Xen. SafeMC is designed to be extensible and reusable, and we hope that it can serve as a building block for future research in this area.

I. Introduction

Despite recent successful developments of self-driving cars in companies such as Google, Uber, and Tesla, the road towards full autonomy of cyber-physical systems still faces many challenges—among them, the ability to respond and adapt promptly to changes during operation. For instance, a self-driving vehicle must adapt its behavior according to the physical environment (such as road conditions or unexpected behaviors of other vehicles) to avoid accidents; similarly, an unmanned aircraft avionics system must adapt its configuration during sudden turbulence or aircraft system failures to allow continued safe operation. Adaptation may involve moving or terminating existing tasks, as well as starting new ones, and this must be done in a *timely* and *safely* manner.

One approach towards modeling and analysis of such adaptive behaviors is to use a multi-mode formalism. In this formalism, the system operates in multiple *modes*. Each mode corresponds to a configuration or behavior, and it can be characterized by a unique set of tasks. Each *mode change* corresponds to a change in the system behavior, in response to either an event from the external environment or an event from within the system. For example, an adaptive cruise control system can be modeled as a multi-mode system with (at least) two modes: (i) the Speed Control mode, in which the speed is set to a predefined value, and (ii) the Time Gap Control mode, in which the speed is computed dynamically to maintain a minimum distance to leading vehicles. The system changes its mode at runtime, depending on the estimated speeds of other vehicles. Once the system has been modeled in this way, a multi-mode analysis can be used to understand its timing behavior.

A critical input to any multi-mode analysis is the specific protocol used for executing mode changes, also known as the *mode-change protocol* (MCP). When the system changes from one mode to another, the set of jobs that are active can come from both modes; as a result, even if each mode is schedulable in isolation, timing violations can occur during this transient interval. By enforcing a certain execution behavior of the system during a mode change – such as aborting certain jobs, or delaying the release of new jobs – the protocol can avoid or minimize the potential overload, and thus avoid timing violations. The real-time community has already developed a variety of MCPs; see, e.g., Real and Crespo [21] or Burns [4] for a survey.

However, in practice it can be difficult to tell which MCP is the best fit for a given system. There are at least three reasons for this. First, while almost all of the existing work presents a careful analysis of the proposed MCPs, most of it lacks a detailed experimental evaluation. This is problematic because, as prior work has shown repeatedly [20, 26, 27], platform overheads (which are usually abstracted away in the analysis) can have a big impact on the performance of a protocol, to the extent that tasks can miss deadlines even when the analysis predicts that they will be schedulable. Second, when papers do contain experimental evaluations, they tend to be done in very different environments – on different platforms, with different applications and different workloads, etc. – which complicates a direct comparison. Finally, existing work often analyzes relatively simple MCPs: for instance, the protocol might prescribe to always drop a certain job first, regardless of which mode the system is transitioning to, or how important the job is for the safety of the system. Because of this, practical applications of MCPs tend to require at least some customization, which comes with a risk of invalidating the analysis and/or destroying the protocol's guarantees.

In this paper, we propose a way to address these problems. Our key insight is that, while the existing MCPs may appear very different at first glance, they actually have much in common. We identify a set of key primitives that operate on the smallest scheduling entity (i.e., a job) and specify the actions that the MCP must perform, as well as the conditions under which the actions should be invoked. These primitives can be composed – at different levels of granularity, such as job, job type, task, task type, or transition – to form an MCP, and we show that a variety of previously proposed MCPs can in fact be expressed using our primitives. Thus, we obtain a way to decompose existing MCPs into 1) an MCP-specific algorithmic core, and 2) a common, MCP-independent runtime system that executes the algorithm. This solves all of the three problems above: the algorithmic core can easily be customized or (in the case of a novel MCP) rewritten from scratch, while the common runtime simplifies experimentation and enables fair comparisons on a real platform.

We present a system called SafeMC that implements our primitives and that provides an easy-to-use specification language (based on XML) that can be used to describe existing and new MCPs. SafeMC also provides a variety of tools for test generation, automatic evaluation, tracing, and performance analysis. We have integrated SafeMC with Xen; to demonstrate its usefulness, we have used it to experimentally compare the performance of five existing MCPs from the literature. We will make SafeMC freely available to the community under an open-source license, and we hope that it can serve as a building block for future research in this area. In summary, this paper makes the following contributions:

- an extensible set of mode change primitives that implement a broad spectrum of mode change behaviors, and that can be composed to form new MCPs (Section IV);
- the design of SafeMC and its tools for efficient specification, execution, and evaluation of MCPs (Section V);
- a prototype implementation of SafeMC in Xen (Section VI); and
- specifications and experimental evaluations of several existing protocols (Section VII), as well as an automotive case study (Section VIII), to demonstrate the benefits of SafeMC.

Our evaluation of the prototype shows that SafeMC can be implemented efficiently with minimal run-time overhead. Through our evaluation and case study, we show how SafeMC can be used not only to specify, evaluate, and compare a broad set of the MCPs but also to design novel protocols that are optimized for specific characteristics and requirements of realistic multi-mode systems.

We begin with a discussion of related work (Section II) and a description of our system model (Section III).

II. Related Work

There exists a long line of work that extends models and timing analysis techniques from the real-time systems literature to accommodate multi-mode behaviors at multiple levels. For example, several task models and schedulability methods have been developed to support variable computation times or execution periods, which is a type of mode change (e.g., [1, 5, 6, 8, 17]). Similarly, automata-based techniques have also been used to model and analyze multi-mode systems [13, 16, 18, 23]. Existing work in this line, however, focuses on the analysis of the multi-mode system for a given mode-change semantics and not the design and evaluation of different MCPs.

Several *mode-change protocols* have been studied over the years, including the recent addition of protocols for criticality-based mode changes [4]; an excellent survey of these techniques can be found in [21]. A primary goal of these protocols is to ensure that no deadlines are violated during a mode change, and they typically are designed based on two aspects: periodicity (where new jobs of unchanged tasks should be released as before, without being affected), and synchrony (where jobs of new tasks can only be released after all the old mode jobs have completed). An important drawback of these protocols is that they are agnostic to the safety requirements and characteristics of the system in each mode, and thus they may be inefficient or lead to safety violations.

To address this drawback, we have previously developed a theoretical semantic framework for MCPs [19] that generalizes these existing protocols and that allows for the design of more application-specific MCPs. SafeMC nicely complements this existing formal framework: the existing work relies on automata semantics to express mode-change behaviors, e.g., via buffer evaluation and intermediate states, which is powerful and useful for formal analysis, but also purely theoretical and difficult to implement; in contrast, the focus in SafeMC is practicality, with an easy-to-use specification language and intuitive semantics. It should be possible to translate SafeMC primitives to (simpler forms of) the theoretical models in [19].

A number of platforms have been developed to support multi-mode behaviors. For instance, Neukirchner et al. [15] provides an implementation of multi-mode monitors for job activations. Run-time mode changes have also been supported in Ada [3]. Recently, a number of implementations for mixed-criticality systems have also been developed (e.g., [10, 11]), which support a special form of mode changes. Azim and Fischmeister [2] discusses a design that utilizes checkpoints and rollback-based mode-change mechanism for efficient mode changes and an implementation in LITMUS^{RT}. None of these implementations provides a general platform for the design, implementation, and evaluation of a broad set of new protocols, which ours provides. To the best of our knowledge, SafeMC is also the first to provide system supports for exploring mode-change protocols at the Xen level.

Prior work has also developed kernel primitives and abstractions that aim to be expressive enough to support a variety of different scheduling schemes. For example, in the hierarchical scheduling setting, Regehr [22] introduces the Hierarchical Loadable Scheduler (HLS) architecture, which permits schedulers to be dynamically composed in the kernel of a general-purpose OS. Similarly, Lackorzyński et al. [12] uses scheduling contexts to enable flexible and efficient implementation of various guest schedulers in the host scheduler. Ford and Susarla [7] presents a CPU inheritance scheduling framework in which arbitrary threads can act as schedulers for other threads, thus providing much greater scheduling flexibility. Language-based scheduling approaches, such as the Bossa framework [14], allow schedulers to be written using a domain-specific language, which can be instantiated in a standard OS. These complementary lines of work focus on the scheduler implementation instead of mode-change protocols, which is our focus. Integration of such approaches with SafeMC is an interesting future direction.

III. System model and goal

Multi-mode system model. A multi-mode system is defined by a set of modes, the initial mode, and a set of transitions among the modes. Each mode has a set of tasks that are active when the system is in this mode. We follow the conventional real-time task model, where a task is characterized by four (per-mode) timing attributes: a period, a deadline, a worst-case execution time (WCET), and a criticality. Each transition is associated with – and can be triggered by – a mode-change request event (MCR), which for simplicity is assumed to be unique for each transition.

Initially, the system begins in the initial mode. At runtime, whenever an MCR associated with an outgoing transition arrives, the system will perform the mode change, according to a given protocol, to move to the destination mode.

Mode-change protocol. A mode-change protocol (MCP) describes the execution behavior of a multi-mode system during a transition from one mode to another, i.e., from the instant an MCR arrives (called *MCR instant*) until the instant where all the new attributes associated with the destination mode are in effect. It specifies, for instance, when to release a new job in the destination mode (e.g., immediately or after some delay), whether to complete or abort existing jobs, and whether to update the existing jobs with the new timing parameters. In general, the desirable mode-change behavior depends on the safety and performance requirements of the system, and it can vary across transitions, tasks, and jobs. For example, while delaying the release of a new task may be acceptable when an aircraft transits from the take-off mode to the cruise mode, it is undesirable for a transition from the cruise mode to the emergency mode, which requires that the emergency jobs be released and executed as soon as possible.

Types of tasks during a mode transition. For convenience, we refer to the set of tasks that are active in the source or destination mode of the transition as the *transition taskset*. As in prior work, each task τ_i in the transition taskset is of one of the following four types:

- Old (**O**): If τ_i is active in the source mode but not in the destination mode.
- New (**N**): If τ_i is active in the destination mode but not in the source mode.
- Unchanged (**U**): If τ_i is active in both modes and all of its timing parameters remain unchanged.
- Changed (**C**): If τ_i is active in both modes and some of its timing parameters are modified in the destination mode.

Types of jobs. We further categorize jobs of a task during a transition into three types:

- *Pending*: Unfinished jobs that are not currently executing at the MCR instant.
- *Executing*: Unfinished jobs that are currently executing at the MCR instant.
- *New*: New jobs to be released after the MCR instant.

By definition, pending and executing jobs are not applicable to tasks of type **N**, whereas new jobs are usually not applicable to tasks of type **O**. We distinguish pending jobs from executing jobs because they may require different actions during a mode change. For instance, if a currently executing job is almost completed, continuing to execute it until completion may incur less overhead, and thus may be better than aborting the job. In contrast, aborting a pending job typically incurs less overhead compared to letting it continue to run until it completes (due to additional context switches).

Goals. To achieve the goals we have set for SafeMC, it must have the following four properties. (1) *Expressiveness:* SafeMC should cover a broad spectrum of mode-change behaviors for general multi-mode systems that not only capture existing MCPs but also novel MCPs that are desirable for a given system; (2) *Composability:* SafeMC should allow efficient MCP construction by composing different primitives that implement different mode-change logics, at various levels of granularity; (3) *Performance:* SafeMC should have low run-time overhead to enable efficient mode changes; and (4) *Usability:* SafeMC should allow users to easily specify and experimentally evaluate different aspects of an MCP.

In the next sections, we first present the primitives that form the core semantics supported by SafeMC, followed by examples to illustrate how they can be composed to form MCPs. We then present the design and implementation of SafeMC, and end with an evaluation of existing protocols and a case study of an autonomous system.

IV. Mode-change primitives

Formally, an MCP specifies the set of *primitives* that are performed on the jobs of each task that is active in the source mode or in the destination mode of the transition. For SafeMC, we have identified a set of key primitives that cover a broad range of existing MCPs (and hopefully many future MCPs as well). SafeMC can also be extended with additional primitives if necessary.

Existing protocols typically specify the same behavior for all tasks of the same type and globally for all transitions in a system. In practice, however, different tasks of the same type, and even jobs of the same task, may have different requirements during a specific mode transition. For instance, consider two changed tasks τ_1 and τ_2 whose criticality levels change upon an MCR: τ_1 is critical in the source mode but non-critical in the destination mode, whereas τ_2 is non-critical in the source mode but critical in the destination mode. Then, while it is often acceptable to delay the release of new jobs of τ_1 (since they are not critical), delaying the release of new jobs of τ_2 may lead to safety violation (since they are critical). Similarly, while it is important not to drop existing jobs of τ_1, dropping some of the existing jobs of τ_2 is often acceptable.

To capture the diverse characteristics of a wide range of multi-mode systems, we need the ability to specify mode-change semantics at different levels of granularity: for a specific transition, for tasks of a specific type, for a specific task, for jobs of a specific type, and for a specific subset of jobs of a given type. We now describe the primitives in detail.

A. Basic primitives

Mode-change primitives describe the actions that are applied to jobs of a specific type, for a particular task, or for all tasks of a particular type during a mode change.

Definition 1 (Mode-change primitive)**.** *A mode-change primitive is defined by an* action A *and a set of* guards G *that specifies the conditions when the action should be applied to the considered jobs during a mode transition. When* G *is empty,* A *is applied to the jobs immediately at the MCR instant; otherwise, depending on the guard type,* A *is applied to the jobs only if, or when,* G *evaluates to true.*

Mode-change actions. MCPs differ quite a bit in the actions they can apply to jobs. We have identified the following key actions for pending and executing jobs:

- CONTINUE: The jobs continue to be executed and scheduled with their current timing parameters.
- ABORT: The jobs are aborted and removed from the system.
- ABORT[K]: The oldest K jobs are aborted and removed from the system.
- UPDATE: The jobs continue to be executed, but their timing attributes (period, deadline, WCET and criticality) are changed to the values defined for the destination mode. This action is typically applied to jobs of changed tasks but not the others.

For new jobs, we have identified the following key actions:

- RELEASE: This action defines the release of new jobs whose parameters are that of the *destination* mode, and it is always associated with a guard. Specifically, the release of the first new job with the destination mode's parameters is delayed until an associated guard becomes true. (Note that the new values could be the same as the old ones.)
- RELEASE_O: This action specifies that the new jobs whose parameters are that of the *source* mode should continue to be released without being affected by the MCR, as long as an associated guard still holds. This action can be also used in conjunction with the RELEASE action for new jobs of old, changed, or unchanged tasks (e.g., in the Idle Time Protocol shown in Fig. 1.)

When no action is specified for new jobs, these jobs are no longer released after the MCR, which typically is the case for new jobs of an old task.

Action guards. The mode-change action defined by a primitive may be associated with a set of guards, which specifies the set of conditions for when that action should be applied. We have identified multiple types of guards, which we define below.

1) TRUE**:** This is a special guard that simply indicates that the action is to be applied immediately. (It is equivalent to having no guard for that action.)

2) OffsetMCR**:** This guard defines an offset relative to the MCR instant. It is specified by a constant threshold value Δ, or one of the symbols OLD_PERIOD, NEW_PERIOD, MIN_PERIOD, and MAX_PERIOD, which denote an offset equal to the task's old period (in the source mode), new period (in the destination mode), and the minimum period and maximum period of all tasks in the transition taskset, respectively.[1] It is evaluated once at the MCR instant, and the action is to be applied after a delay equal to the defined offset from the MCR instant. This guard is often defined for the RELEASE action to specify when to release the first new job, e.g., to release it after a certain delay to prevent an overload during the mode change.

[1]Additional symbols can easily be added, if necessary.

3) OffsetLR**:** This guard defines an offset relative to the last release time of the corresponding task. It has the same syntax and evaluation strategy as an OffsetMCR guard, except that the action is to be applied at the defined offset relative to the most recent release time of the task. This guard is often used to specify when to release new jobs; e.g., it can be used to enforce periodicity of unchanged tasks (using the offset OLD_PERIOD) or to release the first job of a changed task based on either the old period (using OLD_PERIOD) or the new period (using NEW_PERIOD).

4) Backlog**:** This guard defines a formula that compares the current backlog (number of pending and executing jobs) of the corresponding task with a certain threshold Δ or with a certain fraction r of the queue size MAX_VALUE. It is specified as (Op, Δ) or $(\mathsf{Op}, \mathsf{MAX\_VALUE} \cdot r)$, where $\mathsf{Op} \in \{>, \geq, =, \leq, <\}$, Δ is a (non-negative) constant threshold, MAX_VALUE is the symbol denoting the task's queue size, and r is a positive constant ratio. For pending and executing jobs, this guard is evaluated once at the MCR instant, and (only) if the guard holds, the action is applied immediately. For new jobs, this guard is first evaluated at the MCR instant and, if invalid, it will continue to be evaluated whenever a context switch occurs until either the guard becomes true or a new MCR arrives; the action is applied when the guard becomes true (if it does).

A Backlog guard is useful for aborting pending or executing jobs based on the current backlog (i.e., load) of the task when the MCR arrives. For instance, one may abort some pending jobs if the backlog is more than half the queue size. It is also useful for controlling the release of new jobs, e.g., to delay the release of the first new job of the task until the backlog is below a certain threshold.

5) BacklogGlobal**:** This guard defines a formula on the backlog of the global scheduling queue (i.e., the number of pending and executing jobs of all active tasks). It follows the same syntax, evaluation strategy, and usage as the Backlog guard but considers the system-level load; here, MAX_VALUE denotes the size of the global scheduling queue. A special use of this guard is to only release new jobs at the *idle instant*, i.e., when the global backlog is equal to zero.

6) RemainTime**:** This guard defines a formula that compares the remaining execution time of the job with a certain threshold or with its WCET. The formula follows the same syntax as in (4), where MAX_VALUE denotes the job's WCET. This guard is applicable to only pending and executing jobs by definition, and it is evaluated once at the MCR instant. It is useful for aborting jobs based on their remaining execution times, e.g., to abort a job only if its remaining execution time is more than a threshold Δ or a certain fraction r of its WCET.

7–10) Criticality **(**Period**,** Deadline**,** WCET**):** This guard defines a formula on the criticality (period, relative deadline, or WCET) of the corresponding task in the source mode and in the destination mode. It is specified as $(\mathsf{Op}, \mathsf{OLD\_VALUE}, \Delta)$ or $(\mathsf{Op}, \mathsf{NEW\_VALUE}, \Delta)$ or

	JOB TYPE	*Executing and Pending*	*New*		
	TASK TYPE	**OLD/UNCHANGED/CHANGED**	**UNCHANGED**	**CHANGED**	**NEW**
PROTOCOL	**Maximum Period Offset**	CONTINUE	OffsetLR: OLD_PERIOD	OffsetMCR: MAX_PERIOD	OffsetMCR: MAX_PERIOD
	Min. Offset without periodicity	CONTINUE	BacklogGlobal: (=, 0)		
	Min. Offset with periodicity	CONTINUE	OffsetLR: OLD_PERIOD	BacklogGlobal: (=, 0)	BacklogGlobal: (=, 0)
	Asynchronous with periodicity	CONTINUE	OffsetLR: OLD_PERIOD	OffsetLR: OLD_PERIOD	OffsetMCR: Δ
	Asynchronous without periodicity	CONTINUE	OffsetMCR: Δ		

	TASK TYPE	**OLD/UNCHANGED/CHANGED**	**OLD**	**UNCHANGED**	**CHANGED**	**NEW**
PRO.	**Idle Time Protocol**	CONTINUE	[RELEASE_O; BacklogGlobal: (>, 0)]	[RELEASE_O; BacklogGlobal: (>, 0)] [RELEASE; BacklogGlobal: (=, 0)]		[RELEASE; BacklogGlobal: (=, 0)]

Note: Δ is a user-defined threshold per task. The action RELEASE is defined for all new jobs in all protocols except the Idle Time Protocol, but we omit for brevity.

Fig. 1: Specifications of existing mode-change protocols.

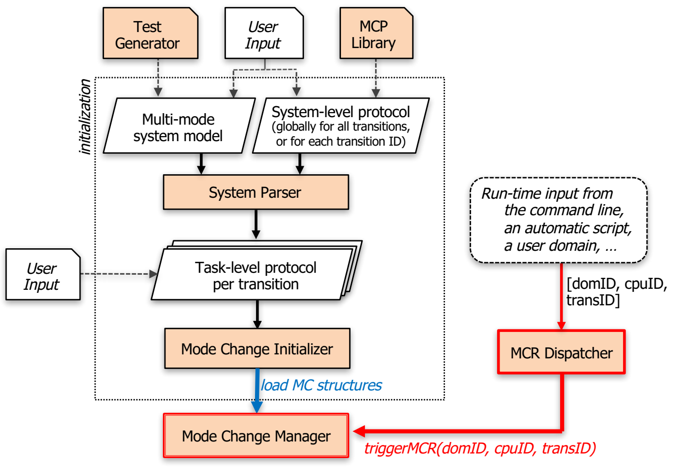
(Op, OLD_VALUE, NEW_VALUE $\cdot r$), where Op $\in \{>, \geq, =, \leq, <\}$; the symbols OLD_VALUE and NEW_VALUE represent the task's criticality (period, deadline, WCET) in the source mode and in the destination mode, respectively; Δ is a constant threshold value; and r is a constant ratio. It is evaluated once at the MCR instant, and the action is applied if the guard holds.

These types of guards are useful for defining an action based on the task parameters. For example, the Criticality guard can be used to release new jobs immediately if the task's new criticality is high (or higher than the old criticality), e.g., using [RELEASE; Criticality: (<, OLD_VALUE, NEW_VALUE)]; it could also be used to abort pending and executing jobs if the old criticality is low (or lower than the new criticality). Since the scheduling priority of a task is typically determined based on its timing parameters, one can easily express a change in the task's scheduling priority at a mode transition in SafeMC.

Finally, we note that thanks to the well-defined semantics and syntax of SafeMC primitives, it is possible to translate the protocol specifications to automata models, e.g., using a similar approach as in [16, 19], thus facilitating the theoretical analysis of new user-defined protocols using verification.

B. Example MCP specifications

To demonstrate that the above primitives are indeed sufficient to express many existing MCPs, we show the specifications of several MCPs from the literature (taken from [21]) in Fig. 1. Note that each protocol is simply a composition of the different primitives defined for the different job and task types. For example, the Maximum Period Offset specifies that all old (pending or executing) jobs should continue to execute, new jobs of unchanged tasks should be released as usual (after a period from the last release) without being affected, and the first new job of changed and new tasks should be released after an offset equal to MAX_PERIOD (the maximum period of tasks in both modes) from the MCR instant. The Minimum Offset without periodicity specifies that old jobs should continue to execute, and the first new job in the new mode (of unchanged, changed and new tasks) will only be released at the idle time (i.e., when the global backlog is 0). A complete description of these protocols is available in [21].



Test Generator
User Input
MCP Library
initialization
Multi-mode system model
System-level protocol (globally for all transitions, or for each transition ID)
System Parser
User Input
Task-level protocol per transition
Mode Change Initializer
load MC structures
Mode Change Manager
Run-time input from the command line, an automatic script, a user domain, ...
[domID, cpuID, transID]
MCR Dispatcher
triggerMCR(domID, cpuID, transID)

Fig. 2: Overall architecture of SafeMC.

Besides existing protocols, we can also construct new protocols, e.g., as discussed above in the use of different primitives. An example of a new protocol that is optimized for a specific system is described in Section VIII (Fig. 5).

V. Design

Next, we describe the design of the SafeMC system, which implements the above primitives and allows users to compose them into new MCPs using a simple specification language. A Xen-based implementation of SafeMC will be described in Section VI.

A. Overall architecture

Fig. 2 shows the high-level architecture of SafeMC. Internally, it is made of multiple components for the specification, initialization and execution of an MCP for a multi-mode system.

At initialization, SafeMC takes as input two specification files: (i) a concrete *multi-mode system model*, and (ii) a *system-level protocol* that defines the mode-change primitives. The protocol can be specified either globally for all transitions or for each transition ID; the former is agnostic of the multi-mode system and thus simpler, whereas the latter requires knowledge of the transition IDs but is useful for systems that need different mode-change behaviors for different transitions. From these input files, the SystemParser generates a concrete *task-level protocol* for each transition of the multi-mode system.

Alternatively, users can also specify a set of task-level protocols for different transitions (that form a multi-mode system) as input directly. This capability is useful for multi-mode systems that require different mode-change behaviors for different tasks, even when they are of the same type.

The ModeChangeInitializer is responsible for setting up the mode-change (MC) data structures that are necessary for correct and efficient operation of the ModeChangeManager at runtime. Based on the generated (or given) task-level protocol of each transition, it builds the MC data structures—which contain the different modes and transitions, together with the specific actions and guards for each task in each transition—and then loads these structures into the ModeChangeManager component. This initial processing enables MCRs to be processed efficiently at runtime, since all corresponding actions and guards are available in the MC data structures.

At runtime, the MCRDispatcher simply waits for new MCRs, which can be initiated interactively from the command line by the system administrator or automatically by some program. Whenever an MCR arrives, it delivers the triggered MCR event with the ID of the corresponding transition (and potentially other information) to the ModeChangeManager.

The ModeChangeManager is responsible for performing the mode changes upon receiving MCR events. It implements all the mode-change primitives defined in Section IV-A, which include various operations such as evaluating the guards for each job (job type) and performing the corresponding actions (e.g., remove some pending jobs from the run queue, disable an old task, update the job/task parameters, release new jobs) based on the guards. Upon a triggered MCR, it looks up the corresponding transition in its MC data structures, scans through all tasks involved in the transition, and performs the actions defined for their jobs based on the action guards.

Besides the core components, SafeMC also contains a set of tools to facilitate the design, experiment, and evaluation of MCPs, such as (1) a TestGenerator for generating multi-mode system models based a certain specified range of input parameters, (2) an MCPLibrary that contains the specifications of common MCPs, and (3) a toolset for tracing and analyzing the MCP performance of an MCP under experiment.

B. Multi-mode system and MCP specification

Both the multi-mode system and the protocols are described in the SafeMC specification language, which is based on XML and follows the semantics and syntax defined in Sections III and IV-A. A multi-mode system model specifies a concrete number of modes, mode transitions, and the set of tasks associated with each mode and their timing parameters. An example specification of a simple system with two modes and two transitions in SafeMC is shown in Listing 1.

A system-level protocol specifies the mode-change primitives for (the different job types of) each task type, either globally for all transitions (independent of the specific multi-mode system that is being considered) or specifically for each transition ID. Most existing protocols, such as the ones in Fig. 1, are global system-level protocols. Listing 2 shows the specification of the Minimum Offset with Periodicity protocol (c.f. Fig. 1) in SafeMC. We can observe that an action is defined for each valid job type (executing, pending, new) for each task type (old, unchanged, changed, and new), and it is to be applied globally to all transitions of any given multi-mode system.

A task-level protocol per transition specifies the concrete mode-change primitives for each task in the transition taskset. This can be given as a user input or generated by the SystemParser from the multi-mode system model and system-level protocol. For example, the generated protocol for the first transition of the example multi-mode system is shown in Listing 3. Observe here that the action is now defined for each concrete task in the transition taskset, and the task's timing parameters and the guard values (e.g., NEW_PERIOD) are concretized. All task parameters are associated with the destination mode, except for old tasks (whose parameters are that of the source mode). For completeness, a changed task also contains the changes in the values (i.e., the field `diff_from_old` gives the new value minus the old value for each changed parameter).

```xml
<sys name="simple">
   <mode id="0">
      <task id="0" wcet="50" period="200" crit="1"/>
      <task id="1" wcet="10" period="100" crit="0"/>
      <task id="2" wcet="20" period="100" crit="0"/>
   </mode>
   <mode id="1">
      <task id="1" wcet="10" period="100" crit="0"/>
      <task id="2" wcet="20" period="150" crit="0"/>
      <task id="3" wcet="5" period="50" crit="1"/>
   </mode>
   <trans id="0" src="0" dst="1"/>
   <trans id="1" src="1" dst="0"/>
</sys>
```

Listing 1: An example multi-mode system specification.

```xml
<mcp name="minimum offset with periodicity">
   <task type="old">
      <action_executing value="CONTINUE"/>
      <action_pending value="CONTINUE"/>
   </task>
   <task type="unchanged">
      <action_executing value="CONTINUE"/>
      <action_pending value="CONTINUE"/>
      <action_new value="RELEASE">
         <guard type="OffsetLR">NEW_PERIOD</guard>
      </action_new>
   </task>
   <task type="changed">
      <action_executing value="CONTINUE"/>
      <action_pending value="CONTINUE"/>
      <action_new value="RELEASE">
         <guard type="BacklogGlobal" Op="="
                threshold="const">0</guard>
      </action_new>
   </task>
   <task type="new">
      <action_new value="RELEASE">
         <guard type="BacklogGlobal" Op="="
                threshold="const">0</guard>
      </action_new>
   </task>
</mcp>
```

Listing 2: An example system-level protocol.

```
<mcp name="minimum offset with periodicity">
   <task type="old" id="0"
             wcet="50" period="200" crit="1">
      <action_executing value="CONTINUE"/>
      <action_pending value="CONTINUE"/>
   </task>
   <task type="unchanged" id="1"
             wcet="10" period="100" crit="0">
      <action_executing value="CONTINUE"/>
      <action_pending value="CONTINUE"/>
      <action_new value="RELEASE">
         <guard type="OffsetLR">100</guard>
      </action_new>
   </task>
   <task type="changed" id="2"
             wcet="20" period="150" crit="0">
      <action_executing value="CONTINUE"/>
      <action_pending value="CONTINUE"/>
      <action_new value="RELEASE">
         <guard type="BacklogGlobal" Op="="
                threshold="const">0</guard>
      </action_new>
      <diff_from_old wcet="0" period="50"/>
   </task>
   <task type="new" id="3"
             wcet="5" period="50" crit="1">
      <action_new value="RELEASE">
         <guard type="BacklogGlobal" Op="="
                threshold="const">0</guard>
      </action_new>
   </task>
</mcp>
```

Listing 3: The generated task-level protocol for the transition from mode 0 to mode 1 of the system in Listing 1.

VI. Implementation

We now describe our current SafeMC prototype, which extends the Xen virtualization platform (version 4.7) [24] and Xen's RTDS scheduler to implement the different components of SafeMC and to support mode changes at the hypervisor level. The RTDS scheduler schedules the virtual CPUs (VCPUs) using the (partitioned or global) EDF algorithm [25]. From the hypervisor's perspective, each 'task' defined in SafeMC corresponds to a VCPU; the period, WCET and remaining execution time of the task correspond to the VCPU's period, replenishment budget, and remaining budget.

We opted for a Xen-based implementation for three main reasons: 1) There is substantial interest in using virtualization to consolidate components in real-time systems; 2) Xen has been adopted in automotive platforms (e.g., GlobalLogic's Nautilus), and we also have projects with colloborators from the automotive industry that use Xen in autonomous driving research; and 3) Since Xen is widely used, our prototype could have applications beyond traditional real-time embedded systems, e.g., real-time cloud applications or IoT. SafeMC is not limited to Xen/EDF; we plan to implement it on other platforms (e.g., Linux/LITMUS) and schedulers in future.

A. Multi-mode data structures and API

The current Xen implementation allows only a single job per VCPU. We extended the RTDS scheduler to support multiple jobs per VCPU (i.e., each VCPU has a list of active jobs) and job-level scheduling run-queue (i.e., a queue of runnable jobs). This support is necessary for multi-mode systems due to the presence of multiple jobs per VCPU during mode changes. All the scheduler's functions – such as `wake()`, `context_save()` and `sleep()` – were modified to support scheduling at the job level (instead of at the VCPU level). In addition, we extended the VCPU data structure to include the VCPU type (old, new, changed, unchanged) and criticality.

The MC data structures stored in the ModeChangeManager consist of a global array of MC structures for all transitions of the multi-mode system. Each transition's MC structure contains the set of VCPUs that are involved, the IDs of domains (VMs) to which they belong, and a pointer to an array of VCPU-level MC structures for the VCPUs in the set. A VCPU's MC structure specifies all details of the mode-change primitives to be applied to its jobs; for example, it has fields for actions (`action_executing`, `action_pending`, `action_new`) and guard structures (with a guard type and a value field), as well as VCPU timing parameters associated with the considering transition.

B. SafeMC core functionality

The components SystemParser, ModeChangeInitializer and MCRDispatcher were implemented as three new modules in Xen's user-space tool stack. The ModeChangeManager was implemented in the hypervisor; here, we extended the RTDS scheduler (`sched_rt.c`) and the common Xen scheduling framework (`schedule.c` and `xc_rt.c`) with the mode-change primitives to support mode changes at run time.

We added two new hypercalls in the hypervisor to support communication from Xen user-space to the hypervisor; once executed, their handlers will invoke function hooks that we added in the scheduler that realize the primitives. The first hypercall, `loadMC`, is used by the ModeChangeInitializer to copy the MC data structures that it built from the protocol specifications into the hypervisor's space (dynamically allocated). These data structures provide all the information that the hypervisor needs to perform mode changes. Although this hypercall is typically invoked during initialization, it can also be called at runtime to add or modify some modes and transitions dynamically; this provides support for systems in which some modes or transitions are only known at runtime.

The second hypercall, `triggerMCR`, is used by the MCRDispatcher to trigger a specific transition whenever a new MCR event arrives; its handler implements (in the scheduler) all the mode-change primitives defined by SafeMC. Once the hypercall is executed, the handler iterates through each VCPU of the corresponding transition, scans through its MC data structure, identifies the actions and associated guards (types and values), evaluates the guards, and performs the actions accordingly. Different types of guards and actions were implemented differently, based on the semantics described in Section IV-A. For example, actions with no guard or a TRUE guard are always performed immediately. Similarly, for VCPUs with no `action_new` defined, the handler will disable their new releases immediately. An OffsetMCR (or OffsetLR) guard is checked only once, and it is implemented as a timer that will fire after the specified amount of time from the time the MCR is received (or from the last job release time);

when the timer expires, the handler will perform the associated action (e.g., release the first new job of the VCPU). For a Backlog (or BacklogGlobal) guard for new jobs, if the guard does not hold, the handler will continue to check the backlog condition of the job queue at each following context switch; once the guard becomes valid, it will perform the action.

C. Toolset for design and evaluation

To facilitate the design and evaluation of MCPs, our prototype also includes a set of tools for test generation, testing automation, tracing, and post-processing.

Test Generation: The TestGenerator was implemented as a new module in Xen's userspace. It randomly generates a multi-mode system model for testing, based on minimum and maximum values for the different parameters (e.g., number of modes, number of transitions, number of VCPUs per mode, VCPU utilization and period). As in existing task generation tools, our test generation only generates the multi-mode system (e.g., modes, set of tasks, and their timing parameters). The actual code of a task and its undo code are application-specific and should be supplied by users. In addition, we also implemented a library of MCPs that includes the specifications of a range of existing protocols (such as those shown in Fig. 1) that users can choose from for their experiments.

Tracing. We extended the Xen tracing framework in the hypervisor to enable the performance evaluation of MCPs. The extended tracing framework can trace several mode-change-related variables and events, such as the time a job is disabled/enabled/updated, the MCR instant, the job queue's backlog at MCR instants, and context switches. It can also trace events that are useful for micro-benchmarks, such as scheduling, context-switch, MC parsing, and job-release overheads. All overheads due to the mode changes are reflected in the collected traces.

Testing automation and post processing. We implemented a set of scripts within Dom0 userspace for automating the evaluation, e.g., scripts for (i) running a specific test for a given protocol, (ii) invoking the ModeChangeInitializer to load the MC structure of one or more transitions into the ModeChangeManager, (iii) initiating MCR events, and (iv) processing traces. In addition, we also developed a number of MATLAB programs for post-processing tracing data and for plotting scheduling and mode-change events and various performance statistics.

D. Limitations

Our prototype currently supports only partitioned multicore scheduling, but it should not be difficult to add global scheduling and complex interactions between queues in future work. Currently, our prototype does not yet support complex rollback mechanisms (e.g., when aborting a job), and it only implements some simple consistency checks. We plan to add rollback features and develop a comprehensive layer in SafeMC to check the consistency of a composition of rules and protocols in future.

VII. Evaluation

We conducted an extensive set of experiments using our prototype to illustrate its applicability. Our main objective is to show how SafeMC can be used to experimentally evaluate and compare different MCPs according to several performance metrics. In addition, we also show how much extra overhead SafeMC introduces to support mode changes.

A. Experimental setup

System configuration. Our prototype ran on an 8-core 3.4 GHz Intel machine. We booted the hypervisor with Domain 0 and one test domain, both running Ubuntu 12.04 (Linux kernel 3.8).[2] The test domain executes the multi-mode system under test and is pinned to the first core, whereas Domain 0 is pinned to the rest of the cores.

We used the TestGenerator to randomly generate a set of multi-mode systems for our experiments, with varying number of VCPUs that are active per mode (4, 8, 16, and 32), and the total utilization of all active VCPUs in each mode of the system was set to be at most 96% to ensure that they are schedulable in each mode *in isolation*. (The extra 4% is to account for potential overheads). The maximum VCPU period was set to be 50ms, and the budget ranged between 1.5ms and 12ms.

The VCPUs were scheduled using the Earliest Deadline First (EDF) algorithm by the RTDS scheduler that we extended with SafeMC. We used the extended Xen tracing framework to collect all data in all experiments for our benchmarks and evaluation.

Protocols for evaluation. We considered five commonly used MCPs in our evaluation: (1) maximum period offset, (2) minimum offset without periodicity, (3) minimum offset with periodicity, (4) asynchronous with periodicity, and (5) asynchronous without periodicity. The primitives defined by these protocols are shown in Fig. 1. For our experiments, the threshold value of the guard OffsetMCR in the last two protocols, (4) and (5), was set to be $\Delta = 5$ ms.

B. Run-time overhead

We ran a series of experiments using each of the five protocols and the generated multi-mode systems, to evaluate the extra overhead introduced by SafeMC based on our prototype. We used an approach similar to [25] to measure the overhead. We recorded the timestamps before and after several relevant functions during each mode transition, such as

- `schedule()`, the main scheduling function,
- `context()`, the context-saving accounting function,
- `repl()`, the timer budget replenishment for a new job release, and
- `mc()`, the mode-change function that processes all MC data structures and performs the actual mode change upon an MCR,

using `gettime_stamp()`, and then computed the difference. For each protocol and each multi-mode system, we took

[2]Note that this is not a limitation; our prototype works for any guest OS that Xen supports.

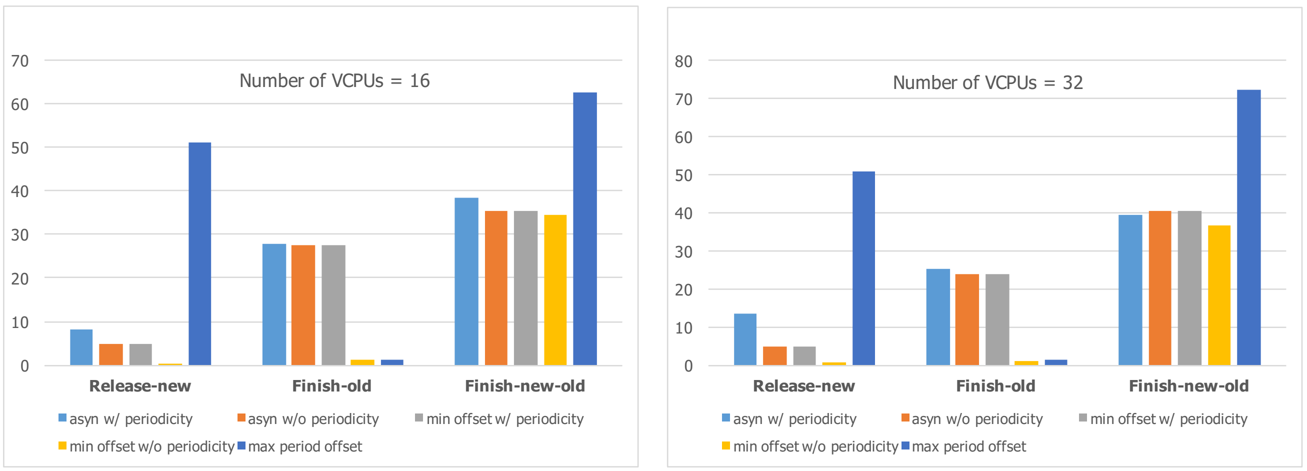


Fig. 3: Mode-change delay of common protocols. Time units are in milliseconds.

1000 measurements (corresponding to 1000 mode transitions), and we report the average.

Table I shows the scheduling, context-switch, release, and mode-change overheads for different numbers of VCPUs per mode. All values are in nanoseconds, and they are averaged over all protocols and all iterations.

#VCPUs	Schedule	Context switch	Release	Mode change
4	540.5	125.4	1025.0	1986.2
8	478.2	86.8	1465.5	3697.5
16	399.8	44.5	1900.8	7063.2
32	366.2	64.0	3267.0	12753.5

TABLE I: Scheduling and mode-change overheads (in ns).

The results show that SafeMC incurs negligible scheduling, context-switch, and release overheads; these overheads are comparable to those of the vanilla Xen's RTDS scheduler (available from [25]). In addition, it has only a small mode-change overhead. We observe that the mode-change overhead increases (close to) linearly with the number of VCPUs, which is expected as the numbers of guards and actions in the mode-change logic also increase. In contrast, there is no clear trend for scheduling and context switches: both scheduling and context-switching are very fast (just a few hundred cycles); at this scale, small platform effects (such as cache or TLB misses) can make a big difference, which is why there is no clear correlation with the number of VCPUs.

Remarks. Since our prototype is based on job-level priority scheduling (EDF), we anticipate that the overheads would be smaller for an implementation based on fixed-task priority scheduling (which is simpler), but that it would follow a similar pattern as the values observed here. In addition, a native implementation of just one protocol could probably avoid some of SafeMC's overheads. However, there is a tradeoff between generality and efficiency; since SafeMC was designed to support experimental comparisons and exploration of protocol space, we went with the former.

C. Performance evaluation of existing protocols

One use case of SafeMC is the experimental evaluation of different MCPs for a given multi-mode system, which is necessary to understand the protocols' tradeoffs and effectiveness. To illustrate this, we performed experiments to evaluate the mode-change latency of the five existing protocols described in the setup, using the following three metrics:

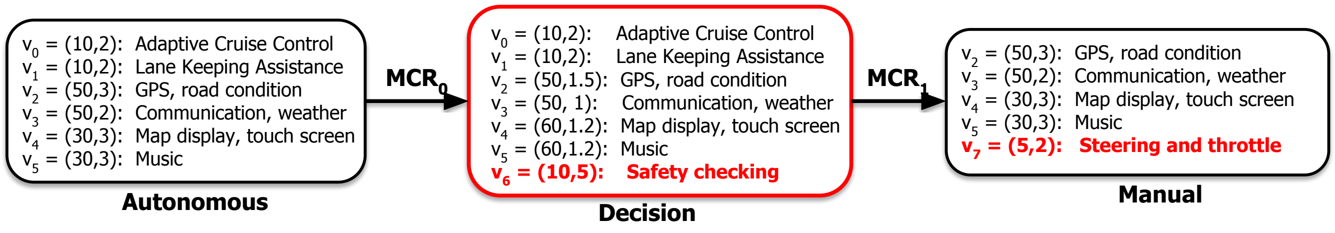
- *Release-new*: The delay from the MCR instant to the instant where all tasks associated with the new mode have released their first new job. This metric measures how fast a new mode is activated.
- *Finish-old*: The delay from the MCR instant to the instant where all pending and executing jobs have completed. This metric measures how fast the system completely leaves the old mode.
- *Finish-new-old*: The delay from the MCR instant to the instant where all pending and executing jobs, as well as the first job of every task in the new mode, have finished. This measures how fast the system completely enters the new mode, without any jobs left over from the old mode.

(We omitted the Idle-Time Protocol because it is simplistic and not well-suited for real-time response to mode-change requests. This protocol *continues to release jobs of all old tasks* and moves to the new mode only when the processor is idle; this could take arbitrarily long and thus is highly impractical.)

Results. Fig. 3 shows the maximum mode-change delay of each protocol across 1000 runs for different VCPU settings. The results show that relative performance among the protocols is similar across all four VCPU settings (4, 8, 16 and 32 VCPUs per mode); due to space constraints, we omit the results for 4 and 8 VCPUs.

It can be observed from the results that the minimum-offset-without-periodicity protocol – which delays the release of all new jobs until the idle instant – appears to be the most efficient according to all three performance metrics. In contrast, the maximum-period-offset protocol – which delays the release of new jobs of changed and new VCPUs for a duration equal to the maximum period of all VCPUs – performs poorly in activating a new mode (Release-new) and in entering the new mode completely (Finish-new-old).

These results show that using the maximum period as the delay offset (as the maximum-period-offset protocol does)



v_0 = (10,2): Adaptive Cruise Control
v_1 = (10,2): Lane Keeping Assistance
v_2 = (50,3): GPS, road condition
v_3 = (50,2): Communication, weather
v_4 = (30,3): Map display, touch screen
v_5 = (30,3): Music
Autonomous
MCR_0
v_0 = (10,2): Adaptive Cruise Control
v_1 = (10,2): Lane Keeping Assistance
v_2 = (50,1.5): GPS, road condition
v_3 = (50, 1): Communication, weather
v_4 = (60,1.2): Map display, touch screen
v_5 = (60,1.2): Music
v_6 = (10,5): Safety checking
Decision
MCR_1
v_2 = (50,3): GPS, road condition
v_3 = (50,2): Communication, weather
v_4 = (30,3): Map display, touch screen
v_5 = (30,3): Music
v_7 = (5,2): Steering and throttle
Manual

Fig. 4: Multi-mode system case study. Here, each $v_i = (P_i, B_i)$ gives the period P_i and WCET B_i (in ms) of the task v_i.

TASK TYPE	OLD	UNCHANGED			CHANGED			NEW
JOB TYPE	***Pend./Exec.***	***Pending***	***Executing***	***New***	***Executing***	***Pending***	***New***	***New***
MCR_0		CONTINUE	CONTINUE	[RELEASE; OffsetLR: OLD_PERIOD]	CONTINUE	ABORT	[RELEASE; OffsetLR: Δ]	[RELEASE; TRUE]
		Tasks: v_0–v_1			Tasks: v_2–v_5			Tasks: v_6
MCR_1	ABORT				CONTINUE	ABORT	[RELEASE; OffsetLR: OLD_PERIOD]	[RELEASE; TRUE]
	Tasks: v_0,v_1 ,v_6				Tasks: v_2–v_5			Tasks: v_7

Fig. 5: Primitives for the hybrid protocol.

can substantially hurt performance: if the current load at the MCR instant is sufficiently small (this was the reason the idle time was reached quickly in the minimum-offset-without-periodicity protocol), this strategy will lead to unnecessary delay. Hence, it seems useful to consider the backlog in determining when to release new jobs, to avoid unnecessary delay.

We can also observe from the results that, since these protocols only release new jobs when there are no more pending and executing jobs, they both can completely leave the old mode much more quickly compared to other protocols (as indicated by the Finish-old values). In other words, while the maximum-period-offset protocol is a poor choice when a prompt activation of the new mode is critical, it is suitable for a mode transition where a fast completion of the originating mode is more important.

Between the two asynchronous protocols, which release new jobs even when old jobs have not yet completed, the protocol without periodicity is faster in releasing new jobs (Release-new). This is expected: this protocol does not need to maintain periodicity, and thus, with a sufficiently small offset (smaller than the task period), it can release new jobs earlier. In addition, the results reveal that the asynchronous-without-periodicity and the minimum-offset-with-periodicity protocols have very similar performance, even though they are composed of very different primitives.

VIII. Case Study

To illustrate how SafeMC can be used to develop novel MCPs for real systems, we conducted a case study of a simple multi-mode system in self-driving cars. Our case study was created based on the self-driving car patent published by Google [9].

A. Multi-mode system description

A self-driving car operates in (at least) three modes of operation: (i) *autonomous* mode, during which the software has complete control of the vehicle; (ii) *manual* mode, during which the driver maintains control; and (iii) *decision* mode, which is an intermediate mode between the autonomous and manual modes, during which various tests are performed to ensure safe control handover to the driver. The functionality and the tasks within each mode are described below.

In each mode, the system executes a subset of the following:

(C1) *Essential control*: Either Adaptive Cruise Control (v_0) and Lane Keeping Assistance (v_1), or Steering and throttle (v_7).
(C2) *Navigation and environmental control*: GPS and road condition sensors (v_2); Communication and weather (v_3).
(C3) *Infotainment control*: Map display and touch screen (v_4); Music (v_5).
(C4) *Safety decision checking*: Safety checking (v_6).

Initially, the system is in the autonomous mode, which executes (C1) to (C3). Whenever the driver initiates a request to take back control, the system first enters the decision mode. In this mode, it executes (C4) to perform necessary checks based on various kinds of information (such as the environment, car speed, road condition/curvatures, lanes, and future predictions) to ensure a safe handover. In addition, it also needs to maintain control of the vehicle; thus, (C1) remains unchanged, but the less critical tasks in (C2) and (C3) have reduced execution budget and larger period to ensure schedulability. While in the decision mode, if the safety check produces positive output, the system then transitions to the manual mode. In this mode, the system does not need to execute (C4), and thus the parameters for (C2) and (C3) are updated. In addition, since the driver has control, (C1) consists of only steering and throttle (v_7). Fig. 4 shows the modes and their tasks' parameters (period and WCET). All values are in milliseconds.

B. Mode-change protocols

We considered two MCPs: (1) the existing minimal offset with periodicity protocol, which was chosen to maintain the periodicity of autonomous driving functions; this protocol was defined globally for both transitions (MCR_0 and MCR_1); and (2) a hybrid protocol, which was newly designed to provide different mode-change semantics required by the different transitions. We describe this hybrid protocol (Fig. 5) in detail.

For the first transition (to the decision mode), the system executes a new task v_6 for safety checks. Since this check is critical for minimizing the delay to the manual mode, it is released immediately. Since the system needs to maintain control of the vehicle, v_0 and v_1 should not be affected by the MCR. Finally, since v_2 to v_5 are less critical, the protocol aborts all pending jobs to avoid overload, continues completing the currently executing jobs to avoid context switch overhead, and releases new jobs after a delay of $\Delta = 1$ ms.

For the second transition (to the manual mode), the protocol releases the steering and throttle control v_7 immediately, as it is essential for the vehicle control. The autonomous functions, v_0 and v_1, are no longer needed; thus, no new jobs are released, and pending and executing jobs can be safely aborted. The logic for the pending and executing jobs of the less critical functions v_2 to v_5 is the same as for the previous transition, and their new jobs are released based on their tasks' old periods.

Remarks. The goal of our case study was to show that SafeMC can be applied to realistic multi-mode systems, and that it can be used to design a completely new class of protocols, e.g., protocols that consider the semantics of specific mode transitions. The hybrid protocol serves as an example, and is not necessarily the best choice for this application. In addition, the minimal-offset-with-periodicity protocol was chosen among the existing protocols instead of the one without periodicity, because (1) it is critical to maintain periodicity for essential autonomous driving functions – i.e., the adaptive cruise control and lane-keeping assistance tasks – when transitioning from autonomous to decision mode (which the hybrid protocol also does), and (2) it performed better than other protocols with periodicity.

C. Evaluation results

We specified the multi-mode system and the above two protocols in SafeMC, and performed experiments using our prototype to evaluate the relative performance of the two protocols. Table II shows the maximum values of the mode-change delay in ms (Release-new, Finish-old, and Finish-new-old), computed across 100 runs, for each transition under the two protocols.

Transition	Protocol	Release-new	Finish-old	Finish-new-old
MCR_0	Existing	32.984	2.005	39.927
	Hybrid	3.441	15.223	26.176
MCR_1	Existing	46.312	8.305	51.036
	Hybrid	0.005	1.908	11.947

TABLE II: Maximum mode-change delay (in ms).

For the transition from the autonomous to the decision mode, our hybrid protocol is substantially more effective in both the Release-new and Finish-new-old metrics. For example, it is almost 10 times faster than the existing protocol in releasing new jobs in the new mode, and thus allows the safety checks to be done much more quickly. In addition, although it delays the completion of old jobs, it can completely transition to the decision mode (i.e., complete both old jobs and the first new job of the new tasks) a lot faster.

We further observe that, for the transition from the decision mode to manual mode, our hybrid protocol outperforms the existing protocol by an order of magnitude across all performance metrics. These results demonstrate that, using SafeMC, it is possible to design and evaluate a whole new class of protocols that are optimized for specific and realistic multi-mode systems.

For completeness, we also conducted experiments for the minimal-offset-without-periodicity protocol, since it offered the best performance among the existing protocols (Section VII) even though it violates the periodicity requirement of our case study. The results show that, for the autonomous-to-decision transition, it performs better in releasing new jobs and finishing old jobs, but worse in completely entering the new mode, compared to the hybrid protocol. For the decision-to-manual transition, however, it performs several times worse than the hybrid protocol in all three metrics. We also note that, unlike the hybrid protocol, the minimal-offset-without-periodicity protocol does not preserve periodicity of the essential autonomous driving functions, and thus it may not be suitable in practice for this particular case study.

IX. Conclusion

In this paper, we have argued that it is often difficult to determine which mode-change protocol is best for a given application: existing work often omits an experimental evaluation entirely, or makes choices that are incompatible with those in other systems; additionally, practical applications often require customizations, which can invalidate the analysis or destroy important properties. To address these problems, we have identified a set of common primitives that are at the heart of many existing MCPs; this allowed us to decompose existing MCPs into an MCP-specific algorithmic "core" and an MCP-independent, shared runtime. We have shown that our primitives can be used to express a wide variety of existing (and hopefully future) MCPs, and we have implemented them in a system called SafeMC. We hope that SafeMC can be an asset both for practitioners and for future research in this area – e.g., by enabling side-by-side experimental comparisons, or by supporting rapid prototyping and customization.

Acknowledgement

This research was supported in part by ONR N00014-16-1-2195, NSF CNS 1703936, CNS 1563873 and CNS 1505799, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-16-C-0056 and HR0011-17-C-0047.

References

[1] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *FORMATS*, 2003.

[2] A. Azim and S. Fischmeister. Efficient mode changes in multi-mode systems. In *ICCD*, 2016.

[3] S. Baruah and A. Burns. Implementing mixed criticality systems in Ada. In *Ada-Europe*, 2011.

[4] A. Burns. System mode changes-general and criticality-based. In *WMC*, 2014.

[5] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *RTSS*, 1999.

[6] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.

[7] B. Ford and S. Susarla. CPU inheritance scheduling. In *OSDI*, 1996.

[8] S. Goddard and X. Liu. A variable rate execution model. In *ECRTS*, pages 135–143, 2004.

[9] L. Gomez, N. Fairfield, A. Szybalski, P. Nemec, and C. Urmson. Transitioning a mixed-mode vehicle to autonomous mode, Dec 2011. US Patent 8,078,349.

[10] H.-M. Huang, C. Gill, and C. Lu. Implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks. *ACM Trans. Embed. Comput. Syst.*, 13(4s):126:1–126:25, Apr. 2014.

[11] Y. S. Kim and H. W. Jin. Towards a practical implementation of criticality mode change in RTOS. In *ETFA*, 2014.

[12] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig. Flattening hierarchical scheduling. In *EMSOFT*, 2012.

[13] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[14] G. Muller, J. L. Lawall, and H. Duchesne. A framework for simplifying the development of kernel schedulers: design and performance evaluation. In *HASE*, 2005.

[15] M. Neukirchner, K. Lampka, S. Quinton, and R. Ernst. Multi-mode monitoring for mixed-criticality real-time systems. In *CODES+ISSS*, 2013.

[16] L. T. X. Phan, S. Chakraborty, and I. Lee. Timing analysis of mixed time/event-triggered multi-mode systems. In *RTSS*, 2009.

[17] L. T. X. Phan, S. Chakraborty, and P. Thiagarajan. A multi-mode real-time calculus. In *RTSS*, 2008.

[18] L. T. X. Phan, I. Lee, and O. Sokolsky. Compositional analysis of multi-mode systems. In *ECRTS*, 2010.

[19] L. T. X. Phan, I. Lee, and O. Sokolsky. A semantic framework for mode change protocols. In *RTAS*, 2011.

[20] L. T. X. Phan, M. Xu, J. Lee, I. Lee, and O. Sokolsky. Overhead-aware compositional analysis of real-time systems. In *RTAS*, 2013.

[21] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[22] J. D. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems*. PhD thesis, University of Virginia, May 2001.

[23] Y. Shin, D. Kim, and K. Choi. Schedulability-driven performance analysis of multiple mode embedded real-time systems. In *DAC*, 2000.

[24] The Xen project. https://www.xenproject.org.

[25] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multi-core virtual machine scheduling in Xen. In *EMSOFT*, 2014.

[26] M. Xu, L. T. X. Phan, H. Y. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS*, 2016.

[27] M. Xu, L. T. X. Phan, O. Sokolsky, S. Xi, C. Lu, C. Gill, and I. Lee. Cache-aware compositional analysis of real-time multicore virtualization platforms. *Real-Time Systems*, 51(6):675–723, 2015.