Achieving Determinism in Adaptive AUTOSAR

Christian Menard*, Andrés Goens*, Marten Lohstroh† and Jeronimo Castrillon*
* Center for Advancing Electronics Dresden (cfaed), TU Dresden, Dresden, Germany
{christian.menard, andres.goens, jeronimo.castrillon}@tu-dresden.de
† Department of EECS, UC Berkeley, USA
marten@berkeley.edu

***Abstract*—AUTOSAR Adaptive Platform (AP) is an emerging industry standard that tackles the challenges of modern automotive software design, but does not provide adequate mechanisms to enforce deterministic execution. This poses profound challenges to testing and maintenance of the application software, which is particularly problematic for safety-critical applications. In this paper, we analyze the problem of nondeterminism in AP and propose a framework for the design of deterministic automotive software that transparently integrates with the AP communication mechanisms. We illustrate our approach in a case study based on the brake assistant demonstrator application that is provided by the AUTOSAR consortium. We show that the original implementation is nondeterministic and discuss a deterministic solution based on our framework.**

***Index Terms*—automotive engineering, reliability and testing, software and system safety, software engineering**

I. Introduction

Designing and developing software for automotive applications is challenging due to stringent safety and real-time requirements. New use cases like the self-driving car have caused a dramatic increase in complexity and computational demands of automotive software. The AUTOSAR[1] consortium addresses the challenges in industrial automotive software design by standardizing the design process, the runtime environment, and the common software framework. The consortium maintains two standards called *Classic Platform (CP)* and *Adaptive Platform (AP)* that serve different goals and requirements. The former is already established in industry and mostly intended for hard real-time applications with low computational complexity deployed on single-core processors. The latter was introduced more recently in order to handle applications with a high computational demand and the need for maintaining ongoing interaction with a changing environment.

One particular challenge that the automotive industry faces is nondeterminism in the software architecture. A deterministic program yields exactly one behavior given an initial state and inputs, whereas a nondeterministic program may yield many. Nondeterminism may be harmless in some applications while it can lead to unintended and unanticipated behavior in others. In either case, unintended or "accidental" nondeterminism tends to impair testability and negatively impact maintainability of the software. This serves as a compelling argument for allowing nondeterminism only when needed [1]. In safety-critical systems, unintended system behavior could translate into physical damage, injury, or even loss of life. For this reason, the nuclear, aeronautics, and railways industries often rely on synchronous languages like LUSTRE [2], Esterel [3], and SCADE [4] to rule out nondeterminism in their designs of safety-critical software [5].

[1]https://www.autosar.org/

```
int main() {
  s = ServiceProxy();

  s.set_value(1);
  s.add(2);
  result = s.get_value();

  std::cout << result.get();
  return 0;
}
```

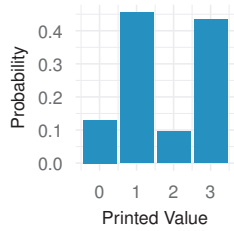

Figure 1. A nondeterministic AUTOSAR Adaptive Platform (AP) client/server application. The client manipulates the server's state variable in a series of (non-blocking) procedure calls. The client prints out one of four different results, distributed as shown in the graph on the right.

In the latest iteration of its well established CP standard, the AUTOSAR consortium introduced support for the logical execution time (LET) paradigm [6], [7]. This can be used to build deterministic software while exploiting the parallelism of multi-core architectures [8]. However, in AP no effective means have been provided for ensuring deterministic execution. On the contrary, AP applications are commonly distributed, and the service-oriented model of AP poses major challenges to the development of deterministic software.

Consider the C++ code in Figure 1 that implements a naive client/server application in AUTOSAR AP. At first glance, C++ being a procedural language, the code suggests that the printed value should be `3`. However, potentially unbeknownst to the programmer that wrote the client, the server implements methods `set_value` and `add` in a non-blocking fashion. And while the server implementation enforces mutual exclusion between the execution of method invocations, by default, the runtime environment maps each invocation to a different thread [9], meaning the order in which the calls are handled is determined purely by the thread scheduler. As a result, no order is enforced on the handling of calls to `set_value`, `add`, and `get_value`, leading to nondeterministic results.

Of course, the client could instead serialize each method call by waiting for the future returned by the server to resolve prior to invoking the next method call; and the server could inform the runtime to use a single thread rather than multiple. However, multi-threading may be necessary to meet performance requirements, yet it is often far from obvious

978-3-9819263-4-7/DATE20/©2020 EDAA

how this may lead to nondeterminism in realistic AUTOSAR applications which are, of course, incomparably more complex than this simple example. Therefore, we argue that the software designer should not be responsible for engineering solutions to concurrency problems in order to achieve determinism. Rather, the underlying model should allow for the exploitation of concurrency in ways that preserve determinism, making it easy to write deterministic programs and requiring explicit directions from the programmer to forgo determinism.

In this paper, we address the lack of an execution model capable of guaranteeing determinism in AUTOSAR AP and show how this can be solved. We make the following contributions:

- We analyze AUTOSAR AP and identify three potential sources of nondeterminism (Section II).
- We propose a solution based on reactors [10], a recently introduced deterministic reactive programming model in which software components are coordinated under a discrete-event semantics. We demonstrate that reactors can integrate with the existing communication mechanisms of AUTOSAR AP and deliver determinism while maintaining compatibility with the standard (Section III).
- We present a case study based on the brake assistant application provided by the Adaptive Platform Demonstrator (APD). We show that this application exhibits nondeterminism that directly translates into problematic behavior. Finally, we describe a deterministic implementation based on reactors that addresses the problem (Section IV).

II. AUTOSAR Adaptive Platform

A. Overview

AUTOSAR Adaptive Platform is a service-oriented architecture (SoA) that is based on a POSIX-compliant operating system. The software stack consists of a middleware that handles communication between services, and the Runtime Environment for Adaptive Applications (ARA) that provides common APIs and services. While the standard does not specify the precise middleware and supports third-party solutions, the AUTOSAR consortium suggests using the SOME/IP protocol [11], [12].

Adaptive AUTOSAR applications are organized in software components (SWCs) that communicate via services that they may provide or request. We call an SWC that provides a service a *server* and an SWC that requests a service a *client*. Client and server roles may be fulfilled by the same SWC. SWCs provide or request services as needed; the binding between clients and servers is determined at runtime by the middleware through service discovery. The dynamic binding of services is the core mechanism for providing adaptivity in AP.

The service interfaces are fully specified at design time and are composed of methods, events, and fields. While events are one-way messages that the server initiates and the client handles, methods are two-way messages that the client initiates and the server responds to. Fields are state variables exposed by the server. Each field may provide a get method, a set method and an event that indicates state changes.

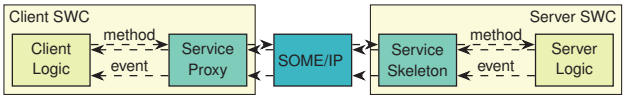

Figure 2. Communication mechanism in AUTOSAR AP. Client and server use auto-generated proxies and skeletons to communicate with their peers.

Figure 2 illustrates the communication mechanisms of AUTOSAR AP. SWCs abstract over the precise middleware by using *proxies* and *skeletons* that are generated from a service description. A skeleton is an abstract interface that a server needs to implement in order to provide a service. A proxy is an object that a client receives when requesting a service. Client and server communicate directly through the proxy and skeleton objects. For instance, the invocation of a method provided by the proxy translates into a message being sent via the middleware to the server, which then translates the message back into a method call. The implementation of the service method is expected to return a future. As soon as the corresponding promise is fulfilled, the server sends a message back to the client.

Applications in AUTOSAR AP commonly consist of multiple SWCs. Each individual SWC can be considered a full program as it is mapped to a process on the target platform during deployment. While the service-oriented communication model of AUTOSAR AP specifies how SWCs interact, it does not specify how SWCs should be implemented. The standard, however, suggests a thread-based coding style.

B. Nondeterminism in AUTOSAR Adaptive Platform

Despite its goal of supporting safety-critical, possibly autonomous applications, AUTOSAR AP uses a model of computation (MoC) that is inherently nondeterministic. We identify three distinct sources of nondeterminism in AP:

1) The suggested programming model for the implementation of individual SWCs is based on threads. Threads, however, make it notoriously difficult to engineer deterministic concurrent software [1]. AUTOSAR AP provides coding guidelines to avoid the problems with threads, but nondeterminism is still likely to creep in, especially when code evolves over time [13].
2) The order in which SWCs process incoming messages is undefined. If two clients call the same method on a service, the calls may be processed in either order.
3) Point-to-point in-order message delivery could be achieved by the middleware and underlying TCP/IP network stack, but this is not a formal requirement in AUTOSAR AP. Even if in-order message delivery is guaranteed, the time required for message transport is still unpredictable.

One provision for deterministic execution that the AUTOSAR AP introduces is the “deterministic client” [14], which provides a task-based programming model for the implementation of SWCs. Because its scope is limited to *individual* SWCs, the solution only addresses the first source of nondeterminism. Applications that consist of multiple communicating deterministic clients can still exhibit nondeterminism via 2) and 3). The solution we outline in the remainder of this paper addresses

all three. It should be noted, however, that the mechanisms for recovering transient errors and ensuring predictable execution time offered by the deterministic client can be combined with the solution we propose.

III. Achieving Determinism in AUTOSAR AP

A. The Reactor Model

SWCs in AUTOSAR AP follow the design pattern of actors, which are concurrent stateful processes that communicate via asynchronous message passing [15], [16]. In the actor model, no constraints are enforced on the ordering of message delivery, but coordination strategies for achieving deterministic actor programs are known [17]. A recently introduced variation of actors, called reactors [10], [18], comprises an execution model based on a discrete events semantics that is deterministic by default, but admits explicit sources of nondeterminism to accommodate sporadic sensors, interrupts, and other nondeterministic components that are indispensable in cyber-physical applications like automotive software.

Unlike actors, communications between reactors occur via events that are associated with tags (also called timestamps). Coordination entails ensuring that all communication between reactors happens in tag order. Reactors are composed out of reactions that can be triggered by input events and may produce output events. The tags of events produced by a reaction are identical to the tags of the events that triggered it; tags denote *logical* time and reactions are logically instantaneous. Reactions can also be triggered by action events, which may emanate from asynchronous resources (e.g., a sporadic sensor) managed within the reactor. Such asynchronously scheduled actions, called *physical actions*, are tagged based on the last observed *physical* time, potentially with an additional delay.

The tag of a physical action can be considered an input to the reactor system. Hence, while physical actions may occur unexpectedly, their occurrence does not compromise the deterministic operation of the system—unless physical actions are used as end points for a communication channel between reactors in the same system. Additional coordination is necessary to preserve determinism in that case. Precisely because actions can be scheduled sporadically, to ensure in-order processing of events, no events are handled before physical time exceeds their tag.

Another difference with actors is that reactors only communicate to one another via channels that connect reactor ports. The communication topology of a reactor program translates into an acyclic precedence graph (APG) that drives the execution. A reactor runtime scheduler is responsible for transparently exploiting concurrency in the APG by mapping independent reactions to separate worker threads. Reactors are capable of performing efficient multi-threaded computation, while the programmer is freed from the burden of managing the interactions between threads, effectively addressing the first source of nondeterminism discussed in Section II.

Because events are timed, the reactor model also allows for the expression of deadlines, making it suitable for specifying programs with real-time constraints. A deadline D is considered violated when an event with tag t triggers a reaction associated with D *after* physical time T has exceeded $t + D$.

Importantly, reactors can also be coordinated deterministically across SWCs, by leveraging safe-to-process analysis known from PTIDES [19], [20]. This addresses the second and third source of nondeterminism discussed in Section II. This approach assumes that each network interface has an associated deadline D. By further assuming that distributed reactors have synchronized physical clocks with a bounded clock synchronization error E (this is the case in AP [21]) and their communication has a bounded latency L, in-order event handling can be assured. Specifically, when a reactor receives a message with tag t from the network, it has to schedule an action with tag $t + D + L + E$ in reaction to which it can later safely output the message. The physical time delay enforced by the scheduler ensures that no message with a timestamp smaller than t is still expected to arrive over the network.

We propose to use reactors as a programming model for designing deterministic SWCs and attach tags to communications between SWCs to allow for the preservation of the deterministic discrete event semantics of reactors across SWCs.

B. Integrating Reactors with AUTOSAR AP

We created a framework called DEAR[2] (Discrete Events for AUTOSAR) that provides a C++ implementation of the reactor model. It implements type-safe mechanisms for the definition of reactors with ports, actions and reactions. This also includes mechanisms for composing reactors to form deterministic programs. The framework further provides an implementation of the runtime scheduler to coordinate the execution of the reactor network. This establishes a foundation for the design and execution of deterministic SWCs.

In order to enable composition of deterministic applications from deterministic SWCs, a mechanism is needed for transporting tagged messages in AUTOSAR AP so that safe-to-process analysis can be performed prior to inserting events into the receiving reactor network. This is challenging since the standard for AUTOSAR AP explicitly specifies the interface that SWCs use for communication. Exposing reactor ports directly to the interface of SWCs would break compatibility with the standard. We can work around this by introducing transactors that translate between the service-oriented interfaces of SWCs and the event-based input and output ports of reactors.

DEAR provides four distinct transactors, each implemented as a reactor and enabling the composition of reactors through regular AUTOSAR service interfaces. The *client method transactor* interacts with a given method of a service interface in the client role. Similarly, the *server method transactor* interacts with a method in the server role. Analogous to methods, a similar pair of transactors for interacting with AP events in the role of clients and servers exists. Since fields are composed of a get method, a set method and an event, interaction with fields requires the use of one event and two method transactors. All four transactors are shown in Figure 3. Given

[2]Available at: https://github.com/tud-ccc/dear

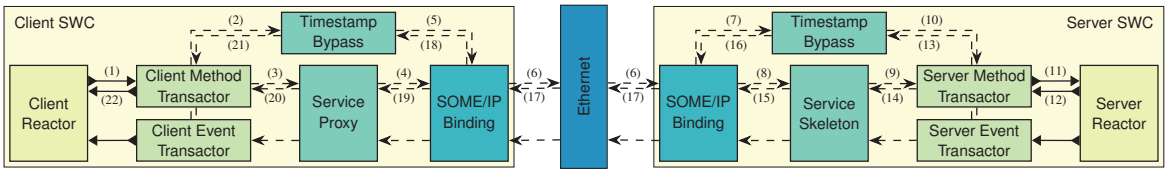

Figure 3. Integration of reactors in AUTOSAR AP. Special reactors (transactors) translate between the reactor implementation of the SWC logic and the service interface that the SWC exposes to its environment.

a service interface, the transactors required for interacting via this particular interface can be automatically generated.

AUTOSAR AP provides no mechanisms for associating metadata like reactor tags with method calls or events. This, however, is required for the transmission of tagged messages between deterministic SWCs. Therefore, we modified the library that binds to the SOME/IP middleware to optionally append tags to outgoing messages and to retrieve tags from incoming messages if available. This modification is not in violation of the standard. It can be seen as the introduction of a new third-party middleware that extends over SOME/IP by allowing the transmission of tagged messages. While the transactors use the regular AUTOSAR AP service proxies and skeletons, for each event occurrence or method call they store a corresponding tag that is picked by the modified SOME/IP middleware prior to transmitting the payload over the network.

The entire process of transmitting tagged messages between SWCs in DEAR is illustrated in Figure 3. The sequence starts with a client that invokes a method call on a remote service. In the reactor implementation, that corresponds to producing an event with tag t_c on the output port connected to the request input port of the client method transactor (1). This input port has a configurable deadline D_c. If the deadline is not violated, the corresponding reaction sends $t_c + D_c$ to the timestamp bypass (2) and invokes the actual method call on the service proxy object (3). Thereby, it forwards the data associated with the incoming event as method arguments. The service proxy calls the SOME/IP binding (4) to prepare a network message. Our modified binding retrieves $t_c + D_c$ from the timestamp bypass (5) and attaches it to the SOME/IP message, which it then sends over the network to the server (6).

Upon receiving the network message on the server side, our modified SOME/IP binding retrieves $t_c + D_c$ from the message and sends it to the local timestamp bypass (7) before invoking the corresponding method call on the service skeleton (8). This method call triggers an interrupt of the server method transactor (9) which retrieves $t_c + D_c$ from the timestamp bypass (10) and schedules an action with tag $t_c + D_c + L + E$ accounting for the worst-case network latency L, as well as the maximum clock skew E between platforms. The reaction to this action produces an event on the output port that forwards the method arguments to the reactor that implements the server logic (11). The server logic then reacts to this event.

The server eventually sends a response by producing an event with tag t_s ($t_s \geq t_c + D_c + L + E$) on the output port connected to the input port of the service method transactor (12). This input port has a configurable deadline D_s. If the deadline is not violated, the corresponding reaction sends $t_s + D_s$ to the timestamp bypass (13) and returns the data associated with the event to the service skeleton (14). The service skeleton calls the SOME/IP binding (15) to create a response message. The binding retrieves $t_s + D_s$ from the timestamp bypass (16) and attaches it to the outgoing message which it then sends over the network to the client (17).

The client SOME/IP binding retrieves $t_s + D_s$ from the message and sends it to the timestamp bypass (18) while forwarding the return value to the service proxy (19). The arrival of the return value triggers an interrupt in the client method transactor (20) which retrieves $t_s + D_s$ from the timestamp bypass (21) and schedules an action with tag $t_s + D_s + L + E$ to again account for transmission latency and clock synchronization error. Finally, the reaction for this action produces an event on the output port of the transactor (22).

While transparently enabling deterministic composition of reactor-based SWCs, our solution leaves open the possibility of composing reactor-based SWCs with regular service implementations that do not communicate via tagged signals. The default behavior of our transactors is to fail when receiving messages without an associated timestamp, but they can also be configured to tag received messages with the physical time at which they are received. This approach treats the arrival of untagged messages the same way as reactors deal with the arrival of sporadic sensor readings. This essentially furnishes backward compatibility with existing service implementations and the ability to gradually introduce reactor-based SWCs.

IV. Case Study: Adaptive Platform Demonstrator

A. Nondeterministic Brake Assistant

The AUTOSAR consortium provides the Adaptive Platform Demonstrator (APD), which is an example implementation of the specification for AUTOSAR AP. It provides a set of demo applications, where the most realistic and advanced application is the brake assistant shown in Figure 4. Unlike what may be expected of a safety-critical application, the brake assistant exhibits nondeterminism that could potentially have fatal consequences. While this demo is not designed for deployment in the real world, it illuminates the presence of uncontrollable and safety-undermining nondeterminism in AP.

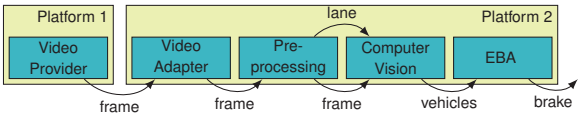

Figure 4. Brake assistant application in APD

The brake assistant consists of a pipeline of five SWCs, distributed across two platforms. Video Provider captures video frames and sends one approximately every $50\,\mathrm{ms}$ (via a proprietary protocol) to Video Adapter, which is running on the second platform. The communication along the remainder of the component chain occurs through AP service interfaces via the SOME/IP middleware. Event notifications are used to transfer data from one SWC to the next and the corresponding event handler stores the data in a one-slot input buffer. Each SWC sets up a periodic callback so that the OS triggers the SWC logic every $50\,\mathrm{ms}$. Each component then reads the current data item from its input buffer, performs some computations, and then communicates the result via an event.

For each frame Preprocessing receives from Video Adapter, it computes a bounding box demarcating the current travel lane. Computer Vision receives from Preprocessing both the lane information as well as the original frame and uses it to detect vehicles in the lane and estimates their distance. It forwards the list of detected vehicles to the EBA component, which in turn decides whether an emergency brake maneuver is required.

The logic of each component processes the last data written to its one-slot input buffer. If there is no data, the SWCs silently stop computation and wait for the next periodic trigger to occur. This introduces nondeterminism as data could get overwritten before it is read by a downstream component, causing entire frames to be dropped. Moreover, since the Computer Vision component reads not one but two inputs, this can lead to misalignment between the video frames and the lane information. We instrumented the brake assistant code to detect and report frame loss and misalignment. Execution on our evaluation platform, consisting of two MinnowBoard Turbot[3] boards connected via an Ethernet switch, confirmed that the described errors indeed occur in a real-world setting. The boards are equipped with an Intel Atom E3845 Quad-Core processor and are officially supported by the APD.

We performed a series of experiments to analyze the prevalence of the described errors. We let the brake assistant processes a total of 100,000 frames and counted dropped inputs and mismatches. Figure 5 plots the obtained results for a total of 20 experiments. Each bar in the figure shows the error prevalence for one instance of the experiment. The results are ordered by error rate for better visibility.

The error rate varies significantly between experiments. In the best case we observed an error rate of 0.018% and in the worst case an error rate of 22.25%. On average we observed 5.60% errors. Also the composition of error types varies significantly. In most experiments frame dropping at Computer Vision was dominant, but sometimes dropped vehicles at EBA or dropped frames at Preprocessing dominated. This underlines the difficulty of assessing the performance and correctness of the brake assistant. It appears the error rate is strongly influenced by the offset between the individual periodic callbacks of the SWCs, which depends on when SWCs are started and is difficult to control.

[3]https://minnowboard.org/minnowboard-turbot-dual-e/technical-specs

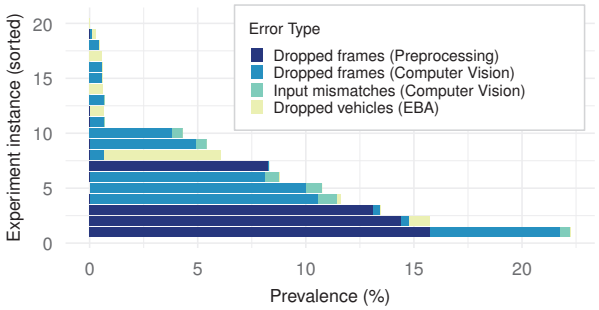

Figure 5. Prevalence of errors for 20 executions of the brake assistant.

B. Deterministic Brake Assistant

DEAR allows us to easily transform the brake assistant application into a deterministic reactor implementation. Since the original implementation separates computational logic from the communication mechanism, transformation requires only a few code changes. We encapsulate the logic of each SWC in a reactor that has one reaction to process incoming events. This reaction calls the original logic to process the data associated with the incoming event and produces an output event. In order to support the transmission of tagged messages between SWCs, each reactor binds to the service interfaces of the SWC using the DEAR transactors. As described in Section III, a carefully chosen deadline on the reaction of each sending transactor ensures that there is an upper bound on how much logical time lags behind physical time. The receiving transactor further accounts for the physical delay of message transmission and ensures that incoming messages are only processed when it is safe to do so.

Since Computer Vision has two inputs, the reaction that calls its logic expects to receive two events with the same tag at both inputs. If only one input is received, this is considered an error. Video Adapter has no well-defined input. It sporadically receives frames over the network sent by the camera. As the timing of the camera cannot be controlled, we implement Video Adapter as a sensor that inserts frames into the reactor network with a tag equal to the physical time of message reception. Once the incoming frame is tagged, subsequent reactions are carried out in a deterministic order.

In order to achieve correct execution, it is important to carefully consider the physical delays imposed by the computations of each SWC as well as by the transport of messages between SWCs. Only if the deadlines associated with each SWC account for its WCET and the specified maximum communication latency and synchronization error are accurate, correct execution is guaranteed. In our implementation, we set the deadlines to $5\,\mathrm{ms}$ for Video Adapter, $25\,\mathrm{ms}$ for Preprocessing, $25\,\mathrm{ms}$ for Computer Vision and $5\,\mathrm{ms}$ for EBA. We further assume a maximum communication latency of $5\,\mathrm{ms}$. Since all SWCs of this application are deployed to the same platform, there is no clock synchronization error to account for. Note that these numbers are estimated upper bounds of delays. More precise values can be obtained from WCET analysis.

With this implementation, we achieve correct and deterministic execution on the MinnowBoard platform. Moreover,

the timed semantics of reactors facilitates reasoning about the worst-case end-to-end latency between receiving a frame and producing an output brake signal. These benefits come at the cost of an extra physical time delay as each SWC needs to account for worst case computation and communication delays. This, however, is not a necessity. For certain applications it is acceptable to deliberately introduce the possibility of sporadic errors by setting deadlines to values lower than the actual WCET. Independent from how deadlines and communication delay are chosen, the reactor semantics guarantees determinism and translates any violation of one of the assumptions directly into observable errors. In contrast to the original brake assistant implementation, the trade-off between end-to-end latency and error rate becomes apparent.

V. Related Work

We are unaware of any publications that address determinism in AUTOSAR AP. In AUTOSAR CP, deterministic execution can be achieved based on the LET paradigm [6]–[8], which can also be extended for use in distributed applications [22]. However, while LET is compatible with the task-based programming model of AUTOSAR CP, it is not easily applicable to the adaptive and reactive programming required in AUTOSAR AP. In particular, LET is a real-time programming paradigm where logical time strictly matches physical time at each task invocation and termination. The reactor semantics, in contrast, provides for a more flexible application design where real-time constraints are only enforced when this is explicitly indicated by a deadline. Time violations are not treated as system failure but become observable errors that can be handled as appropriate for the given application. Another difference is that, while LET tasks always take a non-zero amount of logical time, reactions are logically instantaneous and thus compose without requiring explicit alignment.

Synchronous languages like LUSTRE [2], Esterel [3], and SCADE [4] have been designed for the development of complex reactive systems and are completely deterministic. While the semantics of reactors is also a synchronous one, reactors provide an additional coupling to physical time that allows for the specification of real-time requirements and can be used for deterministic distributed execution without a central coordinator.

VI. Conclusion

We have shown that AUTOSAR AP exhibits nondeterminism in the core elements of its architecture. Our case study of a brake assistant demonstrator application exposes that this could lead to serious malfunction. To address this, we propose to use reactors as programming model for the design of applications in AUTOSAR AP. We introduce the DEAR framework that effectively enables reactor-based programming and coordination between distributed components as shown in our case study. While our approach warrants standard compatibility, we advocate for an extension of the standard that obviates the need for the workarounds we implemented to associate method calls and events with tags.

Acknowledgment

We thank Edward A. Lee for his feedback on an earlier version of this paper. This work was supported in part by the KMU-innovativ project EM-RAM funded by German Federal Ministry of Education and Research (BMBF). The work was further supported in part by the National Science Foundation (NSF), award #CNS-1836601 and the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by Camozzi Industries, Denso, Ford, Siemens, and Toyota.

References

[1] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[2] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[3] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.

[4] G. Berry, "SCADE: Synchronous design and validation of embedded control software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, S. Ramesh and P. Sampath, Eds., Dordrecht: Springer Netherlands, 2007, pp. 19–33.

[5] J.-L. Boulanger, F.-X. Fornari, J.-L. Camus, and B. Dion, *SCADE: Language and Applications*, 1st. Wiley-IEEE Press, 2015.

[6] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*, S. Chakraborty and J. Eberspächer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 103–120.

[7] AUTOSAR, "Specification of timing extensions," *AUTOSAR CP Release 4.4.0*, Oct. 2018.

[8] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the LET paradigm," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2018, pp. 240–250.

[9] AUTOSAR, "Specification of communication management," *AUTOSAR AP Release 19-03*, Mar. 2019.

[10] M. Lohstroh, M. Schoeberl, A. Goens, *et al.*, "Actors revisited for time-critical systems," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19, Las Vegas, NV, USA: ACM, 2019, 152:1–152:4.

[11] AUTOSAR, "SOME/IP protocol specification," *AUTOSAR FO Release 1.5.0*, Oct. 2018.

[12] ——, "SOME/IP service discovery protocol specification," *AUTOSAR FO Release 1.5.0*, Oct. 2018.

[13] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu, "What change history tells us about thread synchronization," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: ACM, 2015, pp. 426–438.

[14] AUTOSAR, "Specification of execution management," *AUTOSAR AP Release 19-03*, Mar. 2019.

[15] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial intelligence*, vol. 8, no. 3, pp. 323–364, 1977.

[16] G. A. Agha, "Actors: A model of concurrent computation in distributed systems," MIT Artificial Intelligence Lab, Tech. Rep., 1985.

[17] M. Lohstroh and E. A. Lee, "Deterministic actors," in *2019 Forum on Specification and Design Languages (FDL)*, IEEE, 2019.

[18] M. Lohstroh, Í. Íncer Romeo, A. Goens, *et al.*, "Reactors: A deterministic model for composable reactive systems," in *Model-Based Design of Cyber Physical Systems (CyPhy'19)*, 2019, To appear.

[19] Y. Zhao, J. Liu, and E. A. Lee, "A programming model for time-synchronized distributed real-time systems," in *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, Apr. 2007, pp. 259–268.

[20] P. Derler, T. H. Feng, E. A. Lee, *et al.*, "PTIDES: A programming model for distributed real-time embedded systems," EECS Department, University of California, Berkeley, Tech. Rep., May 2008.

[21] AUTOSAR, "Specification of time synchronization for adaptive platform," *AUTOSAR AP Release 19-03*, Mar. 2019.

[22] R. Ernst, L. Köhler, and K.-B. Gemlau, "System level LET: Mastering cause-effect chains in distributed systems," in *44th Annual Conference of the IEEE Industrial Electronics Society (IECON)*, Oct. 2018.