Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds.

SEAMLESS SIMULATION-BASED VERIFICATION AND VALIDATION OF EVENT-DRIVEN SOFTWARE SYSTEMS

Tom Meyer
Philipp Andelfinger
Andreas Ruscheinski
Adelinde M. Uhrmacher

Institute for Visual and Analytic Computing
University of Rostock
Albert-Einstein-Straße 22
Rostock, 18059, GERMANY

ABSTRACT

Verification and validation (V&V) are essential concerns in the development of safety-critical distributed software systems. V&V efforts targeting full system implementations rely on testing, which requires real-world deployments and cumbersome analysis to track down issues across distributed software components. Here, we propose a simulation-based development and testing framework for distributed systems following the event-driven architecture (EDA) paradigm. During development, unmodified software components can be executed in their interaction with a simulated environment, allowing for early testing under envisioned deployments. After introducing the interplay of EDA and discrete-event simulation, we present our framework's architecture and the API offered to software components, which closely follows accepted EDA principles. We demonstrate the use of our framework on a medical software system used in the diagnosis of rare genetic diseases. By observing the system's interaction with simulated laboratories, the feedback loop between diagnoses by laboratories and classifications from the software system is evaluated.

1 INTRODUCTION

Software verification and validation (V&V) methods (Rodriguez et al. 2019) are essential parts of the development process of safety-critical software systems, e.g., in medicine (Arcaini et al. 2015). However, these methods struggle to capture the complex interactions among components that are characteristic for modern distributed software systems. For instance, in a self-adaptive software system (Cheng et al. 2008), a component's behavior may change dynamically based on a feedback loop involving other components and an external environment. While models of the system and environment may allow for design-time verification of desired properties, the system's implementation may inadvertently diverge from the design. On the other hand, unit and integration testing typically does not consider complex interactions with the environment. Runtime V&V can detect violations of the expected system behavior during deployment (Tamura et al. 2013), but is insufficient if the consequences of violations in a real-world context cannot be tolerated.

In the present paper, we propose a simulation-centric development approach for distributed software systems, wherein V&V efforts can target real implementations of the components, which are executed in virtual time and interact with a model of the environment (e.g., the users of the system). We target systems that follow the *event-driven architecture* (EDA) paradigm, in which loosely coupled components communicate asynchronously in the form of *notifications* signifying *events*, i.e., significant changes in state (Chandy 2006). This approach maps naturally to *discrete-event simulation* (DES), in which simulated entities evolve along a virtual timeline, by skipping from one instantaneous state change to the next (Law 2013). The use of simulation enables V&V with respect to envisioned deployments at various scales and

978-1-6654-7661-4/22/$31.00 ©2022 IEEE

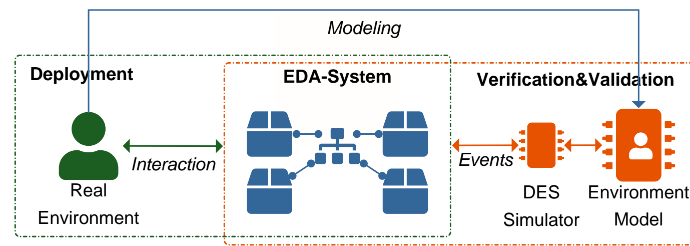


Figure 1: Our approach: an EDA system implementation is executed either in a simulation for V&V (orange box), or in a real-world deployment (green box). During V&V, unmodified components interact with a model of the environment.

under arbitrary influences from the environment. In particular, feedback loops between the EDA system and the environment can be studied without the cost and safety concerns of real-world testing. Importantly, the component code tested in the simulation can be deployed to production without modification (cf. Figure 1).

Related Work Existing approaches for executing distributed software systems in a simulation modify or substitute operating system functionalities to orchestrate the execution of the software components and the time advancement (Jansen and Hopper 2012; Jin et al. 2014; Lamps et al. 2014). We take a different approach, enabling a user-level implementation by making use of the natural fit of EDA systems to the DES paradigm. The closest existing work targeting EDA systems proposes modeling and simulation languages for EDA systems (Clark and Barn 2011). In contrast, we allow components to be implemented using arbitrary general-purpose programming languages and software libraries.

2 SIMULATING EDA IN VIRTUAL TIME

Our goal is the execution of a real EDA *system* in its interaction with either a real or a simulated *environment*, with minimal changes required for transitioning between the two modes of execution. The use of simulation makes it possible to parameterize the envisioned deployment of the system (e.g., execution speed) as well as the environment (e.g., the distribution of incoming user requests). In this setting, methods from experiment design (Kleijnen 1998; Sanchez et al. 2020) can be applied, for example, to systematically assess the behavior of the system when varying parameters of the environment.

In the following, we describe how we connect concepts from EDA and DES to execute unmodified EDA components interacting with a simulated environment. In particular, this involves the translation of placement of occurrences in the EDA system on a virtual timeline, which requires EDA events to be dynamically diverted to a central simulator. Further, we detail how distributed EDA systems are imitated by sequential simulations and describe the constraints that must be satisfied to achieve correctness.

2.1 Overview

In the following, we introduce the central concepts of our approach.

Event-Driven-Architecture (EDA) In EDA, loosely coupled components *observe* events and notify other components about events using messages (Eugster et al. 2003; Mühl et al. 2006). An event is a “significant change in state” (Chandy 2006), whereas the corresponding *notification* message is used to signal the occurrence of the event to other components. Notifications are disseminated by the *notification service (NoSe)* using a publish-subscribe (pub/sub) API (Mühl et al. 2006), in which notification messages are published on a topic and all subscribers to this topic are informed.

Discrete-Event Simulation (DES) In DES, events are used to control the operation of the simulation. A modeled system is simulated by changing the state of *simulation entities* at discrete time points (Law 2013). These time points, signified by events, occur in *virtual time*. Pending events are stored in an *event queue*. The simulator advances virtual time by iteratively dequeueing and executing the event with the lowest timestamp. This execution may update the state of simulation entities and schedule new events by

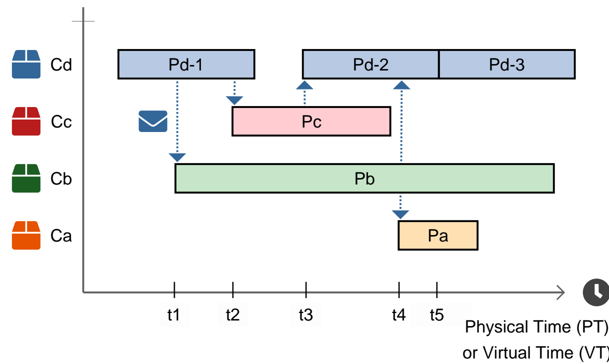


Figure 2: The system's execution during simulation is based on a mapping of physical time to virtual time via a time model.

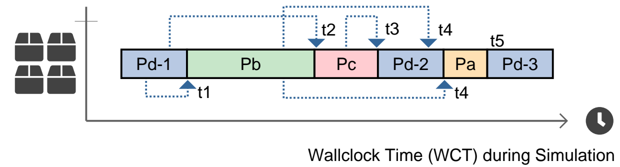


Figure 3: During simulation, processes (which may overlap in virtual time) are executed sequentially in wall-clock time.

inserting them into the event queue. In this manner, virtual time advances from one event to the next in timestamp order, until the event queue is empty or a termination condition is met.

Both in EDA and in DES, events represent changes in state. However, in DES the state changes are *triggered* directly by the simulator, while the focus in EDA is on *observing* and communicating them.

Processes To enable the execution of EDA systems in a DES context, we employ the notion of *processes*. We define a process (bar in Figure 2) as a non-preemptable reaction of a component to a single notification (arrow in Figure 2). Hence, a process represents the execution of component code as a consequence of receiving a notification. Over the course of a running process, new notifications may be published. Components follow this process-based mode of execution independently of whether the EDA system is embedded in a simulation or deployed in the real-world (Figure 1). This notion of a process is roughly analogous to a "service" in service-oriented and microservice architectures.

Notions of Time To temporally relate a sequence of state changes observed in a real EDA system to those observed in a simulation, three different notions of time must be distinguished (Fujimoto 2000). In a real-world deployment, the EDA system and its various processes execute in *physical time* (PT). In contrast, the advancement of time in a simulated execution of an EDA system or process occurs with respect to *virtual time* (VT). During simulation, PT is mapped to VT based on a time model. The execution of the simulation itself requires real-world processing. The time needed to do so is measured in *wall-clock time* (WCT), which is the ordinary notion of time observed in the real world.

Translation between Timelines Figures 2 and 3 contrast the execution and simulation of an EDA system on the different timelines. In the descriptions and our prototypical implementation, the simulation is assumed to be sequential, although parallel or distributed simulation (Fujimoto 2000) could also be applied. In the following, we describe the translation of overlapping EDA processes to a sequential simulation.

Suppose for simplicity of the illustration that processing times in the simulation are chosen to be identical to those observed in a real-world deployment. Then, Figure 2 represents the desired behavior as a mapping from points in PT at which events and processes occur in the real deployment to the same points in VT during a simulation run. The simulation imitates the overlapping execution of processes in VT, while serializing the process execution in WCT (cf. Figure 3). In WCT, the process with the lowest timestamp is always executed to completion before the next process is executed. For instance, processes Pd-1 and Pb are executed sequentially in WCT, whereas in VT, their execution overlaps as it does in PT.

Constraints Our approach puts only mild constraints on the EDA system under development. To guarantee that the component behavior during simulation matches the behavior in a real-world deployment, we require that components do not share memory directly, i.e., all communication among processes relies on notifications. Further, our approach requires the creation of a model of the environment. These constraints are detailed further in Section 3.

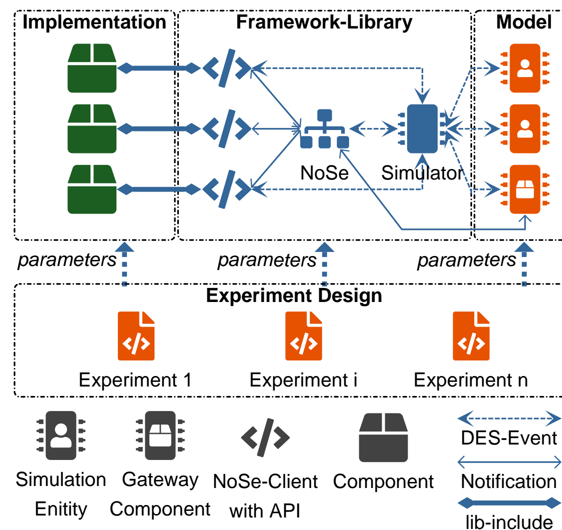


Figure 4: The overall system is composed of a reusable framework library with API, notification service (NoSe) and simulator (blue), a model of the environment (orange), and components, which communicate via the NoSe API (green). The parameters used in individual simulation runs are configured as part of the experiment design.

2.2 Architecture

Figure 4 shows the building blocks of the overall system and their interaction during a simulation study. The EDA components employ a client library with a pub/sub API to interface with a central notification service (NoSe). The implementations of the client library and NoSe are central to the simulation-based execution. As in real world deployment, the NoSe keeps track of subscriptions to the communication channels and routes notifications to the respective subscribers. The client provides an interface to produce notifications for detected events and consume and process events received from a subscribed channel. However, to translate the execution to virtual time, published notifications are first sent to the simulator, which enqueues the notifications as timestamped events in its event queue. The time stamp is calculated according to the time model. Since EDA is a distributed system architecture, we cannot make assumptions on the locations where components are executed. Thus, all communication among components and with the simulator requires the message data to be serialized, reflecting the loosely coupled nature of the EDA concepts. The notifications are only propagated to the receivers once virtual time has advanced to the respective events. We go into more detail on the notification dissemination in Section 2.4.

The environment is represented by a simulation model, which generates and processes events to interact with the EDA system. The model of the environment can be designed freely depending on the requirements of a given verification and validation study. A particular benefit of our simulation-based approach is the opportunity to include complex environment models in order to study feedback loops between EDA system and environment.

Considering the EDA system, the only components that must be modeled are *gateway components*, which act as the interface between the environment and the EDA system (e.g., web front-ends). A model of a gateway component translates simulation events generated by the environment (e.g., user accesses to a website) to simulated EDA events, and vice versa. Apart from gateway components, all EDA components are executed using their unmodified implementations.

To carry out a simulation study, parameters of the environment model as well as the time model that maps process durations from physical time to virtual time are configured and may be varied systematically using established experiment design techniques (Kleijnen 1998; Sanchez et al. 2020).

2.3 Simulation Events

The simulation is controlled entirely using three types of simulation events:

1. Environment events,
2. API events, and
3. Control events.

All these events are exclusively used in the communications (i.e., via socket) between EDA modules (components and NoSe) and the simulator. All events hold a timestamp in virtual time and a data field.

Environment events represent occurrences in the environment. They can originate from the environmental model as well as the EDA system during an interaction with the environment via the modeled gateway. The timestamps of environment events that stem from the EDA system are set based on the advancement of virtual time in the EDA system. An event's data field can be used to hold arbitrary data to be interpreted by the modeled gateway component or the environment model.

API events are used in the EDA communication between NoSe and components. The canonical events include notifications, subscriptions and unsubscriptions. However, other broker protocol messages such as connection handshakes could be included as well. We recall that an EDA event is a state change observed by a component. A representation of the EDA event is disseminated to other components as a *notification* message. The main API event represents this dissemination by encapsulating the notification with a timestamp in virtual time, which is generated either implicitly or based on a component-specific measure of progress by our library (see Section 3). The resulting simulation event is then rerouted to the simulator. The API event's data field carries the original notification message consisting of header and body with an identifier of the notification's publisher component in the header. Once the notification event is executed by the simulator, the publisher continues execution. Subsequently, only the notification itself is needed to continue, whereas the encapsulation is discarded for compatibility with the communication flow in a real-world deployment. Other API events are routed through the simulator in the same fashion (see Section 2.4 for a more detailed description of the dissemination process).

Finally, to schedule and serialize processes correctly, simulator, NoSe and NoSe clients schedule *control events*. For instance, if a notification arrives at a component that is blocked due to a currently running process, the delivery of the notification must be postponed to a later point in virtual time. Consider the scenario from Figure 2:

1. Component Cc sends a notification to component Cd, which starts a process Pd-2 at component Cd at virtual time t3.
2. Component Cb sends another notification while Pd-2 is still running at virtual time t4. Assuming component Cd can only execute one process at a time (as we require for state isolation; see Section 3.), this notification is blocked and cannot start a new process.
3. The notification from Component Cb has to be requeued to a time, i.e., virtual time t5, when Pd-2 will have completed.

In a real-world deployment, the NoSe handles (re-)dissemination of notifications to the components. However, in a simulation run, event dissemination is controlled by the simulator and triggered by simulation events. Hence, a new simulation event is required for dissemination retries (from blocked events), which can be achieved by generating a control event that reschedules notifications within the NoSe's event queue.

2.4 Implementation Specifics

While developing a prototype of the framework, we discovered some details that are common for any implementation. Here we discuss these framework specifics excluding DES implementation details.

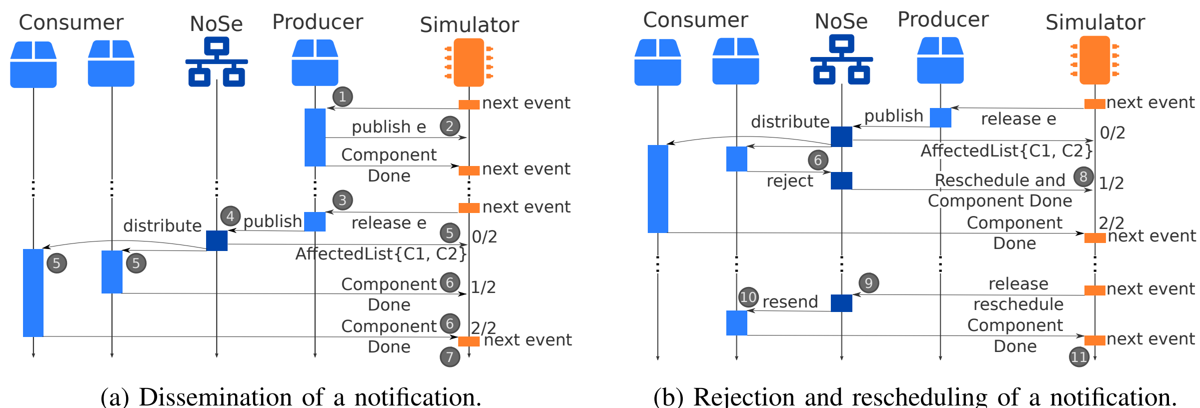
Initialization Since the EDA components and the NoSe are distributed, the initialization must enable communication between the components and the simulator. Since the simulation experiment determines

the involved components, the starting of components is initiated by the simulator. In contrast, due to the simulator's centrality, the communication connections to the simulator are initiated by the NoSe and the individual components. Additionally, since components are likely to attempt to connect to the NoSe during startup, the NoSe is initialized before any component is started. Only after all components have signaled the completion of initialization, the simulator begins executing events.

Component synchronization Since all events are controlled by the simulator, it must be aware of all components known to the NoSe. Based on this knowledge, notifications are disseminated as follows. The simulator executes a notification event which starts a process in a component (step 1 in Figure 5a). During the process another notification is published (2) which is scheduled as a simulation event by the component client before sending it to the NoSe. This event is dequeued later in the simulator (in the meantime other events might be scheduled) and a simulation event informs the client to continue (3). Now, the conventional EDA execution flow continues, and the client sends the event to the NoSe (4) as an API event. From there, notifications are disseminated as API events to all subscribed components (5). Simultaneously, a list of all recipients is sent to the simulator (same step, also 5) to track which components need to signal their completion. All components that accept the event execute the corresponding process and send an event signaling completion to the simulator (6). Once all components in the affected list of step 5 have responded, the current simulation step is finished, and the next event can be dequeued (7). Components that are blocked due to running processes send a rejection to the NoSe (6 in Figure 5b) instead of a completion event to the simulator. Rejected events are rescheduled in the simulator, and a completion event in the name of the rejecting component is sent (8). When a previously rejected event is executed by the simulator, it is once again sent to the NoSe (9) where it is forwarded to the blocking component (10). If the component is not busy anymore, the corresponding process is executed and a completion event is sent to the simulator (as in 6 in Figure 5a). Once the completion event has been received, the simulator can execute the next event (11).

3 SYSTEM DEVELOPERS' PERSPECTIVE

Time Model. The execution of an EDA system in virtual time allows a time model to be chosen as part of the experiment design. The time model maps the progress of processes to virtual time. For instance, the durations may be determined based on the processing speed expected from the hardware to be used once the system is deployed. However, to achieve transparency, the process durations can also be measured on the fly on the hardware that executes the simulation, by using the process durations observed in wall-clock time as the virtual durations. The point in virtual time at which a notification is published can then be calculated as $VT_{\text{notify}} = WCT_{\text{notify}} - WCT_{\text{process}} + VT_{\text{process}}$, where $WCT_{\text{notify}} - WCT_{\text{process}}$ is the duration from process start to the notification in wall-clock time, and VT_{process} is the process start time in virtual



(a) Dissemination of a notification.

(b) Rejection and rescheduling of a notification.

Figure 5: The main API events during dissemination of notifications in simulation mode.

time used as an offset. Of course, durations can also be scaled to experiment with faster or slower hardware in the envisioned deployment. As a downside of relying on processing durations measured in wall-clock time, the simulation results become machine-dependent and non-deterministic. An alternative is to record application-specific measures of progress within components, (e.g., the number of database records updated as a reaction to an event) based on which process durations can be calculated deterministically according to a chosen time model. To offer these modes of time advancement, we slightly extend the traditional pub/sub API (Mühl et al. 2006) (see Figure 6). Optional arguments allow a component to quantify the progress made throughout the current process, from which the NoSe client computes a corresponding virtual time according to the time model.

These functions cover all basic pub/sub functionality. However, in future work we plan to extend the API to include the configuration of quality of service options and callbacks for error handling.

State Isolation To maintain consistency between the components' states in the simulation and a real-world deployment, processes of separate components cannot share memory directly; instead, all state must be shared via notifications. Similarly, concurrent processes within a single component cannot share state. Due to the serialization of processes during simulation, any race conditions that would be observed in the deployed system would otherwise not be reflected in virtual time. For this reason, we prohibit component level concurrent processes entirely and require the blocking behavior of components.

Environment Modeling The discrete-event model used to represent the environment in the simulation can be designed freely according to the needs of the verification and validation study. To closely represent environment behavior that may be observed during deployment, the model should be designed to follow real-world data, e.g., from surveys or measurements. As a trivial example, the environment model could simply replay user requests recorded in a previous deployment of the system, or generate requests according to a probability distribution fitted to the data. However, a key benefit of the simulation-based approach is the capability of representing *feedback* between the EDA system and a dynamic environment. For instance, user requests may depend on the responses received from the EDA system as reactions to prior requests. The model developer may also include hypothetical behavior in the environment model in order to carry out so-called *what if* analyses (Banks 2000), e.g., to study the system's behavior under bursts of user requests or the robustness of the system when faced with erroneous input data. The creation of the model can draw on established methods from the field of modeling and simulation (**?**).

4 CASE STUDY

It is assumed that 350 million people worldwide are affected by rare diseases (Faviez et al. 2020). As most of these are of genetic origin (Faviez et al. 2020), whole-exome sequencing (WES) has become the standard tool for diagnosing such diseases. The diagnosis process can be roughly outlined as follows (Quintáns et al. 2014): A physician sends a genetic sample from the patient, e.g., blood, to a genetic laboratory

```
1  interface NoSeClient
2   // connect to NoSe at 'address'
3   fn connect(address[, progressMeasure]);
4   // disconnect from NoSe
5   fn disconnect([progressMeasure]);
6   // disseminate observed event
7   fn publish(notification[, progressMeasure]);
8   // subscribe to a filter to receive notifications
9   fn subscribe(filter[, progressMeasure]);
10  // register callback function for received notifications
11  fn register_notify(notificationCallback);
12  // unsubscribe from previous subscription filter
13  fn unsubscribe(filter[, progressMeasure]);
```

Figure 6: API as pseudocode.

that prepares the sample and sequences the patient's exome. A reporter subsequently reviews the genetic variants, i.e., the deviations of the patients' DNA from a reference genome, to identify the disease-causing variants and reports the findings to the physician.

However, in a typical WES case, 25 000 to 75 000 (Negishi et al. 2015) variants have to be narrowed to only a handful of disease-causing variants. Variant prioritization methods aim at supporting the reporter by predicting the pathogenicity-likelihood of the variants using classification models (Quintáns et al. 2014). To this end, these models are trained on data from curated variant-pathogenicity associations from databases, e.g., ClinVar (Landrum et al. 2014), in combination with annotations describing the biological context of the variants, e.g., their evolutionary conservation (Eilbeck et al. 2017).

In the following, we demonstrate our prototype in a scenario in which the variant-pathogenicity associations from the reviewing process of the reporters are also considered when learning a variant prioritization method. In this scenario, the medical environment of physicians and reporters interacts with our implementation of a self-learning system. This interaction introduces a feedback loop in which the self-learning system uses final variant assessments (created as part of diagnoses) as training data to learn prioritization methods which, in turn, are used to support the same reporters in their assessment. Such feedback loops are common for self-adaptive software systems (Cheng et al. 2008) (like our learning system) and under active research (Weyns et al. 2021). They tend to amplify output that is fed back to the inputs, as in our case, where diagnoses should improve later diagnoses while simultaneously posing a risk of amplifying diagnosis errors. Given the importance of the feedback loop behavior, thorough tests should be conducted during the development cycle and lifetime of the software to ensure robustness. With this case study, we illustrate the use of our framework and the analysis of feedback dynamics.

4.1 EDA-Prototype

The learning system contains two main workflows: 1. The classification pipeline with patient variants as input and their predicted pathogenicity-likelihood as output, and 2. an adaptation loop which monitors database changes and diagnoses to find the best points in time for learning runs. The classification pipeline can be integrated seamlessly in a web frontend where reporters filter variants according to their workflow. Rankings can then be displayed as an additional column in the currently listed variants. This frontend is the interface between environment (human interaction) and system (EDA implementation), hence it will be a gateway component that has to be maintained in a version for simulation runs and in a version for the real environment. As far as the system is concerned, the reporter workflow starts by uploading the patient's information and his or her variants at the frontend. After the upload, the variants are annotated with information describing the biological context of the variants. With the annotations as features, the current classifier calculates a pathogenicity score by which the variants can be ranked. The variants can now be returned to the frontend to be displayed, filtered and diagnosed. Finished diagnoses are sent to the physicians for patient treatment but also collected by our system as part of the last step in the reporter workflow, while respecting legal constraints. The accumulating variant assessments are monitored in the second workflow (the adaptation loop), together with changes in public databases. If the accumulating changes justify a new training process, a machine learning run is configured and executed. The performance of the resulting classifier is tested in a final step. If the newly trained classifier's ranking precision is superior, it is used in future reporter interactions.

4.2 Simulation Experiment

We exercise the feedback loop among the EDA system and the reporters using synthetic case data. Each variant in a generated case is associated with a point in a two-dimensional feature space based on which a binary classification is carried out using logistic regression. Both the initial training data and the reports are affected by random errors, which we describe below. Since the true class membership of each generated variant is known, we can assess the propagation of errors through the system.

Environment Model We defined the simulation model of the environment as a DEVS model (Van Tendeloo and Vangheluwe 2018). The model consists of the three components of Physician, Laboratory reporter, and Consensus, each realized as an atomic DEVS model. The Physician component generates synthetic cases with exponentially distributed delays (see Figure 7). Part of its state is a seed used for generating cases. Cases are forwarded to Laboratory reporters who annotate cases with features later used by the learned classifiers. Laboratory reporters collect the cases and request pathogenicity scores from the EDA software system. The scores received from the software system are then compared to the reporters' assessments.

We define two types of reporters, which differ in their reaction when their assessments deviate from the classifier's scores. If the number of deviating assessments exceeds a deviation threshold, *cautious* reporters consult their peers at the same laboratory to determine a consensus as the final assessment. Otherwise, they accept the classifier's suggestion. In contrast, *bold* reporters always maintain their own assessment.

Each model's state consists of a variable that shows the phase the model is in, e.g., $idle$, or $assess$, and further information $?x$ to keep track of data or other information, e.g., cases or a prioritizing list of variants. As usual in DEVS, the external transition function δ_{ext} defines how the model reacts to incoming events, whereas the internal transitions δ_{int} are triggered by the passage of time. A time advance function (ta) defines how long each state persists per se. At the moment an internal transition takes place, an output is produced by the output function λ. From the perspective of the environment, the EDA system acts as a DEVS model that takes case data and diagnoses and returns variant rankings. For the case study, we assume that the processing time of these actions is negligible.

In the case study, we focus on the effects of the consensus mechanism on the propagation of erroneous assessments. Figure 8 shows the distribution of the synthetic data used to generate cases. The classes of benign (top left) and pathogenic (bottom right) variants are drawn from two-dimensional normal distributions with covariance 0.03 and mean $(0.25, 0.75)$ and $(0.75, 0.25)$, respectively. We model errors by a probability p of assigning the classification "benign" without consideration of the variant's actual class. For instance, if p equals 1, all variants are assessed to be benign. This type of probabilistic error is used when generating

Physician:
δ_{ext} = (∗, ∗, ∗) → (∗)
δ_{int} = (idle, ∗) → (idle, ∗)
ta = (idle, ∗) → exponential(lambda)
λ = (idle, ?x) → case(?x)

Laboratory reporter:
δ_{ext} = ((∗, idle, ?x), ∗, case) → (∗, systemRequest,appendX(?x, case))
((∗, idle, ?x), ∗, rankedVariants) → (∗, assess, appendX(?x, rankedVariants))
((cautious, idle, ?x), ∗, consensus) → (cautious, reassess, assessX(?x, consensus))
δ_{int} = (∗, systemRequest, ∗) → (∗, idle, ∗)
(bold, assess, ?x) → (∗, idle, cleanUp(?x))
(cautious, assess, ∗) → (∗, idle, ∗)
(cautious, reassess, ?x) → (∗, idle, cleanUp(?x))
ta = (∗, systemRequest,∗) → ε
(∗, assess, ∗) → ε
(∗, reassess,∗) → ε
(∗, idle, ∗) → ∞
λ = (∗, systemRequest, ?x) → case(?x)
(bold,assess, ?x) → { rankedVariants(?x), report(rankedVariants(?x)) }
(cautious, assess, ?x) → evalRankedVariants(?x)
(cautious, reassess, ?x) → { rankedVariants(?x), report(rankedVariants(?x)) }

Consensus:
δ_{ext} = ((idle, ?x), ∗, variants) → (assess, appendX(?x, variants))
δ_{int} = (assess, ?x) → (idle, cleanUp(?x))
ta = (assess,∗) → ε
(idle, ∗) → ∞
λ = (assess, ?x) → evaluatedConsensus(?x)

Figure 7: Excerpt from the environment DEVS models.

the initial data used to train the classifier as well as in the assessments made by individual reporters or as a consensus. Initially, the classifier is trained using 500 samples with $p = 0.1$. The same value for p is used in the assessments by individual reporters. Since we assume that a consensus leads to more accurate assessment, we set $p = 0.05$ in this case. We populate the environment with two cautious reporters and one bold reporter. Case generation and assessment begins after one day of virtual time, each case involving the assessment of 100 gene variants. The times between new cases are drawn from an exponential distribution with a mean of three hours. The classifier is newly trained once ten new reports have been received, the training data being comprised of the initial training data and all assessments by reporters up to this point in time. To allow us to study the effects of erroneous assessments on future classifications, the bold reporter produces highly erroneous assessments throughout the first three days of activity, i.e., days 1 through 3. During this time span, we set $p = 1.0$, i.e., the bold reporter assesses all variants as benign. We now consider two possible behaviors of the cautious reporter by setting d to two different values. With $d = \infty$, the cautious reporters never obtain a consensus opinion, even if their own assessments deviate severely from the classifier's scores. In contrast, with $d = 10$, a consensus opinion is obtained if more than 10 of the assessments in the current case deviate.

Results We measure the error propagation based on the F1 score, which accounts for the precision and recall of the binary classifier within a single scalar. Figure 9 shows the F1 scores observed throughout one week of virtual time each for the two scenario variants, each curve showing averages across 16 simulation runs. We observe that with $d = \infty$, the errors introduced by the bold reporter promptly and severely reduce the quality of the classifications. Since cautious reporters naively accept the assessments suggested by the classifier, the errors introduced by the bold reporter propagate into the reports and any subsequent classifier training. Once the bold reporter begins generating high-quality reports at day 4, the F1 score gradually recovers. On the other hand, with $d = 10$, the F1 scores are much less affected by the erroneous reports. In this case, the bold reporter affects the training process negatively only through its own reports, while the cautious reporters rely on the consensus results to still generate high-quality training data.

This feedback analysis shows a basic test scenario with our simulation framework i.e. testing the feedback behavior under the influence of bad actors. During the software development, the components can be further improved, and complex environments can be modeled to focus on more specific behavior patterns. For example: we can analyze changes in the learning sample usage based on differing laboratory reputations, and validate the impact on the classification score. This way, the software can be verified early and validated systematically with a model of the environment.

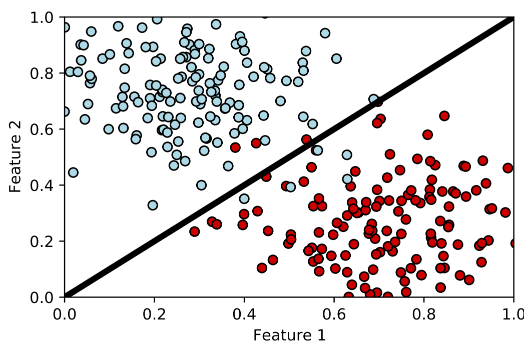


Figure 8: Binary classification in our case study. Due to the symmetric distribution of the two classes' features, an optimal linear separation of the classes is achieved by the diagonal (black line).

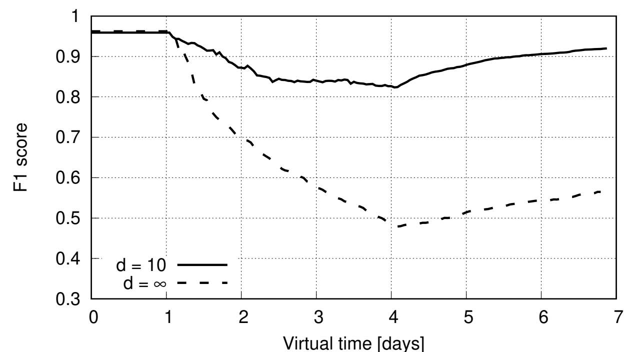


Figure 9: F1 score of the classifier in our case study over virtual time, varying the deviation threshold d. The frequent reliance on a consensus triggered with $d = 10$ limits the error propagation.

5 CONCLUSION

We described a framework that allows EDA systems to be tested in their interaction with simulated environments. We outlined the mapping from physical time to virtual time as a basis for integrating EDA system processes and environment actions in a discrete-event simulation. Additionally, we described a generic architecture that defines the communication among a simulator core, EDA components, and simulated entities. We detailed the constraints required to guarantee that the simulation behavior is analogous to production use. In a case study, a prototypical implementation of our framework was used to analyze the feedback-dynamic between a self-learning EDA system and an environment of users n a medical context.

Our approach can be expanded in several ways. On the technical level, complex EDA systems may require moving beyond a centralized notification service. In general, publish/subscribe functionality may be offered by an entire network of notification service brokers. Further, applications may demand functionalities such as different quality of service levels for notifications. To minimize the gap between simulated EDA systems and their deployment, we intend to make use of the functionalities of existing middlewares for message-based communication (e.g., Mosquitto (Light 2017)) by extending the middleware with the facilities required for simulation. While our current focus is on medical systems, our intention in the long term is to facilitate the development, debugging, and analysis of a wide range of safety-critical event-driven systems.

ACKNOWLEDGMENTS

This research was funded by the state of Mecklenburg-Vorpommern, Germany via the European Regional Development Fund (ERDF) of the European Union as part of the project “IDEA-PRIO-U” (TBI-V-1-354-VBW-122).

REFERENCES

Arcaini, P., S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene. 2015. “Formal Validation And Verification Of A Medical Software Critical Component”. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. September 21st-23rd, Austin, USA, 80–89.

Banks, J. 2000. “Introduction To Simulation”. In *Proceedings of the 2000 Winter Simulation Conference*, edited by P. A. Fishwick, K. Kang, J. A. Joines, and R. R. Barton, 9–16. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Chandy, K. M. 2006. “Event-Driven Applications: Costs, Benefits And Design Approaches”. In *Gartner Application Integration and Web Services Summit*. June 12th-13th, Barcelona, Spain.

Cheng, B. H. C., H. Giese, P. Inverardi, J. Magee, R. de Lemos, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. 2008. “Software Engineering For Self-Adaptive Systems: A Research Road Map”. In *Dagstuhl Seminar Proceedings*, edited by B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, 1–13. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Clark, T., and B. S. Barn. 2011. “Event Driven Architecture Modelling And Simulation”. In *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*. December 12th-14th, Irvine, USA, 43–54.

Eilbeck, K., A. Quinlan, and M. Yandell. 2017. “Settling The Score: Variant Prioritization And Mendelian Disease”. *Nature Reviews Genetics* 18(10):599–612.

Eugster, P. T., P. A. Felber, R. Guerraoui, and A. Kermarrec. 2003. “The Many Faces Of Publish/Subscribe”. *ACM Computing Surveys (CSUR)* 35(2):114–131.

Faviez, C., X. Chen, N. Garcelon, A. Neuraz, B. Knebelmann, R. Salomon, S. Lyonnet, S. Saunier, and A. Burgun. 2020. “Diagnosis Support Systems For Rare Diseases: A Scoping Review”. *Orphanet Journal of Rare Diseases* 15(1):1–16.

Fujimoto, R. M. 2000. *Parallel And Distribution Simulation Systems*. Wiley.

Jansen, R., and N. J. Hopper. 2012. “Shadow: Running Tor In A Box For Accurate And Efficient Experimentation”. In *Distributed System Security Symposium (NDSS)*. February 5th-8th, San Diego, USA.

Jin, D., Y. Zheng, and D. M. Nicol. 2014. “A Parallel Network Simulation And Virtual Time-based Network Emulation Testbed”. *Journal of Simulation* 8(3):206–214.

Kleijnen, J. P. 1998. “Experimental Design For Sensitivity Analysis, Optimization, And Validation Of Simulation Models”. *Handbook of Simulation* 173:223.

Lamps, J., D. M. Nicol, and M. Caesar. 2014. "Timekeeper: A Lightweight Virtual Time System For Linux". In *Conference on Principles of Advanced Discrete Simulation*, edited by J. A. H. Jr., G. F. Riley, and R. M. Fujimoto, 179–186. New York, USA: Association for Computing Machinery.

Landrum, M. J., J. M. Lee, G. R. Riley, W. Jang, W. S. Rubinstein, D. M. Church, and D. R. Maglott. 2014. "ClinVar: Public Archive Of Relationships Among Sequence Variation And Human Phenotype". *Nucleic Acids Research* 42(D1):D980–D985.

Law, A. M. 2013. *Simulation Modeling And Analysis*. Fifth ed. New York, USA: McGraw-Hill Education.

Light, R. A. 2017. "Mosquitto: Server And Client Implementation Of The Mqtt Protocol". *Journal of Open Source Software* 2(13):265.

Mühl, G., L. Fiege, and P. Pietzuch. 2006. *Distributed Event-based Systems*. Springer-Verlag.

Negishi, Y., F. Miya, A. Hattori, K. Mizuno, I. Hori, N. Ando, N. Okamoto, M. Kato, T. Tsunoda, M. Yamasaki, Y. Kanemura, K. Kosaki, and S. Saitoh. 2015. "Truncating Mutation In NFIA Causes Brain Malformation And Urinary Tract Defects". *Human Genome Variation* 2:15007.

Quintáns, B., A. Ordóñez-Ugalde, P. Cacheiro, A. Carracedo, and M. Sobrido. 2014. "Medical Genomics: The Intricate Path From Genetic Variant Identification To Clinical Interpretation". *Applied & Translational Genomics* 3(3):60–67.

Rodriguez, M., M. Piattini, and C. Ebert. 2019. "Software Verification And Validation Technologies And Tools". *IEEE Software* 36(2):13–24.

Sanchez, S. M., P. J. Sanchez, and H. Wan. 2020. "Work Smarter, Not Harder: A Tutorial On Designing And Conducting Simulation Experiments". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K. G. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, 1128–1142. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Tamura, G., N. M. Villegas, H. A. Müller, J. P. Sousa, B. Becker, G. Karsai, S. Mankovskii, M. Pezzè, W. Schäfer, L. Tahvildari, and K. Wong. 2013. "Towards Practical Runtime Verification And Validation Of Self-adaptive Software Systems". In *Software Engineering for Self-Adaptive Systems II*, edited by R. de Lemos, H. Giese, H. Müller, and M. Shaw, 108–132. Berlin, Heidelberg: Springer.

Van Tendeloo, Y., and H. Vangheluwe. 2018. "Introduction To Parallel Devs Modelling And Simulation". In *Proceedings of the Model-Driven Approaches for Simulation Engineering Symposium*, edited by A. D'Ambrogio and U. Durak, 1–12. San Diego, USA: Society for Computer Simulation International.

Weyns, D., B. Schmerl, M. Kishida, A. Leva, M. Litoiu, N. Ozay, C. Paterson, and K. Tei. 2021. "Towards Better Adaptive Systems By Combining Mape, Control Theory, And Machine Learning". In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. May 22nd-30th, Madrid, USA, 217–223.

AUTHOR BIOGRAPHIES

TOM MEYER is a Ph.D. candidate in the Modeling and Simulation group at the University of Rostock. His e-mail address is tom.meyer@uni-rostock.de.

PHILIPP ANDELFINGER is a postdoctoral researcher in the Modeling and Simulation group at the University of Rostock. His e-mail address is philipp.andelfinger@uni-rostock.de.

ANDREAS RUSCHEINSKI is a Ph.D. candidate in the Modeling and Simulation group at the University of Rostock. His e-mail address is andreas.ruscheinski@uni-rostock.de.

ADELINDE M. UHRMACHER is a professor at the Institute of Visual and Analytic Computing, University of Rostock, and head of the Modeling and Simulation group. Her e-mail address is adelinde.uhrmacher@uni-rostock.de.