Event-based Simulation for Transient Systems with Capture Replay to Predict Self-Adaptive Systems (Work in Progress Paper)

Sarah Stieß
sarah.stiess@iste.uni-stuttgart.de
Institute of Software Engineering, University of Stuttgart
Stuttgart, Baden-Württemberg, Germany

Stefan Höppner
stefan.hoeppner@uni-ulm.de
Institute of Software Engineering and Programming Languages, Ulm University
Ulm, Baden-Württemberg, Germany

Steffen Becker
steffen.becker@iste.uni-stuttgart.de
Institute of Software Engineering, University of Stuttgart
Stuttgart, Baden-Württemberg, Germany

Matthias Tichy
matthias.tichy@uni-ulm.de
Institute of Software Engineering and Programming Languages, Ulm University
Ulm, Baden-Württemberg, Germany

ABSTRACT

Cloud-native systems are dynamic in nature as they always have to react to changes in the environment, e.g., how users utilize the system. Self-adaptive cloud-native systems manage those changes by predicting how future environmental changes will impact the system's service level objectives and how the system can subsequently reconfigure to ensure that the service level objectives stay fulfilled. The farther the predictions look into the future, the higher the chance that good reconfigurations can be identified and applied. However, this requires efficient exploration of potential future system states, particularly exploring alternative futures resulting from alternative system reconfiguration. We present in this paper an extension to the Slingshot simulator for Palladio component models to efficiently explore the future state space induced by environmental changes and reconfigurations. The extension creates snapshots of simulation states and reloads them to explore alternatives. We show that Slingshot's event-based publish-subscribe architecture enables us to extend the simulator easily and without changes to the simulator itself.

KEYWORDS

event-based simulation, capture-replay, Palladio, Slingshot, self-adaptive system

ACM Reference Format:
Sarah Stieß, Stefan Höppner, Steffen Becker, and Matthias Tichy. 2023. Event-based Simulation for Transient Systems with Capture Replay to Predict Self-Adaptive Systems (Work in Progress Paper). In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3578245.3585029

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0072-9/23/04.
https://doi.org/10.1145/3578245.3585029

1 INTRODUCTION

Cloud-native systems are subject to continuous changes to their environment, e.g., the rate of user requests, the type of user requests, and the amount of stored data. In spite of all those changes, systems still need to satisfy service level objectives (SLOs) [4] while minimizing cost. Due to all those continuous changes, such systems may not have a steady-state.

Self-adaptive cloud-native systems use reconfigurations to adapt themselves w.r.t. those environment changes by employing a digital twin architecture as we described in a previously published paper [13]. They can do this particularly well if they predict the effects of multiple environmental changes and reconfiguration rules, capturing them in a tree containing future states and changes between them. They then use those predictions to choose the best reconfigurations responding to upcoming environmental changes.

The Palladio ecosystem, its component model [11], and simulation approaches such as Slingshot [8] already enable predicting the effect of individual rules and sequences of rules on the system's behavior and SLOs.

In this paper we present ongoing work on developing such a self-adaptive cloud-native System based on the the Slingshot simulator and the Palladio component model. An exploration component systematically executes simulations to predict how the system fares after different environmental changes and how executing different reconfiguration rules as reactions improve (or worsen) the system's behavior. The result of the exploration component is a graph of states resulting from sequences of environmental changes and reconfiguration rules. Creating this graph efficiently is the key challenge as conducting the simulations is very costly. Hence, our aim is to avoid beginning the simulation from the starting state for each different sequence of environmental changes and reconfiguration rules. The contribution of this paper is an extension of the Slingshot simulator which allows to snapshot simulation states and restart a simulation from the snapshot. The introduced exploration component can simulate one sequence of changes and rules, backtrack to a previous state in the tree, re-load the snapshot, and continue the simulation with a different change. As this is still work in progress, we cannot yet provide concrete result regarding efficiency and performance.

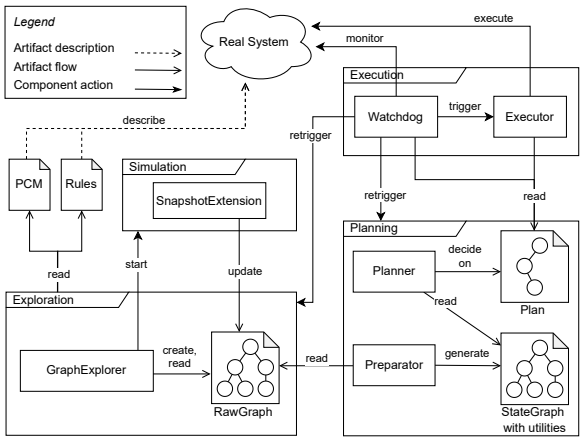

Figure 1: System architecture overview.

The remainder of this paper is structured as follows: Section 2 gives an overview of the complete self-adaptive system we are developing and presents details on simulation and exploration and the involved artifacts. Afterwards, in Section 3, we detail our approach of using snapshots and simulation to explore a number of potential future system developments. Section 4 discusses related works and Section 5 presents a conclusion of our work and outlines future work based on our current results.

2 ARCHITECTURE

In this section, we outline all parts of our self-adaptive system to allow an understanding of the system as a whole and the role simulation plays in it.

The architecture of our self-adaptive system is centered around the four main activities of the system, namely *Simulation*, *Exploration*, *Planning* and *Execution*, as depicted in Figure 1. Simulation is focused on simulating how the system might develop over the next few minutes based on an arbitrary start state. Exploration uses simulation to explore several potential futures and aggregates them into a directed acyclic graph that depicts which different system developments are possible. The artefact that holds this graph is called *RawStateGraph*. We explain the makeup of the RawStateGraph in more detail in Section 2.2. Planning is concerned with using the results from Exploration to decide on optimal system actions, and Execution applies planned actions to the real system at appropriate times. The former step requires some pre-processing to compact the RawStateGraph into a format that is better manageable.

In the remainder of this section, we give a detailed description of the simulation and involved artifacts . Details on the other components are given throughout the remainder of this paper.

2.1 Simulation

We use the Slingshot Simulator [8], that simulates a system with discrete event-simulation [1] and queueing networks (QN) [5]. Using QNs implies that the simulator implements modeled resources as queues. Discrete event-simulation entails that all things that might happen during interpretation of a system model — such as a user request entering or leaving the system, or a user request requesting

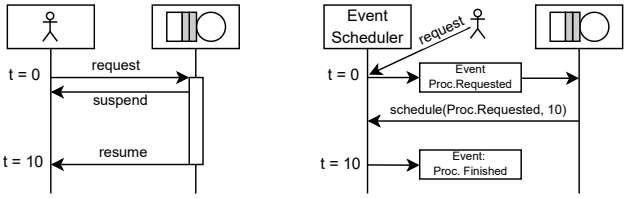

Figure 2: Simulation worldviews, as described in [6]. Process-oriented on the left and event-oriented on the right.

a resource — are described with events. As this is about *discrete* event-simulation, each event occurs at a certain point in time.

There are different views on discrete event-simulation. A process-oriented view, as sported, e.g., by Simulizar [2] and an event-oriented view, as sported, e.g., by Slingshot [8]. Understanding the differences between them is helpful for understanding the following sections. Thus, we recap the worldviews as described by Carson [6], in the following paragraphs.

For the process-oriented worldview, each user request is a separate process. As depicted on the right of Figure 2, a user request traverses the simulated model and requests a resource at point in time $t = 0$. The resource — which behaves like a queue — calculates how long it takes to process the user request, sends the request's process to sleep and wakes it up after the calculated amount of time units passed (10 for this example), thereby simulating the processing time. Once the processing time is over, the user request resumes the traversal of the simulated model.

In the event-oriented worldview, each user request is an entity. Events describe how the user request traverses the simulated model. As depicted on the right of Figure 2, a user request enqueues at a resource at point in time $t = 0$, as represented by the event *Proc.Requested*. Just as with process-orientation, the resource calculates the duration until the request is processed. However, instead of sending a process to sleep, the simulation adds the event *Proc.Finished*, representing the end of the processing, to the future event list. Once the simulation time reaches the respective point in time the event is consumed, in this example at $t = 10$. Importantly, each entity carries its context, e.g. a request knows which resource to request, and what the next processing step is.

The event-oriented worldview fits our purpose better. It enables us to create a system where we can resume simulation from any state of the simulated system. This process is detailed in Section 3. We therefore settled on using the Slingshot Simulator [8].

In addition to the event-oriented worldview, Slingshot has an event-driven architecture [8], i.e., Slingshot uses Publish-Subscribe to distribute events. The performance impact of this design is not yet evaluated. Slingshot consists of a core component that manages the future event list and distributes the events, and arbitrary extensions that may subscribe to events and process them. The extensions are either stateless or stateful. The latter are, e.g., for processing resources that must remember the enqueued requests.

2.2 RawStateGraph and RawModelState

As depicted in Figure 1, the RawStateGraph exists as the central artifact inside Exploration and is the link between Exploration and

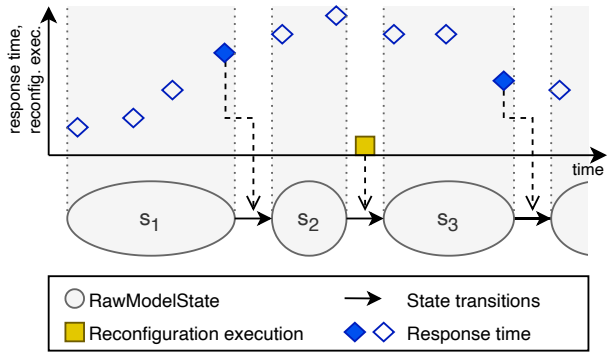

Figure 3: Excerpt of a RawStateGraph (lower half) and the underlying runtime measurements (upper half).

Planning. It is also a data structure we designed to represent states of our self-adaptive system and transitions between the states. We use the prefix "Raw" because Planning processes the raw graph further into the actual StateGraph.

The RawStateGraph consists of *RawModelState*s and transitions. A RawModelState represents one state of the self-adaptive system. It consists of an architecture configuration, an environment context, runtime measurements, and a snapshot of the simulator state.

Figure 3 visualizes the relation between runtime measurements and the RawStateGraph. The upper half of the figure shows response time measurements, marked with ◊ and ⧫. The ⧫ marks values with a significant difference from the previous values. In reality, we would measure other runtime measurements as well. The □ in the diagram marks the execution of a reconfiguration. The lower half of the figure shows an excerpt of a RawStateGraph, consisting of three successive RawModelStates (s_1, s_2 and s_3) and a bit of s_3's successor state, as well as the transitions between the states. Each RawModelState encompasses multiple runtime measurement values, as indicated by the dotted vertical lines. The system transition into a new state, if a runtime measurement changes significantly, e.g. the ⧫ that caused the transition from s_1 to s_2 and from s_2 to s_3, or the architecture configuration changes, e.g. the □ that caused the transition s_2 to s_3. Both are marked with a dashed arrow. Environment changes cause transitions as well, but are not depicted.

Regarding the snapshot, it contains the state of a simulator's stateful constituents, such as the queues. We take it just before the transition to the next RawModelState, e.g., for the state s_2 in Figure 3, a snapshot is taken just before the execution of the reconfiguration. The snapshot can be used to restore the simulator to that point in time. We have multiple possible snapshots for each RawModelState, because RawModelStates last a certain duration, such that the state of the simulator changes while we stick to the same RawModelState. There is only one possible architecture configuration and environment context for each RawModelState because changes in either cause a transition. The next RawModelStates then contains the changed architecture configuration or environmental context.

3 STATE GRAPH EXPLORATION

This section covers Exploration and Simulation (c.f. Figure 1). Figure 3 from the previous section shows a linear sequence of states, i.e. one future, no alternatives. However, we want to continue into alternative futures as well, e.g. by branching off after s_2. To achieve this, we must restore the stateful constituents of the simulator to what they were like when transitioning from s_2 to s_3, select a different change, based on probabilities provided by the user, and apply the selected change. After the new change, the system will likely behave differently, i.e. ends up in a new state, leaving us with a different future to save and explore.

The crucial aspect of the exploration is to restore the stateful constituents of the simulator to a previous state. We achieve this with the snapshots we take of previous simulator states.

We realize snapshots with an approach based on the event-oriented Slingshot Simulator [8]. We named the approach *capture-replay*.

In Slingshot, we can snapshot the state of the simulator by *capturing* a set of simulation events. The set does not contain all past events, but only those required for restoring. Later on, we can *replay* the captured events on an empty simulation run to restore the simulator state at the time of capturing. After restoring we can continue the simulation.

Alternative approaches have been considered and are discussed in Section 3.3.

3.1 How does this work in detail?

In this section, we describe how we realized the capture-replay approach with Slingshot [8] and discuss how it works in detail.

As shown in Figure 1, Exploration uses Simulation to gather information about the system behavior. In addition, we hook a component for snapshots into Simulation. Due to using the Slingshot simulator and Slingshot's Event-driven architecture, hooking into the simulation is easy. The snapshot component is designed as a Slingshot Extension. It subscribes to events, that might trigger a snapshot and introduces additional snapshot-related events and the logic to take a snapshot.

The SnapshotExtension is designed to be as noninvasive as possible. We restricted changes to the Slingshot simulator to a few additional access operations to extract the upcoming events from the future event list. For the simulation itself, our SnapshotExtension is purely optional. We can detach it and still use the simulator.

The Snapshot Extension is split into three parts: one part, to check for snapshot triggers, a Camera to take the snapshots, and a Recorder to remember important events of the past.

The trigger-checking part subscribes to measurement and reconfiguration related events, evaluates them, and triggers a snapshot, if necessary. Once the snapshot creation got triggered, the Camera creates a snapshot with the events from the future event list.

If all extensions involved in the simulation of a system were stateless, events from the future event list would be sufficient to initialize the next simulation run. However, in Slingshot, the queues for the processing resources and the calculators for monitoring durations, e.g., response time, are stateful. To restore their states as closely as possible, we introduce the Recorder. We record past events, instead of accessing the queues directly because it offers

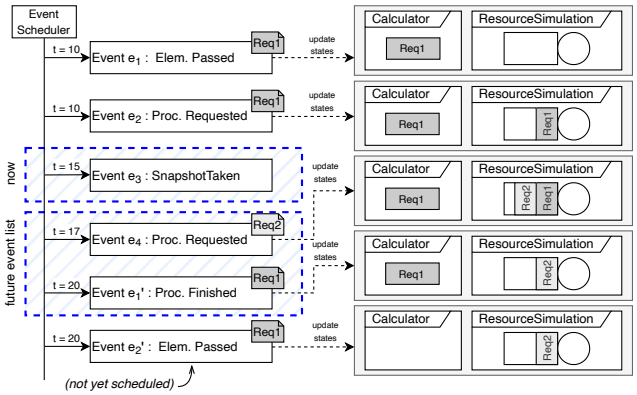

Figure 4: Events of an event-oriented Simulation. Two requests enter a queue, one request leaves. At $t = 15$ a snapshot is taken.

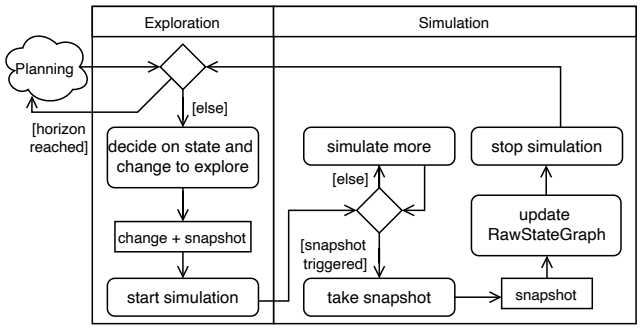

Figure 5: Exploration / Simulation loop.

more independence. We need not alter any existing Slingshot component, and are less likely to be affected by changes of Slingshot's internal implementation. Currently, the Recorder remembers events about a request arriving at a resource until the request is processed and events that start a duration calculation until the duration is calculated. Both are observed by subscribing to the respective events.

Figure 4 shows an event trace on the left, and stateful resources on the right. The current simulation time is $t = 15$ and the *SnapshotTaken* event is the latest published event. Thus e_1 and e_2 are in the past, e_4 and e_1' are in the future event list and e_2' is not yet scheduled, as it is only created after consumption of e_1'. For the snapshot at $t = 15$, the Recorder remembers e_1 until e_1' is published and e_2 until e_2' is published .

Without the Recorder, a Snapshot created at $t = 15$ in Figure 4 consists of the events e_4 and e_1'. Event e_2' is only scheduled after consumption of e_1' and thus not yet in the future event list. Upon initializing a Simulation run on $\{e_4, e_1'\}$, the following would happen: At $t = 2$ of the new run, e_4 requests the resource for *Req2*. The resource deems itself idle and schedules an event at the end of the processing. At $t = 5$, e_1' announces, that the user request *Req1* is done processing. However, the resource does not know about *Req1*, because the resource's pre-replay state got lost. As a result, some measurements are incorrect or missing.

With the Recorder, the Snapshot at $t = 15$ consists of e_4 from the future event list and the recorded e_1 and e_2. Upon initializing a new Simulation on these events, the following would happen: At $t = 0$ of the new run, we schedule e_1 and e_2. Event e_1 puts an entry for Req1 into the Calculator and e_2 puts the request into the resource's queue, thereby recreating the state of the stateful components as close as possible. The resource schedules a new event (not in the figure) to announce when Req1 is processed. The new event replaces e_1', which we excluded from the snapshot. At $t = 2$ of the new run, e_4 requests the resource for Req2. The resource has Req1 in the queue and schedules the next event with the appropriate delay. Later on, when the replacement event for e_1' and e_2' are published, the stateful extensions have the correct states and can process the events. Finally, the measurements taken from the initialized simulation run are similar to the measurements of the initial simulation run after $t = 15$. For models without randomness, the measurements may be identical except for minor errors due to disarray in the queues of FCFS resources.

On an additional note, we do not replay all events as an exact copy of their original. Some events require slight changes, e.g., for a partially processed processing request, we reduce the demand to reflect the ongoing processing at the time of the snapshot.

3.2 Exploration of the RawStateGraph

The exploration involves the activities Exploration and Simulation depicted in Figure 1. It loops across Exploration and Simulation as depicted in Figure 5. At first, the GraphExplorer component decides on the change to explore, feeds the change as well as the snapshot to start on into the simulator, and starts a simulation. The simulation runs until a snapshot gets triggered. Then, the SnapshotExtension takes the snapshot, updates the RawStateGraph, and stops the simulation. This repeats until the time horizon is reached, in which case the explorer hands the graph to Planning. Planning dictates the time horizon.

The change to explore is either an architecture reconfiguration or an environmental change. The GraphExplorer decides on the next change based on probabilities that the user provides. The likelier a change in reality, the likelier its exploration.

We derive the snapshot triggers from the possible transitions. We have three types of transitions in the RawStateGraph: reconfigurations, context changes, and significant changes in measurements. The significance level is a parameter provided by the user. With these transitions, we need three types of triggers for snapshots:

(1) significant runtime measurement changes.
(2) reactive reconfigurations.
(3) time intervals (proactive reconfiguration/context change).

We observe (1) and (2) during the simulation. In both cases we take a snapshot, stop the simulation and continue with the exploration loop. The rationale behind (1) is, a significant change in measurements might be the prelude to a reconfiguration soon after, and we want to see what happens if we reconfigure proactive before a reactive reconfiguration is required. The rationale behind (2) is, in case of a reactive reconfiguration, we want to explore what happens if we ditch the reconfiguration or if we replace it with another change. The alternatives will either support the decision to

do the reconfiguration, or provide better decisions. They are also useful to reason about the decision later on.

If neither (1) nor (2) are observed for some time, i.e. nothing new happens, we do not gain any knowledge, therefore we have trigger (3). We take a snapshot after a predefined interval and enforce either an environmental change or reconfiguration. The rationale for the environmental change is to disturb the system, e.g. with a sudden spike of user requests. The rationale for the reconfiguration is to figure out whether additional reconfigurations may optimize the system further, or not.

3.3 Discarded alternatives

Aside from the capture-replay approach, we considered two other approaches. The other approaches do not require the event-oriented simulation worldview. Instead, they might work with both event- and process-oriented worldviews. In addition, they use less and also less specific data. Mostly, because we assume that specific information about the user requests, as we use them for the capture-replay approach, is not available. Consequently, we assume that the discarded approaches would produce worse results than the capture-replay approach, yet would require more effort to implement them. Thus, we discarded them.

Both of the discarded approaches have the same base idea. The second one is an improvement of the first one. The underlying assumption is, all knowledge about the state of the simulation is obtained through monitoring. We simulate the processing resources with QNs, thus we can measure the length of the queues. However, we cannot know which requests were waiting, because the simulators tend to capsule the queues in such a way that we cannot directly access their content without further effort. As we do not know the requests, we cannot put them back into the queues when attempting to restore a previous state. Instead, we try to at least recreate the former length of the queues. If the simulated PCM instances contain randomness, this introduces errors, e.g. if the size of a user request is determined randomly.

The first discarded alternative was an approach where a new simulation would started and run until the run passes through the previous state once again. It is the most straightforward and least-effort approach. During the new simulation run, we would monitor the queue lengths until they are the same as in the state we want to recreate. Then we would apply the change and continue into the alternative future, just as described earlier in this paper. The main disadvantage of this first approach is that starting the simulation anew for each future takes a lot of time.

The second discarded approach was an attempt to counter this disadvantage. We considered to ignore the time it takes to process requests, until we would have the correct queue lengths. Instead of starting the simulation with the incoming user requests as they are modelled, we would modify the first few requests, such that they pass through some of the queues without delay. We would send these delayless user requests in until the queues would have the desired length. Then we would start the actual simulation with processing time and everything.

Compared to the first discarded approach, we expect this approach to be faster. However, we assume the approach to require a lot of effort to correctly tweak the delayless user requests. In addition, up to here we thought about queues only and did not yet consider other stateful constituents of the simulation, such as the calculators for response time and other duration.

4 RELATED WORK

The related work is structured into three areas, namely simulation, state-space exploration, and formalisms for self-adaptive systems. We excluded works, whose results cannot contribute to the creation of the state graph we desire. In particular, the excluded works encompass AI-based approaches to planning for self-adaptive systems, or simulation approaches with steady-state analyses, as those only deliver results, but no intermediate steps to create a state graph.

In the area of simulation, we looked into existing simulators such as SimuLizar [2], Slingshot [8], and line [10]. They are of interest because they simulate the behavior of self-adaptive systems based on Palladio models. SimuLizar [2] is a process-oriented simulator and Slingshot [8] is an event-oriented simulator. Both procure measurements for transient states as well, instead of only one average measurement value for the steady-state, as analytical solutions usually do. Measurements for transient states are important to us, as we need them to build a state graph. Line [10] transforms the Palladio models into fluid models and applies math to solve them for different points in time, thereby getting both transient and steady-state measurements [10]. However, neither of the three simulators is sufficient for our intention, at least not with their current features. They consider one path through the state space only and do not explore alternatives. We also looked into *parallel discrete event simulation* (PDES) [7]. In PDES a simulation is split into multiple, parallel process. With an optimistic PDES approach, events may get out of order. To cope with such errors, the PDES regularly save the simulation state, and restores the simulation to an earlier state if an out of order event occurs. The method, i.e., saving and restoring states, is similar to our approach. However their motivation and goals differ from ours. PDES uses snapshots only to rollback an erroneous simulation. We restore a simulation to previous states to explore different futures.

In the second area, state-space exploration, we looked into PerOpteryx [9] and work of Smit and Stroulia [12]. PerOpteryx [9] is a tool to explore and compare different architecture configurations. It is also based on Palladio. However, it is intended to optimize a system at design-time and does not consider reconfigurations at runtime. Thus, it differs from our desire to explore architecture configurations of self-adaptive systems at runtime. Smit and Stroulia [12] make reconfiguring decision for a self-adaptive system based on a state space graph of the system's states. They construct the graph from simulation results, though they do not state which simulation tool they use. Their states consist of performance measurements, the degree of satisfaction of service level objectives, etc., and are merged based on similarity, such that the total number of states is reduced. During runtime they monitor the system, match the runtime states to the simulated states and decide reconfiguration based on the state-space graph. Thus, their general idea is similar to ours. In addition, they support our argument, that knowing intermediate and alternative system states improves explainability [12]. Regarding the state graph, they create it a priori and schedule simulations to update the graph for later, such that they need not care

about the time it takes to run the simulations [12]. Updates are necessary in case of unknown states that are not yet in the state graph. As a downside, their planning is always limited to the known states. Contrary to them, our approach attempts to run simulations for state graph updates immediately. To make this work, our simulation must be fast. However, if successful, this generates a more accurate plan.

In the area of formalisms for self-adaptive systems, Becker [3] formalize the states of a self adaptive system as a tuple of architecture configuration, context scenario, and runtime measurements. Transitions from one state to the next are caused by changes in either of them. We partially based our RawModelState on Becker's formalization. However, we do not transition on every change in measurement values. Thus, our states are coarser-grained than theirs.

5 CONCLUSION & FUTURE WORK

In this paper, we presented a novel approach for simulating and exploring the behavior of a self-adaptive system. The approach is based on an event-based simulation. We create snapshots of a system and record currently active events to resume simulation from any given system state. This capture-replay mechanism is used to explore multiple potential future system developments, enabling more robust planning of reconfigurations.

At this time our simulation can produce a raw state graph containing all explored system states and transitions between them. We are currently working on fully implementing the self-adaptive system, as outlined in Section 2. This includes implementation of planning that takes the RawStateGraph as input and produces a plan and execution which takes the plan and uses it to control the system.

Planning is separated into two different phases (see Planning in Figure 1) First the state graph is compacted by removing duplicate states and a utility for each state within the graph is calculated that describes how desirable the state is. Then, based on the state graph with utilities, an optimal path to follow is selected and passed to the executor.

Compacting the state graph is done to reduce the overall size of the state graph that needs to be analyzed. States are considered duplicates if the difference between system measures within them lies below a threshold for all measures, and they follow one another based on changes in the system environment. Chains of such states are condensed into a single state. In this phase, we also intend to analyze the state graph to detect emergent effects such as e.g., bottleneck shifts [14]. If such effects are detected, the affected section of the state graph is labeled accordingly so it can be considered during the creation of the optimal path.

The utility of a state is calculated via a function that takes all measurement data within the state as input. It also incorporates the utilities of states that follow after the considered state. This is done to enable greedy planning.

Planning then starts from the source state of the state graph. The utility of all following states are considered and the state with the highest utility is selected. This process is repeated until the simulation horizon is reached. If a graph section is selected that is labeled as containing emergent behaviors an appropriate countermeasure is applied to it before adding it to the selected path.

After a complete path is selected this path is passed on for application to the system.

Execution of an optimal path takes the optimal path as input and tries to follow this path by either waiting for system state changes or applying the planned reconfigurations if a reconfiguration edge is reached. To do so we constantly monitor the real system and check if the system state matches the tip of the planned path (see Watchdog in Figure 1). If this is the case the edge connecting the tip and its following state is executed (see Executor in Figure 1). If the edge represents an environment change this means waiting for the change to occur. If the edge represents a reconfiguration this means applying the reconfiguration to the system and then waiting for the system the reach the predicted following state.

If the system state does not match the tip of the planned path we either have to re-plan or redo simulation and planing completely. Re-planning can be done if the system state matches a different state within the state graph not present on the selected optimal path. Redoing simulation and planning must be done if the state is completely unknown.

As a further optimization we intend to use the waiting time during execution to trigger further simulation and planning down the currently followed path.

ACKNOWLEDGMENTS

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 453895475.

REFERENCES

[1] J. Banks, J. S. C. II, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation, 5th New Internatinal Edition*. Pearson Education, 2010.

[2] M. Becker, S. Becker, and J. Meyer. Simulizar: design-time modeling and performance analysis of self-adaptive systems. In SE'13, 2013.

[3] M. W. Becker. *Engineering self-adaptive systems with simulation-based performance prediction*. PhD thesis, Dissertation, Paderborn, Universität Paderborn, 2017, 2017.

[4] S. Becker, G. Brataas, and S. Lehrig. *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications - The CloudScale Method*. Springer, 2017. DOI: 10.1007/978-3-319-54286-7.

[5] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[6] J. Carson. Modeling and simulation worldviews. In *Proceedings of 1993 Winter Simulation Conference - (WSC '93)*, pages 18–23, 1993. DOI: 10.1109/WSC.1993.718024.

[7] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990. DOI: 10.1145/84537.84545.

[8] J. Katic, F. Klinaku, and S. Becker. The slingshot simulator: an extensible event-driven PCM simulator (poster). In SSP'21, 2021.

[9] A. Koziolek, H. Koziolek, and R. Reussner. Peropteryx: automated application of tactics in multi-objective software architecture optimization. In *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, QoSA-ISARCS '11, pages 33–42, Boulder, Colorado, USA. Association for Computing Machinery, 2011. DOI: 10.1145/2000259.2000267.

[10] J. F. Pérez and G. Casale. Assessing sla compliance from palladio component models. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 409–416, 2013. DOI: 10.1109/SYNASC.2013.60.

[11] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016.

[12] M. Smit and E. Stroulia. Autonomic configuration adaptation based on simulation-generated state-transition models. In *2011 37th EUROMICRO Conference on*

Software Engineering and Advanced Applications, pages 175–179, 2011. DOI: 10.1109/SEAA.2011.36.

[13] S. Stieß, S. Becker, F. Ege, S. Höppner, and M. Tichy. Coordination and explanation of reconfigurations in self-adaptive high-performance systems. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '22, pages 486–490, Montreal, Quebec, Canada. Association for Computing Machinery, 2022. DOI: 10.1145/3550356.3561555.

[14] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *ICAC'05*, 2005. DOI: 10.1109/ICAC.2005.27.