Towards Systematic Live Experimentation in Software-Intensive Systems of Systems

Ilias Gerostathopoulos[1], Tomas Bures[2], Sanny Schmid[1], Vojtech Horky[2], Christian Prehofer[1], and Petr Tuma[2]

[1]Fakultät für Informatik
Technische Universität München
Munich, Germany
{gerostat, schmidsa, prehofer}@in.tum.de

[2]Charles University in Prague
Faculty of Mathematics and Physics
Prague, Czech Republic
{bures, horky, tuma}@d3s.mff.cuni.cz

Abstract

As the size, variation, and sophistication of software-intensive systems-of-systems grows, so does the uncertainty inherent to their design and development. To deal with this issue, we propose a framework for systematic experimentation based on declarative specification connected with system architecture. The focus is on how to specify experiments that allow systematic exploration of the space of alternative configurations at runtime. Since such experiments should be launched on live systems, extra care needs to be taken in preventing damages when experimenting with the systems. Therefore, we also focus on how to quantify the direct and the indirect cost associated with each experiment execution (which needs to be included in a cost-benefit analysis for system adaptation) and on how to gradually roll out an experiment via a number of different stages. We use the development of a route planner system as an example to motivate and exemplify our approach.

Keywords

Systematic experimentation; uncertainty; system architecture

1. INTRODUCTION

There is a clear trend of devices getting more and more connected to the Internet. Connected systems such as cars, home appliances, and office buildings are also increasingly becoming smarter: they are more aware of user preferences and environment situations; able to adjust their operation to save power and money; serve users in personalized ways. As the size, variation, and sophistication of such software-intensive systems-of-systems (siSoS) grows, so does the uncertainty inherent to their design and development.

Two broad sources of uncertainty can be distinguished [5]. *Internal uncertainty* refers to the uncertainty stemming from the complexity of the software system itself: large codebases, multiple development teams, diverse development practices and standards across teams, frameworks featuring multiple levels of abstraction are all factors that hinder the understanding of how a siSoS operates internally. It is not wrong to assume that in many real-life software-intensive systems, even their own developers and software architects may not be able to have a complete and accurate understanding of how the systems they build operate or will operate when brought to production (due to e.g. scale effects). This is exacerbated in the case of siSoS, which feature managerial independence, heterogeneity, and emergent behavior.

External uncertainty stems from the complexity and unpredictability of the physical environment (natural disasters, etc.) and of the non-software-controlled parts (network, mechanical, chemical parts, etc.) of siSoS. This also includes the unpredictable behavior of human users and the unreliability of human operators. Again, maintaining a complete and accurate understanding of the interplay of the different physical and mechanical parts in a system, while taking into consideration a large number of possible user interactions, is a close-to-impossible task even for the most skilled system architects.

Despite their high internal and external uncertainty, siSoS still need to be developed, integrated, tested, and delivered to customers at an increasing pace. This is a direct effect of the speed-up of innovation cycles and the pursue of continuous deployment (i.e. every several minutes or hours) seen in many products delivered today [8]. Rapid development cycles make it hard for system architects and developers to manage the growing internal and external uncertainty of the systems they are building.

In this paper, we focus on how to deal with the uncertainty inherent to the design and development of siSoS. We argue that we need methods and tools which

(i) allow developers to test their understanding of the system they are building;
(ii) enable the system under construction to adjust itself not only in response to the environment in general, but also to reflect the gradually maturing understanding of the developers.

Such methods should transcend the traditional barrier of testing environments to be also applicable to production environments, where the effects of scale and emergent behavior [16] can be better studied. Furthermore, due to the large scale and complexity of the siSoS, monitoring has to be done from within the siSoS. Any self-observation naturally impacts system behavior and service quality due to probing overhead, increased communication overhead and because the siSoS may need to execute alternative behavior to assess whether it gives better results than the regular behavior. A careful assessment of the negative impact of self-observation in system and service performance and output quality is therefore needed.

In response to these needs, and inspired by ideas in evidence-based engineering and data-driven evolution [3], we propose a framework for systematic experimentation with live systems based on declarative specification connected with system architecture. By “live” we mean systems deployed in production environments. We focus on how to specify experiments that allow systematic exploration of the space of alternative configurations, as well as on how to quantify the direct and the indirect cost associated with each experiment execution—which needs to be included in a cost-benefit analysis for system adaptation. We also provide a work-in-progress prototype of a tool implementing the framework in Python and some results of our initial experiments using the tool.

Although our approach is applicable to any kind of software system one can experiment with, it is particularly beneficial for experimenting with siSoS. This is because (i) the full-scale emergent behavior of siSoS can only be observed and studied in production environments, and (ii) the managerial independence of the individual systems hinders the comprehensive experimentation of system-of-systems in testing environments.

The rest of the paper is structured as follows. Section 2 introduces the example system used for motivating and exemplifying our approach. Section 3 describes the main ideas behind systematic stage-based experiment execution in live siSoS. We then describe our work-in-progress implementation of a tool that reifies these ideas in Section 4, together with indicative experimentation results. Section 5 overviews related work in experiment-driven development and performance-related issues, and Section 6 concludes with a summary of contributions.

2. RUNNING EXAMPLE

To illustrate our approach for systematic live experimentation in siSoS, we describe here an example system from the mobility domain, which we will use throughout the paper.

In our running example, a number of cars navigate in a city. Each car has an itinerary consisting of a number of destinations to reach within certain time constraints. For route planning, each car uses an external routing service accessible via the mobile network. The route planner takes as input the current position of the car (as monitored by on-board sensors, e.g. GPS) and its target destination (typed in by the driver) and provides the fastest route to the destination as output. To calculate the fastest route, it runs a Dijkstra algorithm on the graph representing the map of the city consisting of nodes/intersections and edges/streets. The cost of each edge is calculated based on *static* information, i.e. streets length and maximum permissible speed on every street, and on *dynamic* information, i.e., traffic level on each street. The latter is estimated based on travel information sent by the cars: upon reaching an intersection (end of a street), each car sends the time it took to navigate the street to the route planner. This serves as an indication of the traffic level in the street, with higher travel times in a street than normal (i.e. calculated with the static information) indicating high traffic in this street. To deal with the dynamic changes in the city traffic that might make certain routes more or less preferable, each car periodically polls the route planner for a route and re-routes itself if needed.

Even though the above system is quite simple compared to real-life route planners, which typically work with historical traffic data and even real-time predictions, it still features a number of challenges in its design, implementation, and testing that are related to both internal and external uncertainty factors:

- How often should each car poll the route planner? Polling more often might provide faster routes but also increases the communication overhead and the computation load on the planner.
- How can the route planner ensure that it has up-to-date information for the traffic level in all streets? Since the traffic information is estimated by the travel times of cars themselves, the planner might not have up-to-date information on the streets less traveled. A possibility for the planner is then to route certain cars through these streets to collect information, which can potentially cause annoyance to the drivers.
- How much should the route planner rely on static information and how much on dynamic information in determining the routes? Relying more on the latter will make planning react quicker to traffic jams but may also result in creating new traffic jams by re-routing many cars through the same alternative routes.

The above questions serve just to illustrate the different design decisions and tradeoffs, mainly between system quality and performance, that have to be accounted for when developing such a system. Apart from thoroughly testing the system in testing environments via experimenting with the different decisions and tradeoffs, we may also need to resolve some of them in production. In the following, we provide a framework aiming to support this.

3. SYSTEMATIC EXPERIMENTATION

Experimentation in siSoS requires a systematic approach in which the choice of system variants (at design time or at runtime) and optimization of system's parameters needs to be based on rigorous and statistically justified experiments. In the basic case this means modifying a part of a system and collecting data to evaluate the effect of a particular choice.

Given the large-scale nature and the close interaction with humans and the physical environment, siSoS cannot be easily replicated in testing environments, which means that the experimentation has to be typically performed on a live system. This however implies that the experimentation may negatively affect the system—first because a choice being experimented with can be suboptimal to the actual current version of the system, second because the process of collecting and evaluating the experimental data cannot be separated from the system and itself may be quite costly. For instance, in case of collecting performance data, the instrumentation of the system can yield a significant slowdown. (In one of our previous experiments, we showed that even a very efficient instrumentation of all methods in SPECjbb2015 yielded 400% slowdown [7].) The cost of experimentation gets also projected in energy consumption, which is especially felt on battery powered devices. Finally, experiments that involve humans typically come with a high inherent cost which stems from the fact the system requires a human user to do some extra work (e.g. provide feedback on satisfaction) or may lower user appreciation (e.g. by navigating a vehicle through an unusual route or bringing the car to a traffic jam in an unexplored street).

We thus propose the systematic experimentation as a closed-loop process which, in addition to executing experiments, continuously monitors the effect of the experimentation on the running system and controls (or even stops) the experimentation should the system be negatively impacted beyond a certain limit.

To correctly capture and execute the systematic and statistically relevant experimentation, we conceptualize the experiment in this section. Then in Section 4, we propose a tool-supported framework for automatically performing systematic experimentation based on a set of well-defined experiments.

3.1 Quantifying the cost of an experiment

We view the experiment as a time-constrained process which changes the system in a defined way and observes and evaluates the effect of the changes. To account for the negative impact on the utility of a system, we define the experiment as a workflow of stages. Each stage is associated with a cost (i.e. the amount of the negative impact on the system's utility). Splitting the experiment to smaller pieces makes it easier to measure and quantify their cost. Further, a subsequent stage would typically exploit evidence collected in some previous stage to estimate its cost.

To illustrate this on the running example, assume that the routing algorithm lends itself to parameterization which has the potential to increase user satisfaction, but requires heavier computation (e.g. to re-evaluate routes more often, to consider more routes as potential alternatives or to extend the horizon of simulation to predict the traffic in the near future). In this respect, the experiment would have the following stages:

1st stage: Evaluate the cost of instrumentation and performance data collection on the HW platform where the experiment is going to be executed. This can be done by instrumenting a dummy function and iterating its execution multiple times to obtain statistical confidence about the expected value of the instrumentation overhead in a single instance. Though this experiment will require some computation capacity, its impact on the system is rather minimal because the experiment is rather small-scale compared to the processing connected with the car navigation.

2nd stage: Instrument the existing routing algorithm to get the baseline performance of its individual components. The estimated cost here takes into account the expected frequency of executions of the instrumented code and the cost of instrumentation determined in the 1st stage.

3rd stage: Modify the parameters of the routing algorithm, instrument it and execute it in a limited scale (e.g. use the modified version for only 1% of cars). This provides an estimation of the cost of a larger scale deployment of the experiment in the 4th stage. The estimated cost of the 3rd stage takes into account the baseline performance of the individual components of the routing algorithm from the 2nd stage and an estimation of where the algorithmic complexity increases given the particular experiment.

4th stage: Execute the experiment in a larger scale (e.g. use the modified version of the routing algorithm for 20% of cars) to get representative data that reflect potential resource contention due to unforeseen interactions at the larger scale.

5th stage: Execute the experiment to a full extent, i.e. use the modified version of the routing algorithm for all cars. The experiment should only reach this stage if there is clear indication that the utility of the system (measured, e.g., a function of user satisfaction rate and degree of resource utilization) will not sink below a certain limit (termed *utility threshold* in the next Section).

3.2 Structure of an experiment

From a structural perspective, we see the experiment as consisting of the following:

- Input parameters—parameters of the experiment (e.g. in case of exploring alternative routes, the experiment parameter can be the maximal time a driver can be delayed for the sake of exploring traffic in less-travelled streets of the city).
- System utility—a measure of how well the system performs. An experiment is evaluated based on how much it improves the baseline utility (i.e. the utility before the experiment is executed).
- System utility threshold—a limit on system utility below which the experiment is not allowed to continue in executing any of its stages. This serves to maintain system's dependability in face of potential negative effects (i.e. the cost) of the experiment. The expected effect of a stage on the utility is captured by the *cost of the stage* as explained in the structure of the stage below.
- Stages—different steps of the experiment as explained in Section 3.1. The stages are connected in an oriented graph, where edges in the graph are guarded by conditions over input parameters and outputs of previous stages (Figure 1). The structure of a single stage is described below.
- Result—decision or quantification (backed by statistical evidence) whether and how much a particular parameter setting or a particular system variant increases/decreases system utility w.r.t the baseline utility.

Structure of a stage:

- Input parameters—come from the instantiation of the experiment or from previous stages.
- Output values—values that are provided as a result of the stage.
- Function to execute in the stage—this typically means modifying the system, collecting data from it and computing output values.
- Cost dependencies—specify which input parameters the stage cost depends on. This allows skipping the stage if experiment results for the stage are already known in the particular or similar settings of the parameters.
- Cost of the stage—this quantifies (based on input parameters) the expected impact on executing the stage on the system's overall utility.

3.3 Tackling uncertainty

Structuring the experiments in a workflow of stages and quantifying the cost of individual stages allows decoupling the coordination and control of the experiment from the description of the experiment itself. This creates a closed loop control in which effects of a stage are first assumed and then continuously measured w.r.t. to the system utility. This allows the experimentation framework to guarantee the dependability of the system being experimented with.

Evidence collected by experiments can be used both for off-line evolution and on-line adaptation. Contrary to the traditional view of experimentation, the inherent uncertainty in the design of the experiment itself (in particular the estimation of the stage cost) gives rises to parallel evolution/adaptation in two areas: (i) the system itself and (ii) the specification of experiments which come along with the system. In particular, in (ii), the data collected about the effects of an experiment stage can be further used retroactively to learn and fine-tune the cost estimation of a stage.

Though we have not systematically explored and implemented this learning of costs, this potentially allows extending the structure presented in Section 3.2 by replacing the direct specification of stage's cost with only indicators of what the actual stage cost depends on (formally, we assume that the stage cost is an unknown function of *cost dependencies*).

To illustrate the concept, consider optimizing the traffic prediction service by routing certain cars via alternative (possibly worse) routes to explore the traffic levels and flow. Obviously, this is connected with the cost of reducing the satisfaction of selected drivers. As such, the system has to carefully scale the extent of the experiment—it should estimate the cost of each alternatively routed car (taking into account the driver profile) and keep the overall cost below the system utility threshold. Similarly, the experiment is connected with computation cost needed for simulations to select the alternative routes that would have the highest positive impact on the overall system utility if explored. As these costs cannot be quantified upfront, the experiment designer can only indicate what they depend on (e.g. number of unexplored routes where traffic is not available, driver's profile, etc.). Specifically, this experiment would have a structure as outlined below:

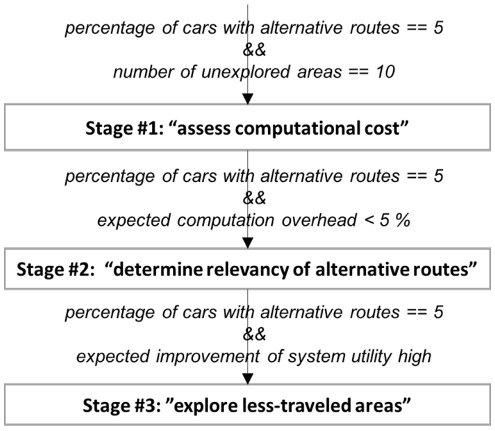
percentage of cars with alternative routes == 5
&&
number of unexplored areas == 10
Stage #1: "assess computational cost"
percentage of cars with alternative routes == 5
&&
expected computation overhead < 5 %
Stage #2: "determine relevancy of alternative routes"
percentage of cars with alternative routes == 5
&&
expected improvement of system utility high
Stage #3: "explore less-traveled areas"

Figure 1. Stages in execution of example experiment.

- Input parameters: percentage of cars with alternative routes.
- System utility: *function(average of trips overhead, user satisfaction rate, degree of resource utilization)*. Trip overhead is the actual duration of a trip versus the theoretical case where routing is performed based only on static map data (i.e. length and maximum speed of each street). User satisfaction is directly related to the number of complaints drivers issue when annoyed. Resource utilization is quantified via the CPU computation cost of the route planner.
- System utility threshold: *90%*
- Result: *system utility when experimenting / baseline system utility*. The baseline system is the system where no cars have alternative routes.

In all the stages of the experiment, the input parameters are the same: *percentage of cars with alternative routes*. The individual experiment's stages are depicted in the graph of Figure 1 and detailed below (note that we have not included actual costs, as these are learned over time):

- Stage #1: "assess computational cost"
  - Output values: *assessed computation cost of simulations to find out which parts of the city would have positive impact on planning if explored*
  - Function: Instrument the component of the route planner responsible for identifying "less traveled" routes and gather data to infer the overhead of its computation.
  - Cost dependencies: *number of unexplored areas*
- Stage #2: "determine relevancy of alternative routes"
  - Output values: *expected improvement of system utility if areas are explored*
  - Function: Determine less traveled areas and calculate differences between current routes and routes that take into account information on less traveled areas.
  - Cost dependencies: *computation cost assessed in phase #1*
- Stage #3: "explore less-traveled areas"
  - Output values: *system utility*, as specified in experiment definition
  - Function: Configure planner to route a certain percentage of cars through the less traveled areas.
  - Cost dependencies: *expected improvement of system utility*

4. PROTOTYPE TOOL & EXPERIMENTS

In this section, we first describe how we have reified the main concepts of our experimentation approach in a dedicated tool. We then give a brief overview of the testbed we set up to test our approach and on our initial experiments with using the tool.

```
experiments_seq = [
   {
      "knobs": {
         "reroute_frequency": 0
      },
      "ignore_first_n_results": 5000,
      "sample_size": 10000
   },
   {
      "knobs": {
         " reroute_frequency": 5
      }
      "ignore_first_n_results ": 5000
      "sample_size": 10000
   },
   … # more experiments here
]

primary_data_provider= {
   "type": "kafka_consumer",
   "kafka_uri": "http://kafka:9092",
   "topic": "trip-overhead",
   "serializer": "JSON",
   "data_reducer": trip_overhead_data_reducer
}

secondary_data_provider= {
   "type": "kafka_consumer",
   "kafka_uri": "http://kafka:9092",
   "topic": "router-load",
   "serializer": "JSON",
   "data_reducer": performance_data_reducer
}

def trip_overhead_data_reducer(state, newData):
   cnt = state["trips_count"]
   state["avg_overhead"] =
      (state["avg_overhead"] * cnt +newData["overhead"]) / (cnt + 1)
   state["trips_count"] = cnt + 1
   return state

def performance_data_reducer(state, newData):
   cnt = state["load_count"]
   state["avg_load"] =
      (state["avg_load"] * cnt +newData["load"]) / (cnt + 1)
   if state["avg_load"] > 20:
      raise StopIteration("routing got too expensive")
   state["load_count"] = cnt + 1
   return state

def evaluator(resultState):
   return – (resultState["avg_overhead"] + resultState["avg_load"])

```

Listing 1: Example of specifying a sequence of experiments in RTX.

4.1 Work-in-progress Implementation

To enable experimentation with live systems, we have been developing an open-source tool in Python, called Real-Time Experimentation[1] (RTX) [14]. So far, RTX can be used for specifying and running a number of experiments on a *base-level system*. Following the structure of Section 3.2, an RTX experiment consists of (i) a number of input parameters, (ii) a function to measure system utility based on the data coming from the base-level system, and (iii) a specification of the amount of incoming data that need to be analyzed. To avoid hysteresis effects (due to that the effect of a change in the base-level system may not be directly observable), an RTX experiment also includes a specification of the amount of data that need to be first skipped

[1] Available at: https://github.com/Starofall/RTX

before the data analysis starts. An experiment's system utility threshold and stages are currently not part of an RTX experiment definition; we detail on our plans to support both in Section 4.2. Finally, we have chosen not to include a "results" structure in the definition of an experiment, since we wanted to be agnostic w.r.t the baseline configuration (and corresponding utility) of the base-level system. Such results can though be easily generated by running two different RTX experiments—one with the baseline configuration, one with the experimental one—and comparing their resulting utility values.

Let us illustrate how RTX can be used in specifying and running experiments in the running example (our base-level system). Consider that we want to experiment with the frequency of re-routing requests on the route planner. In this case, the user specifies a sequence of experiments (Listing 1, lines 2-15), each of which sets a value to the same input parameter (called *knob* in RTX) and executes until it has received data points equal to the "ignore_first_n_results" (line 6) plus "sample_size" (line 7).

To evaluate the system utility that corresponds to an experiment, one or more data providers have to be specified, together with their corresponding *reducer* functions. The latter summarize the data as they come and store them to a "state" Python variable. This allows us to combine analysis results from different sources as inputs to a multivariate system utility function, the *evaluator* function in RTX.

In our example, two data providers have been specified (Listing 1, lines 19-25 and 27-33). The first one provides trip overheads (see Section 3.3), the second provides data measuring the CPU load of the router. Although RTX currently supports Kafka [10] as the default data provider mechanism, we are working on supporting also other publish-subscribe platforms such as MQTT. Each data provider has a dedicated reducer function (lines 35-40 and 42-49). Both functions in our example simply return the moving average of their input values. Finally, the evaluator function (lines 51-52) returns the inverse sum of the two averages, as the ultimate measure of system utility.

4.2 Planned Extension: Stages

We have so far focused on creating a working prototype of an experimentation tool that is generic and extensible. We detail on our plan to include stage-based execution of experiments here.

Essentially, an experiment stage can be seen as a regular RTX experiment, since a stage's structure closely resembles that of an experiment (Section 3.2). We therefore plan to build on the existing abstractions offered by the tool to implement stage-based execution in the following way:

- Each stage will be an RTX experiment.
- The execution of stages will be controlled by a custom experiment strategy that essentially implements the control structure of Figure 1. We have already provided support for three concrete implementations of experiment strategies in RTX, namely sequential, step, and self-optimizer [14]. The implementation of the custom strategy will be the responsibility of the RTX user (developer or tester).
- Each stage specification will contain a cost function, implemented as an *evaluator*.
- At any point during the execution of a stage, the tool should be able to monitor the cost on the system and halt the stage execution (and, optionally the execution of the experiment as a whole) if a specified cost threshold is exceeded. We have already provided a way to do this in RTX by raising a *StoptIteration* exception (Listing 1, lines 46-47, highlighted).

```
54. def trip_overhead_data_reducer(state, newData):
55.   if newData["isExperiment"]:
56.     cnt = state["trips_count "]
57.     state["avg_overhead"] =
58.       (state["avg_overhead"] * cnt +newData["overhead"]) / (cnt+1)
59.     state["trips_count "] = cnt + 1
60.     return state
61.   else:
62.     return state
```

Listing 2: Illustration of using a Boolean JSON field for distinguishing between data coming from cars using the experimental router vs the baseline one.

- When rolling out an experiment incrementally (as e.g. in the 3rd stage of Section 3.1), it is important to be able to distinguish between data coming from the components being experimenting with and the rest of the components. For example, we should be able to distinguish between data coming from cars routed with the experimental router and data coming from cars routed with the baseline router. For this, we plan to set a JSON-based protocol to allow the base-level system to tag the data it submits to RTX. An example of how this can work is depicted in Listing 2 (line 55, highlighted).

4.3 Indicative Experiments Using RTX

To evaluate the feasibility of running experiments using RTX, we have implemented a testbed[2] based on the running example. Our testbed uses SUMO, a state-of-the-art discrete-event mobility simulator and TraCi, a Python interface to SUMO. We have simulated 750 cars moving in a city with approx. 450 streets and 1200 intersections. The cars continuously travel from one randomly picked destination to another. They are all controlled via TraCI. For routing, they use our own Python-based router, which takes into account both static information (street lengths and maximum speeds) and dynamic information (estimated traffic per street, based on data provided to the router by the cars). The router can also optionally route certain cars to less traveled streets to obtain more up-to-date information on the travel level in these streets. In our setting, the simulation runs forever, to emulate a real-life live system.

We have equipped our testbed with several parameters that can be tuned at runtime, including re-routing frequency, measured as the number of SUMO ticks between re-routing: the smaller the number, the higher the frequency. We have also enabled the testbed to generate data that can be used in RTX: each car publishes its trip overhead upon reaching a destination (on the "trip-overhead" Kafka topic); the duration of router execution per SUMO tick is also monitored and sent to RTX (via publishing it on the "router-load" Kafka topic). We have used Kafka for both setting parameter values to the testbed at runtime and getting data emitted from the testbed.

We have successfully run a number of different experiments and managed to optimize the testbed at runtime via observing the promising regions for each parameter value and running more focused experiments. We report here our results with experimenting with the re-routing frequency parameter.

As depicted in Figure 2, we performed 50 experiments with different values of the re-routing frequency, starting from re-routing every 100 SUMO ticks and ending at re-routing every 2 ticks. Each experiment was specified with 10.000 ignore size and 10.000 sample size. We measured both the average trip overhead

[2] Available at: https://github.com/Starofall/CrowdNav

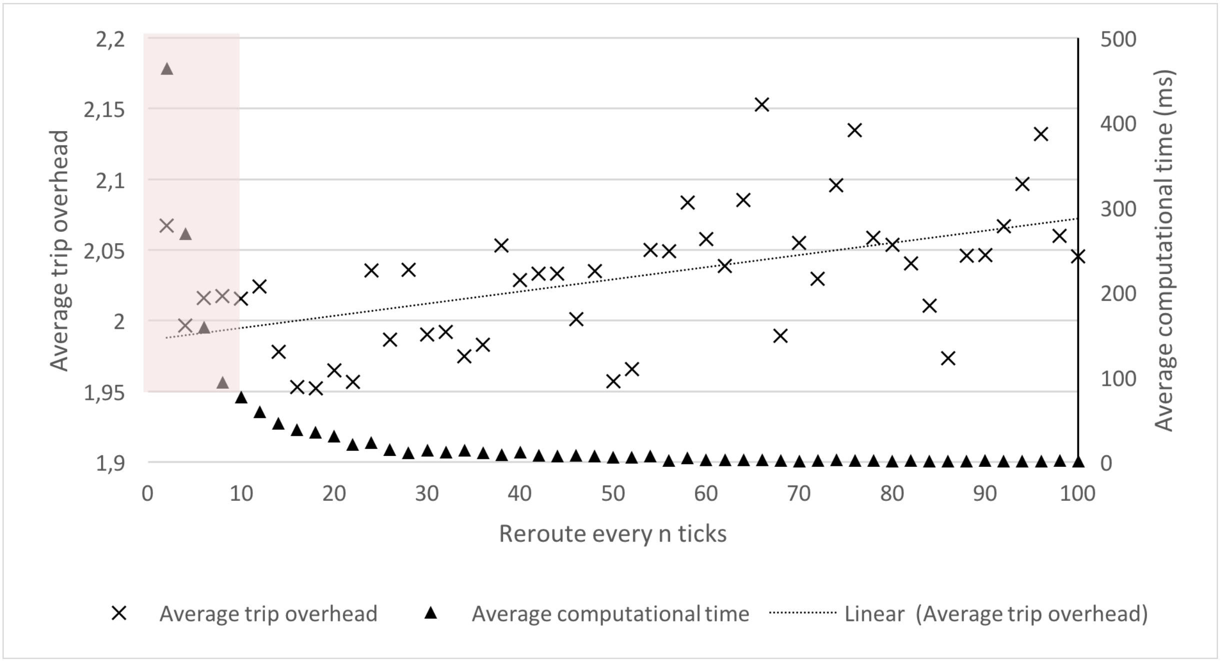

Figure 2. Indicative results when experimenting with the "re-route frequency" parameter.

(left axis) and the average computational time of the router (right axis). The results show that if we re-route more often, the average trip overhead is reduced (there is a linear trend despite the high variance). The computational time is almost stable up to a point where it increases rapidly. Having a threshold on the maximum computational time and using the cut-off mechanism of the tool described in the previous sections we can stop an experiment before it causes any damage to the base-level system, by making it operate in the dangerous zone highlighted on the left part of Figure 2.

5. RELATED WORK

Our approach touches upon several software engineering areas, including runtime self-adaptation, experiment-driven development, and performance engineering. We focus on the closely related work on the last two areas here; for a comparison to self-adaptation approaches we refer the reader to [14].

Experiment-driven development is present as an idea in different conceptual frameworks proposed in literature, namely evidence-based software engineering (SE) [3], data-driven SE [1], and value-based SE [2]. Evidence-based SE is a recently proposed vision of being able to validate any new development or change to a system from the perspective of the value it delivers. In this, new developments and changes are evaluated based on performing end-user experiments (e.g. A/B testing [9]). Data-driven SE is a practice that focuses on continuous collection of data to quantify metrics related to produce quality and make estimates of post-release failures early in the development cycle. Finally, value-based SE is a related practice that focuses on increasing a company's business value by improving the economic efficiency of the software they develop. Our approach draws inspiration from all the above conceptual frameworks and strives to provide an open-source, tool-supported framework as a next step towards realizing experiment-driven development.

As proposed in our work, systematic experimentation requires continuous monitoring of experimentation effects. Such monitoring is known to influence the system under observation, as explained for example in [11]. In some cases, such overhead can be separated from the useful measurement data—this has been done for example in the HPC context in [12]—however, the overhead then still impacts the overall system behavior. We therefore need to consider monitoring methods that reduce overhead by dynamically managing the instrumentation probes.

Measurement frameworks can manage the instrumentation probes either by enabling and disabling data collection, or even by inserting and removing probe code as needed. Both methods reduce the monitoring overhead, as has been illustrated for example in the context of the Kieker monitoring framework [6, 15], or with SPASS meter [4]. As we show in our work focused in particular on dynamic instrumentation management [7], probe insertion can introduce perturbations that persist even after complete probe removal, however, these perturbations are likely to be negligible in the class of systems we consider.

Even without extensive overhead management, limited monitoring has been shown to achieve reasonable overhead for example in potentially complex enterprise systems [13]. This leads us to believe that on the monitoring side, our work has generally feasible objectives.

6. CONCLUSION

In this paper, we have argued for the need of a systematic experimentation approach to deal with the uncertainties inherent to the design and development of many relevant software-intensive systems-of-systems. We proposed to specify experiments connected with system architecture, in particular tunable parameters of a system, which can enhance the understanding of system behavior. We argued for the need to launch such experiments in production environments, where the cost of experimenting has to be carefully evaluated. To do this, we have proposed a stage-based experiment execution that gradually rolls out an experiment to a system. We have described our initial prototype tool reifying these ideas. In the future, we intend to fully implement stage-based execution in our tool and evaluate it on a real-life case study.

REFERENCES

[1] Bird, C. et al. 2011. Empirical Software Engineering at Microsoft Research. *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work* (New York, NY, USA, 2011), 143–150.

[2] Boehm, B.W. 2006. Value-Based Software Engineering: Overview and Agenda. *Value-Based Software Engineering*. S. Biffl et al., eds. Springer Berlin Heidelberg. 3–14.

[3] Bosch, J. and Olsson, H.H. 2016. Data-driven Continuous Evolution of Smart Systems. *Proc. of SEAMS '16* (2016), 28–34.

[4] Eichelberger, H. and Schmid, K. 2014. Flexible resource monitoring of Java programs. *Journal of Systems and Software*. 93, (Jul. 2014), 163–186.

[5] Esfahani, N. et al. 2011. Taming uncertainty in self-adaptive software. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (2011), 234–244.

[6] van Hoorn, A. et al. 2009. *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*. Department of Computer Science, Kiel University, Germany.

[7] Horký, V. et al. 2016. Analysis of Overhead in Dynamic Java Performance Monitoring. *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (New York, NY, USA, 2016), 275–286.

[8] Humble, J. and Farley, D. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.

[9] Kohavi, R. et al. 2009. Online Experimentation at Microsoft. *Third Workshop on Data Mining Case Studies and Practice* (2009).

[10] Kreps, J. et al. 2011. Kafka: A distributed messaging system for log processing. *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB'11)* (2011), 1–7.

[11] Malony, A.D. 1990. *Performance Observability*. University of Illinois at Urbana-Champaign.

[12] Malony, A.D. and Shende, S.S. 2004. Overhead Compensation in Performance Profiling. *Euro-Par 2004 Parallel Processing*. M. Danelutto et al., eds. Springer Berlin Heidelberg. 119–132.

[13] Parsons, T. et al. 2006. Non-intrusive end-to-end runtime path tracing for J2EE systems. *Software, IEE Proceedings*. 153, 4 (Aug. 2006), 149–161.

[14] Schmid, S. et al. 2017. Self-Adaptation Based on Big Data Analytics: A Model Problem and Tool. *Proc. of 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'17)* (Buenos Aires, Argentina, May 2017), 102–108.

[15] Waller, J. et al. 2014. Application Performance Monitoring: Trade-Off between Overhead Reduction and Maintainability. *Proceedings of the Symposium on Software Performance 2014* (Stuttgart, Germany, Nov. 2014), 1–24.

[16] Wolf, T.D. 2007. *Analysing and Engineering Self-Organising Emergent Applications*. Katholieke Universiteit Leuven.