Chapter 12
Simulation

A Methodology to Evaluate Recommendation Systems in Software Engineering

Robert J. Walker and Reid Holmes

Abstract Scientists and engineers have long used simulation as a technique for exploring and evaluating complex systems. Direct interaction with a real, complex system requires that the system be already constructed and operational, that people be trained in its use, and that its dangers already be known and mitigated. Simulation can avoid these issues, reducing costs, reducing risks, and allowing an imagined system to be studied before it is created. The explorations supported by simulation serve two purposes in the realm of evaluation: to determine whether and where undesired behavior will arise and to predict the outcomes of interactions with the real system. This chapter examines the use of simulation to evaluate recommendation systems in software engineering (RSSEs). We provide a general model of simulation for evaluation and review a small set of examples to examine how the model has been applied in practice. From these examples, we extract some general strengths and weaknesses of the use of simulation to evaluate RSSEs. We also explore prospects for making more extensive use of simulation in the future.

12.1 Introduction

The creation and study of simulations is a traditional activity performed by scientists and engineers, aimed at understanding something about the "real world," in which the real world is too complex, too expensive, or too risky to directly understand well [29]. Consider two examples: a computer program that forecasts the weather and a wind tunnel containing a scale model of an airplane. In weather forecasting,

R.J. Walker (✉)
Department of Computer Science, University of Calgary, Calgary, AB, Canada
e-mail: walker@ucalgary.ca

R. Holmes
David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada
e-mail: rtholmes@cs.uwaterloo.ca

predictions about the weather are needed in advance in order to plan; while an unexpected rainy weekend is unpleasant, imagine an unexpected hurricane arriving. In the wind tunnel, avionics engineers can measure properties of a proposed airplane's performance; this avoids the high cost of constructing a prototype real airplane, avoids the risk to a real test pilot's life, and avoids the necessity of locating the precise physical conditions somewhere in the real world that are of interest.

Essentially, a simulation is an imitation of the functioning of one system by the functioning of another, typically simpler one; a simulation involves executing a model of behavior with specific inputs to obtain the resulting outputs. In other words, we seek to abstract away those details of the real system that are too complex or that otherwise are not considered important for what is being studied.

The word "simulation" can refer to the general, abstract idea ("simulation is a common methodology"); a specific instance in which the methodology is applied ("the simulation was conducted as follows"); and a particular execution of a specific model ("we observed interesting phenomena recorded during the third simulation"). Some research fields differentiate *simulation modeling* [6, 11, 29] as the activity that creates the static model that is then dynamically driven to produce the results, i.e., during the "simulation." While in principle this overloading of the term can confuse the reader, the context in which the term is used generally disambiguates the meaning.

Simulation is performed for three main purposes: (1) to estimate the answer to a problem whose exact computation would be too expensive to solve directly; (2) to explore the range of behaviors attainable from the model for a set of inputs that are representative in some sense; or (3) to predict a set of outputs that can then be compared against reality, for the sake of evaluating the model. Cases 2 and 3 involve evaluation and will be most pertinent to this chapter.

For recommendation systems in software engineering (RSSEs), few authors [5, 15, 23, 36] make mention of the term "simulation," often referring to their studies as simply "experiments" or "evaluations" [4, 12, 13, 18, 19, 21, 37]. An *evaluation* involves an examination of something to assess its merits. An *experiment* involves following a disciplined procedure to test a hypothesis, usually under controlled conditions. A simulation involves imitating the behavior of some process, usually for the purpose of study. Thus, experiments and simulations can be used in evaluation, and simulations can be used in order to conduct experiments. But a simulation need not involve experimentation (an exploration does not involve testing a hypothesis) nor even an evaluation (watching an animated simulation may be simply aesthetically pleasing).

Perceptions of the value of simulation can color the accepted usage of the term. For example, some disciplines make a distinction between "the use of computer techniques to perform calculations, on the one hand, and [proper] computer simulation, on the other" [16, p. 128]. Winsberg [35] claims that two characteristics distinguish "mere number crunching" from "true simulation":

1. the use of a variety of techniques to draw inferences from the numbers; and
2. the application of expertise and judgment to decide which results are reliable.

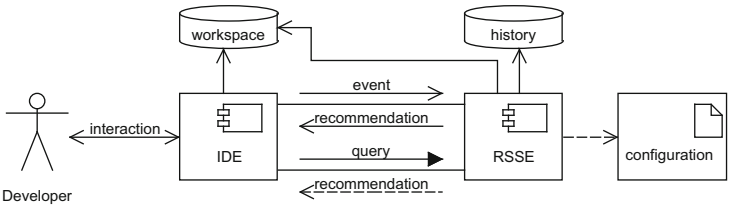

Fig. 12.1 A model of a typical RSSE

The essential point is to say that the algorithmic production of data does not imbue it with validity [2, p. 67]: garbage-in/garbage-out. A serious simulation must be designed with careful consideration of its underlying model and choice of inputs; triangulation of the results—in which different methodologies are applied to address a research question—is most likely to ensure that they are meaningful [8].

The remainder of the chapter is structured as follows. Section 12.2 describes a general model for the use of simulation in evaluating RSSEs. Section 12.3 describes examples from the RSSE literature that have made use of simulation for evaluation, focusing specifically on that use, and referring to our general model. Section 12.4 summarizes the lessons learned.

12.2 A General Model of Simulation for Evaluation of RSSEs

RSSEs come in many varieties, with differing characteristics, differing purposes, and differing design decisions [26, 27]. Nevertheless, consider the model shown in Fig. 12.1, which represents a common arrangement in many RSSEs. In it, a developer interacts with an integrated development environment (IDE) in order to perform development tasks. This interaction may explicitly involve asking the RSSE for recommendations (query/response), or the developer's activities may cause events to be reported to the RSSE, which in turn can cause changes to occur in the IDE (to announce recommendations). During these activities, the IDE will typically interact with some internal representation of the programs and other artifacts upon which it operates (the "workspace," which may include a version control system or other repositories); some RSSEs will also directly access this representation. Many RSSEs are configurable in some form, which we represent as an artifact upon which the RSSE depends. Furthermore, the RSSE may record and later utilize a history of information: for example, past decisions by this developer or decisions by others.

This is a potentially complex situation. The workspace and history can be large and can differ significantly between organizations; the human developer can be unpredictable; the IDE can contain bugs. Simulation can most obviously be used here in two ways: to determine how the developer will react to certain situations and to determine how the RSSE will react to certain situations. Unlike other simulation contexts (like the wind tunnel or in weather forecasting), we will typically have

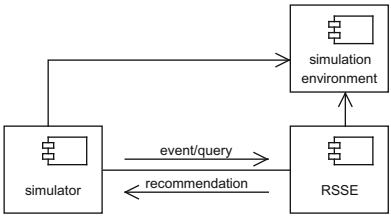

Fig. 12.2 A generic model for the typical simulation scenario for an RSSE

the RSSE already in hand, so the simulation will either involve (a) constructing an artificial workspace/history/configuration in order to see how the developer will react or (b) imitating the developer and the environment around the RSSE to examine how the RSSE behaves.

As a standard means of simplifying the situation, we note that the RSSE receives inputs and produces outputs, but that the real sources and sinks of that data can be imitated; this results in the generic simulation model of Fig. 12.2. In it, the developer is removed along with the IDE to be replaced with a *simulator* that generates events/queries and receives responses (likely recording these somehow); in practice, the simulator is simple and may not even be automated. A *simulation environment* for an RSSE is a combination of workspace, history, and configuration, appropriate for the particular RSSE being studied. Note that this generic simulation model will work for RSSEs that are not well described by the model of Fig. 12.1; its only assumption is that the RSSE takes input (explicit and/or implicit) and produces output.

This model is analogous to the standard model of *unit testing*, in which a software unit of functionality (e.g., a class) provides an interface that can be called, that can have data passed to it, that returns output, and that may depend on other units of functionality; we want to isolate the unit of interest, and so the other units that it depends upon are eliminated in favor of ones constructed to collect information and/or to return specific values to the unit under test. These replacement dependencies are often called *stubs* (they come in many varieties each using different names).

The RSSE is analogous to the unit under test, the simulator is the driver, and the simulation environment is the stub that replaces the other dependencies of the RSSE. Different scenarios can be explored by adjusting the content of the simulator and simulation environment. As with choosing the extent of a given unit in unit testing, the researcher can adjust the boundary between the RSSE and the simulation environment to achieve different purposes: for example, one might choose to have the RSSE construct and modify a real history over an extended interaction sequence, rather than just initializing a synthetic history directly.

An alternative simulation scenario makes sense in some settings (see Fig. 12.3). In this scenario, the RSSE is not present, but the researcher wants to evaluate the reaction of the developer to potential recommendations, possibly derived from data previously collected from them. In this case, the RSSE itself is simulated: its recommendations may be computed offline or synthesized and can be presented

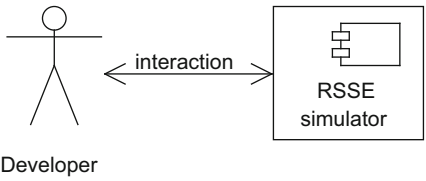

Fig. 12.3 A generic model for an alternative simulation scenario for an RSSE

directly by a human being, as a paper prototype, or as a mocked-up program. This case is not currently common in the RSSE literature, but it is not unknown in other areas—so-called Wizard of Oz experiments [17] are one variation on this idea.

Given either of these setups, it is necessary to determine with what inputs the simulation will be driven and what to do with the outputs that result. Ultimately, these decisions are important and should be based on the purposes for conducting the specific simulation. The chapter henceforth focuses on the case where the RSSE's context is being imitated (as per Fig. 12.2).

12.2.1 Inputs and Outputs

Let V be the set of possible simulation environments for a specific RSSE. In addition, let Q be the set of possible queries on the RSSE. Then the set I of possible inputs will be $I \subseteq Q \times V$ (it is a subset because not all queries may be possible for all simulation environments). Furthermore, let R be the set of possible recommendations that the RSSE could possibly produce, and let M be the set of meta-results (like the time needed to perform its calculations) that are not derivable from R. Then the set O of possible outputs from the RSSE will be $O \subseteq R \times M \times V$, where V is included as the RSSE could modify the simulation environment. Thus, we can see the RSSE as defining a function $f : Q \times V \mapsto R \times M \times V$. For most RSSEs, the input space and output space will be too large to evaluate exhaustively; even sampling thoroughly such a large space will be infeasible in general [31].

While the need to abstract away from the full input and output spaces ought to be obvious, there is the danger of oversimplification that can lead to poor generalizability and questionable meaningfulness [9, 32]. The inputs tried and outputs obtained ought to comprise a representative sample of the possible inputs and outputs—that is, the results ought to generalize to the full space, or at least the subset of that space that is considered most important. There are three basic approaches to obtain representative samples. (1) Consider the full input space abstractly, without concern for the relative likelihood that a given query will occur in practice. (2) Consider the intended application, where we have knowledge or assumptions of realistic inputs and can judge whether a given input is likely. (3) Select inputs so that the resulting outputs are representative of the output space, which can require either that the function f be invertible or that a search process be followed to find an input that can obtain a given output. Hybrid approaches between

the three basic ones are also conceivable. In general, the desire is to sample more heavily those regions of the full (input or output) space that are more likely to occur in practice. But sometimes, the researcher is interested in determining the overall characteristics of the space, such as whether problematic states can ever result.

Simulations of RSSEs often consist of multiple trials of single-step simulations: each trial i consists of selecting $q_i \in Q$ and $v_i \in V$ to obtain $r_i \in R$, $m_i \in M$, and $v'_i \in V$. But it is also possible to have each trial involve multiple steps, where $v_{i,j}$ is $v'_{i,j-1}$ obtained from the previous step. In this way, emergent behavior of an RSSE that alters its environment can be investigated; this is obviously only of interest where $v_{i,j} \neq v'_{i,j-1}$ for at least some trials. Whether single-step or multiple-step trials make more sense depends greatly on the characteristics of the specific RSSE and the purpose of the simulation. In practice, many RSSEs do not directly modify their environment, although they are used within environments that change over time, for example, where a version control system tracks code modifications and the RSSE uses this information.

12.2.2 Characterizing the Results

How a researcher should characterize the results of simulation trials depends on what the purpose is for conducting them. Such purposes could involve (1) description of individual results without reference to an external notion of what would be good; (2) indication of under what conditions certain classes of results occur; or (3) assessment of the quality of the results.

As an example of simple, descriptive summarization, if the execution time of the RSSE is to be described, standard descriptive statistics will often suffice (i.e., minimum, maximum, mean, standard deviation), but sometimes, a graphical plot of the execution time may be better—especially if the researcher has noticed that the execution time appears to have a relationship with other factors. These kinds of characterizations depend heavily on the numeric nature of the (meta-)data being characterized and are not appropriate for categorical data, in particular, which is what a recommendation would typically consist of.

Simulation can also be used to explore the behavior of the RSSE without concern about the "correctness" of its recommendations. This can be appropriate in situations where the general properties of the RSSE's behavior, relative to the inputs, are of interest. For example, if one wished to determine under what input conditions the RSSE would provide no recommendations, simulation could be used to probe the input space to address the question. In practice, such questions are usually supplementary to asking about the "correctness" of the recommendations.

In many settings, we want to assess the quality of the resulting recommendations. Quality is an imprecise term that may possess both objective and subjective elements; as a result, different stakeholders can have significantly different opinions about the quality of a recommendation. Consider that the needs of a novice are often very different from those of an expert: the same recommendation given to each

would likely attain different opinions as to its quality. Furthermore, users' needs depend heavily on the context of their applications: in some contexts, incorrect recommendations would be disastrous; in others, recommending all possible answers is irrelevant as long as a single expected recommendation is actually recommended; in yet others, the order of individual recommendations will matter.

In the typical RSSE simulation scenario, a key purpose is to avoid the use of collecting subjective assessments and the complications outlined in the previous paragraph, so some means of determining the "right answer" (i.e., the *expected recommendation*) is needed that would be expected to be recommended by an ideal oracle with perfect knowledge (see Sect. 12.2.2); this can then be used to assess the objective aspect of the quality. Expected recommendations are often derived from data collected from the real world; note that this does not automatically mean that the expected recommendations are objectively "correct" though (see Sect. 12.2.2). A variety of measures are available to characterize the objective quality, with varying levels of detail and appropriateness (see Sect. 12.2.2). But one must be careful: this purely quantitative approach to assessing quality may not result in an accurate reflection of the user experience. Ultimately, human-participant studies are needed to determine whether the quantitative analysis of quality agrees with the reality; this would be a form of triangulation.

In Chaps. 11, 13, and 9, Said et al. [28], Tosun Mısırlı et al. [33], and Murphy-Hill and Murphy [24] (respectively) expand on the idea of a more complete evaluation of an RSSE, often involving real developers.

Determining Expected Recommendations

Most commonly, it is the quality of the RSSE's recommendations that is to be evaluated; this requires knowing, assuming, or otherwise estimating the expected recommendations. With a given expected recommendation for a given input, the researcher can compare the RSSE's actual recommendation against the expected recommendation for the given input.

It is generally problematic to determine the expected recommendations. In many situations, it is impossible or impractical to automatically generate the expected recommendations; otherwise, the RSSE would use that algorithm and the "RSSE" would no longer qualify as a recommendation system (instead it would compute the correct answer). There are three possibilities for determining expected recommendations: (1) asking human participants for their assessments; (2) using a variety of other automated approaches comparatively; or (3) using data collected previously from the real world.

Human participants from an appropriate population (e.g., students, experts, etc.) can be asked to either provide the correct answers or judge whether the RSSE's answers are correct. When the definition of correctness is dependent on the context of the task for which recommendations are being produced, it is important that the participants understand the context of the task in order to make such judgments. Detailed instruction and training exercises with feedback are common techniques

for ensuring that participants have a common understanding of the tasks to be performed. It is also important that the study design avoid the participants' biases: their tendency to assume that the recommendations are correct, or their wish to provide the answers that they assume are desired by the experimenters. Ideally, the experimenters should avoid giving any hint of their own opinions and avoid indicating whether the RSSE being studied is their own. When the definition of correctness being used is subjective, the human participants will tend to differ in their opinions. Often, the majority opinion is interpreted to be the expected recommendation (particularly for categorical data); other options include using the average (for numerical data) or allowing multiple possible expected recommendations. But when participants' opinions differ, there exists a threat to validity of the results: it may be that the experimenters failed to instruct the participants sufficiently, or that the task is too subjective. Minor variations in opinions are often ignored without serious problems. But even when participants' opinions agree, there is no guarantee that no threat to validity exists; it could be that all the participants share the same bias, and so a systematic error exists in the experiment. In either case, it is best practice to be explicit that the threat exists.

Attempting to estimate the expected recommendation on the basis of other automated approaches (e.g., other RSSEs) is fraught with danger. First, if based on published results of the other approaches' application to the same data, there is a strong chance that the current RSSE will have been developed with the knowledge of those results—it has been overfitted to this data. Second, triangulation of other RSSEs' recommendations is no guarantee of the correctness of those other RSSEs nor of their "averaged" results; a researcher will be biased in evaluating novel recommendations not in the union of the recommendations from the other RSSEs (see previous paragraph). As argued in the literature [3,7], although determining the ground truth for expected recommendations may be costly, reference to the ground truth is the only way that an evaluation of quality can approach a lack of bias.

Real Data

For many RSSEs, the researcher can take advantage of some sort of real-world data in producing the set of queries and/or the simulation environment. For example, a repository of open-source programs can serve either to produce queries that ask about source constructs or as the simulated workspace from which the RSSE will draw its knowledge. If a reasonable argument can be made that the data thus used is representative, or at least not a bad representation of other inputs (because it is not too trivial, too large, or too biased in some way), such real data can often avoid the high costs and questionable validity of constructing that data artificially.

For some RSSEs, a repository of data is available that contains both examples of real queries and the corresponding examples of expected recommendations; that is, the repository D consists of pairs $d_i = (q_i, e_i)$ of queries q_i and corresponding expected recommendations e_i. (Note that the generality of these extracted examples is dependent on the representativeness of the source of the data, too.) If the RSSE

need not draw upon a workspace or history, each query q_i from the repository can be posited to the RSSE, and its actual recommendation a_i can be compared against the expected recommendation e_i. If the RSSE does need to draw upon a workspace or history, it is important to separate the data used for querying from the data used to form the workspace or history—otherwise, the RSSE would already have access to the expected answer for that query, which will generally not be a representative simulation situation. The typical approach to this process divides the real data into a training set and an evaluation set. When this process is repeated with k different partitions (often chosen at random) and measures of quality are averaged over each, it is called a k-fold cross-validation [22]. In many RSSE situations, a truly random partition of the data cannot be chosen, but instead all data before a given point (for example, a particular timestamp) are used as the training set, and the remaining k data items are used for the evaluation set; this is called k-tail evaluation [21].

It is possible to draw both the queries and the simulation environment's data from the same data items, but this is necessarily a tricky proposition. Given a data item $e_i \in D'$ that will ultimately serve as an expected recommendation, one can construct its corresponding query q_i by extracting a subset of the information from e_i. If that set of information is too perfect, this will not be a fair evaluation. Thus, the researcher must define a transformation $\mathscr{T} : D' \rightarrow Q$ that will obfuscate the original identity of e_i from the RSSE. For example, elements can be removed from the set of extracted information, elements can be added to it, or elements can be modified to hide their nature. The details of fair and appropriate transformations depend heavily upon the application context, the design of the RSSE, and the research questions being addressed. A researcher can expect to have difficulty convincing reviewers that the chosen transformation is not biased.

Evaluating the Objective Aspect of Quality

Researchers are often interested in the objective quality of the recommendations produced by an RSSE. As discussed above, a careful choice of the inputs (both queries and simulation environment) is important to obtain meaningful results, and availability of the expected recommendations is needed for the sake of evaluating the quality. But in the absence of perfect agreement or perfect disagreement between the actual and expected recommendations, one needs a way to assess the quality. This is often done with measures borrowed from the field of information retrieval, but several aspects of their use is important to note: (a) there are many such measures available with differing strengths and weaknesses; (b) summarizing a set of observations of agreements/disagreements between actual and expected recommendations necessarily eliminates information—the fewer in quantity that the resulting summary measurements are, the more information that has been lost; (c) the measurements are correct only for the specific inputs and outputs for the RSSE, and these will generalize to other situations only if the inputs were representative of those other situations; and thus (d) comparing two RSSEs requires

that the same measures be used on each, and that they be collected in identical situations.

A useful notion in evaluating quality is the confusion matrix, shown below:

		Expected	
		Yes	No
Actual	*Yes*	TP	FP
	No	FN	TN

For any given item, the two dimensions represent what the expected recommendation is (either "Yes," it is in the set of interest, or "No," it is not) and what the actual recommendation from a given RSSE and simulation environment was. Where the expected and actual recommendations agree, we have a *true* recommendation from the RSSE—either a true positive (TP) or true negative (TN); in case of disagreement, we have a *false* recommendation from the RSSE—either false positive (FP) or false negative (FN). Typical RSSEs will provide multiple recommendations in a given situation, so each recommendation in the set can then be classified within a confusion matrix; this results in a four-valued characterization of the quality of that recommendation.

Often, people use a characterization of quality with fewer values, in which the confusion matrix is reduced to other measures [34]—for example, the precision and recall, or just the F-measure (the harmonic mean of the precision and recall). To summarize the overall quality of recommendations from a set of trials, one can either populate a single confusion matrix (called microevaluation) or use a means of summarizing the individual quality measures [30] (called macroevaluation): for example, a simple approach is to take the mean over the individual measures.

One can introduce schemes to weight certain cells in the confusion matrix, allowing us to account for contexts in which, say, false positives are problematic. Such a scheme ought to possess some a priori justification, such as empirical knowledge of the application context, in order to achieve construct validity [10, 20].

Variations on the standard idea of the confusion matrix are possible. For example, one can use matrices with higher dimensionality in order to simultaneously compare multiple RSSEs with expected recommendations. One can allow more than two outcomes: some RSSEs are explicit about recommending to do something, recommending not to do something, or making no recommendation [e.g., 15]. One can permit the confusion matrix cells to represent fuzzy sets: the probable count of cases that fall within a cell. For example, recommendations often come with confidence values attached to them that can be interpreted as the probability that the recommendation is right. Each cell of the confusion matrix would then be a sum of the probabilities of the recommendations that fall therein.

In Chap. 10, Avazpour et al. [1] expand on the notion of quality, exploring a variety of other measures.

12.3 Experience with Simulation to Evaluate RSSEs

We proceed to examine four papers from the literature on RSSEs that have applied simulation for the sake of evaluation. These four papers use simulation in different problem contexts and in different ways; common strengths and weaknesses of simulation for evaluation can be seen from these. Section 12.3.1 examines eROSE (originally called ROSE) [37]; its evaluation solely involved simulation, derived from data collected in industrial version control systems. Section 12.3.2 examines Strathcona [15]; its evaluation used simulation only to generalize the results from its formal experiments. Section 12.3.3 examines Gilligan+Suade [12]; its evaluation made heavy use of simulation, derived from data previously collected during a formal experiment. Section 12.3.4 examines an unnamed approach for recommending development environment commands [23]; its design started from a simulation, derived from data collected from actual industrial use of an IDE. It is particularly interesting as it is a rare case in which the RSSE itself was simulated.

Only enough detail is provided to describe the application problem addressed and the solution pursued in order to contextualize the use of simulation in the evaluation of the research. In addition, each subsection attempts to emphasize the evaluation problem that the research attempted to address through simulation, details of the simulation procedure followed, and threats to the validity of the results as reported by the authors of each paper.

12.3.1 Recommending Programmatic Entities to Change: eROSE

Real software systems tend to be large and complex. As a result, developers can have trouble recognizing dependencies between different parts of a system: in some cases, they fail to see explicit dependencies due to excess visual noise from other code; in other cases, the dependencies are too subtle to easily detect. As a result, when the developer modifies one part of their system, they are liable to overlook other parts that should also be changed. Automated analyses of the control- and data-flow within the system can help in some cases, but undecidability can limit the effectiveness of such analyses.

The Recommendation System

Instead of analyzing the structure or runtime of the program to look for dependencies, past human experience can be leveraged as recorded in a version control system. By detecting that two (or more) entities have tended to change together in the past, the hypothesis is that they are likely to change together again in the future.

Thus, in detecting the fact that a developer has modified one or more entities in the program, recommendations for other entities to change can also be made. This is the premise of the eROSE tool [37].

eROSE mines the history to locate commits: sets of changes simultaneously submitted, where a change consists of a change type and an entity. Commits must be inferred, particularly for repositories using CVS (which versions only individual files, and so groups of files simultaneously submitted can be detected only indirectly). Furthermore, entities are considered the same (and hence two versions of the same entity) if and only if their names (or signatures, for methods) and the names of their structural ancestors are identical. The presence of two (or more) changes occurring frequently enough together leads eROSE to infer a rule that a change to one entity likely ought to be accompanied with a change to that (or those) other entities. CVS supports branching to allow independent development paths of different variants of a system; in some cases, changes within a branch are merged back into the main trunk, resulting in apparent commits involving very large numbers of entities. eROSE heuristically ignores commits that involve too many entities, in order to avoid considering merges.

In general eROSE takes a set of changed entities as the query from the developer's IDE, mines for rules involving those entities, and makes recommendations about other entities to investigate. For each recommendation it indicates two measures of relevance: the confidence, representing the frequency with which the rule has applied previously for equivalent queries, and the *support count*, indicating how many cases have gone into constructing the rule.

The Evaluation Problem

eROSE operates by finding other entities to recommend once the developer has made a change to a system. If real developers were provided with eROSE while they performed change tasks, very little empirical data would be collected relative to the large amount of time required to conduct the experiments. Furthermore, eROSE has many aspects that could be and were evaluated, including: the length of history to be analyzed; the kind of scenario in which it is being applied; the kinds of changes to be analyzed from history; the minimum thresholds of confidence and support count at which to make a recommendation; and the specific system to which it is being applied. Simulation is a promising methodology to apply here in order to assess a wide variety of situations at relatively low cost.

How Simulation Was Used

A set of industrial software systems, each with an available change history, were selected. The general simulation procedure that was used followed four steps: (1) a time-limited portion of the history was designated as the training set to be mined by eROSE (the training set formed the simulation environment V); (2) the remainder

of the history was used to collect a set of commits T; (3) each commit $t \in T$ was partitioned into a query q and an expected recommendation e; and (4) eROSE was given each query, and the actual recommendation was compared against the expected recommendation.

Several variations on this general procedure were followed. In particular, the manner of partitioning the commits into queries and expected recommendations was varied according to three conjectured usage scenarios.

1. *Navigation* involved the simulated developer making a single change and recommending to them a set of other changes. For each commit t_i, $|t_i|$ distinct queries q_j were formed such that $|q_j| = 1$.
2. *Error prevention* involved the simulated developer making a set of changes but missing one. Again, $|t_i|$ distinct queries were formed, but each one contained the entirety of the commit except for one change; hence, each $q_j = t_i - \{e_j\}$ for some unique e_j.
3. *Closure* involved checking that eROSE would not recommend additional changes when the whole commit was used as the query. This involved one query for each commit, for which $e_i = \emptyset$.

The general simulation procedure was then followed for several cases.

- The effects, on the quality of eROSE's results, of selecting thresholds for confidence and support count were explored for each of the usage scenarios. For each scenario, different levels were selected representing the appropriate trade-off between precise recommendations (few wrong recommendations) and complete recommendations (few missing, correct recommendations). This can be seen as a configuration phase.
- With the preferred settings for a given usage scenario in place, the quality of eROSE's results was then evaluated for that scenario.
- The effects of the level of granularity of the entities being analyzed and reported were considered. The configurations from the previous item were repeated, but for which the entities were considered only in terms of files, rather than individual functions or variables. The quality of eROSE's results was then evaluated and compared against those from the previous item.
- The effects of restricting changes to consider only alteration ("maintenance") events, as opposed to addition or deletion of entities, were considered. Two conditions were compared: where only maintenance events were considered, and where all change events were considered. The navigation scenario with its preferred configuration was again used to instantiate the general procedure.
- The effects of differentiating the kinds of change events, as opposed to treating them all as generic change events, were considered. The configuration from the previous item, in which all change events were considered was used again; this time, two conditions varied this configuration: whether the kinds of change events were differentiated or not.
- The effects of the duration of the project's history were considered on the quality of the recommendations, both in terms of looking at the intervals from the project

start until a set of specific moments and in terms of looking at the intervals of a specific length prior to a set of specific moments. This investigation was restricted to two projects.

Reported Threats to Validity

Zimmermann et al. [37] report four possible threats to validity in their work on eROSE. (1) More than 100,000 commits on eight large open-source systems were studied. Although these systems differ in terms of domain (and likely also in terms of development processes, design issues, etc.), the results may not be representative of all systems. This issue plagues much research in software engineering, regardless of methodology applied or problem context investigated. (2) Transactions do not record ordering information about individual changes. Zimmermann et al. express concern that different frequencies for specific orderings of changes could affect results for the navigation and error prevention usage scenarios. (3) Transactions are not assessed for their quality. Any commit that is not filtered out by the branch-merging heuristic is used by eROSE. This is potentially problematic since developers sometimes make bad decisions, and so the expected recommendations extracted from the history would differ from the true expected recommendations. But since bad decisions will make their way into the version control system far less frequently than good decisions, the effect on the expected recommendations is likely small. (4) There is a difference between a recommendation being correct and it being useful. Assessing the usefulness of recommendations would require a different methodology to be applied.

12.3.2 Recommending Usage Examples for an API: Strathcona

Developers frequently make use of libraries and frameworks to create software applications. Libraries and frameworks provide application programming interfaces (APIs) specifically for this purpose. For nontrivial cases, understanding how to correctly utilize an API can be difficult: particular subtypes must be provided, particular objects must be created, and particular methods must be called in particular orders. Examples are often used by developers to understand usage scenarios for APIs, but this requires (a) that the examples exist and (b) that the developer know how to locate the appropriate example for their needs.

The Recommendation System

To overcome these weaknesses, a recommendation system can be created that allows the developer to specify the kind of example of interest. Many forms of specification are possible, but few are developer-centric, placing a high burden on the developer to be precise and accurate. Instead, the fact that the developer is specifically trying to interact with the API means that they will have a partial implementation of what interests them, a *skeleton*. This skeleton may not even compile, but hopefully describes certain details of the interaction that matter to the developer. Furthermore, for many APIs, source code already exists that uses it in some form. By extracting information from the skeleton and looking for existing source code that also contains (some of) the same information, we can hope to provide meaningful examples to the developer.

This is the basic idea behind the Strathcona example recommendation system [15]. Strathcona utilizes only the structural facts it locates in the skeleton (the developer's class containing the skeleton, plus the skeleton's supertypes; types used in the skeleton; methods called), but this is often enough to locate even uncommon examples. Strathcona was configured to weigh the importance of particular facts; these heuristics were determined over time with informal experimentation.

The Evaluation Problem

Many aspects of Strathcona could be and were evaluated. For example, user studies were conducted to determine whether developers would be able to interpret and utilize the recommended examples. The generalizability of these studies was of concern: they were expensive and focused on a small set of tasks that (while of differing levels of complexity) were not definable as representative of all possible tasks. In particular, to reduce variability between subjects, skeletons were provided to the participants.

How Simulation Was Used

In order to generalize from these user studies, a simulation was conducted to evaluate Strathcona's ability to return appropriate examples given varying skeletons.

The general simulation procedure consisted of four steps: (1) code fragments of between 10 and 20 lines in length were selected at random from the repository that Strathcona was using (these were the expected recommendations); (2) the set of structural facts used by Strathcona were extracted from the selected code fragment (with references eliminated to entities that would easily identify examples, such as private methods); (3) subsets of these filtered structural facts were selected to form queries; and (4) it was recorded whether the Strathcona server was able to locate the target answer amongst the top ten matches.

The general procedure was followed in two variations: (1) all subsets of the structural facts were used, even when these were clearly generic (e.g., calls to methods on `String`) and (2) the structural facts were restricted to eliminate those that were deemed generic, then all subsets of the restricted structural facts were used. The purpose of the second variation was to determine how many important facts needed to be known by a developer in order to locate the target answer.

A graph was provided for each of the two variations, in which the number of facts in the query was plotted against the occurrence rate of the target answer, for each of the (four) randomly selected code fragments.

Reported Threats to Validity

Although the simulation part of the evaluation was conducted by Holmes et al. specifically to address threats to validity arising in other parts of their evaluation, the simulation itself possessed three reported threats to validity. (1) It is unknown if the queries (which are the essence of what is extracted from the skeletons) that are successful in actually recommending the expected recommendation are representative of the queries that a developer would form in practice. (2) The examples selected for the simulation all contained slightly fewer than the average number of structural facts. It is unknown if this biased the results in a serious way. (3) The simulation focused on the APIs provided by the Eclipse IDE. Although the paper describes informal experience with other APIs, it is possible that the results are not representative of general APIs.

12.3.3 Recommending Dependency Treatments During Reuse: Gilligan+Suade

While software reuse has long been pursued for its potential benefits for productivity and quality, traditional approaches require that the needed form of reuse of functionality be predicted ahead of time and be explicitly designed for. When unpredicted reuse scenarios occur, developers will often pursue a *pragmatic* process of reuse, involving copying and modifying the source code that provides their needed functionality.

During such a task, the developer attempts to balance the desire to reuse as much code as possible that implements the desired functionality, with the desire to avoid reusing as much irrelevant code as possible. A pragmatic reuse task involves the developer navigating the source code for potential reuse, following dependencies

between elements, and considering the cost of eliminating each element versus the (future) cost of retaining each element. The Gilligan tool [14] allows the developer to record their decisions about whether to retain or eliminate elements while investigating the functionality to be reused; the result is a plan about the pragmatic reuse task that can be semiautomatically enacted at any point, and atomically undone if the results are unsatisfactory, to be revised when necessary.

The process of creating a pragmatic-reuse plan is an iterative one in which the goal is to find the ideal dependencies at which to "cut away" unwanted functionality while minimizing the effort required to repair or replace the resulting dangling references. This is a complex decision process hampered by four factors: cutting a given dependency may eliminate relevant functionality; cutting a given dependency may incur high costs to repair the resulting dangling reference; *not* cutting a given dependency may fail to eliminate irrelevant functionality; and not cutting a given dependency may force us to cut a dependency in an even worse situation. Analyzing the possibilities requires both local and nonlocal reasoning to determine good dependencies at which to cut. Gilligan does not directly aid in making these decisions, but only in recording them and analyzing whether the overall plan is complete.

The Recommendation System

Developers are generally good at local reasoning about repairing dangling references, and low-cost analyses are unlikely to improve upon the manual approach. In contrast, nonlocal reasoning on the dependency graph is much more difficult as standard tools provide at most localized views of this information. An opportunity exists for a recommendation system that draws on knowledge of transitive dependencies and a model of cost and relevance to suggest where good or bad cuts could be made. This is the central idea of the Gilligan+Suade tool [12].

Gilligan+Suade takes a partial pragmatic-reuse plan and the dependency graph of the system from which functionality is to be reused; it makes recommendations to cut dependencies (*reject* the depended upon entity) or not to cut dependencies (*accept* the depended upon entity). For a given depended upon entity, it may make no recommendation about its treatment. The recommendations are revised as the developer makes additional decisions or revises previous decisions.

These recommendations are based on two heuristic measures for each entity: its *structural relevance* [25] and its *reuse cost*. The structural relevance (as defined by the Suade tool [25]) is based on heuristics that attempt to characterize the local shape of the dependency graph; dependencies from elements that possess fewer dependencies are each deemed more relevant, and dependencies back to entities already marked as being reused are also deemed more relevant. The reuse cost is based on the number of descendants of a given entity in the dependency graph, weighted by the length of the shortest path to each of them.

The Evaluation Problem

Human participant studies involving the performance of pragmatic reuse tasks are expensive to design and run, as the tasks cannot be trivial if they are to be meaningful. A set of such experiments had been conducted previously [14] during which interaction data from developers' actions with Gilligan were recorded. Rather than repeat such actual experiments again, it was desired to make use of the recorded data. A recommendation system for these decisions had been specifically requested by the study participants to enhance the usefulness of Gilligan.

How Simulation Was Used

Two simulation phases were performed. In the first phase, each experimental session was replayed by executing the developer's decisions in chronological order, in order to reconstruct the partial pragmatic-reuse plans at each moment. The recommendations that would have been displayed by Gilligan+Suade were computed for each instant at which a decision was actually made.

The sessions for which the data was reused ultimately resulted in successful pragmatic-reuse plans; "correct" decisions were deemed to be those for which at least 75 % of the developers agreed, and these were treated as the expected recommendation in each situation. Each actual recommendation was thus compared against the expected recommendation and the developer's immediate decision, which they sometimes changed (even multiple times) later. The quality of the recommendations was reported in terms of agreement or disagreement with the expected recommendations; the presence of the cases where no recommendation was made inhibited the use of standard quality measures.

The second simulation phase was performed because the authors perceived that the behavior of an actual developer in performing a pragmatic reuse task could be quite different in the presence of the RSSE: the results of the first simulation phase might have had little external validity. The second phase thus involved one of the authors (Holmes) using Gilligan+Suade to repeat the experimental tasks to see how it would affect his decision behavior and whether the tasks would still be successful.

Holmes spent little effort confirming cases with recommendations and never disagreed with these, but focused carefully on the (minority of) cases where no recommendation was forthcoming. The result was successful completion of both tasks according to the criteria of the original human-participant experiment, but at the cost of a small expansion of the code having been reused.

Reported Threats to Validity

Holmes et al. [12] explained that their rationale for conducting the second phase of the simulation was the concern that the presence of the recommendation system could have affected the developer actions enough to have invalidated the use of

the recorded data. Nevertheless, the second phase of the simulation involved only one author following a largely mechanical procedure in order to gain confidence that the results were promising. This involved a small set of tasks that may not have been representative of all tasks, by a "participant" who possessed a biased perspective.

12.3.4 Recommending Development Environment Commands

IDEs have grown increasingly complex. With their increased usage and increased ease of extension, a plethora of tools have been added to them. While the existence of the right tool for a given task is of benefit to the developer performing that task, that benefit can be realized only if the developer is aware of the existence of the tool, of how the tool can be activated, and of how to use the tool. At some point, tools become hard to find within an IDE because of their large numbers: this is the proverbial "finding a needle in a haystack." If the developer knows that the tool exists, simple navigation strategies like searching and browsing will likely suffice to find it. But if the developer is unaware that a tool exists, that a better tool exists than the one they are using, or that a simpler command exists for activating that tool, they will not even know to perform a search or browse.

The Recommendation System

When a developer learns how to make use of a tool, their initial attempts can be awkward and inefficient; upon discovering a more effective approach, they abandon their original style of usage for the one that they see as better. This gives rise to a detectable pattern over time. Patterns of interaction by the current developer can be compared to such patterns from other developers. If those other developers eventually abandoned a style of usage that the current developer is also following, the improved style of usage adopted by those others can be recommended to the current developer. This is the premise of the work of Murphy-Hill et al. [23]. Their proposed recommendation system is proactive: it observes the developer's interactions with an IDE and makes recommendations when it can.

The Evaluation Problem

Murphy-Hill et al. were in possession of a large repository of data concerning developers' interactions with an IDE. Their problem was to decide on an algorithm (out of eight possibilities postulated) that would most effectively leverage this data to recommend novel commands to developers and, ultimately, to determine whether such a recommendation system would be useful in practice.

How Simulation Was Used

Two simulations were performed. The initial phase involved an automated simulation in which k-tail evaluation (see Sect. 12.2.2) over the data repository was performed. The second phase involved a human participant-based evaluation of the usefulness of a set of recommendations; this also was essentially a simulation. Both simulations focused on commands in the Eclipse IDE (e.g., commands involving the use of CVS; commands involving the editing and refactoring of Java source code).

For the automated simulation, the general procedure was as follows. The data repository was factored into interaction histories for individual developers. For a given value of k, the last k commands discovered by each developer was determined, by detecting the first occurrence of each command in the interaction history for a given developer. (Developers without k command discovery events were immediately eliminated from consideration.) The full interaction history for a developer prior to the first of the k command-discovery events formed the simulated history (i.e., training set) for the recommendation system. The expected recommendations would then be the k commands themselves. The recommendation system was configured in turn with each of the postulated algorithms. The authors chose to suppress cases where one or more of the algorithms were unable to deliver recommendations, for example, due to insufficient history in the training set.

Three variations on this generic procedure were followed. The standard variation is as described above; it assumes that the first use of a given command led to the developer deeming the command to be useful. Examination of the interaction histories indicated that, in some cases, the command was not repeated again, and so the assumption of usefulness was likely not correct. This led to two other variations: (1) k-tail multi-use, in which commands that are not repeated are ignored and do not contribute to the k commands sought and (2) k-tail multi-session, in which commands that are not repeated in different development sessions are ignored and do not contribute to the k commands sought. (The multi-session variation is strictly more conservative than the multiuse variation.)

For the simulation involving human participants, a set of recommendations was generated from the dataset by each algorithm and for each participant. Two populations of participants were sampled: experts and novices. Each recommendation was presented verbally to a participant who was asked to rate the novelty of the command and to explain the rationale for this rating.

Reported Threats to Validity

Murphy-Hill et al. [23] report one key threat to validity for the automated simulation: the inability to determine whether the recommendations were actually useful, since the expected recommendations were constructed by inferring behaviors in the previously recorded interaction traces. For the simulation involving human participants, they report one other key threat: the fact that the recommendations were delivered to participants by a human experimenter could have influenced the

participant to be more willing to accept them; an automated RSSE would likely find more resistance from its users.

12.4 Lessons Learned

Every nontrivial empirical study necessarily has some weaknesses: the space to be explored is effectively infinite, while a study must be finite. Nevertheless, each of the studies described in the previous section points to issues about which we need to worry and strategies that we can apply to address these. We discuss here four lessons we can learn from these studies.

12.4.1 Triangulation

One convenient but misguided interpretation of the papers described in the previous section would be that simulations suffice for evaluation, and that through the use of simulation, one can avoid user studies altogether. But simulations can only be as good as the model, assumptions, and data that go into them. While being careful in designing and running a simulation can go a long way towards the validity of its results, this is not enough to ensure that some issue has not been overlooked.

Every empirical methodology has inherent strengths and weaknesses; this is equally true of simulation. The strengths can be eroded by a poor study design, and the weaknesses can be mitigated in some circumstances. Triangulation is an approach in which multiple evaluations are conducted, each using different methods and/or on different data sources in order to improve the generalizability of the findings. When the threats to validity of the individual evaluations differ and the results support the same conclusion, the threats are mitigated overall.

In the works on Strathcona, Gilligan+Suade, and the approach of Murphy-Hill, we see that simulation was used in combination with other studies for the sake of triangulation. Sometimes simulation was used to mitigate the threats accrued from other evaluations (in the case of Strathcona); sometimes simulation was used as the first step in collecting evidence before a human-participant study was conducted (in the other two cases).

In the work on eROSE, triangulation consisted of repeating the same kind of study on multiple software systems and performing different kinds of study to evaluate different aspects of the approach. Zimmermann et al. point out that their methodology was unable to address the actual usefulness of the recommendations, and further, that their study could be affected by bad developer decisions recorded in history. The purpose of their study was to examine how often eROSE could produce correct recommendations; there is no obvious alternative means to obtain a set of expected recommendations for simulation purposes. Further triangulation involving

user studies could help to mitigate these threats, but the contribution of their work was already large, so it is not surprising that this last step was not taken.

It would be wrong to think that the authors of the work on eROSE chose the wrong path for triangulation. In the other three papers, the overall study was smaller and less thorough, sometimes necessitated by the available data source from which to extract expected recommendations. In the end, it is easy to say, "they should have done more studies"; it is much harder to judge when this demand is excessive. Further studies beget further studies; such is the nature of science. We all have limited resources to expend; how best to make use of those resources depends on our goals and how important the answers are. As an area matures, it is natural for reviewers to expect stronger results.

12.4.2 Quality of Real Data

The availability of real-world data with which to evaluate or populate a simulation is an important factor in pursuing a simulation. As such, it is natural to believe that whatever real-world data one has in possession will be good enough. Much of science revolves around this fact: all real data is imperfect.

Most data that is used in evaluating or driving a simulation has been automatically recorded by some program. All programs contain bugs. Any useful program will execute on a real computer; real computers contain bugs. Humans make mistakes, and this can affect the quality of any data that was written down by a human.

All real data was recorded in a real setting. Particular people doing particular things in a particular context and at a particular time. In using this data for other purposes, one has to assume that differences between the original setting and the setting being simulated are not significant, but this assumption can be false.

Sometimes the real data does not provide the information that is desired. In the case of eROSE, Zimmermann et al. tell us that they would have liked to know about the order in which entities were changed, but that this was not recorded and no inference could be made to recover this information. There is no obvious way to overcome this limitation, but it is also unclear that the limitation would be a significant one, so following a more expensive route to collect this data might be unwarranted.

In the case of the work by Murphy-Hill et al., the recommendations were not received by the participants as well as had been predicted by the initial simulation. The data quality could have been an issue. Murphy-Hill et al. assumed that evidence of learning commands could be inferred from the available interaction traces; they were surprised to find that some participants claimed that they were already using recommended commands, as a recommendation should not happen in that situation. Either the data was wrong, their tool contained bugs, or the participants' claims were false—all problematic cases to deal with. There is no obvious way that they could have avoided this issue a priori. Perhaps with further study, the nature of the issue

will be discovered as well as how it can be avoided. Another potential issue was that there could have been a time gap between when the trace data was recorded, and when the recommendations were made; in this time gap, the developers could have discovered the command and started to use it. Ensuring that the data being leveraged is very fresh and that it contains all of the participant's interactions would be about the only way to avoid this, but could be difficult to enforce in most studies.

12.4.3 The Importance and Dangers of Assumptions

We all make assumptions. Sometimes we are aware of our assumptions, and sometimes we make them implicitly. Assumptions are absolutely needed when the evidence available does not permit inference. But obviously, assumptions can be wrong. When assumptions are explicit, one needs to decide whether it is worthwhile to invest time into testing them. When assumptions are implicit, they cannot be directly tested, so triangulation in general is the best means to discover any consequences arising from false ones.

In the case of eROSE, assumptions were made about its usage scenarios. These were necessary to drive the simulations; they were presumably derived from the experience of Zimmermann et al. rather than actual data. While these assumptions appear reasonable, there is no guarantee that they would really occur in practice or that other important scenarios would not occur. This is an inherent weakness of the methodology used that could only be mitigated through other methodologies or other data sources containing direct evidence of such usage scenarios.

In the case of Strathcona, the sensitivity of the approach to the amount of input facts was assessed through simulation. The simulation attempted to use as queries all subsets of facts of a small set of examples. This implicitly assumes that these queries are representative of what developers would typically provide. Perhaps a better design would have been to non-exhaustively select subsets of facts from a larger number of examples; this could have avoided the combinatorial explosion problems arising from trying all subsets. Follow-on studies could have been performed on the more problematic cases to see whether human participants could handle them well enough.

12.4.4 Effects from the Presence of the RSSE

In most cases, the real-world data that a researcher uses in driving or assessing a simulation was necessarily collected without the RSSE being present. The hope is that the RSSE will not alter the essential decisions that were made, but perhaps allow them to be made faster and with greater confidence.

In the case of Gilligan+Suade, this assumption about the recorded data was challenged. The second simulation phase found that the results from the tool seemed

better than predicted by the initial simulation. The presence of the RSSE in the decision process appeared to actually alter nontrivially the decisions made by the "participant." The effect at work could be that developers will make sub-optimal decisions when a complex decision process is not well supported. Thus, the recorded choices were not the "gold standard" that were assumed, but merely good enough for the developers to have completed the task; the presence of Gilligan+Suade apparently improved the decision process. To determine whether this effect was real or an artifact from a biased investigation would require follow-on study.

In the case of the work of Murphy-Hill et al., they point out the fact that their simulation that involved human participants was not completely natural, as a human was giving them the recommendations. Recommendations delivered by an RSSE could be received with less trust, or be deemed annoying if they were delivered at the wrong time. They suggest that social tagging of recommendations be supported as a developer is more likely to pay attention to a recommendation seconded by a trusted colleague.

12.5 Conclusion

Simulation is an important empirical technique used in many areas of science and engineering. Simulation can serve to explore a complex system or to evaluate it. Simulation involves the imitation of some part of a system, in order to avoid the complexities, risks, or costs involved in directly evaluating the system.

We have specifically examined the use of simulation for the evaluation of RSSEs. We have presented a general model of simulation for evaluation of RSSEs that applies to a typical situation in which RSSEs are to be evaluated; this typical situation is analogous to the standard setting of unit testing of software, with drivers and stubs. Variations on this model were mentioned briefly, including the alternative of simulating the RSSE itself to assess developer actions.

Four examples from the literature were described that involved simulation for evaluation of RSSEs. Three of these involved the typical simulation scenario in which the context of the RSSE was simulated to drive the actual RSSE. One of them used the alternative simulation scenario of simulating the RSSE itself as well as its simulation environment.

All empirical methodologies have strengths and weaknesses. A typical simulation has the advantage that a much wider of range of behavior can be examined than could be through user studies, at a lower cost, with greater control, and with greater reproducibility. A typical simulation has the disadvantage that it involves driving a model, which must be assumed to serve as a valid abstraction of the system of interest; when this assumption fails to hold, the conclusions drawn from the simulation may not be valid.

To mitigate this problem, simulations are often used in combination with other forms of evaluation (i.e., in triangulation). Simulations are sometimes used to generalize the results of human-participant studies. Simulations are commonly used as a first step before more expensive, alternative methodologies are pursued.

All the example RSSEs that we have described made use of real-world data to drive their simulations. Real-world data has the advantage that it has not been contrived, so it can be claimed to represent at least some aspect of the real world. Unfortunately, real-world data does not eliminate the need to be cautious in its application. Real-world data has to be collected and recorded and there is no guarantee that this process is free of errors. The data may violate important assumptions of the simulation, despite being "real." The real-world context in which the data was collected could be radically altered were the RSSE added to it, thereby reducing the validity of the results of the simulation. Still, real-world data would generally be more reliable than synthetic data, and thus is highly sought after. The increasing availability of high-quality, real-world data needed to assess the quality of recommendations will only serve to make simulation an even more feasible option on the road ahead.

The future of simulation for evaluation of RSSEs looks bright. An interesting possibility, hinted at in some of the studies described in this chapter, is to directly simulate a limited range of behaviors of developers. From recorded traces of tool interactions, one may be able to extract common behaviors and use these to model the "representative developer" over an extended time. Time will tell if this is more than a dream, but we believe that there is promise.

Overall, simulation is an exciting option with growing importance as an exploration and evaluation technique for RSSEs.

References

1. Avazpour, I., Pitakrat, T., Grunske, L., Grundy, J.: Dimensions and metrics for evaluating recommendation systems. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 10. Springer, New York (2014)
2. Babbage, C.: Passages from the Life of a Philosopher. Longman, Green, Longman, Roberts, & Green, London (1864)
3. Bachmann, A., Bird, C., Rahman, F., Devanbu, P., Bernstein, A.: The missing links: bugs and bug-fix commits. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 97–106 (2010). doi:10.1145/1882291.1882308
4. Bavota, G., De Lucia, A., Marcus, A., Oliveto, R.: Automating Extract Class refactoring: an improved method and its evaluation. Empir. Software Eng. (2013). doi:10.1007/s10664-013-9256-x (in press)
5. Bird, C., Zimmermann, T.: Assessing the value of branches with what-if analysis. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 45:1–45:11 (2012). doi:10.1145/2393596.2393648
6. Collinson, M., Monahan, B., Pym, D.: Semantics for structured systems modelling and simulation. In: Proceedings of the ICST International Conference on Simulation Tools and Techniques, pp. 34:1–34:8 (2010). doi:10.4108/ICST.SIMUTOOLS2010.8631
7. Cossette, B.E., Walker, R.J.: Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 55:1–55:11 (2012). doi:10.1145/2393596.2393661

8. Erzberger, C., Prein, G.: Triangulation: validity and empirically-based hypothesis construction. Qual. Quan. **31**(2), 141–154 (1997). doi:10.1023/A:1004249313062
9. Feinstein, A.H., Cannon, H.M.: Constructs of simulation evaluation. Simulat. Gaming **33**(4), 425–440 (2002). doi:10.1177/1046878102238606
10. Foss, T., Stensrud, E., Kitchenham, B., Myrtveit, I.: A simulation study of the model evaluation criterion MMRE. IEEE Trans. Software Eng. **29**(11), 985–995 (2003). doi:10.1109/TSE.2003.1245300
11. Hlupic, V., Irani, Z., Paul, R.J.: Evaluation framework for simulation software. Int. J. Adv. Manuf. Technol. **15**(5), 366–382 (1999). doi:10.1007/s001700050079
12. Holmes, R., Ratchford, T., Robillard, M., Walker, R.J.: Automatically recommending triage decisions for pragmatic reuse tasks. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 397–408 (2009). doi:10.1109/ASE.2009.65
13. Holmes, R., Walker, R.J.: Customized awareness: recommending relevant external change events. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 465–474 (2010). doi:10.1145/1806799.1806867
14. Holmes, R., Walker, R.J.: Systematizing pragmatic software reuse. ACM Trans. Software Eng. Methodol. **21**(4), 20:1–20:44 (2012). doi:10.1145/2377656.2377657
15. Holmes, R., Walker, R.J., Murphy, G.C.: Approximate structural context matching: an approach to recommend relevant examples. IEEE Trans. Software Eng. **32**(12), 952–970 (2006). doi:10.1109/TSE.2006.117
16. Hughes, R.I.G.: The Ising model, computer simulation, and universal physics. In: Morgan, M.S., Morrison, M. (eds.) Models as Mediators: Perspectives on Natural and Social Science, No. 52 in Ideas in Context, Chap. 5. Cambridge University Press, Cambridge (1999). doi:10.1017/CBO9780511660108.006
17. Kelley, J.F.: An iterative design methodology for user-friendly natural language office information applications. ACM Trans. Inform. Syst. **2**(1), 26–41 (1984). doi:10.1145/357417.357420
18. Kim, S., Zimmermann, T., Whitehead Jr., E.J., Zeller, A.: Predicting faults from cached history. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 489–498 (2007). doi:10.1109/ICSE.2007.66
19. Kononenko, O., Dietrich, D., Sharma, R., Holmes, R.: Automatically locating relevant programming help online. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 127–134 (2012). doi:10.1109/VLHCC.2012.6344497
20. Marcus, A., Poshyvanyk, D., Ferenc, R.: Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Trans. Software Eng. **34**(2), 287–300 (2008). doi:10.1109/TSE.2007.70768
21. Matejka, J., Li, W., Grossman, T., Fitzmaurice, G.: CommunityCommands: command recommendations for software applications. In: Proceedings of the ACM Symposium on User Interface Software and Technology, pp. 193–202 (2009). doi:10.1145/1622176.1622214
22. Mosteller, F.: A k-sample slippage test for an extreme population. Ann. Math. Stat. **19**(1), 58–65 (1948). doi:10.1214/aoms/1177730290
23. Murphy-Hill, E., Jiresal, R., Murphy, G.C.: Improving software developers' fluency by recommending development environment commands. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 42:1–42:11 (2012). doi:10.1145/2393596.2393645
24. Murphy-Hill, E., Murphy, G.C.: Recommendation delivery: getting the user interface just right. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 9. Springer, New York (2014)
25. Robillard, M.P.: Topology analysis of software dependencies. ACM Trans. Software Eng. Methodol. **17**(4), 18:1–18:36 (2008). doi:10.1145/13487689.13487691
26. Robillard, M., Walker, R.J.: An introduction to recommendation systems in software engineering. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 1. Springer, New York (2014)
27. Robillard, M.P., Walker, R.J., Zimmermann, T.: Recommendation systems for software engineering. IEEE Software **27**(4), 80–86 (2010). doi:10.1109/MS.2009.161

28. Said, A., Tikk, D., Cremonesi, P.: Benchmarking: a methodology for ensuring the relative quality of recommendation systems in software engineering. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 11. Springer, New York (2014)
29. Sánchez, P.J.: Fundamentals of simulation modeling. In: Proceedings of the Winter Simulation Conference, pp. 54–62 (2007). doi:10.1109/WSC.2007.4419588
30. Sebastiani, F.: Machine learning in automated text categorization. ACM Comput. Surv. **34**(1), 1–47 (2002). doi:10.1145/505282.505283
31. Shepperd, M., Kadoda, G.: Using simulation to evaluate prediction techniques. In: Proceedings of the IEEE International Symposium on Software Metrics, pp. 349–359 (2001). doi:10.1109/METRIC.2001.915542
32. Teorey, T.J., Merten, A.G.: Considerations on the level of detail in simulation. In: Proceedings of the Symposium on Simulation of Computer Systems, pp. 137–143 (1973)
33. Tosun Mısırlı, A., Bener, A., Çağlayan, B., Çalıklı, G., Turhan, B.: Field studies: a methodology for construction and evaluation of recommendation systems in software engineering. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 13. Springer, New York (2014)
34. van Rijsbergen, C.J.: Information Retrieval. 2nd edn. Butterworth–Heinemann, London (1979)
35. Winsberg, E.: Simulated experiments: methodology for a virtual world. Philos. Sci. **70**(1), 105–125 (2003). doi:10.1086/367872
36. Ye, Y., Yamamoto, Y., Nakakoji, K., Nishinaka, Y., Asada, M.: Searching the library and asking the peers: learning to use Java APIs on demand. In: Proceedings of the International Symposium on Principles and Practice of Programming in Java, pp. 41–50 (2007). doi:10.1145/1294325.1294332
37. Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. IEEE Trans. Software Eng. **31**(6), 429–445 (2005). doi:10.1109/TSE.2005.72