This preprint has not undergone any post-submission improvements or corrections. The Version of Record of this contribution is published in Software Architecture, 16th European Conference, ECSA 2022, Prague, Czech Republic, September 19-23, 2022. Published by Springer Nature Switzerland AG

Feature-based Investigation of Simulation Structure and Behaviour*

Sandro Koch, Eric Hamann, Robert Heinrich, and Ralf Reussner

KASTEL – Institute of Information Security and Dependability, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{firstname.lastname}@kit.edu

Abstract. Reusing a simulation or parts of a simulation is difficult as often simulations are tightly coupled to a specific domain or even to the system analysed by the simulation. In this paper, we introduce a specification approach that allows simulation developers to model the structure and behaviour of a simulation with a domain-specific modelling language. The specification is used to compare a simulation or parts of a simulation to identify features that can be reused. The results show that our approach can find similar features based on their architectural structure and behaviour. Our approach enables developers to identify and then reuse simulation features.

Keywords: simulation reuse, feature compare, simulation specification, domain-specific modelling language

1 Introduction

Due to the increasing complexity of system properties, simulations are becoming more complex. The rise in complexity of a simulation requires that previously implemented simulation features are reused in subsequent simulation projects to save time and resources. The specialisation of simulations for a particular domain or system impedes reusability for other domains or systems. The decomposition of a simulation into distinct features allows the developer to manage and reuse features individually. A *simulation feature* is an abstraction of a property to be analysed. A *simulation component* comprises the implementation of a simulation feature (i.e., packages, classes, and simulation algorithms). To identify simulation features that could be reused, the developer can compare the architectural structure (i.e., classes, interfaces) of the corresponding simulation component to an architecture specification (e.g., component diagram) on a syntactic level. An identical structure, however, is not sufficient to determine a reusable simulation feature [7]. The developer also has to consider the behaviour of a simulation

* This work was supported by the Federal Ministry of Education and Research (BMBF) under the funding number 01IS18067D, and the KASTEL institutional funding.

component; ergo, they must determine whether the discovered simulation component is a semantic match (i.e., has the desired behaviour) when compared to another simulation component. Manually analysing a simulation component is time-consuming and error-prone.

In this paper, we focus on *Discrete-Event Simulation* (DES). DES is a type of discrete simulation where states only change at instantaneous points in time. We provide an approach to support the decomposition of DES by specifying simulation components in structure and behaviour. We employ a modelling approach with metamodels to specify simulation components. To identify pre-existing simulation components, we compare the specifications of simulation components regarding the structure and behaviour. The identification of simulation components is divided into two stages: First, we compare simulation components based on their structure; we convert the specifications into graph notation and do a graph-isomorphism analysis to identify similar structures. Second, we compare simulation components based on their behaviour by converting the specification to *Satisfiability Modulo Theories* (SMT) notation and then utilising an SMT-Solver to identify similar behaviour.

The paper is structured as follows: We introduce the problem statement in section 2. Our specification language, the structure comparison, and the behaviour comparison are presented in section 3. In section 4, we evaluate (i) our specification metamodel by investigating its applicability using a case-study-driven approach and (ii) the accuracy of our comparison approach by comparing different specifications of simulation components. The paper concludes with a discussion of related work in section 5, a summary and a description of future work in section 6.

2 Problem Statement

The specification of a simulation feature can be derived from its requirements. The implemented simulation component must meet these requirements. Whether a component can be reused is determined by whether it meets the given specification, i.e., whether its structure and behaviour fit the desired criteria. We identified three problems when simulation components need to be compared to find and reuse already existing components instead of developing these components from scratch. ***P1****: Depending on the complexity of a simulation component, comparing its structure at the code-level is time-consuming.* Thus, we require a specification-based approach to describe the structure of a simulation component. Developers can use the specification approach to identify a similar structure of simulation components across different simulations. ***P2****: The structure of a simulation enables that a component can be technically integrated (i.e., matching interfaces); however, the exact behaviour of the simulation can differ.* To gain more insight, information about the component is required; thus, we require that the specification approach also supports the specification of a simulation's behaviour. Comparing simulation components requires comparing the components at the code-level. Code contains details that are not relevant to identifying a matching component. ***P3****: Due to the*

irrelevant details at the code-level, it is costly to compare and identify simulation components. This problem results in unused components that developers could reuse, resulting in redeveloped components. Thus, we require the approach to identify an identical component specification in a set of component specifications. Also, we require that the approach can identify a component with an identical structure and behaviour, although entities, attributes, and events are named differently.

3 Specification Language and Feature Identification

This section presents our contributions to address the problems in section 2.

3.1 Specification Metamodel

Our contribution to address the specification part of problems **P1** and **P2** is the definition of a *Domain-Specific Modelling Language* (DSML) to describe the structure and behaviour of a simulation component. To process the models specified in the DSML, we define a metamodel as underlying abstract syntax of the language. The generalisation we make is always referring to entities and attributes on the type-level, i.e., referring to static objects instead of specific instances. We also exclude the specification of simulation outputs because the simulation result does not impact simulation behaviour. In the definition of the metamodel, we separated elements concerned with the structure of a simulation from those concerned with its behaviour. We define the *structure* of a simulation as the set of basic building blocks: events, entities and attributes. While there are different definitions of the term *behaviour*, we define the behaviour of a simulation as the effects of events on the state of the simulation world, i.e., the changes to attributes triggered through events. A *Simulation* contains a set of *Entities* and *Events*, and each entity contains a set of typed *Attributes*. Additionally, we model a *writes* relationship between events and attributes to describe which attributes affect an event (i.e., delay and when it is fired).

Two additional concepts are necessary to specify the behaviour of a simulation. A simulation changes the simulation world during its runtime. To describe those changes, the metamodel must allow a specification of changed attributes as part of the simulator specification. Attribute changes can be linked to events during which they occur since in DES, such changes can only happen at events. This is always the case in DES because an event is defined as any point in time that marks a change in the simulation world. The state of the simulation world is affected by the order and time that events are scheduled and events may cause other events to be scheduled with a certain delay. The *Schedules* and *WritesAttribute* classes represent the aforementioned two additional concepts.

3.2 Structure Comparison

To address problems **P1** and **P3**, identifying simulation components based on their structure, we compare the structure of two components based on their

specification. We use a graph-based representation of these specifications with annotated nodes and edges. Entities, events and attributes are represented as nodes while schedules- and writes-relationships as well as parent-child relationships between entities and attributes are represented as edges. The graph contains the entire specification of schedules- and writes-relationships. However, the presence of the behavioural specification does not affect the structural comparison as the schedules- and writes-relationships are annotated to the edges of the *reads* and *writes* dependencies of the *events*. We consider two simulation specifications structurally similar if their graph representations are isomorphic, i.e., if there is a bijection between the structural elements (i.e., entities, attributes, and events) of both simulations. Regarding entities and attributes, a graph isomorphism ensures the simulation worlds of both simulation components can store the same information.

3.3 Behaviour Comparison with SMT

To address the problems **P2** and **P3**, identifying simulation components based on their behaviour, we compare their behaviour specification. While a description of the structure of simulation components with the structural metamodel holds enough information to employ a graph-based structural comparison, the use of expressions in the behavioural metamodel makes this approach not viable for behaviour comparison. The expressions in the specification can be entirely expressed as first-order logic formulas, and they can be used as part of SMT instances. We will use representations of those expressions as SMT formulas to build SMT instances whose satisfiability/validity is coupled to the behavioural similarity of two events.

Representing behaviour in SMT: To compare schedules- and writes-relationships to determine the behaviour of a simulation componnent, we need to capture the effect of these concepts on the simulation world as SMT formulas. Schedules-relationships indirectly affect attributes in the simulation world by specifying scheduled events that can affect attributes or schedule other events. We consider two schedules-relationships to have the same behaviour if they always schedule the same event with the same delay (cf. listing 1.1). All SMT formulas in SMT-LIB syntax shown here include variable declarations for all attributes accessed in those formulas. We consider two writes-relationships to have the same behaviour if they affect the attribute in the same way. For n writes-relationships from event A to attribute C with condition-expressions $C_{1..n}$ and write-functions $F_{1..n}$ listing 1.3 shows the combined SMT formula to describe the effect A has on C.

```
// condition:
(declare-fun waitingPassengers () Int)
(assert (> waitingPassengers 0))
// delay:
(declare-fun delay () Double)
(assert (= delay 15))
```

Listing 1.1. Delay specification

```
// all read-attributes
(declare-fun ...)


(assert (not (= C_A C_B)))
(assert (not (= D_A D_B)))
```

Listing 1.2. General schedule comparison

Comparing schedules-relationships: Without loss of generality, we assume that for every pair of events, E_1 and E_2, there is at most one schedules-relationship from E_1 to E_2. The following concepts can be extended to multiple schedules-relationships by finding a bijection between the schedules-relationship from E_1 to E_2 in both simulator specifications. This is possible because the effect of each schedules-relationship on the simulation world is self-contained and independent of other schedules-relationships, a property not present with writes-relationships. With this assumption and a given mapping of events, we can compare the (unique) schedules-relationship from event A to event E in simulator S_1 with the schedules-relationship from event B to event F in simulator S_2, where A and B, as well as E and F, need to be a structural match.

```
(declare-fun old () Int)
(declare-fun value () Int)

// additional inputs
(declare-fun ...)

(assert (=> (C_1) (= value F_1)))
...
(assert (=> (C_n) (= value F_n)))

(assert (=> (not (or C_1 .. C_n))
    (= value old)))
```

Listing 1.3. General write specification

```
// simulator S_1
event A {
  reads Z.waitingPassengers
  writes Z.waitingPassengers = 0
    when waitingPassengers != 0
}
// simulator S_2
event B {
  reads Z.waitingPassengers
  writes Z.waitingPassengers = 0
}
```

Listing 1.4. Writes-relationships with identical behaviour

With the same logic, we can compare the conditions of the schedules-relationships. Let C_A and C_B be the condition-expressions of the schedules-relationships from events A and B, respectively, and D_A and D_B the delay-expressions. Then the behaviour of the schedules-relationships is identical if the SMT formula shown in listing 1.2 is not satisfiable. If the formula is satisfiable, there is an assignment of input variables for which the condition- or delay-expressions evaluate to different values. With the SMT-LIB command `(get-model)` a solver can output such an assignment of input variables. This enables our approach to identify whether two events have the same behaviour and generate an assignment of attribute values to demonstrate that they do not.

Comparing writes-relationships: For schedules-relationships, we assumed that there is at most one schedules-relationship between one event and another. For write-relationships, we cannot make a similar assumption that there is at most one write-relationship in event A that writes to attribute C because the effect of A on C is the result of the combination of all write-relationships from A to C. Therefore, write-relationships (to a single attribute) cannot be compared separately. First, we present an example with a single write-relationship to an attribute and then extend the concept to a general formula. Listing 1.4 shows two events writing to a (matched) attribute *waitingPassengers*. Although the conditions alone are not equivalent, it shows that the effect on the attribute is the same for both events.

4 Evaluation

In this section, we present the evaluation of our approach to specify and compare simulation components.

4.1 Evaluation Goals and Design

The evaluation of the specification DSML and comparing approach follows the Goal Question Metric (GQM) approach [1]. The first goal **G1** is to evaluate whether our DSML for the specification of simulations, which covers structural and behavioural information, is able to specify components of real-world simulations. The second goal **G2** is to evaluate, whether our approach can identify similar components based on their structure. Our last goal **G3** is to check, whether our approach can identify similar components based on their behaviour.

The questions to be answered are: **Q1** – Can our DSML model the structure of a real-world simulation? **Q2** – Can our DSML model the behaviour of a real-world simulation? Even if each component could be modelled with our DSML, it is only a vehicle to enable us to compare the components of the case study. Therefore, we need to answer the following question: **Q3** – Can our approach identify simulation components when compared to other components identical in structure and behaviour?

M1 Applicability: To answer the questions **Q1** and **Q2**, we use the following metric. For the case study, we randomly selected ten simulation features and we modelled the implementation of the features (i.e., components) using our metamodel. Then, we identify the number components that could be modelled. By investigating the number of components that could be modelled, we can infer the applicability of our DSML to the case study.

M2 Accuracy: To answer question **Q3**, we use the following metric, and a scenario-based evaluation. We use the components derived for **M1** to find matching components. First, we compare the structure using the graph-isomorphism approach. Then, if a structural match is identified, we compare the behaviour using our SMT-based approach. We determine the accuracy by calculating the metric F_1 score, which is a harmonic mean of precision and recall. Identifying t_p, t_n, and f_n are scenario-specific; thus, we explain how we identify them when we introduce the scenarios. Given the number of true positives, false positives, and false negatives, precision and recall are calculated as $precision = \frac{t_p}{t_p + f_p}$ and $recall = \frac{t_p}{t_p + f_n}$. F_1 score is calculated as the harmonic mean of precision and recall: $f_1 = 2\,\frac{precision \times recall}{precision + recall}$.

Case study: We selected a publicly available case study as we want to model the specification of the internal structure and the behaviour as precisely as possible. The simulation framework Camunda is a workflow and simulation engine based on the Business Process Modelling Notation 2 (BPMN2). Due to the size of the Camunda BPM Platform (over 500,000 lines of code), we could not model the simulation as a whole; therefore, we focused on ten features of the simulation.

Scenarios: We developed two scenarios where we test whether our approach can find a simulation component when compared to other components. In addition to the specification of the ten simulation components (Ft_1 to Ft_{10}). To verify that our structure comparison does not take the names of the entities, attributes, and events into account, we obfuscated them (O_1 to O_{10}). *The first scenario,* S_1 compares the components Ft_1 to Ft_{10} with each other to find the correct match. If the correct component is identified, we count it as t_p, if a wrong component is identified, we count it as f_p, and if the component cannot be identified, we count it as f_n. *The second scenario* S_2 compares each component Ft_1 to Ft_{10} with each obfuscated component O_1 to O_{10} to find the correct match. If the correct component is identified, we count it as t_p, if a wrong component is identified, we count it as f_p, and if the component cannot be identified, we count it as f_n.

4.2 Evaluation Results and Discussion

Applicability The ten components (Ft_1 to Ft_{10}) contain a total amount of 19 entities. We were able to model all 19 entities with our DSML. Besides the entities, the components also contain 26 events in total. Almost every component contains an event called *execute* as an initial event. The behaviour of this event is different for each component; thus, we had to model each individually. We were able to model each of the 26 events. We designed the DSML to model simulations regarding their structure and behaviour. The results show that we can at least model the selected components of the case study.

Accuracy Regardless of identical components (S_1) or obfuscated components (S_2), the results for scenarios S_1 and S_2 show, that all 20 components could be found. No component was missing or misinterpreted. These results lead to a score of precision, recall, and F_1 of 1.0. In total, 20 components were identified by our approach. The overall results for our evaluation are 1.00 for precision, 1.00 for recall and 1.00 for F_1. The results for comparing simulation components are promising. In a set of individual components, we can identify components that match regarding structure and behaviour. These preliminary results are encouraging, but we have to model more case studies before we can determine whether our approach can be applied to different types of DES.

5 Related Work

In this section, we list related approaches and research concerned with decomposing simulations, reuse in simulation, and the description and comparison of discrete event simulations. In software engineering, the decomposition and composition of software is a well-researched field, but none of these approaches considers the semantics of analyses or simulations. In contrast to our work, the extracted modules do not necessarily represent a semantically cohesive module (i.e., feature). The FOCUS approach gives mathematical semantics for structure and behaviour of software systems [6], it also supports the representation of

quality properties and domain-specific properties [4]. However, these approaches are too broad and too ambiguous for non-domain experts to model DES. Heinrich et al. [3] propose a reference architecture for DSMLs used for quality analysis. However, their architecture focuses only on the input models of the quality analysis. Approaches like first order predicate logic [8] investigate logical implications for various forms of logic. Clarke et al. [2] investigate the satisfaction of temporal logic formulas by automata, and Richters et al. [5] check the consistency of object structures regarding data structures (e.g., class structure). In contrast, our approach allows the straightforward transformation of declarative expressions into SMT-instances and their comparison with an SMT-solver.

6 Conclusion

In this paper, we present a domain-specific modelling language for decomposing discrete event simulations by specifying the architectural structure and the behaviour of simulation features. We evaluated our approach by specifying simulation features of an open-source simulation. The findings show that our approach can identify similar structure and behaviour in simulation components. In this work, however, we have only tested the applicability of our approach to one case study. We plan to model more simulations to investigate our approach's applicability further.

References

1. Basili, V., Caldiera, G., Rombach, D.: The goal question metric approach. Encyclopedia of software engineering pp. 528–532 (1994)
2. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems p. 244–263 (1983)
3. Heinrich, R., Strittmatter, M., Reussner, R.: A Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis. IEEE Transactions on Software Engineering p. 26 (2019)
4. Maoz, S., et al.: OCL Framework to Verify Extra-Functional Properties in Component and Connector Models. In: 3rd International Workshop on Executable Modeling, Austin. p. 7. CEUR, RWTH Aachen (2017)
5. Richters, M., Gogolla, M.: Validating UML models and OCL constraints. In: UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference. pp. 265–277. Lecture Notes in Computer Science, Springer (2000)
6. Ringert, J.O., Rumpe, B.: A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. International Journal of Software and Informatics pp. 29–53 (2011)
7. Talcott, C., et al.: Composition of Languages, Models and Analyses, chap. 4, pp. 41–60. Springer (2021)
8. Tomassi, P.: An introduction to first order predicate logic. In: Logic, pp. 189–264. Routledge (1999)