A Toolchain for Simulation Component Specification and Identification

Sandro Koch(✉) and Frederik Reiche

KASTEL – Institute of Information Security and Dependability, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{sandro.koch,frederik.reiche}@kit.edu

Abstract. Reusing a simulation or parts of it is difficult, because simulations are tightly coupled to a specific domain or even to the analysed system. In a set of simulation components, either publicly available or from internal repositories, it is difficult for simulation developers to find simulation components that can be reused in a new context. They have to understand the structure and the behaviour of a component to determine, whether it fits for the new context. To address this problem, we introduce our toolchain that allows simulation developers to specify the structure and behaviour of a simulation component. We utilise a state-of-the-art graph database and an SMT theorem prover to compare a simulation components. This allows simulation developers to compare and search for simulation components that can be reused instead of being redeveloped.

Keywords: simulation reuse · component compare · simulation specification · domain-specific modelling language

1 Introduction

The specification of a software architecture, e.g. UML class models, is an abstraction of the actual code of the software system. The software architecture covers the structure of the software system; for the behaviour of a system, a different type of model is necessary. In the context of a simulation, the behaviour of the simulation and the behaviour of the system are very similar. For a software system's performance simulation, the developer must understand how to implement the simulation and how a performance simulation functions. To reduce the complexity and the effort of implementing a simulation, especially reimplementing already existing parts of a simulation, we proposed our approach to specify the structure and behaviour simulation components [10]. We use the specification of simulation components to identify other components with similar structures

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project number 499241390 (FeCoMASS) and by the Federal Ministry of Education and Research (BMBF) under the funding number 01IS18067D (RESPOND).

and behaviour. Our approach allows the simulation developer to compare and find simulation components they can reuse in subsequent simulation projects. We decompose a simulation into individual features that allow the simulation developer to manage and reuse them individually. A *simulation feature* is an abstraction of a system's property that the simulation can analyse, for example, the property *throughput* to simulate the performance of a system. We focus on implementing a simulation feature, the *simulation component*. A simulation component comprises packages, classes, and simulation algorithms.

The toolchain we present in this paper allows simulation developers to specify the structure and behaviour of a simulation component by using model-based editors. We use the specification to compare simulation components to find simulation components that the simulation developer can reuse in a different context. To compare simulation components, we use two approaches. First, our tool uses the specification to compare the structure of the simulation components to identify whether the compared simulation components are structurally identical. However, an identical structure is insufficient to determine a reusable simulation feature [18]. Therefore, we also implemented the second step, comparing a simulation component based on the behaviour.

This paper is structured as follows: In Sect. 2 we present approaches related to this work. In Sect. 3 we introduce our toolchain: First, we present the specification part in Sect. 3.1, and then we present the identification part in Sect. 3.2. Finally, in Sect. 4 we conclude the paper, and in Sect. 5 we present the next steps we planned for the toolchain.

2 Related Work

For source code comparison we found the tool JPlag which can find similarities in Java, C#, C, and C++ source code [14]. JPlag is used to detect software plagiarism. Gitchel et al. [7] developed the tool SIM, which compares source code written in C, Java, Pascal, and Lisp. Its approach is similar to JPlag, both use a tokeniser approach for comparing the source code. SIM is also able to take the correctness, style and uniqueness of the code into account. Measure Of Software Similarity (MOOS) is another tool that can compare source code [17]. In contrast to JPlag and SIM does MOOS support 26 different programming languages [1]. These tools focus on similarities regarding the structure of the source code, in contrast to our work is the behaviour is not part of their analyses.

Another approach, FOCUS, by Ringert et al. [16] provides a mathematical semantics for the specification of structure and behaviour of software systems. FOCUS can also specify quality and domain-specific properties of a software system [11]. Graphical approaches such as UML-based Activity Diagrams, Flow Diagrams or Activity Cycle Diagrams can be used to describe the structure of a simulation and specify the flow of events [2]. FOCUS and UML are too broad, they can model any kind of software system; therefore, they require additional training for non-domain experts to model *Discrete-Event Simulation* (DES).

The refinement of relations and various forms of simulation dependencies are investigated by Milner [12]. Clarke et al. [4] investigate the satisfaction of temporal logic formulas by automata, and Richters et al. [15] check the consistency of object structures regarding data structures. The *Discrete Event System Specification* (DEVS) formalism [22] is a formal approach to describing and analysing discrete event systems. Other approaches, like Condition Specification Language (CSL) [13] or the OMNeT++ framework [20], combine simulation specifications with a description language. These approaches use general-purpose languages like C or Java for their specification, comparing these specifications would require to compare the structure and behaviour on the source-code level. Our approach allows the straightforward transformation of declarative expressions to *Satisfiability Modulo Theories* (SMT)-instances and their comparison with an SMT-solver.

For the specification of an architecture for distributed simulations that allow the interoperability and reuse [9] of simulations, the *High-Level Architecture* (HLA) was developed by the Modelling and Simulation Coordination Office of the US Department of Defence. Another specification approach, the *Functional Mock-up Interface* (FMI) [3] standard, helps to define an interface for exchanging information and coupling between heterogeneous software systems used for Model Exchange and Co-Simulation. Both approaches, the HLA and the FMI, can be combined to facilitate the reuse of simulation models in complex engineered systems [6]. However, in contrast to our work, these approaches lack the ability to compare and identify simulation components.

3 The Toolchain for Simulation Specification and Simulation Component Identification

We separated our toolchain to specify and identify simulation components into two parts. Figure 1 depicts the third-party tools we used and the tools we developed to realise the specification and identification of simulation components.

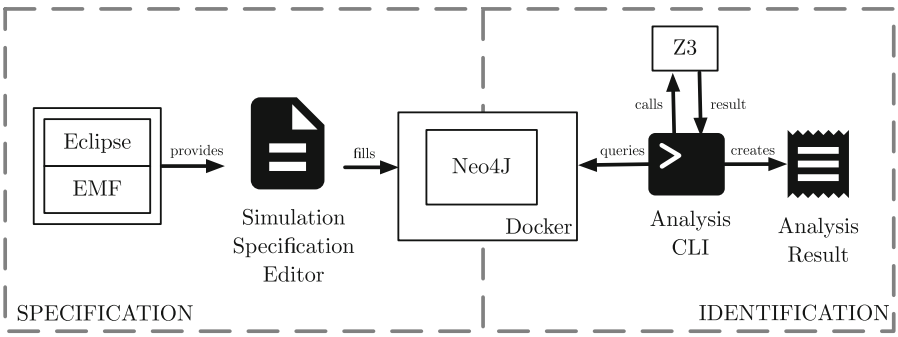

Fig. 1. Specification and Analysis Toolchain

Our contributions are the *Simulation Specification Editor*, the *Analysis Command Line Interface (CLI)*, and the *Analysis Results*; they are depicted in black. This section details how we implemented the toolchain and how it is used. First, in Sect. 3.1, we present how one can specify simulation components with our tooling. Second, in Sect. 3.2, we present how one can use our tooling to identify similar simulation components.

3.1 Specification of Simulation Components

The Simulation Specification Editor is based on our metamodels from our previous work [10]. Events, entities, and attributes are the three elements that make up what we refer to as the structure of a simulation. According to this viewpoint on the structure of a simulation, an event is nothing more than a different object devoid of any behavioural characteristics. The structure of a discrete event simulation can be modelled using the metamodel that is shown in Fig. 2. A *Simulation* is made of a collection of *Entities* and *Events*, and each *Entity* is made of a collection of typed *Attributes*. In addition, *Events* can read *Attributes* of *Entities*. This dependency indicates which attributes are affected by an event. Because performing a read operation on an attribute has no impact on the simulation world, this relationship is considered to be a component of the structural metamodel.

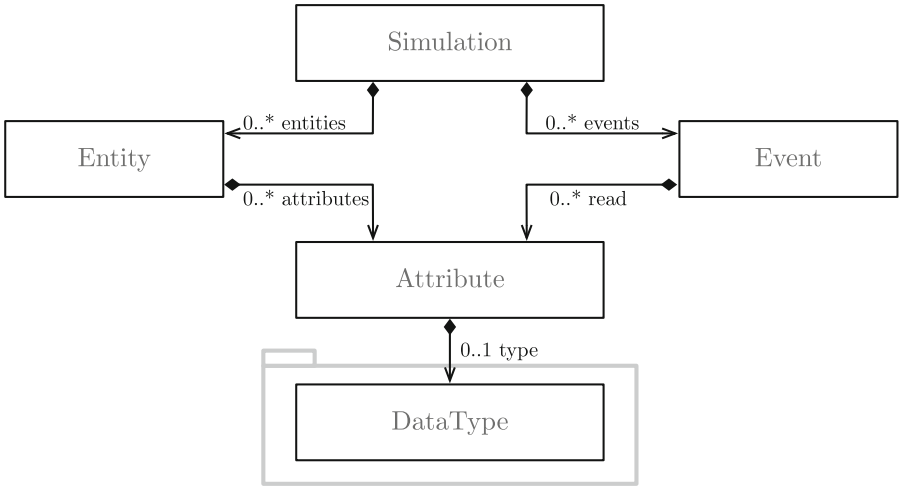

Fig. 2. Metamodel for Specifying the Structure of Simulation Components

The metamodel used to describe the behavioural aspects is displayed in Fig. 3. Even though the term behaviour can have several distinct meanings, we define the behaviour of a simulation to be the impacts of events on the state of the simulation world, i.e., changes to attributes that are triggered through events.

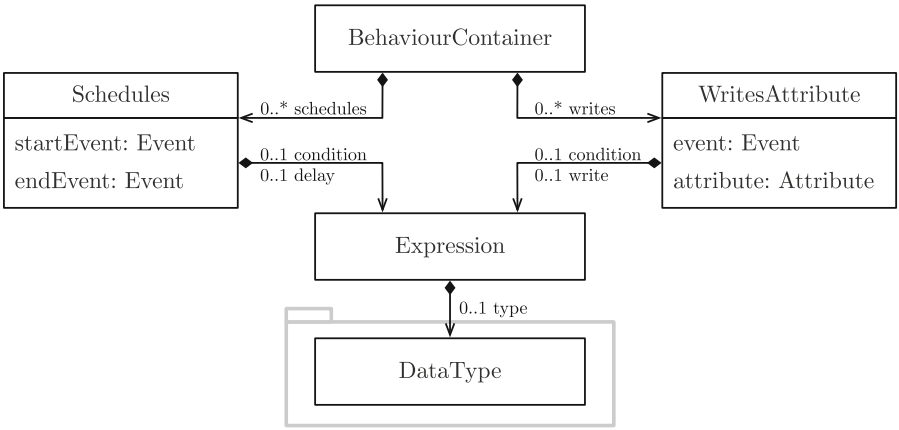

Fig. 3. Metamodel for Specifying the Behaviour of Simulation Components

When attempting to specify the behaviour of a simulation, in addition to the structure of the simulation, two additional notions are required. (i) During its execution, a simulation will modify the simulation world. The metamodel must be able to specify modified attributes in order for it to express changes to the simulation world. Furthermore, the metamodel must be able to provide to model the modification of attributes in order for it to express changes to the simulation world. These changes are included as part of the simulation's specification. (ii) In DES, attribute changes can only take place during events. Furthermore, an event can occur at any time, which signifies a change in the simulation world. The order in which events are scheduled and the times at which they occur indirectly will influence the simulation world's status. Events will cause other events to be delayed in their scheduling.

In order to specify a simulation, we implemented the metamodels in the *Eclipse Modelling Framework* (EMF)[1]. EMF is an extension of the Integrated Development Environment (IDE) Eclipse. EMF provides graphical editors to create metamodels, and it also provides developers with code generators to create code stubs of the metamodel classes and tree-editors for the metamodels.

As shown in Fig. 1, EMF provides the editor to create the simulation specification. We utilised the tree-based editors so that the simulation developer could model a simulation component graphically. The simulation developer can model the structure and behaviour of a simulation component in the tree editor. It is necessary to model both: the structure and the behaviour of a simulation component to compare and identify identical components [10]. Figure 4 shows the tree editor in Eclipse to specify simulation components.

Each simulation component is stored in a `*.structure` file. The developer can edit these files with the *structure tree-editor*. Each node in the editor is created with a unique *ID* and *name* property. The *root node* represents the

[1] https://www.eclipse.org/modeling/emf/.

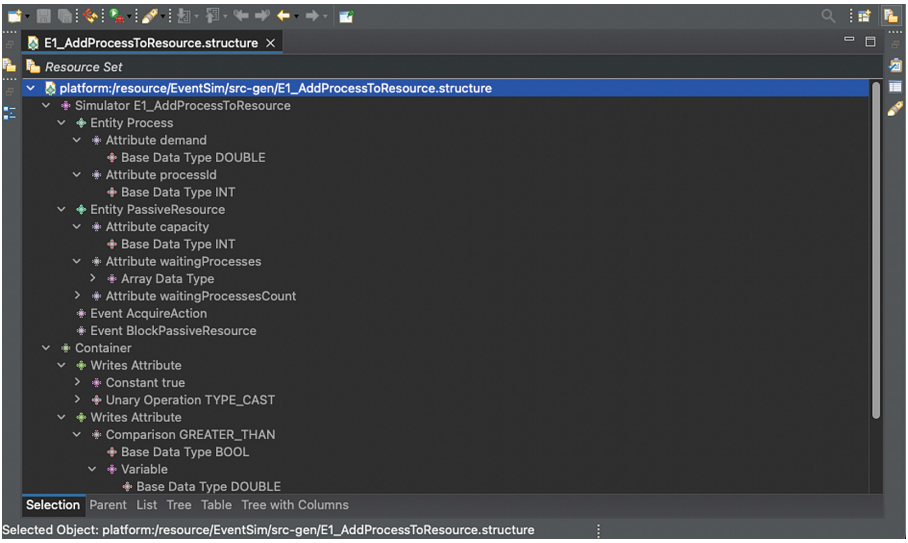

Fig. 4. Simulation Specification Editor

simulation component; the developer can add a description in addition to the ID and name. The root node contains *entities* and *events*. All entities have *attributes* that are either *base datatypes* like integers or booleans, or *arrays* or *enums*. Each event can reference any number of attributes to indicate a *reads* relationship.

In order to separate the structure from the behaviour, the behaviour is modelled in a separate section, but it is stored in the same file. The metamodels of structure and behaviour are modelled according to the reference architecture for domain-specific modelling languages [8]. The modular structure of the metamodels allows us to maintain and extend the metamodels separately and use an editor that references both metamodels. The behaviour consists of *writes attribute* and *schedules* relationships. According to our metamodel, each writes attribute is linked to one event. When the event is fired, the writes attribute contains a condition; if this condition is true, the referenced attribute gets changed. How an attribute is changed is also modelled in the writes attribute.

Events are also able to schedule other events. To model the scheduling of events, developers can add the *schedules* node to the tree editor. A schedules node references the causing event and the event to schedule. In order to determine whether the simulation component schedules an event, the node also contains the condition and a reference to the attributes that are evaluated.

For the specification to be used for comparison, we transform the specification into a graph (cf. Koch et al. [10]). In Fig. 5, we show an example of the structural information stored in the graph. The stored simulation component represents the simulation of a traffic light. The graph contains one *entity*, two *attributes* and two *events*. The entity *TrafficLight* represents the simulated traffic light. The *TrafficLight* contains the two attributes *colourTraffic* and

waitingPedes. The attribute *colourTraffic* represents the colour of the traffic light (red or green), and the attribute *waitingPedes.* represents the number of pedestrians that wait at the traffic light. Besides the entity and the two attributes, the graph also contains the events *PedestrianRed* and *PedestrianGreen*. The event *PedestrianGreen* can change the colour of the traffic light, as it has a writes relation on the attribute *colourTraffic*. The event *PedestrianGreen* also reads the number of waiting pedestrians. How we represent the structure and behaviour of simulation components differs from how we differentiate structure and behaviour when we compare them. The behaviour information is part of the graph (i.e., schedules- and writes relations), and the expressions are annotated on these relations. However, when our tool compares the structure of the simulation components, the annotated information about the behaviour is discarded; thus, Fig. 5 only contains the structural information (i.e., nodes and edges). The usage of expressions in schedules- and writes-relationships, which reflects a paradigm orthogonal to the graph notation, is why the behaviour specification of the simulation component cannot be compared using a graph-based method (cf. [10]). The behaviour is stored as annotations on the schedules- and writes-relations; thus, the graph-isomorphism approach cannot determine the behaviour's similarity. The expressions representing the simulation component's behaviour are first-order logic statements. We transform these statements into SMT statements (as introduced in [10]).

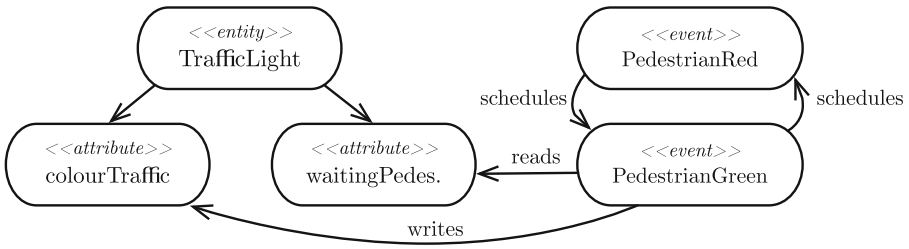

Fig. 5. Graph-representation of Structural Elements

The specification in its graph form is stored in the graph database Neo4J[2] as shown in Fig. 1. Our tool provides an interface to save the specification in the database. For convenience, we recommend running the Neo4J database in a Docker container.

Although the database is used to store the transformed specifications and to perform the analysis to compare the specifications structurally, the user can visualise each stored graph via the Neo4J UI. Figure 6 shows the graphs of six simulation components of various sizes. The blue nodes represent a simulation component. The yellow nodes represent the entities of a simulation component. The red nodes represent the events of a simulation component. The grey nodes represent

[2] https://neo4j.com/.

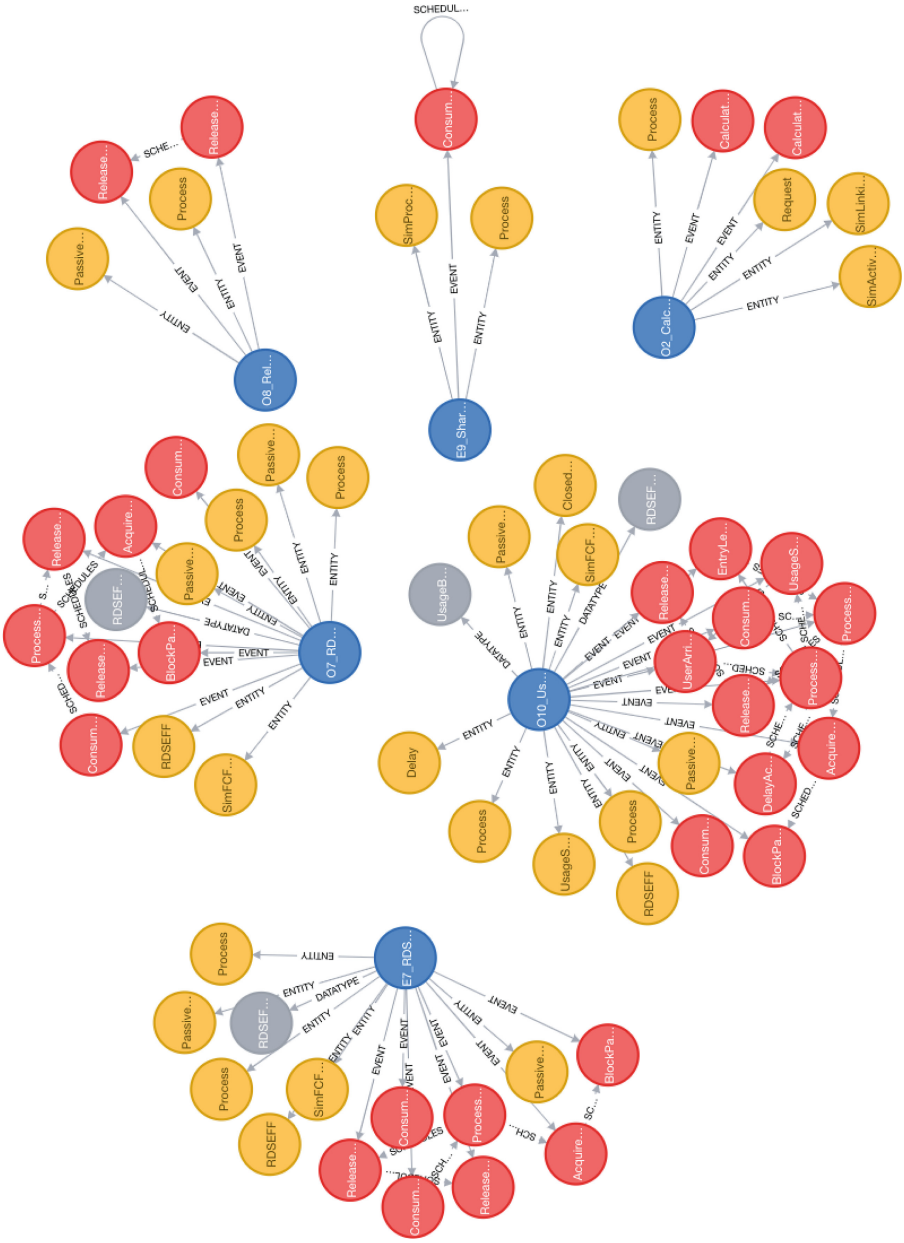

Fig. 6. Simulation Specification Graph Visualisation (Color figure online)

the datatypes of the simulation component. Reads- and writes-relationships are represented by arrows between events and datatypes. Arrows between events also represent schedules-relationships. Although it is possible to modify the graphs

in the Neo4J user interface, the editor does not automatically update the specification in the tree editor based on the changed graph. Therefore, we recommend using only the tree-based editor to modify the specifications.

3.2 Identification of Simulation Components

The specification of structure and behaviour of simulation components can serve as documentation for the simulation. The analysis developers can use these specifications to understand the software better, and if necessary, they can compare these specifications manually. However, besides the specification of simulation components, our toolchain can also compare these specifications regarding their structure and behaviour. To extend the specification's purpose and allow the identification of similar simulation components, we present a second tool. Our second tool, the Analysis CLI, utilises the graphs derived from the simulation specifications. Therefore, it accesses the Neo4J database to identify identical simulation components based on their specification. Comparing two simulation components is separated into two steps. First, the Analysis CLI performs a graph-isomorphism analysis [19] in the Neo4J database. This analysis checks whether the nodes and edges of a graph A can be mapped onto another graph B, i.e. whether the structure is identical. The graph B that is searched can have the same number of nodes and edges or a higher number of nodes and edges. We use the graph-isomorphism implementation by Cordio [5], which is available as plugin for the Neo4J database, for the subgraph analysis. Second, if the graph-isomorphism identifies a structural match, the Analysis CLI proceeds with the behaviour analysis. The behaviour information stored in the behaviour metamodel is transformed into SMT statements based on the SMT-LIB standard[3]. These statements are then analysed by an SMT-Solver. For our toolchain, we use the *Z3 Theorem Prover* by Microsoft [21].

Tool Setup: The user of our tool can access the functionality of our tooling via a CLI. We developed the CLI to enable the user to compare the specifications of simulation components regarding their structure and behaviour. The CLI acts as an interface so that the user does not have to invoke the graph-isomorphism and behaviour analysis manually. Before the user can compare the specifications, they must install the Z3 Theorem Prover and provide the path to it. They can either extend the systems `PATH` variable manually or they use the following command of the CLI:

```
sim-compare z3 <PATH TO libz3.dylib>
sim-compare z3java <PATH TO libz3java.dylib
```

Listing 1.1. Z3 Theorem Prover Setup

Whether the user must manually modify the `PATH` variable or invoke the commands of our CLI depends on the used operating system. In the context of

[3] https://smtlib.cs.uiowa.edu/.

the Analysis CLI, we tested the prerequisite commands for the operating system MacOS. For more information and further instructions, please visit the official Z3 website[4] or the GitHub page[5].

Furthermore, the user must have access to a running Neo4J instance with the installed graph-isomorphism plugin. The default setting is that the tool assumes that a standard Neo4J instance is running locally with the default user and password. If the user wants to use a different configuration of the Neo4J database, they can use the following commands:

```
sim-compare neoip <IP>
sim-compare neopw <PASSWORD>
```

Listing 1.2. Neo4J Setup

With the command `neoip`, the user sets the IP according to their Neo4J installation. If they use another password, with the command `neopw` the user can change the password corresponding to their Neo4J installation.

Analysis: Before the user can compare two specifications, they need to know which simulation specifications are available for the analyses. Therefore, we provide the command `list`, which prints all simulation components that are stored in the Neo4J Database. The print shows the names of the simulation components.

```
sim-compare list
```

Listing 1.3. List all Simulation Components

The user can compare two simulation components at a time with the information on which simulation components are available. The command `compare <SIM_A> <SIM_B>` allows the user to compare two simulation components:

```
sim-compare compare <SIM_A> <SIM_B>
```

Listing 1.4. Compare Simulation Components Command

The two parameters, `<SIM_A>` and `<SIM_B>`, represent the names of the specifications of the two simulation components the user wants to compare. Although the user can modify the entries in the database, i.e. change the structural and behavioural specifications; we recommend avoiding using the Neo4J interface to modify the entries. The changes are not part of the specification model; therefore, the changes will get lost when the database gets updated. The `compare` command first invokes the structural comparison of the graphs by using the graph-isomorphism plugin. If the graph-isomorphism analysis yields a positive

[4] https://www.microsoft.com/en-us/research/project/z3-3/.
[5] https://github.com/Z3Prover/z3.

result, the schedules- and writes-relationships are transformed into SMT statements. Based on these SMT statements, the Z3 performs a satisfiable analysis, i.e., a behavioural comparison.

Results: The results depend on whether the structural and behavioural analysis is successful. After invoking the `sim-compare compare <SIM_A> <SIM_B>` command, the analysis result can have four outcomes. Figure 7 depicts the sequencing of the analysis and the possible results. In the remainder of this section, we go through the sequence, and we present the different results, depending on the structural and behavioural analysis.

After the user starts the analysis, the two specifications are first compared regarding their structure. The first result specifies whether the graph-isomorphism analysis yields a negative result, i.e. they do not match structurally. Listing 1.5 shows the result, when the simulation component `SIM_A` is compared to `SIM_B` and the graph-isomorphism yields no result.

```
Compare SIM_A and SIM_B
No isomorphism between simulator graphs!
```

Listing 1.5. No Subgraph Found

The second result can be that the graph-isomorphism yields a positive result. Listing 1.6 shows an excerpt of the result of the successful graph-isomorphism analysis. Instead of the output `No isomorphism between simulator graphs!` the analysis proceeds and starts the behavioural analysis by transforming the schedules- and writes-relationships into SMT statements. The graph-isomorphism analysis can have multiple mappings of nodes and edges; thus, each mapping needs to be analysed. The currently analysed mapping is indicated by the placeholder `n`, and the total number of mappings is indicated by the placeholder `m`.

```
Compare SIM_A and SIM_B
...
Testing mapping n out of m:
```

Listing 1.6. Successful Subgraph Analysis

After the graph-isomorphism analysis, each mapping is analysed regarding the matching behaviour. As graph-isomorphism can yield more than one result, each result will be compared. The analysis proceeds until the SMT-Solver finds a solution or the behaviour is not identical. Listing 1.7 shows the results for a mapping that is not identical (`SMT status: UNSATISFIABLE`).

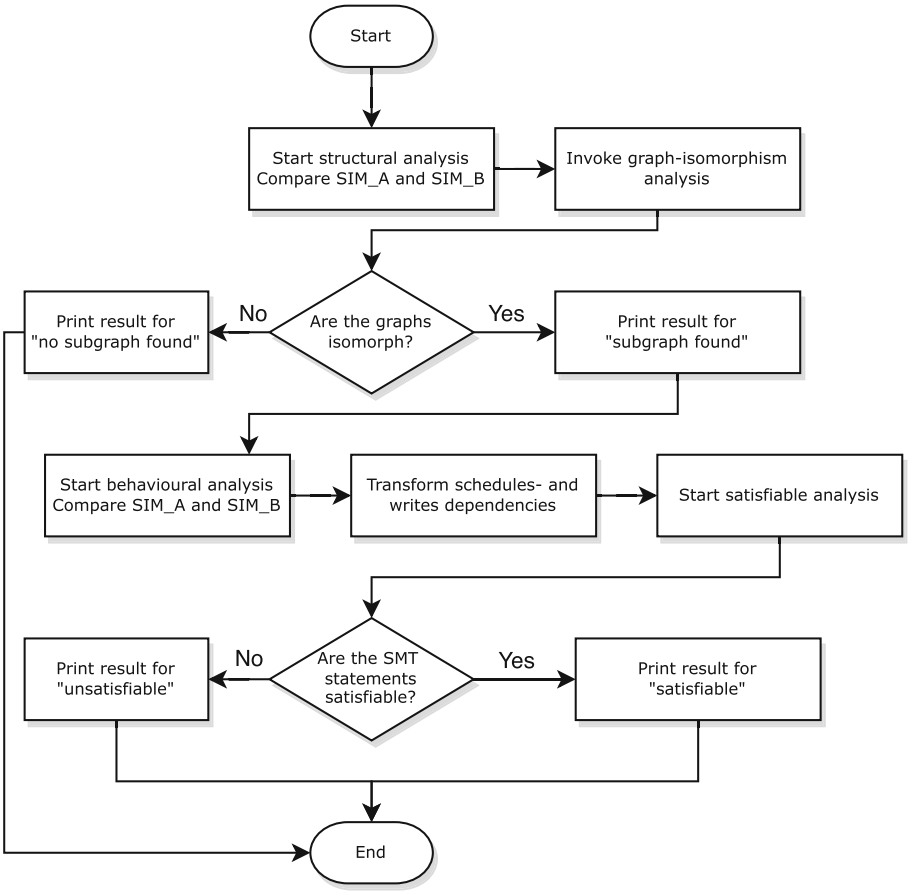

Fig. 7. Sequencing of the Analysis

```
Compare SIM_A and SIM_B
...
Testing mapping n out of m:
Comparing 'XYZ writes demand' with 'ABC writes demand'
SMT status: UNSATISFIABLE
```

Listing 1.7. Behaviour does not match

If the graph-isomorphism analysis was successful and the behaviour is identical, the results show a mapping of the events and entities that yielded the result. Listing 1.8 shows the result of a successful graph-isomorphism and behaviour analysis.

```
...
Testing mapping n out of m:
Comparing 'XYZ writes demand' with 'ABC writes demand'
Behaviour identical with mapping:
[Event] EventA = EventC
...
[Entity] EntityA = EntityZ
...
```

Listing 1.8. Matching Behaviour

4 Conclusion

In this paper, we present a toolchain for specifying and comparing components of discrete event simulations. The specification allows the developer to model the simulation components' structure and behaviour. During the development phase, our toolchain helps the developers find already implemented components of a simulation by comparing the desired specification to the specifications of already existing simulation components. Thus, the developers can avoid reimplementing simulation components that already exist. Also, before the implementation phase or during maintenance, our toolchain can be used; it can help the software architect to find simulation components with different designs that have the required structure and behaviour of the project at hand. Thus, they can analyse the found simulation components regarding their design to determine which design has already been done and is already used. In order to compare the specifications, the models are transformed into a graph notation and then stored in a graph database. We run a graph-isomorphism analysis based on the graph notation to find similar structures of specified simulation components. Suppose the structural analysis yields a positive result, i. e. the compared graph is isomorph; the toolchain starts the behaviour comparison. The behaviour comparison converts the specifications of the simulation components into SMT-notation, which we utilised to analyse the specifications regarding similar behaviour. Finding related simulation components allows software architects to reuse existing simulation components while reducing the effort required to create new simulation components. Thus, they can reuse simulation components that they otherwise would implement again.

5 Future Development

The evaluation of our approach in [10] showed that it is possible to specify and compare simulation components based on their structure and behaviour. To extend our evaluation, we have to model more components of simulations of different domains. Our tree editor is cumbersome when modelling many simulation components. Thus, we have to improve the usability of the editor to be able to model more than a hand full of entities and events. It is hard to track complex

writes- and schedules-relationships of events, which makes the modelling process prone to errors. Therefore, we plan to implement a graphical or textual language to specify simulation components more quickly. Furthermore, our similarity analysis needs to identify specifications that match less than 100%. Thus, we plan to extend our approach so that we can identify simulation specifications that do not match perfectly. This would enable us to help the software architects in the system design to explore more alternative designs.

References

1. Ahadi, A., Mathieson, L.: A comparison of three popular source code similarity tools for detecting student plagiarism. In: ACM International Conference Proceeding Series, pp. 112–117. Association for Computing Machinery (2019). https://doi.org/10.1145/3286960.3286974
2. Balsamo, S., Marzolla, M.: Simulation modeling of UML software architectures. In: 17th European Simulation Mulitconference, vol. 3, pp. 562–567. Society for Modelling and Simulation International, SCS European Publishing House (2003)
3. Blockwitz, T., et al.: Functional mockup interface 2.0: the standard for tool independent exchange of simulation models. In: Proceedings of the 9th International MODELICA Conference, 3–5 September 2012, Munich, Germany, vol. 76, pp. 173–184 (2012). https://doi.org/10.3384/ecp12076173
4. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**, 244–263 (1983)
5. Cordio, S.: csb/neo4j-plugins/subgraph-isomorphism at master · msstate-dasi/csb. https://github.com/msstate-dasi/csb/tree/master/neo4j-plugins/subgraph-isomorphism. Accessed 01 July 2022
6. Falcone, A., Garro, A.: Distributed co-simulation of complex engineered systems by combining the high level architecture and functional mock-up interface. Simul. Model. Pract. Theory **97**, 101967 (2019). https://doi.org/10.1016/j.simpat.2019.101967. https://www.sciencedirect.com/science/article/pii/S1569190X19301005
7. Gitchell, D., Tran, N.: Sim: a utility for detecting similarity in computer programs. SIGCSE Bull. **31**(1), 266–270 (1999). https://doi.org/10.1145/384266.299783
8. Heinrich, R., Strittmatter, M., Reussner, R.: A layered reference architecture for metamodels to tailor quality modeling and analysis. IEEE Trans. Softw. Eng. **47**, 26 (2019)
9. IEEE: 1516-2010 - IEEE Standard for Modeling and Simulation High Level Architecture (HLA). Technical report (2010). https://doi.org/10.1109/IEEESTD.2010.5553440
10. Koch, S., Hamann, E., Heinrich, R., Reussner, R.: Feature-based investigation of simulation structure and behaviour. In: Gerostathopoulos, I., Lewis, G., Batista, T., Bureš, T. (eds.) European Conference on Software Architecture, pp. 178–185. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-16697-6_13
11. 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)
12. Milner, R.: Communication and Concurrency. PHI Series in Computer Science. Prentice Hall (1989)

13. Overstreet, C., Nance, R.: A specification language to assist in analysis of discrete event simulation models. Commun. ACM **28**, 190–201 (1985). https://doi.org/10.1145/2786.2792
14. Prechelt, L., Malpohl, G., Philippsen, M.: Finding plagiarisms among a set of programs with JPlag. J. Univ. Comput. Sci. **8**(11), 1016–1038 (2002)
15. Richters, M., Gogolla, M.: Validating UML models and OCL constraints. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 265–277. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40011-7_19
16. Ringert, J.O., Rumpe, B.: A little synopsis on streams, stream processing functions, and state-based stream processing. Int. J. Softw. Inform. **5**, 29–53 (2011)
17. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: local algorithms for document fingerprinting. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 76–85 (2003)
18. Talcott, C., et al.: Composition of languages, models, and analyses. In: Heinrich, R., Durán, F., Talcott, C., Zschaler, S. (eds.) Composing Model-Based Analysis Tools, pp. 45–70. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81915-6_4
19. Ullmann, J.R.: An Algorithm for Subgraph Isomorphism. Technical report 1 (1976). https://doi.org/10.1145/321921.321925
20. Varga, A.: Omnet++. In: Wehrle, K., Güneş, M., Gross, J. (eds.) Modeling and Tools for Network Simulation, pp. 35–59. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12331-3_3
21. Z3Prover: z3: The Z3 Theorem Prover (2019). https://github.com/Z3Prover/z3
22. Zeigler, B.P., Prähofer, H., Kim, T.G.: Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, 2 edn. Academic Press, San Diego (2000). http://www.gbv.de/dms/goettingen/302567488.pdf