Assessing the value of incomplete deadlock verification in Model-Driven Engineering

Felix Cammaerts[1,*], Monique Snoeck[1]

[1]*LIRIS, Naamsestraat 69, 3000, Leuven, Belgium*

Abstract

Model-Driven Engineering (MDE) involves automatically generating code from a conceptual model. However, despite accurate translation of requirements into models and correct transformation of models into code, ensuring internal consistency of the model remains a challenge. Inconsistencies can arise among various elements of a conceptual model. While certain errors can be identified through simple semantic checks (e.g., circular inheritance relationships, non-deterministic statecharts), other error detection problems encounter decidability issues or necessitate complex verification algorithms that may suffer from state explosions. One particularly arduous verification problem involves checking concurrent statecharts for deadlocks. To address this problem we proposes a divide and conquer algorithm that starts from two statecharts and incrementally adds more statecharts. Although the algorithm cannot address all possible deadlock issues, it demonstrates the capability to detect a significant number of them. This paper provides an initial evaluation of the algorithm, by evaluating the algorithm's utility, conceptual models submitted as homework by students in previous years within an MDE course are employed. The results reveal that students' conceptual models often contain deadlock situations. Consequently, the paper concludes that even an algorithm that cannot detect all deadlock situations, as presented in this research, can already detect a significant number of potential deadlock situations and thus can serve as an initial step in mitigating them while providing valuable feedback to students. The paper also describes the next steps to be undertaken to use the algorithm as part of an educational tool for novice modellers.

Keywords

Deadlock detection, Formal verification, Education, MERODE

1. Introduction

The cost of poor software quality in the United States in 2020 was estimated at a staggering $2.08 trillion [1]. Operational failures accounted for a significant portion, reaching $1.56 trillion, which increased from $1.275 trillion in 2018. Software bugs can have various consequences, from frustrating customers to catastrophic outcomes. Reports on software bugs and faults continue to emerge, with the latest example mentioned on the "Risks to the Public" forum being a bug in a life-sustaining heart pump [2].

For the industry, adequate software testing is the most widely used technique ensuring quality assurance, typically achieved through validation and verification processes.

As low-code and no-code software development approaches gaining ground [3], more software is developed according to Model-Driven Engineering (MDE) practices. In MDE code is automatically generated from a model, verification entails checking the internal consistency of the model. For example, statecharts should only have accessible states, and circular inheritance relationships should be avoided. Many of these mistakes can be identified through simple semantic checks that can be defined by means of Object Constrain Language (OCL) [4] in the meta-model of the modelling approach. However, more complex semantic checks, such as deadlock situations can not be modelled by OCL and require a procedural verification.

The preferred language of MDE practitioners is UML [5]. This language is used for object-oriented modelling, but can also be used for artefact-centric modelling [6][7]. In both approaches, systems are designed as a collection of object types each of which is equipped with a statechart defining the lifecycle of objects. While such systems are prone to deadlock errors, research on deadlock verification is quite scarce in this domain. A possible reason could be that while there have been attempts to provide UML with formal semantics, leading to the definition of fUML[1], these are not based on a formal process algebra, making it challenging to perform deadlock detection in UML-based modelling approaches.

The importance of algorithmic verification should not be underestimated. Previous research has demonstrated that "reading" models without actually running them does not allow to detect errors easily, unless modellers have at least 10 to 20 years of modelling experience [8]. In the absence of algorithmic verification for deadlock, software developers have to rely on (time consuming) software testing to detect possible errors. Algorithmic verification would thus allow for saving precious time in the software development process.

In search for an algorithm, one can consider different formalisms to verify concurrent processes for deadlocks. A prominent and well-researched formalism is Petri Nets, which is mostly used in the domain of process modelling [9]. As statecharts can be translated into Petri nets, it is worthwhile to consider the work that has been performed in this domain, such as the work of Raedts et al. [10], Chu et al. [11], Huang et al. [12], and Jeng et al. [13]. Besides Petri Nets also other or additional techniques are used such as decomposition [14], graph reduction [15], and WF-nets [16]. However, all these proposed verification approaches apply to a single process. In case of collaborative processes, such as is the case for concurrent statecharts, these approaches require the separate processes to be integrated to a single process to allow for deadlock verification. Moreover, these algorithms do not support the provision of feedback about the root cause of the deadlocks in the individual processes in the collaboration, and especially not if those originate from statecharts. There thus remains work to be done to develop a practical algorithm that can verify a collection if interacting statecharts for deadlock errors while providing the modelers with actionable feedback about the root causes in the source statecharts.

In the past, research on MERODE has focused on creating a UML-based MDE approach that is grounded in process algebra with the goal of facilitating model checking. MERODE makes use of a combination of class diagrams and statecharts in line with UML, but formalises this with the process algebra of Communicating Sequential Processes (CSP) [17]. While some theoretical work has been performed to check MERODE models for deadlocks [18], this work is incomplete and only reports whether a model contains a potential deadlock or not, instead of providing the

---

[1]More information can be found at: https://www.omg.org/spec/FUML/1.5/About-FUML/

specific conditions under which the deadlock would occur, making it of little practical use.

This paper contributes to the state of the art in deadlock checking of UML-based models by taking a next step towards addressing deadlock situations in Model-Driven Engineering within the MERODE approach by presenting a practical algorithm capable of detecting deadlock situations in conceptual models. These deadlock situations arise from the parallel execution of separate artifacts within these models. The present paper introduces the *Algorithm for Deadlock Detection Leveraging Incomplete Feedback* (ADD-LIFE), and explores the prevalence of deadlock situations within MERODE models. Ultimately, the ADD-LIFE algorithm would be used to provide feedback to students. Before implementing the algorithm in a didactic modelling tool, the present research evaluates the feasibility of developing such an algorithm and assesses its utility, thus addressing the following research questions:

- RQ1: What is a feasible way to algorithmically perform (partial) checks on conceptual models for deadlock, livelock, deadlock imminent and livelock imminent situations?

- RQ2: How many deadlock, livelock, deadlock imminent and livelock imminent situations can be identified using such an algorithm on conceptual models from students?

The subsequent sections of this paper are organized as follows: section 2 presents an overview of related work concerning deadlock detection in object-oriented conceptual models and discusses the prior work on the MERODE approach. Section 3 presents the algorithm that was developed as answer to RQ1. Section 4 presents the answer to RQ2 by applying the algorithm to student homeworks from the past 5 years. Section 5 offers a further discussion of the results, providing insights and interpretations and also addresses limitations. Lastly, section 6 concludes the paper by summarizing the findings. It also outlines the next steps to be taken in further developing and refining a tool that would use the algorithm.

2. Related work

2.1. Deadlock detection in conceptual models

In Model-Driven Engineering (MDE), software development revolves around creating domain models as abstract representations of the desired software behavior, rather than directly writing source code. These domain models serve as conceptual models that capture the expected behavior of the software system and are used to automatically generate runnable code.

Roughly speaking, three types of errors can occur during the MDE process. First, the conceptual models themselves may contain errors or misinterpretations of the requirements (type 1 errors). Second, a model may contain internal inconsistencies, such as inaccessible states in statecharts, unlinked classes in class diagrams, or deadlock situations resulting from backward inaccessibility or interactions between different statecharts (type 2 errors). Finally, transformation errors (type 3 errors), may result in the generated code being erroreous and not correctly implementing the conceptual model is was generated from. The work presented in this paper focuses on type 2 errors, and more specifically different types of deadlocks as type 2 errors in a set of concurrent and interacting state charts.

Zhang et al. [19] have introduced a framework that ensures the absence of deadlock situations when transforming SLCO models (which are based on state machines) to code. This framework guarantees that the semantic behavior of the model is faithfully preserved in the generated code. The work is part of the SLCO language framework, which focuses on ensuring error-free translation from SLCO models to code [20]. SLCO thus focuses on type 3 errors and does not guarantee internal consistency within the model itself (type 2 errors). While in the domain of process modelling most of the work focuses on checking a single process for deadlock (e.g. using Petri Nets as formalism), some of the work in this domain is related to checking state charts as well. Takaki et al. [21] proposed the EVA algorithm, which verifies a workflow's consistency with the state charts that constitute the default lifecycles of the data artifacts manipulated by the process. They also introduced an algorithm to transform cyclic workflow diagrams into acyclic ones, enabling the application of the EVA algorithm. The paper demonstrates that EVA can determine the consistency of lifecycles and identify defects in a real large enterprise application system. However, these approaches primarily address the consistency of generated code and the surrounding business processes, rather than the internal consistency of the conceptual model of the data artefacts. Artefact-centric process models define systems as a collection of business artefacts having lifecycles defined by means of statecharts[22]. Calvanese et al. [23] investigated safety properties using Satisfiability-Modulo-Theories (SMT) agnostic to the initial database structure, making it applicable to artifact-centric systems. While they address type 2 errors for concurrent state charts, their research identified three classes of systems for which safety is decidable, limiting the applicability of their results to certain models. Deutsch et al. [24] implemented WAVE, a tool for static analysis of a subset of data-aware process models. However, their verification checks only apply to that specific subset of models, as other models are not within the PSPACE complexity class. Thus, these approaches are limited and cannot be universally applied to all conceptual models that define a series of concurrent state charts.

While object interaction in UML is binary (one object sends a message to one other object), in MERODE, object interaction has been defined by means of joint participation to common events, thus allowing for multi-party interaction. The semantics of this interaction have been formalised by means of Communicating Sequential Processes (CSP) [17]: when an event is triggered, all processes that have this event in their input alphabet will react on it.

Numerous studies have already investigated deadlock detection for CSP, including works by Huang et al. [25], Lima et al. [26], Isobe et al. [27], and Ladkin [28]. These papers propose algorithms that automatically detect deadlock situations in CSP-based systems. However, these algorithms assume that the number of processes is known in advance (e.g. a system with 5 dining philosophers who share 5 forks). While addressing type 2 errors, the assumption of a fixed number of objects does not hold in the broader context of information systems, where objects can be created and ended, thus adding processes to a system or removing them (i.e. philosophers and forks can be added and removed, thus resulting the need for checking systems that may have an arbitrary number of philosophers and forks).

Despite the many works on safety and deadlock checking either based on Petri Nets, artefact-centric modelling or CSP, to the best of our knowledge, no prior research has introduced an approach for static analysis of deadlock detection in conceptual models, particularly when considering the time and space complexity challenges that arise with large-scale models.
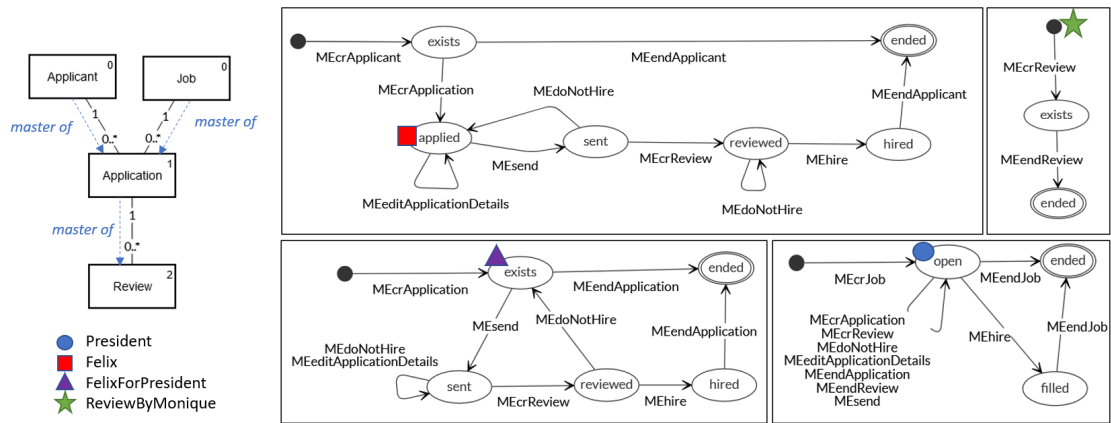
Figure 1: Model with possible deadlock situation. The tokens show the system's state after the sequence crJob (President), crApplicant (Felix), crApplication (FelixForPresident)

2.2. Existing work on MERODE

The MERODE approach is an MDE approach that uses class diagrams and finite state machines. The class diagram represents the structural aspects by means of object types and their associations, while the finite state machines model the behavioral aspects. In MERODE, the class diagram takes the form of an Existence-Dependency Graph (EDG), where object types are connected through associations that express existence dependency. The EDG can be obtained from a regular UML class diagram by reifying the associations that do not express existence dependency (ED). Assume for example a many-to-many association 'applies for' between jobs and applicants: in MERODE, this results in a class 'Application' that is ED on both 'Job' and 'Applicant', see Figure 1. In the EDG, all classes are thus arranged in a master-dependent order.

Each class has a state chart that defines its lifecyle. Object interaction happens through joint participation to events. Assume that in the example of Figure 1 three objects have already been created: `President` is an instance of Job and is in the state *open*; `Felix` is an instance of Applicant and is in the state *applied*, and `FelixForPresident` is an instance of Application and is in the state *exists*. `ReviewByMonique` has not yet been created. Joint participation to events means that each object that has been defined to participate in the event must validate the event and possibly react to it. For example, when the *send* event occurs, the `FelixForPresident` object will react to it and move from the *exists* to the *sent* state, but, since each application is connected to a single applicant, the *send* event also appears in the lifecycle of the `Felix` object. Hence, the occurrence of the *send* event will also transition `Felix` from the *applied* to the *sent* state and `President` from the *open* to the *open* state. However, transitions only happen if the objects are in the right state. In line with CSP's definition of interaction, all participants to an event must accept an event for it to be accepted overall. If `Felix` is not in the *applied* state but in the *exists* state, the event would not be executed for either of the thee objects because the state of `Felix` does not allow the event to be executed.

Making use of the lifecycle implications of existence dependency, three approaches have been defined to enhance the internal consistency between different views of a software model

[29, 30]. The *consistency by construction* approach infers specifications in one view based on user-defined specifications in another view, for example by automatically adding default lifecycles when creating a class (see the lifecycle of Review in Figure 1). The *consistency by monitoring* approach checks newly defined specifications for consistency at the moment they are added, preventing the creation of cyclic inheritance relationships, for example. This conforms to the aforementioned use of OCL [4] to specify constraints on the meta-model to prevent model errors. The *consistency by analysis* approach uses algorithms to check for inconsistencies between different deliverables and reports any identified issues. These approaches enable modelers to freely construct various artifacts while ensuring consistency.

As the default statecharts do not impose any other sequence constraint than creation having to happen first and ending last, in combination with the a-cyclicity of the EDG, such system has been proven to be deadlock-free [18]. However, the default statecharts often fail to capture the full complexity of the requirements, such as for example having a blocked state for a bank account, thus requiring the definition of more detailed statecharts. The introduction of additional sequence constraints however holds the risk of introducing deadlock problems. To check for these errors, the *consistency by construction* approach cannot be applied. Therefore, *consistency by analysis* and *consistency by monitoring*, both relying on algorithmic checks for model consistency, are to be used. The *consistency by analysis* approach performs checks when the user requests model verification on their conceptual model, while the *consistency by monitoring* approach verifies the new conceptual model immediately upon user-initiated changes. For example, when the user attempts to add an ending transition to a non-final state, the consistency by monitoring approach will not allow this transition to be added. The addition of a new state will cause a problem of inaccessibility and deadlock but is not prevented to ensure usability of the tool. But when the user then requests a model check from the system, the consistency by analysis will inform the user about the inaccessibility of a state.

Several verification checks for the MERODE approach already exists in MERLIN [31], a tool to develop MERODE models with. These checks include verifying the accessibility of states, ensuring object types have attributes, confirming the determinism of statecharts, and the absence of forward inaccessible states. When considering a single object type, the MERLIN tool can detect potential deadlock situations by verifying the absence of backward inaccessible states. However, MERLIN has no algorithm for checking a collection of statecharts for deadlock problems. Previous research on MERODE has already proposed an algorithm to check a complete model for deadlock freedom [18], be it with some simplifications to circumvent problems arising from unbounded interleaving. Unbounded interleaving results from maximum many cardinalities. Indeed, in the case of a 1-to-many association, an instance on the 1 side may have an unbounded number of instances on the many side running concurrently. In our example, an applicant may have an unbounded number of concurrent applications. When translating this into automata theory, a full deadlock check in the MERODE approach would require automata able to deal with unbounded interleaving semantics, i.e. push-down automaton conforming to a context-free (CF) grammar. Such a pushdown automaton can be derived from on the class diagram and state diagrams of the conceptual model. To find possible deadlock situations in the conceptual model, one then needs to check whether all possible input sequences (strings) are accepted by the automaton. For one input sequence of length n, a pushdown automaton has a time complexity of n^3 [32]. Since not just one input sequence has to be checked, but all possible

input sequences, this clearly turns into a computationally intractable approach.

To avoid the complexities associated with pushdown automata, the algorithm therefore replaces the 0/1.* cardinalities on the side of the dependent with 0/1..1. This forces the dependent class to exist sequentially, thus exhibiting an iteration instead of unbounded interleaving behaviour (for the given example: an application can have a sequence of applications). As iteration belongs to regular expression grammars, this can be handled by means of an finite state automaton. The algorithm works by calculating the parallel composition of the concurrent statecharts of a master with the iteration of its dependents. Once the parallel composition has been calculated, the algorithm verifies if all events remain fireable in the resulting finite state machine. If all events are still firable, the system consisting of these concurrent statecharts, is said to be deadlock-free. Thus, it does therefore not ensure the absence of potential deadlock situations in a object-oriented conceptual model; it merely ensures the existence at least one deadlock-free execution trace and the presence of all events in at least one of these traces. Therefore, a more comprehensive approach is needed to detect deadlock situations that may arise from the combined behavior of multiple object types in the system.

3. Defining a feasible algorithm

The parallel execution of multiple interacting statecharts may result in deadlocks. This means they have reached a state where none of them can make further progress, rendering the system unresponsive and incapable of executing any further actions. While some deadlock situations, such as those resulting from backwards inaccessible states within a single statechart, are relatively easy to detect, the complex deadlock scenarios that arise due to the interplay between concurrent statecharts are more difficult to detect.

An experienced modeller may notice, by merely inspecting the FSMs of Figure 1, that an applicant will be able to have at most one application. Such problem identification requires the calculation of the system behaviour as the parallel composition of the concurrent statecharts. An experienced modeller might be able to do this mentally. Junior modelers, however, struggle a lot more to do this type of verification by just viewing and mentally interpreting the models.

Various types of deadlock situations can be identified within a model's global state. The first type is the *deadlock* situation, in which no further steps can be executed. A second type is the *deadlock imminent* situation, where one or more steps can still be executed, but this will eventually lead to a *deadlock* situation. The third type is the *livelock* situation, which allows steps to be executed but lacks a sequence of steps that allows for progress towards a final state. Lastly, the fourth type is the *livelock imminent* situation, in which all possible sequences of steps will lead to a *livelock* situation.

The given example contains deadlock and livelock situations. For instance, the sequence of events *MEcrApplicant*, *MEcrApplication*, *send*, *review*, *doNotHire* would place the `Applicant` object in the *reviewed* state and the `Application` object in the *exists* state. In this global state of the model, for the `Applicant` object only the *hire* and *doNotHire* events can be executed, while for the `Application` object only the *send endApplication* event are permitted. Consequently, the system is in a deadlock state, as no further events can be executed. It's worth noting that neither `Applicant` nor `Application` contain a deadlock situation on their own.

Additionally, the same example contains a livelock situation. The sequence of events *MEcrApplicant*, *MEcrApplication*, *send*, *doNotHire* places the `Applicant` object in the *applied* state and the `Application` object in the *sent* state. In the *applied* state, an `Applicant` can execute either the *send* or *editApplicationDetails* event, while in the *sent* state, an `Application` can execute the *doNotHire*, *review*, or *editApplicationDetails* events. It is still possible to execute the *editApplicationDetails* event, but this doesn't change the state for neither object. As it is the only remaining executable event, there is no way to terminate both the `Applicant` and the `Application`, thus resulting in a livelock situation. Again, neither the `Applicant` nor the `Application` object type has a livelock situation on its own.

To address RQ1, we develop an algorithm capable of detecting deadlock problems in MERODE models with an acceptable running time. The algorithm exploits the fact that the class diagram is a directed acyclic graph (DAG) organizing object types along ED thanks to the reification of associations until they express existence dependency. In such DAG we can identify top and bottom object types. A top object type is not dependent on any other object type, while a bottom object type has no dependents. A formal definition of top and bottom object types is given in Definition 18b of [18]. In Figure 1, the object types have been numbered from top to bottom, starting with 0. To compute a parallel statechart based on the concurrent statecharts of a subset of the EDG, starting from a top object type and ending at a bottom object type, all the paths in the EDG hierarchy are computed. In our example, there are two top object types (`Job` and `Applicant`) and one bottom object type (`Review`). There are two paths, one running from `Applicant` through `Application` to `Review` and one that starts at `Job` and ends at `Review`.

The proposed algorithm, calculates a parallel statechart that represents the global behaviour in the following way. First, all cardinalities of "many" are replaced by "one" so as to replace interleaving by iteration. Next, for each master object type, the parallel composition with an iteration of each dependent object type is calculated. The FSM representing the iteration of an object type is obtained by simply rerouting the transitions that end in a final state to the initial state and relabelling this state as both initial and final. This process is repeated iteratively until a complete path from top to bottom has been calculated. Finally, the parallel combination of the paths is calculated. This is identical to the algorithm presented in [18]. However, the algorithm in [18] removes backwards inaccessible states after each calculation step. This immediate pruning of backwards inaccessible states in each intermediate result helps to avoid rising computational time due a state explosion. However, this pruning also removes the useful information about the exact combination of states that are causing a deadlock situation.

To provide modelers with detailed feedback, it is important to keep the backward inaccessible states in the results as they represent the deadlock states in the global system behaviour. To determine the type of deadlock, it can be checked whether other deadlock states are reachable from the identified deadlock state. If no other (deadlock) states are reachable, it indicates a *deadlock* situation. If another deadlock state is reachable, then we deal with a *deadlock imminent* situation. If the identified deadlock state is still reachable from itself, this indicates a *livelock* situation. If at least one (other) livelock state is reachable, it indicates a *livelock imminent* situation. The transitions towards the problematic states provide useful information as to which scenarios ultimately lead to deadlock/livelock.

In summary, the ADD-LIFE algorithm follows the following steps:

1. Calculate all paths from top object types to bottom object types.

2. For each found path in step 1, apply Algorithm 1 (this algorithm is adjusted from algorithm A.1 as described in [18] and takes into account referential integrity contstraints) iteratively for each object type in the path to obtain the parallel behaviour of all the statecharts in the path.

3. For all the calculated statecharts obtained from step 2, apply Algorithm 2 (this algorithm is adjusted from algorithm A.1 as described in [18]) iteratively for each statechart to obtain the parallel statechart that models the combined behaviour of the entire system.

4. Identify the backwards inaccessible states in the resulting parallel statechart, these are the deadlock states.

5. Classify the resulting deadlock states.

Algorithm 1 The algorithm to combine the statechart of a master with the statechart of a dependent.

Require: objectType2 is existence dependent of objectType1. FSM M1 is the statechart modelling the behaviour of objectType1, FSM M2 is the statechart modelling the behaviour of objectType2.
Ensure: FSM M is the parallel statechart of FSM M1 = $(\Sigma_1, Q_1, \Delta_1, q_1, F_1)$ and FSM M2 = $(\Sigma_2, Q_2, \Delta_2, q_2, F_2)$.
$M \leftarrow (\Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, \Delta, q_1 \times q_2)$ such that

    for $a \in \Sigma_1 \cap \Sigma_2 \wedge a \notin d(\Sigma_1)$ **do**   ▷ d contains the subset of deleting event of the alphabet of objectType1
        if $\Delta_1(q_1, a) = q_1' \wedge \Delta_2(q_2, a) = q_2'$ **then**
            $\Delta((q_1, q_2), a) \leftarrow (q_1', q_2')$
        end if
    end for
    for $a \in \Sigma_1 \setminus \Sigma_2$ **do**
        if $\Delta_1(q_1, a) = q_1'$ **then**
            $\Delta((q_1, q_2), a) \leftarrow (q_1', q_2)$
        end if
    end for

Algorithm 2 The algorithm to combine the statechart of combined FSMs of two subparts of the EDG.

Require: M1 is the parallel statechart of a path of the EDG, M2 is the parallel statechart of a path of the EDG
Ensure: FSM M is the parallel statechart of FSM M1 = $(\Sigma_1, Q_1, \Delta_1, q_1, F_1)$ and FSM M2 = $(\Sigma_2, Q_2, \Delta_2, q_2, F_2)$.
$M \leftarrow (\Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, \Delta, q_1 \times q_2)$ such that

    for $a \in \Sigma_1 \cap \Sigma_2$ **do**
        if $\Delta_1(q_1, a) = q_1' \wedge \Delta_2(q_2, a) = q_2'$ **then**
            $\Delta((q_1, q_2), a) \leftarrow (q_1', q_2')$
        end if
    end for
    for $a \in \Sigma_1 \setminus \Sigma_2$ **do**
        if $\Delta_1(q_1, a) = q_1'$ **then**
            $\Delta((q_1, q_2), a) \leftarrow (q_1', q_2)$
        end if
    end for
    for $a \in \Sigma_2 \setminus \Sigma_1$ **do**
        if $\Delta_2(q_2, a) = q_2'$ **then**
            $\Delta((q_1, q_2), a) \leftarrow (q_1, q_2')$
        end if
    end for

Regarding computing time, the calculation of combined statecharts and the generation of graphs were completed within a few seconds when just a single path is verified. However, if

multiple instantiations of an object type are allowed or the entire parallel statechart is calculated (instead of just the paths in the EDG), the computing time grows exponentially. Estimations indicate that calculating the combined statechart for a single model would take several days under these conditions. This is in line with what was discussed in Section 2, concerning computational time when using push-down automata for recognizing CF grammars.

4. Validation of utility

In order to answer RQ2, the algorithm was run against students' homework solutions of the last 5 academic years. As we can see from Table 1, on average 45% of the student models contain at least one deadlock situation.

Table 1
Total number of deadlock models per academic year.

Academic year	18-19	19-20	20-21	21-22	22-23	Total students solutions
Deadlock-free models	10	10	17	9	5	51
Deadlock models	8	12	10	7	4	41
Percentage of deadlock models	44%	55%	37%	44%	44%	45%

Table 2
Results of the Kruskal-Wallis test and Spearman correlation coefficient.

Deadlock type	Deadlock	Livelock	Deadlock imminent	Livelock imminent
Kruskal-Wallis (p-value)	0.73	0.59	0.65	0.87

Artefact type	Nr. of Classes		Nr. of States	
Student models (Spearman ρ)	-0.10		-0.02	

As each year a different assignment was used, we also investigate whether the prevalence of deadlock situations of a certain type depends on the provided exercise. A Kruskal-Wallis test can be used for ordinal data (in this case count data) to compare multiple independent samples (students from different academic years) when there are more than 2 samples (5 academic years). This Kruskal-Wallis test is used to determine whether the means of all the different academic years, in which a different exercise was given for the homework, are equal or not. A Kruskal-Wallis test works under the null hypothesis that all group means are the same. The results in Table 2 shows all p-values to be non-significant. Furthermore, some models might be more prone to deadlock situations, due to them consisting of more object types or more states than others. Therefore, a correlation between the number of classes and states in the models compared to the total number of deadlock situations in the model has been calculated too. For this we use the Spearman correlation coefficient, as the number of classes/states and deadlock situations are ordinal data. Also here, we see no significant correlation.

5. Discussion

To achieve a feasible algorithm in response to RQ1, two concessions had to be made regarding exhaustive verification for deadlock: interleaving resulting from cardinalities of many is replace by iteration by changing the maximum cardinality to one and the verification is limited to single paths from top to bottom. In [18] a different choice was made: backward inaccessible states are removed from parallel statechart at the cost of losing detailed information as to how many deadlock situations are present and the in which states of the concurrent statecharts they appear. An important limitation of the ADD-LIFE algorithm is that the made concessions affect the completeness deadlock state detection. This means that other deadlock situations may exist beyond those identified by the algorithm. However, due to the state explosion when doing a full calculation, it is practically impossible to know which percentage of deadlock situations have been identified with the proposed algorithm. A second limitation is the limited external validity, as the research focuses on conceptual models created within a specific MDE approach based on CSP. For MDE approaches based on a different process algebra, the algorithm can be adjusted to take into account the different language semantics. Herein, the general idea of allowing only a limited amount of instantiations of the object types to avoid computational issues remains.

For RQ2, the analysis of the results reveals that almost half of the student models contain deadlock situations. While it is possible that the prevalence of deadlock situations may vary across academic years, the Kruskal-Wallis test results in Table 2) did not indicate any significant differences between different years. Additionally, no correlation was found between the number of classes or states in a model and the occurrence of deadlock situations (Table 2), which shows that the complexity of a model, as different assignments might come at different difficulty levels, is not per se related to the prevalence of deadlock situations. While deadlock is mentioned and illustrated when discussing homework results, the topic is not addressed in a systematic way during the course. Students are expected to extrapolate the examples to their own solutions and detect them through adequate testing. However, research has found that the students' manual testing efforts are often below par, as evidenced by their use of an insufficient number of test cases to assess the accuracy of requirement modelling [33]. This leads us to believe that raising students' awareness via automated feedback of potential deadlock situations can be of help in reducing the prevalence of such situations, since students do not bump into these deadlock situations when testing their models on their own.

To enhance internal validity, student models from different academic years were included, to verify that the prevalence of deadlock situations cannot be attributed solely to one specific exercise or year. Again, external validity is limited by the focus on MERODE models. Further research could check the models made by students at other universities where the tool is also used, and investigate other modelling approaches.

6. Conclusion and future work

In this paper, we presented the ADD-LIFE (Algorithm for Deadlock Detection Leveraging Incomplete Feedback) algorithm, which aims to automatically detect deadlock situations in conceptual models. The algorithm calculates the global behaviour resulting from individual

concurrent statecharts and addresses state explosion and time complexity by limiting the number of instantiations per object type and peforming a partial check only.

To evaluate the utility of the algorithm, we applied it to student models collected over multiple years from an MDE course. We identified four types of deadlock situations: deadlock, livelock, deadlock imminent, and livelock imminent. No significant differences were found in the number of deadlock situations among student solutions from different years. Additionally, there was no correlation between the number of classes or states and the occurrence of deadlock situations in the models. This suggests that deadlock errors are quite generally present, thus motivating the need for algorithmic assistance for detecting them.

Two main lines of future research can be developed from this research. First, considering that many student models contain deadlock situations, the ADD-LIFE algorithm could be implemented in the modelling tool to provide automatic feedback to both lecturers and students on the presence of deadlock in their models. Students can use this feedback to learn which modelling choices introduce deadlock situations, while lecturers can use such a tool to aid them in highlighting difficult to find deadlock situations in student homeworks.

The algorithm can be easily integrated with the tooling support that already exists for the MERODE approach. Since the *consistency by construction*, *consistency by monitoring* and *consistency by analysis* approach are already present in the MERLIN tool, the algorithm's feedback can be added to these other checks. After successful integration with the MERLIN tool, we can investigate the impact on students learning process. This can be researched by means of small experiments, but since the tool allows for logging students' interaction, we can evaluate to what extent the new feature is use and whether or not it decreases the prevalance of deadlock situations for the student groups who make use of ADD-LIFE's feedback.

A second line of research could further develop the approach so that it can also be used for other MDE approaches. For instance, according to the UML standard for class diagrams, additional constraints can be modelled, such as preconditions. Further research could thus extend the evaluation by applying the algorithm to models from different courses or modeling approaches. However, it is worth noting that the algorithm may not be suitable for evaluation in MDE approaches based solely on UML without a formal process algebra. Finally, futher research could explore the algorithm's applicability in real-world MDE scenarios outside of academic settings. Overall, while the algorithm's evaluation provides valuable insights, further research is needed to assess its effectiveness and generalizability in diverse MDE contexts.

Acknowledgments

References

[1] H. Krasner, The cost of poor software quality in the US: A 2020 report, Proc. Consortium Inf. Softw. QualityTM (CISQTM) (2021).

[2] The RISKS Digest Volume 33 Issue 01, https://catless.ncl.ac.uk/Risks/33/01#subj1, 2022. Accessed: 2023-07-06.

[3] S. Käss, S. Strahringer, M. Westner, Practitioners' perceptions on the adoption of low code development platforms, IEEE Access 11 (2023) 29009–29034. doi:10.1109/ACCESS.2023.3258539.

[4] OCL, http://www.omg.org/spec/OCL/, 2017. Accessed: 2023-09-07.

[5] C. Verbruggen, M. Snoeck, Practitioners' experiences with model-driven engineering: a meta-review., Softw Syst Model 22 (2023) 111–129. URL: https://link.springer.com/article/10.1007/s10270-022-01020-1. doi:https://doi.org/10.1007/s10270-022-01020-1.

[6] M. Estañol, M.-R. Sancho, E. Teniente, Ensuring the semantic correctness of a bauml artifact-centric bpm, Information and Software Technology 93 (2018) 147–162. URL: https://www.sciencedirect.com/science/article/pii/S0950584917301404. doi:https://doi.org/10.1016/j.infsof.2017.09.003.

[7] M. Snoeck, C. Verbruggen, J. D. Smedt, J. D. Weerdt, Supporting data-aware processes with merode., Softw Syst Model 23 (2023) 111–129. URL: https://link.springer.com/article/10.1007/s10270-023-01095-4. doi:https://doi.org/10.1007/s10270-023-01095-4.

[8] G. Sedrakyan, S. Poelmans, M. Snoeck, Assessing the influence of feedback-inclusive rapid prototyping on understanding the semantics of parallel uml statecharts by novice modellers, Information and Software Technology 82 (2017) 159–172. URL: https://www.sciencedirect.com/science/article/pii/S0950584916303044. doi:https://doi.org/10.1016/j.infsof.2016.11.001.

[9] R. M. Dijkman, M. Dumas, C. Ouyang, Formal semantics and analysis of BPMN process models using Petri nets, Queensland University of Technology, Tech. Rep (2007) 1–30.

[10] I. Raedts, M. Petkovic, Y. S. Usenko, J. M. E. van der Werf, J. F. Groote, L. J. Somers, Transformation of BPMN Models for Behaviour Analysis., MSVVEIS 2007 (2007) 126–137.

[11] F. Chu, X.-L. Xie, Deadlock analysis of Petri nets using siphons and mathematical programming, IEEE Transactions on Robotics and Automation 13 (1997) 793–804.

[12] Y. Huang, M. Jeng, X. Xie, S. Chung, Deadlock prevention policy based on Petri nets and siphons, International Journal of Production Research 39 (2001) 283–305.

[13] M. Jeng, X. Xie, M. Zhou, M. Fanti, Deadlock detection and prevention of automated manufacturing systems using Petri nets and siphons, Deadlock resolution in computer-integrated systems (2005) 233–281.

[14] Y. Choi, J. L. Zhao, Decomposition-based verification of cyclic workflows, in: International Symposium on Automated Technology for Verification and Analysis, Springer, 2005, pp. 84–98.

[15] H. Lin, Z. Zhao, H. Li, Z. Chen, A novel graph reduction algorithm to identify structural conflicts, in: Proceedings of the 35th Annual Hawaii International Conference on System Sciences, IEEE, 2002, pp. 10–pp.

[16] W. M. van der Aalst, A. Hirnschall, H. Verbeek, An alternative way to analyze workflow graphs, in: Advanced Information Systems Engineering: 14th International Conference, CAiSE 2002 Toronto, Canada, May 27–31, 2002 Proceedings 14, Springer, 2002, pp. 535–552.

[17] C. A. R. Hoare, Communicating sequential processes, Communications of the ACM 21 (1978) 666–677.

[18] G. Dedene, M. Snoeck, Formal deadlock elimination in an object oriented conceptual schema, Data & Knowledge Engineering 15 (1995) 1–30.

[19] D. Zhang, D. Bošnački, M. van den Brand, C. Huizing, B. Jacobs, R. Kuiper, A. Wijs, Verifying

atomicity preservation and deadlock freedom of a generic shared variable mechanism used in model-to-code transformations, in: Model-Driven Engineering and Software Development: 4th International Conference, MODELSWARD 2016, Rome, Italy, February 19-21, 2016, Revised Selected Papers 4, Springer, 2017, pp. 249–273.

[20] S. de Putter, A. Wijs, D. Zhang, The SLCO framework for verified, model-driven construction of component software, in: Formal Aspects of Component Software: 15th International Conference, FACS 2018, Pohang, South Korea, October 10–12, 2018, Proceedings 15, Springer, 2018, pp. 288–296.

[21] O. Takaki, T. Seino, I. Takeuti, N. Izumi, K. Takahashi, Verification of evidence life cycles in workflow diagrams with passback flows, International Journal On Advances in Software 1 (2008).

[22] R. Hull, Artifact-centric business process models: Brief survey of research results and challenges, in: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", Springer, 2008, pp. 1152–1163.

[23] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, From model completeness to verification of data aware processes, Description Logic, Theory Combination, and All That: Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday (2019) 212–239.

[24] A. Deutsch, R. Hull, V. Vianu, Automatic verification of database-centric systems, ACM SIGMOD Record 43 (2014) 5–17.

[25] S.-T. Huang, A distributed deadlock detection algorithm for CSP-like communication, ACM Transactions on Programming Languages and Systems (TOPLAS) 12 (1990) 102–122.

[26] L. Lima, A. Tavares, S. C. Nogueira, A framework for verifying deadlock and nondeterminism in UML activity diagrams based on CSP, Science of Computer Programming 197 (2020) 102497.

[27] Y. Isobe, M. Roggenbach, CSP-Prover—A proof tool for the verification of scalable concurrent systems, Information and Media Technologies 5 (2010) 32–39.

[28] P. B. Ladkin, B. B. Simons, Static deadlock analysis for CSP-type communications, in: Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems, Springer, 1995, pp. 89–102.

[29] M. Snoeck, C. Michiels, G. Dedene, Consistency by construction: the case of MERODE, in: Conceptual Modeling for Novel Application Domains: ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM, Chicago, IL, USA, October 13, 2003. Proceedings 22, Springer, 2003, pp. 105–117.

[30] M. Snoeck, G. Dedene, Existence dependency: The key to semantic integrity between structural and behavioral aspects of object types, IEEE Transactions on software engineering 24 (1998) 233–251.

[31] M. Snoeck, MERLIN: An Intelligent Tool for Creating Domain Models, in: Research Challenges in Information Science: 14th International Conference, RCIS 2020, Limassol, Cyprus, September 23–25, 2020, Proceedings 14, Springer, 2020, pp. 549–555.

[32] A. V. Aho, J. E. Hopcroft, J. D. Ullman, Time and tape complexity of pushdown automaton languages, Information and Control 13 (1968) 186–206.

[33] G. Sedrakyan, M. Snoeck, S. Poelmans, Assessing the effectiveness of feedback enabled simulation in teaching conceptual modeling, Computers & Education 78 (2014) 367–382.