Spectrum - The Journal of Operational Applications in Defense Areas

Journal homepage: https://spectrum.ita.br

Operational Analysis and Logistics Engineering: Decision Support

Inconsistency detection methods for statecharts and sequence diagrams: a systematic literature review

Matheus Cogo, Carline Muenchen, Christopher Cerqueira, and Emilia Villani
Instituto Tecnológico de Aeronáutica (ITA), São José dos Campos/SP – Brazil

Article Info

Article History:

Received	28 May 2025
Revised	28 June 2025
Accepted	24 July 2025
Available online	23 September 2025

Keywords:

systematic literature review
model-based systems engineering
model consistency
prisma

E-mail adresses:

cogo@ita.br
chris@ita.br
evillani@ita.br

Abstract

During model-based systems engineering or software engineering activities, diagrams representing use cases (sequence diagrams) and diagrams representing object behaviors (state machine diagrams or statecharts) can conflict with each other in what is called an inconsistency. Detecting these inconsistencies is crucial to check if a given specification is realizable through the behavior that was conceived to meet it. This paper provides a systematic literature review of inconsistency detection methods for UML state machine diagrams and sequence diagrams. The selection process is aided by an open-source machine-learning tool, and resulted in the qualitative synthesis of 27 works. The included publications offer methods to tackle the detection of horizontal-semantic behavior inconsistencies.

I. INTRODUCTION

Systems engineering (SE) activities have become increasingly more "model-driven" in the past 20 years [1]. The shift from document-based engineering to model-based systems engineering (MBSE) or model-driven engineering (MDE) was sparked by several reasons, such as ambiguity reduction and improved traceability, but it was the advent and popularization of digital tools, methods and languages that promoted and effectively enabled this transition [1]. UML (Unified Modeling Language), SysML (Systems Modeling Language) and, more recently, Arcadia are the most prominent examples of MBSE-enabling languages.

The mere adoption of MBSE or MDE, however, did not solve all problems intrinsic to the engineering of intricate and highly coupled systems. Just as consistency issues can arise in document-based approaches, SE models may also exhibit them. Logically, an *inconsistency* represents a contradiction. If any two propositions defined in a model are not simultaneously true, it is said that the model is not consistent [2]. Model consistency is then defined as the absence of inconsistencies, and may be taxonomically branched into several types such as horizontal or vertical consistency and semantic or syntactic consistency [3]. A SE model is often composed of several abstraction facets of a system called *viewpoints* [4]. These viewpoints may evolve concurrently and even overlap as development goes on. As a consequence of poor viewpoint integration, inconsistencies may surface either in the structure of a model or in its behavior.

Of particular interest to this paper is the Arcadia MBSE method [5], implemented in the Capella tool, which extensively uses both Sequence Diagrams (SDs) and Statecharts (SCs, originated from [6]) to model the system's behavior. This systematic literature review (SLR) provides an overview of available inconsistency detection methods, and aims to answer the following research question: which methods are suitable to be implemented in the Arcadia-Capella environment as a plugin for the detection of behavior inconsistency between SDs (representing use cases) and SCs (representing the system components' behavior)?

Detecting inconsistencies between SC diagrams and SDs has the practical effect of checking if a given use case (as defined by the Arcadia methodology) proposed to fulfill a certain capability is realizable by the behavioral logic of that system's components. If any inconsistencies are found, the model should have its behavior changed to match the expected use case or vice-versa. The same applies to forbidden sequences of events.

In general, an inconsistency may also arise not due to a semantic divergence, but from a syntactic one. For example, SCs transitions should contain the same events' names as SDs messages' names. The Capella tool, however, already

eases the detection and handling of these, as the diagrams are feature-rich and internally linked with their metamodels. In fact, the tool user may implement validation rules that navigate through the metamodel to check if a specific element has all the linkage expected for that type. For those reasons, this SLR is concerned only with inconsistencies derived from semantic contradictions.

A previous survey on model consistency [2], carried out in 2001, has tackled the issue on a broader *management* level, including not only the inconsistency detection methods but its diagnosis, handling and tracking with a focus on software engineering. An SLR conducted in 2009 presented inconsistency management methods for UML while explicitly showing which types of inconsistencies and diagrams were supported for each method analyzed [7]. An SLR conducted in 2017 [8] followed a similar strategy, noting the lack of CASE-integrated or tool-integrated methods for detection and handling of inconsistencies. In fact, it was found that almost 90% of studies were not tried in industrial settings [8]. Yet another SLR [9] published in 2017 classifies inconsistency managing methods by their paradigm and, most noticeably, by their compliance to some quality features proposed, like support for all kinds of inconsistencies. This paper differs slightly from those due to the need of surveying updated, state-of-the-art methods for a more specific and narrow use case: the horizontal-semantic behavioral inconsistency detection between SCs and SDs inside the Capella tool. Naturally, any method would, in theory, be implementable in the Capella platform, but it is beneficial to know during the research phase which methods require intermediate steps - like translation of diagrams to a third construct - before development begins. It also important to know the methods' limitations, as they could hinder the usage of the Capella tool. As a last remark, some of them may also require external tools, which could be an obstacle when developing an open-source plugin.

The rest of the paper is structured as follows: Section 2 will provide the reader with the SLR methodology used, such as the eligibility criteria, search strategy and selection process. Section 3 presents the SLR results, such as the study selection and its characteristics. Section 4 contains a discussion and an interpretation of the results found. Section 5 contains final remarks in regards to the methods surveyed and to future works.

II. METHODS

The SLR is guided by the Preferred Reporting Items for Systematic reviews and Meta-Analyses (PRISMA) 2020 statement [10]. For the data collection step, the sources used are Web of Science, Scopus and IEEE Xplore. The search strategy protocol is presented in Table I. The search string reflects the need to identify a model inconsistency detection (or management) method suitable for SCs (or similar behavior-modeling diagrams, such as petri-nets, state machines or finite automata) and SDs.

After data collection a data set preparation is required. By using ASReview Data Tools, an extension of ASReview, the individual data sets sourced from the aforementioned databases can be merged into a single RIS file with their duplicates removed.

Table I. Search protocol

Search parameter	
Data sources	Web of Science, Scopus, IEEE Xplore
Search string	*consisten* AND (model* OR diagram*) AND (statechart* OR "state machine*" OR petri* OR automata) AND sequenc*
Search fields	Title, abstract, keywords
Period	1995-2023
Languages	English
Document type	All

ASReview is an open-source machine-learning tool [11] that enables the quick systematic labeling of papers into two categories: relevant and irrelevant. The labeling decision is up to the user, and each input is used to train the model so that the next entry shown has a higher chance of being relevant. The screening step can then be prematurely stopped (compared to a manual screening process) once a stopping criteria [11] is met. The labeling of entries is done according to the eligibility criteria from Table II, applying exclusion rules to titles, abstracts and keywords provided by ASReview. The stopping criteria chosen was to label at least 33% of the sample's total size and hit 50 consecutively irrelevant publications. Although somewhat arbitrary, this criteria is considered to be conservative enough and is also being used in similar SLR publications [12]. During the screening procedure, whenever the authors disagreed on whether or not to include a publication, it was included to be further analyzed in the full-text eligibility assessment step. Similarly, whenever the search string keywords appeared in an unclear context but the authors were not confident enough to exclude it using the E1 criterion from Table II, the publication was included in the full-text analysis.

The next step is an eligibility assessment where the sample is refined by reapplying inclusion and exclusion rules on the full-text analysis of the sample's content. Lastly, a few more related works are included in the sample by using the forward and backward snowball sampling method [13], a technique consisting of exploring references and citations that match the inclusion and exclusion criteria.

The most impactful factors that may threaten the SLR validity are missing studies due to a biased search protocol and relevant studies being excluded at the screening process. The risk of the first factor is considered to be low, as the search string used is similar to that of previous SLRs [2] [7] [8] - in fact, in some cases the usage of asterisks as suffixes and prefixes of keywords cause the initial sample to be even more inclusive.

The second factor is considered to be of moderate risk due to two main reasons: there was limited, misleading or confusing phrasing in abstracts; the machine-learning approach may not have presented all relevant papers before the stopping criteria was met.

Table II. Eligibility criteria

Criterion	Description
E1	Out of scope due to different keyword meaning.
E2	The inconsistency detection method is not defined in sufficient details to replicate it.
E3	Full-text or abstract not available at the time or in another language.
E4	The inconsistency detection is referenced but is not part of the research effort.
E5	The method supports only a translation or conversion between diagrams.
I1	The method is applied semantically, horizontally between statecharts (or similar) and sequence diagrams.

III. RESULTS

In this section the SLR selection process flow diagram is presented, containing the sample size changes that occurred in each step. The results are then qualitatively presented. Due to size limitations, a quantitative analysis will not be included in this paper.

A. Selection process

Figure 1 contains the selection process flow diagram and the sample size of each step. During the screening step, from 442 articles initially identified, 118 were excluded based on Table II. Out of these, 105 were excluded due to E1; 3 were excluded due to E4; 9 were excluded due to E5; 1 publication was excluded due to E3. From the remaining 324 publications, 245 were considered not relevant because the stopping criteria was met. 7 publications were added back to the sample after divergences between the authors during application of exclusion rules. During the eligibility assessment step, out of 86 articles, 62 were excluded after the full-text analysis. Out of these, 31 were excluded due to E1; 2 were excluded due to E2; 16 were excluded due to E3; 11 were excluded due to E4 and 2 due to E5. Finally, during the snowball sampling step, 3 papers were included into the final sample. These steps produced a final sample size of 27 relevant papers which were included in the qualitative synthesis.

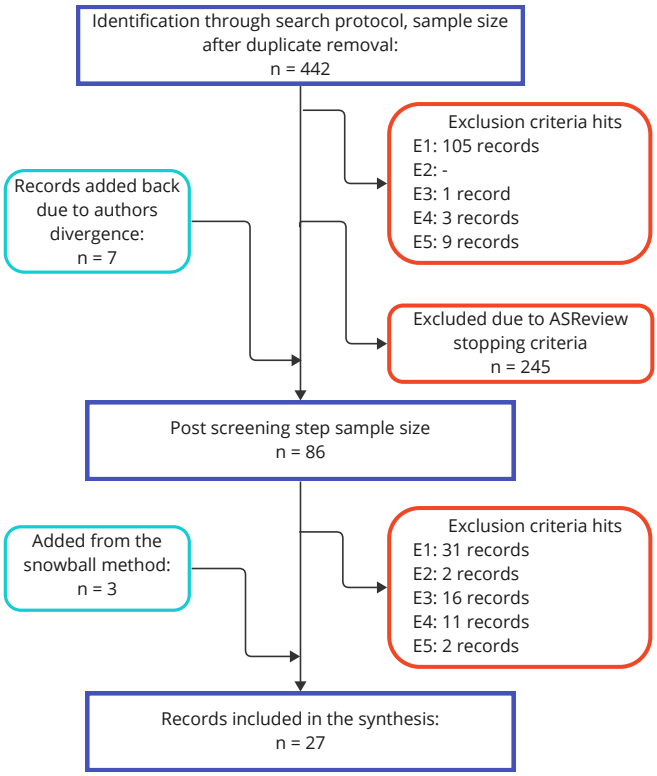

Fig. 1. Selection process flow diagram

B. Synthesis of methods surveyed

The following paragraphs synthesize the methods' characteristics, exposing the rationale behind the technique used, their use cases and gaps identified by the original authors. In Section IV the applicability of methods will be discussed based on parameters like easiness of implementation and UML features supported.

Litvak, B. et al [14]: describes an algorithm to check consistency based on semantic equivalence of both diagrams (e.g. lifelines and state machine objects). This algorithm is implemented as the Behavioral Validator of UML (BVUML) tool. It supports pseudostates, and is not integrated into a CASE/MBSE tool.

Lucas, F. et al [7]: employs a transformation language (IQVT-Maude) that maps the semantics between distinct metamodels and defines rewriting rules. These rewriting rules can be used to check the semantic consistency of state machines and SDs, since lifeline messages can, for example, be expressed as class methods or state machine triggers. Even though the method is implemented using an EMF metamodel, a full integration into a CASE/MBSE tool is not realized.

Yokogawa, T. et al [15]: uses the Labelled Transition System Analyser (LTSA) model checker to detect inconsistencies in state diagrams and SDs that were converted to processes. Presents no support for hierarchy in SDs and no CASE/MBSE tool integration.

Phuklang, S. et al [16]: translates state machine diagrams and SDs to a process representation described by the Communicating sequential processes (CSP) language, which is then fed into a Failures-Divergence Refinement (FDR) model checker, similarly to the method proposed in [15]. No mention of features supported. Integrated into a standalone Java application.

Matsumoto, A. et al [17]: extension of a previous work [18] similar to [15] that allows for hierarchical structures in SDs, such as combined fragments (the two types allowed are *alt* and *loop*). The translated CSP from SDs and state machine diagrams are checked with a FDR model checker.

Yokogawa, T. et al [19]: extension of previous works [17] [15]. The method used is the same as [17], but the

types of combined fragments increased to include the *alt*, *opt*, *par*, *loop*, *strict* and *seq* operators. Provides a counterexample if an inconsistency is detected. Not integrated with a CASE/MBSE tool.

Shinkawa, Y. [20]: translates state machine diagrams and SDs into Colored Petri-nets (CPN), then verifies the correctness via two sets of formal rules (Method-based and State-based Consistency). Translation rules exist for all UML state machine diagram pseudostates and for *alt*, *opt*, *par*, *loop*, *critical*, *break* and *seq* SDs fragments. There are no case studies and the formal rule set is not implemented in an algorithm or tool.

Shinkawa, Y. [21]: extension of [20]. Proposes three perspectives on correctness (consistency, completeness and soundness) and methods to verify them by a mapping to CPNs. There are no case studies or examples and the formal rule set that maps diagrams to CPNs is not implemented in an algorithm or tool.

Tan, H. et al [22]: maps the parallel regions of a state machine diagram and parallel fragments of SDs to Labeled Petri-nets, proposing a formal rule set to verify consistency between the two constructs. No mention of other types of SDs fragments (other than *par*) or pseudostates. Not integrated into a CASE/MBSE tool.

Diethers, K. et al [23]: uses the UPPAAL model checker to detect inconsistencies. SCs and SDs are loaded into *Vooduu*, a plugin implemented in the Poseidon UML tool, which translates them into UPPAAL behavior and observer automata.

Zhao, X. et al [24]: translates state machine diagrams into *Split Automata*, a formalism proposed by the authors that enables inconsistency detection with the SPIN model checker and mitigates the state explosion problem associated with flattened automata. It is implemented into a MagicDraw plugin. Other features of UML, such as pseudostates or combined fragments in SDs, are not mentioned.

Wang, H. et al [25]: maps SCs into Finite State Processes and SDs into a trace of messages, then uses the LTSA tool to verify the consistency. Does not mention UML features such as pseudostates or fragments and is not integrated into a CASE/MBSE tool.

Lam, V. et al [26]: encodes both SDs and SCs diagrams in the π-calculus process algebra formalism. The Mobility Workbench (MWB) tool then checks if the created specifications are weakly open bisimilar or not. The example provided contains a SD with an *alt* fragment, but no composite states or pseudostates in SCs. Not implemented in a CASE/MBSE tool.

Gongzheng, L. et al [27]: describes SCs using XYZ/E then translates that description to Promela. SDs are mapped to Linear Temporal Logic (LTL) specifications. Both are then jointly analyzed with the SPIN model checker. Mapping rules are given for some of the SD fragments (*alt*, *opt*, *par*, *loop*, *neg*) and SCs may have hierarchical states. Provides no mapping rules for pseudostates and no integration with a CASE/MBSE tool.

Gherbi, A., Khendek, F. [28]: branches semantic consistency into three different definitions (Behavioral, Concurrency-related and Time Consistency) and proposes an algorithm that generates a *schedulability model* with SCs and SDs as input. The proposed implementation does not support sub-states (hierarchy), pseudostates or combined fragments. Not integrated to a CASE/MBSE tool.

Hammal, Y. [29]: detects time and semantic inconsistencies by mapping SCs to Petri-nets (using the semantic formalism of a previous work [30]) and then comparing its reachability graph to the SD specification model. Not integrated to a CASE/MBSE tool.

Yao, S., Shatz, S. [31]: proposes the Extended Colored Petri-Nets (ECPN) to verify if the *behavior sequences* produced by the SD ECPN are included in behavior sequences produced by the SC ECPN. A flattening strategy is also included to deal with hierarchical SCs, but pseudostates and guards support are proposed as future work. Not integrated to a CASE/MBSE tool.

Kawakami, Y. et al [32]: extension of previous work [33]. Models state machine diagrams and SDs as boolean formulae through the proposed formalism. Checks the consistency using the symbolic model checker SMV. Does not support UML SCs features, and supports only a subset of combined fragment operators. Not integrated to a CASE/MBSE tool.

Kaufmann, P. et al [34]: extension and revision of [35]. Translates the semantics of state machine diagrams and SDs to propositional formulas that are then fed into a SAT solver. The implementation was developed with the Eclipse Modeling Framework (EMF), which is also used to define Capella metamodels. Not all UML features are included in the formalism, though, like hierarchical states, combined fragments or evaluation of guards. Implemented in a standalone tool.

Xie, Y. et al [36]: definition of LTL specifications from *Collaboration-Contracts*, which in turn come from SDs. SCs are translated into the Promela language to be used in the SPIN model checker. Some of the UML state machine diagram features (*Initial*, *Final*, *Fork/Join* and composite states) have mapping rules to the Promela language. Similarly, combined fragments from UML SDs are not considered in the translation to Promela. Not integrated to a CASE/MBSE tool.

Xuandong, L. et al [37]: checks if a scenario mapped by a Message Sequence Chart (MSC) can (or can not, in the case of forbidden scenarios) be generated from a Petri-net run. No explicit support for fragments in the scenario description. Implemented in a Java tool with partial support for the Rational Rose tool (importing of UML diagrams).

Xuandong, L. et al [38]: similar to [37], checks consistency between bMSCs (basic MSCs) and Petri-nets with emphasis on timing consistency, that is, a scenario specification has timing constraints that a Timed Petri-net shall fulfill. The check is done by an algorithm, but a full implementation of the method is not included. Like [37], there is no explicit support for fragments.

Choi, J. et al [39]: exposes how state machine diagrams and SDs can be represented with MARTE (Modeling and Analysis of Real-Time and Embedded systems) annotations, and proposes the UMCA (UML/Marte timing Consistency Analyzer), a tool that can extract timing requirements from SDs. These requirements are then implemented in the UPPAAL and TIMES model checkers. There is no explicit

support for combined fragments. Partial integration with the Papyrus Modeling environment (importing of diagrams).

Straeten, R. et al [40]: the behavioural inconsistency detection is presented as part of a formal approach for model refinement using Description Logics (DLs). SD traces and Protocol State Machines (PSMs) are then translated into the $\mathcal{SHIQ}$ DL. After translation, it can be checked if a given message occurs at least once or always occurs. Only supports a subset of state machine pseudostates, and there is no explicit support for fragments. Integrated into the Poseidon UML tool.

Haga, S. et al [41]: checks inconsistencies between UML state machine diagrams and SDs through process algebra. A set of entities and the interaction transition graph (ITG) is defined for the Structure-Behavior Coalescence (SBC) process algebra. Formalisms to translate loops, sequence compositions, alternative compositions and parallel compositions from UML diagrams to SBC ITGs are given. According to a previous work from the same authors [42], SBC-SMDs do not contain components such as pseudostates due to the higher-level nature of the algebraic description. There are no explicit translation rules from these pseudostates to SBC-ITGs, however. The method is not integrated into a CASE/MBSE tool.

Knapp, A., Mossakowski, T. [43]: uses the *Institution* formalism to check if interactions such as between SCs and SDs are realizable. In their example, a state machine is constructed as a composite structure and this structure is, in turn, converted to a set of traces. If these set intersect with the SD traces, it is said the interaction is realizable and thus the diagrams are horizontally consistent. Not integrated into a CASE/MBSE tool.

Mithun, M., Jayaraman, S. [44]: uses SCs as "design-time" models and SDs as "run-time" test cases to check the consistency of Java programs, with a similar algorithm to [14]. The algorithm relies on the Java Interactive Visualization Environment (JIVE) to generate the run-time traces of applications.

IV. DISCUSSION

A great deal of work has been put in detecting semantical inconsistencies between SCs and SDs. Most of the included publications have employed formal approaches in their methods, and a majority of those have used model checkers. These methods can check the *static consistency* [8] of diagrams, which means the behavior can be proved to be consistent without executing it. The main advantage of these methods is that the exhaustive search of a specific trace (e.g. a SD) in a generator of traces (e.g. an SC) also leads to assertive temporal specifications. It would be beneficial, for example, to know if a forbidden use case is ever allowed to happen in a given component modeled in an SC without manually testing all possible behaviors, or if a small interaction between two components can happen in all generated traces. This assertiveness, however, comes at a cost: the mapping of UML constructs to a formal language adequate for model checkers is complex. For this reason, most studies consider only a subset of SD features or a subset of SC features. These subsets may not be adequate to represent Capella/Arcadia models without loss of expressiveness. And even if expressiveness is not lost, a seamless experience is not feasible if the user is prohibited from using all diagram features, like pseudostates and combined fragments. The mapping procedure itself is also cumbersome and computationally expensive due to the state explosion problem.

In contrast to static consistency, some methods check the dynamic or symbolic consistency [8]. A *dynamic consistency* check implies detecting inconsistencies between diagrams during the simulation or execution of the behavior. The benefit of this approach is that the generated trace (e.g. runtime output from an SC) can be compared to the specification trace (e.g. use case), provided there is semantic equivalence between the metamodels. The downside is that, without performing an exhaustive search, an event-firing strategy needs to be devised - like the random walk method proposed in [45] that randomly fires transitions until a fault detection is verified. Another strategy commonly used [14] [7] [44] is to fire SD messages as events in the execution of the state machines. Lastly, a user could also select which triggers to send to a simulation at a given timestamp. The generated trace from this execution could then be compared to predefined traces that shall or shall not be included in the desired behavior.

V. FINAL REMARKS

For future works, it is necessary to study the feasibility of employing model checkers along with fully-featured UML diagrams, i.e., including all pseudostates, composite states and combined fragments. As long as the formal language translation is hidden from the end user, exhaustive search processes could significantly aid the detection of inconsistencies.

Alternatively, a dynamic consistency method can also be implemented if SD traces could be generated from SCs modeled in Capella, and then compared to use cases previously modeled. This is part of an ongoing work to develop a plugin that brings execution capabilities for Capella statecharts.

REFERENCES

[1] C. Haskins, "A historical perspective of mbse with a view to the future," vol. 1, 2011.

[2] G. SPANOUDAKIS and A. ZISMAN, "Inconsistency management in software engineering: Survey and open research issues," pp. 329–380, 12 2001.

[3] M. Usman, A. Nadeem, T. hoon Kim, and E. suk Cho, "A survey of consistency checking techniques for uml models." IEEE, 12 2008, pp. 57–62.

[4] I. J. S. . Software and systems engineering, *[ISO/IEC/IEEE 42010:2011] Systems and software engineering — Architecture description*, 1st ed., ser. International Standard. ISO; IEC; IEEE, 2011, vol. ISO/IEC/IEEE 42010:2011.

[5] J.-L. Voirin, *Model-based system and architecture engineering with the arcadia method*. London, Kidlington, Oxford: ISTE Press ; Elsevier, 2018, oCLC: 1013462528.

[6] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 6 1987.

[7] F. J. Lucas, F. Molina, and A. Toval, "A systematic review of uml model consistency management," *Information and Software Technology*, vol. 51, pp. 1631–1645, 12 2009.

[8] F. ul Muram, H. Tran, and U. Zdun, "Systematic review of software behavioral model consistency checking," *ACM Computing Surveys*,

vol. 50, pp. 1–39, 3 2018.

[9] D. Allaki, M. Dahchour, and A. En-nouaary, "Managing inconsistencies in uml models: A systematic literature review," *Journal of Software*, vol. 12, pp. 454–471, 6 2017.

[10] M. J. Page, J. E. McKenzie, P. M. Bossuyt, I. Boutron, T. C. Hoffmann, C. D. Mulrow, L. Shamseer, J. M. Tetzlaff, E. A. Akl, S. E. Brennan, R. Chou, J. Glanville, J. M. Grimshaw, A. Hróbjartsson, M. M. Lalu, T. Li, E. W. Loder, E. Mayo-Wilson, S. McDonald, L. A. McGuinness, L. A. Stewart, J. Thomas, A. C. Tricco, V. A. Welch, P. Whiting, and D. Moher, "The prisma 2020 statement: An updated guideline for reporting systematic reviews," 3 2021.

[11] R. van de Schoot, J. de Bruin, R. Schram, P. Zahedi, J. de Boer, F. Weijdema, B. Kramer, M. Huijts, M. Hoogerwerf, G. Ferdinands, A. Harkema, J. Willemsen, Y. Ma, Q. Fang, S. Hindriks, L. Tummers, and D. L. Oberski, "An open source machine learning framework for efficient and transparent systematic reviews," *Nature Machine Intelligence*, vol. 3, pp. 125–133, 2 2021.

[12] F. van Ommen, P. Coenen, A. Malekzadeh, A. G. E. M. de Boer, M. A. Greidanus, and S. F. A. Duijts, "Interventions for work participation of unemployed or work-disabled cancer survivors: a systematic review," *Acta Oncologica*, pp. 1–12, 4 2023.

[13] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering." ACM, 5 2014, pp. 1–10.

[14] B. Litvak, S. Tyszberowicz, and A. Yehudai, "Behavioral consistency validation of uml diagrams." IEEE, 2003, pp. 118–125.

[15] T. Yokogawa, S. Amasaki, K. Okazaki, Y. Sato, K. Arimoto, and H. Miyazaki, "Consistency verification of uml diagrams based on process bisimulation." IEEE, 12 2013, pp. 126–127.

[16] S. Phuklang, T. Yokogawa, P. Leelaprute, and K. Arimoto, "Tool support for consistency verification of uml diagrams," pp. 606–609, 2017.

[17] A. Matsumoto, T. Yokogawa, S. Amasaki, H. Aman, and K. Arimoto, "Consistency verification of uml sequence diagrams modeling wireless sensor networks." IEEE, 7 2019, pp. 458–461.

[18] H. MIYAZAKI, T. YOKOGAWA, S. AMASAKI, K. ASADA, and Y. SATO, "Synthesis and refinement check of sequence diagrams," *IEICE Transactions on Information and Systems*, vol. E95.D, pp. 2193–2201, 2012.

[19] T. Yokogawa, A. Matsumoto, S. Amasaki, H. Aman, and K. Arimoto, "Synthesis and consistency verification of uml sequence diagrams with hierarchical structure," *Information Engineering Express*, vol. 6, p. 529, 2020.

[20] Y. Shinkawa, "Inter-model consistency between uml state machine and sequence models." vol. 2. SciTePress - Science and and Technology Publications, 01 2011, pp. 135–142.

[21] ——, "Evaluating behavioral correctness of a set of uml models," in *International Conference on Software Paradigm Trends*, vol. 2. SCITEPRESS, 2012, pp. 247–254.

[22] H. Tan, S. Yao, and J. Xu, "Behavioral consistency analysis of the uml parallel structures," pp. 287–292, 2011.

[23] K. Diethers and M. Huhn, "Vooduu: Verification of object-oriented designs using uppaal," pp. 139–143, 2004.

[24] X. Zhao, Q. Long, and Z. Qiu, "Model checking dynamic uml consistency," pp. 440–459, 2006.

[25] H. Wang, T. Feng, J. Zhang, and K. Zhang, "Consistency check between behaviour models." IEEE, pp. 470–473.

[26] V. S. W. Lam and J. Padget, "Consistency checking of sequence diagrams and statechart diagrams using the π-calculus," pp. 347–365, 2005.

[27] L. Gongzheng and Z. Guangquan, "An approach to check the consistency between the uml 2.0 dynamic diagrams." IEEE, 8 2010, pp. 1913–1917.

[28] A. Gherbi and F. Khendek, "Consistency of uml/spt models," pp. 203–224.

[29] Y. Hammal, "A formal methodology for semantics and time consistency checking of uml dynamic diagrams," pp. 78–85, 2009.

[30] ——, "A formal semantics of uml statecharts by means of timed petri nets," pp. 38–52, 2005.

[31] S. Yao and S. Shatz, "Consistency checking of uml dynamic models based on petri net techniques." IEEE, 11 2006, pp. 289–297.

[32] Y. Kawakami, T. Yokogawa, H. Miyazaki, S. Amasaki, Y. Sato, and M. Hayase, "Symbolic model checking of interactions in sequence diagrams with combined fragments by smv," *vol*, vol. 4, pp. 1692–1695, 2010.

[33] S. HARADA, T. YOKOGAWA, H. MIYAZAKI, S. Yoichiro, and M. HAYASE, "A tool support for verifying consistency between uml diagrams by smv," in *ITC-CSCC: International Technical Conference on Circuits Systems, Computers and Communications*, 2009, pp. 897–900.

[34] P. Kaufmann, M. Kronegger, A. Pfandler, M. Seidl, and M. Widl, "Intra- and interdiagram consistency checking of behavioral multiview models," *Computer Languages, Systems and Structures*, vol. 44, pp. 72–88, 12 2015.

[35] ——, "A sat-based debugging tool for state machines and sequence diagrams," pp. 21–40, 2014.

[36] Y. Xie, D. Du, J. Liu, and Z. Ding, "Towards the verification of services collaboration." IEEE, 2009, pp. 428–433.

[37] X. Li, J. Hu, L. Bu, J. Zhao, and G. Zheng, "Consistency checking of concurrent models for scenario-based specifications," pp. 298–312, 2005.

[38] L. Xuandong, W. Linzhang, Q. Xiaokang, L. Bin, Y. Jiesong, Z. Jianhua, and Z. Guoliang, "Runtime verification of java programs for scenario-based specifications," pp. 94–105, 2006.

[39] J. Choi, E. Jee, and D.-H. Bae, "Timing consistency checking for uml/marte behavioral models," *Software Quality Journal*, vol. 24, pp. 835–876, 9 2016.

[40] R. V. D. Straeten, V. Jonckers, and T. Mens, "A formal approach to model refactoring and model refinement," *Software and Systems Modeling*, vol. 6, pp. 139–162, 6 2007.

[41] S. W. Haga, W.-M. Ma, and W. S. Chao, "Inconsistency checking of uml sequence diagrams and state machines using the structure-behavior coalescence method." IEEE, 10 2022, pp. 1–6.

[42] ——, "Formalizing uml 2.0 state machines using a structure-behavior coalescence method." IEEE, 10 2022, pp. 174–179.

[43] A. Knapp and T. Mossakowski, "Uml interactions meet state machines-an institutional approach," in *7th Conference on Algebra and Coalgebra in Computer Science (CALCO 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[44] M. Mithun and S. Jayaraman, "Comparison of sequence diagram from execution against design-time state specification." IEEE, 9 2017, pp. 1387–1392.

[45] C. Schwarzl and B. Peischl, "Static- and dynamic consistency analysis of uml state chart models," pp. 151–165, 2010.