Techniques and Challenges for Trace Processing from a Model-Checking Perspective

Vincent Ribaud, Ciprian Teodorov, Zoé Drey, Luka Leroux, Philippe Dhaussy

Techniques and Challenges for Trace Processing from a Model-Checking Perspective

Vincent Ribaud, Ciprian Teodorov, Zoé Drey, Luka Le Roux and Philippe Dhaussy

***Abstract*—Despite the high-level of automation offered by model-checking techniques for proving that a system satisfies its specification, if one property is violated the designer is left with a counterexample trace to understand. In this paper, we overview ten families of techniques used to diagnose a system behavior relying on traces. However, whereas these techniques are highly effective and are largely used, they are either not yet available in the context of model-checking or they are not adapted to the particularities of this verification technique. To address this, we have identified three very challenging problems hindering the concurrent systems diagnosis process. This helped us to define a roadmap for future research directions in our team.**

***Index Terms* — execution trace, model-checking, diagnosis, verification.**

I. Introduction

As its name suggests, model-checking combines modeling and checking. A system-under-study (SUS) is represented through a formal model, for instance concurrent state-machines, and this model is used to exhaustively explore the SUS possible configurations. Then properties to be satisfied by the SUS have to be specified, and weaved with the SUS model. The weaving results in a Labeled Transition System (LTS) whose exploration permits to assert if a property is satisfied or not. If the property is violated the model-checker produces a counter-example trace.

The main advantage of this method is that it is completely automated. However, once the existence of a problem is proved, the designer is left with a counterexample trace that only exhibits the problem. From this point, the designer needs to find and apply the right corrective measures. Conceptually the path from the counterexample to the correction is straightforward: understand what caused the behavior exhibited by the counterexample and fix it. However, in reality a large number of factors render this "simple" task very challenging. Two of the most challenging, and easy to understand, factors amongst these are: *a)* the size of the counterexample (from a few execution steps to millions or even more) that quickly can exceed the human capacity of analysis; and b) the semantic gap between the formalisms used to express the system and the low-level counterexample trace.

Manuscript received December 8, 2014.

Vincent Ribaud is with Lab-STICC at Université de Bretagne Occidentale (corresponding author ; e-mail: ribaud@ univ-brest.fr).

Ciprian Teodorov, Zoé Drey, Luka Le Roux and Philippe Dhaussy are with Lab-STICC at ENSTA Bretagne (e-mail: firstname.lastname@ensta-bretagne.fr).

The field of trace-based diagnosis (Section II), provides a rich state-of-the art integrating a wide variety of concepts from many domains ranging from data processing, software engineering and verification to distributed systems and visualization.

In this study, we review ten families of techniques (Section III), which have been grouped in three clusters. The first one is focused on techniques striving to *reach an explanation* – find the cause. The second cluster regroups *knowledge-based techniques* aiming to ease the system understanding. And lastly, the third cluster overview hands-on techniques used for *observing and manipulating the SUS*. Whereas not exhaustive, the set of techniques presented here are covering techniques highly effective and largely used in industrial projects.

In the model-checking context, however, these techniques are either not available due to engineering constraints, or they do not adapt well to the constraints of this verification technique. Moreover, in some cases even the techniques that are currently integrated in model-checking toolkits do not always provide enough information (at the right abstraction level) to enable the developer to easily identify and understand the causes of failures.

In Section IV, we have overviewed three major challenges hindering the diagnostic process, which pose the basis for future research directions in this field. These challenges are:

1. The *semantic gap*, refers to the discrepancy between the formalisms used during the model-checking verification process.
2. *Multi-level trace interpretation* refers to the level of detail at which the traces are built and the perspective from which they are analyzed. In this case we note mainly the lack of tools adapted at crossing the boundaries.
3. *Inconsistency robustness* refers to the scalability of trace diagnosis in the context of present and future massively parallel computing systems, which might need radically new diagnosis approaches.

II. Diagnosis for Transition System

Generally, the diagnosis encompasses any activity that provides information about the SUS, including analysis, observation, proofs, testing, etc. We agree with the Merriam-Webster on-line dictionary that defines diagnosis as an "investigation or analysis of the cause or nature of a condition, situation, or problem." (http://www.merriam-webster.com). For the purpose of this study, we use a narrower vision of

diagnosis, in which symptoms are materialized by traces and processing traces leads to causes. Nevertheless, the techniques presented often transcend this narrow view addressing the diagnosis problem generally.

In this context we consider that the existence of the trace set is the starting point of the diagnosis, and we do not focus our attention on the techniques employed to produce such a set. Suffice to say that the set of traces can be either implicitly (executable system) or explicitly (counterexample) defined and that we have the possibility to add/remove elements to/from the set iteratively. However, the illustrative traces presented are mostly issued from reachability analysis and model-checking.

A. The Problem Space: Traces and Errors

Transition systems are used as models to describe the behavior of systems. They are basically directed graphs where nodes represent states, and edges model transitions, i.e., state changes. They are the basis of many program analysis techniques, and in particular of model checking: "*Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model* [1]." A property is formal if it is described in a precise and unambiguous manner, using a property specification language.

When a property is falsified, it can be due to different causes. There may be a modeling error, meaning that the model does not reflect the system. If diagnosis reveals no inconsistencies between the system and its model, there may be a design error or a property error. In the case of a design error, an execution trace will be presented as a counterexample showing that the property is falsified and the model has to be improved. It may be the case that upon studying the trace, the formal property does not reflect the informal requirement from which it stems and the property has to be improved. In all cases, all properties have to be re-checked for the model.

Several approaches exist to address the problem above and this paper is a non-exhaustive attempt to present popular techniques and their application for diagnosis using model-checking traces.

B. The Solution Space: Trace-Based Diagnosis Techniques

The solution space consists of any technique that addresses the diagnosis problem. We have identified a broad range of techniques that we have classified in 10 families as follows:

a) *Counterexample Processing*: groups the techniques that attempt SUS diagnosis through the use of negative witness traces (traces exhibiting some unwanted behavior).

b) *Trace Analysis*: groups the techniques that rely on the analysis of a set of traces, including specialized query languages.

c) *Data-mining*: focuses on trace analysis techniques that attempt to automatically uncover and classify hidden structures (patterns) present in large execution traces.

d) *Pattern Analysis*: groups the techniques focusing on the identification of previously specified conditions (patterns).

e) *Model-based Trace Diagnosis*: focuses on high-level trace reification and their interpretation through the use of model-based engineering tools and formalisms (e.g. UML).

f) *Ontology-driven Diagnosis*: groups research activities related to knowledge classification and retrieval in the context of system diagnosis.

g) *Model Transformation*: groups the techniques using rule-based transformations (e.g. rewriting) in the diagnosis process. Whereas not directly related to the diagnostic problem, these techniques play a central role in many diagnosis activities.

h) *Testing*: groups the techniques that rely on the supervised, scenario-directed system-execution for unraveling unwanted behaviors.

i) *Debugging*: groups the family of techniques that focus on an interactive diagnosis process, offering the diagnostician the tools to observe, control and modify the behavior of the system.

j) *Visualization*: groups all techniques enabling the visual representation of the artifacts the diagnostician encounters, from traces to highly specific diagnosis reports.

These techniques arose as answers to problems and if obviously some techniques address related problems, we did not succeed to organize techniques within a classification. However, rather than presenting the list techniques in alphabetic order, they are clustered accordingly a proximity of usages. The first cluster groups techniques that aim to *reach an explanation*, relying on the principle that, in order to explain something (like an error), one has to identify its cause. This cluster includes counterexamples processing, and trace analysis. But we might look for explanations that help to better understand the SUS and this will be a second cluster, which contains *knowledge-based techniques*, such as data-mining, pattern analysis, model-based diagnosis, and ontology-driven diagnosis. Finally, the third cluster contains techniques related to the *observation and manipulation of the SUS* and it includes: model transformation, testing, debugging, and visualization.

Besides these clusters and families of techniques, we acknowledge the existence of many more other techniques that were deliberately left out of this study. Such techniques include: *automated diagnosis*, an important topic in the artificial intelligence field, and *fault diagnosis*, a major concern in modern control theory and practice.

The following section briefly overviews a number of research results related to techniques considered.

III. A HIGGLEDY-PIGGLEDY SET OF TECHNIQUES

A. Counterexamples

A major advantage of model-checking is the production of

counterexamples as a proof that the system violates the specification. Counterexamples are represented as witness traces that describe the system behavior "responsible" for the property violation.

A counterexample trace provides a bunch of low-level information and indicates the symptom of an error in the SUS (or a problem with the specification of the SUS). Different approaches exist for relating the symptom to a possible cause and most approaches share the idea of finding variations of the counterexample either with the symptom reproduced or without the problem.

Groce and Wisser [2] use an automated method for finding multiple versions of an error (*negatives*) and similar executions that do not produce an error (*positives*). The negatives and positives are analyzed together to produce a succinct description of the error elements, including portions of source-code that distinguish failing and successful executions.

Ball, Naik and Rajamani [3] use the model checker as a subroutine to localize the error cause in an error-trace, stemming from correct traces; and generate multiple error traces having independent causes; and implemented this error analysis technique in the SLAM tool.

Sharygina and Peled [4] propose the notion of the neighborhood of a counterexample, consisting of a tree of execution paths, where the original error-trace is one of them; and suggest that an exploration of this region may be useful in understanding an error. In this case, while a testing tool aids the exploration, it does not offer automatic analysis.

Jin, Ravi and Somenzi [5] produce an enhanced error trace as an alternation of fated (forced) and free segments. The fated segments show unavoidable progress toward the error while the free segments show choices that, if avoided, may have prevented the error. Segmentation raises the questions of whether the fated segment should indeed be inevitable and whether the free segments are critical in causing the error.

B. Trace Analysis

Trace-based analysis is the analysis of programs based on their trace, that is, their execution history. History is the sequence of states (e.g., the value of the program's memory) in which a program is at each step of its execution, until it stops (hence the trace may be infinite). The idea of trace-based analysis originates from Cousot and Cousot who defined its theoretical foundations [6]. Later, Colby and Lee defined a framework named "trace-based analysis" as an alternative to the traditional "state-based" analysis [7]. Contributions of their approach are to provide a direct way to prove program equivalences and to enable compositional analysis of programs. Trace-based analysis is at the core of the model-checkers that use trace equivalence to verify properties [1]. Groce et al. also exploit traces of log information (variable values, strings denoting a program point) to increase the scalability of program analysis [8].

Whereas counterexample-processing works on a single or a few traces, trace analysis aims to gather, compare and analyze a large set of traces.

Querying traces can be applied in any domain where execution traces can be recorded. There are two types of query-based debugging, those that operates a posteriori on traces and those where the query is weaved with the source program and parameterize the trace recording.

Martin, Livshits, and Lam [9] propose PQL (Program Query Language), a language intended to query sequences of events associated with a set of related objects. They developed both static and dynamic techniques to find solutions to PQL queries. The static analyzer finds all potential matches conservatively using a context-sensitive, flow-insensitive, inclusion-based pointer alias analysis. Static results are also used to reduce the scope of dynamic analysis. The dynamic analyzer instruments the source program to catch all violations precisely as the program runs and to optionally perform user-specified actions.

Goldsmith, O'Callahan, and Aiken [10] propose Program Trace Query Language (PTQL), a language based on relational queries over program traces. Produced traces result from an automatically instrumentation by a tool PARTIQLE that monitor particular properties. Given a PTQL query and a Java program, PARTIQLE instruments the program to execute the query on-line.

Classical property specification languages, such as PSL[11], can also be considered in this family. In this case the verification tools using these languages can be seen as the query execution runtime, and the witnesses produced would be considered the query result.

C. Data-mining

Data mining strives to uncover recurring patterns or hidden structures through automated analysis of data samples. In the context of model-checking industrial problems, often the diagnostician is confronted with sheer amount of data (long counter-examples, huge LTS graphs), which by far exceeds the human capacity of analysis. In these cases, efficient data-mining can improves the analysis limits, as well as the accuracy of the results.

Approaches such as statistical debugging heavily rely on data mining mainly for fault localization. In such approaches the model in instrumented to produce statistical execution profiles, these are mined to pinpoint the location of suspicious code [12]. Parsa, Naree and Koopaei [13] present an approach that combines multiple execution traces into an execution graph, similar to the LTS, which is analyzed for detecting bug-relevant sub-graphs. The novelty of the approach resides in the use of weighted execution graphs integrating failing and passing executions, which enable finer analysis.

Many such approaches are present in the literature, relying on a wide variety of data mining techniques, like formal concept analysis, association rules, n-gram analysis, etc.

These techniques are largely used for software debugging and test and they can be coupled with model-checking verification approaches. However, in such a coupling the potential complementarity of the approaches is ignored, mainly the data-mining tools won’t benefit from the huge amount of information produced during a model-checking run.

Ge, Mantel and Crégut address this gap [14] for Time Petri net model-checking through a ranking algorithm inspired by a divergence metric from information theory and a numerical statistics algorithm computing an importance factor for the model transitions.

Moreover, while model-checking provides an automated solution for model verification its biggest limitations is known as state-space explosion [15] due to the exponential increase in number of states with respect to the number of interacting components. Whereas the focus of this paper is not on this problem, it is important to understand that due to this exponential increase in complexity any diagnosis tool used in the context of model-checking has to scale to huge amounts of data and ideally offer the user "big-data"-like functionalities, as introduces by Fan and Bifet [16].

D. Pattern Analysis

Pattern analysis groups a broad fan of pattern-based approaches. In the model-checking community, property pattern refers immediately to the work pioneered by Dwyer, Avrunin and Corbett [17] who proposed a pattern repository[1] and a pattern-based approach to the presentation, codification and reuse of property specifications for finite-state verification. Many researchers deepened this work. For instance, Smith, Avrunin, Clarke and Osterweil [18] developed PROPEL, intended to make the job of writing and understanding properties easier by providing templates that are represented using both "disciplined" natural language and finite state automata. Konrad and Cheng [19] extended Dwyer and al. work to real-time specification patterns expressed in terms of three commonly used real-time temporal logics, and a structured English grammar that includes support for real-time properties.

Another family of patterns stem from the work of Gamma, Helm, Johnson and Vlissides [20] (software design-patterns). In the distributed and real-time research community, analysis patterns, architectural patterns, design patterns have been extensively proposed. Pattern usage is shifted at a higher use during the analysis and design phases to guide developers in creating conceptual models of embedded systems, e.g., by providing structural and behavioral templates. Konrad, Cheng and Campbell [21] research explores how object-oriented modeling notations can be used to represent structural and behavioral information as part of commonly occurring object analysis patterns. This research also investigates how UML-based conceptual models of embedded systems can be automatically analyzed using the Spin model checker for adherence to properties specified in linear temporal logic. It has been stated that research on design patterns, domain-specific languages, and product-line architectures are particularly relevant to, and complementary with the Model-Driven Engineering field [22].

Model-checking of programs is a hot-topic in formal methods community. Static program analysis [23] is an entire discipline, but an alternative is runtime analysis, which is based on the idea of concluding properties of a program from program traces.

Examples are Java PathFinder 1 (JPF1) [24], an attempt to bridge the gap between Java and the PROMELA language of SPIN [25] or Java PathFinder (JPF2) [26] which model checks the java bytecode directly. Havelund [27] explains that runtime analysis is based on the idea of executing the program once, and observing the generated run to extract various kinds of information. This information can then be used to predict whether other different runs may violate some properties of interest, in addition to demonstrating whether the generated run violates such properties. Dynamic analysis algorithms detect the violated property based on the idea of recognizing the pattern related to the type of problem, e.g. a locking pattern for a deadlock detection property or a race pattern for a data race detection (a data race is the simultaneous access to an unprotected variable by several threads).

E. Model-based Diagnosis

Model-based traces are defined as trace behavioral models of a system during its execution. According to [28], "*an important feature of model-based traces is that they provide enough information to reason about the executions of the system and to reconstruct and replay an execution (symbolically or concretely), exactly at the abstraction level defined by its models.*" Hence, model-based trace analysis offers dynamic analysis features related to the investigation of the relationships between the execution traces of a system and its models, measuring how various model features materialize in a system run, and finding the differences between two or more system runs. Major challenges are the complexity and length of the models and execution traces and visualization attempt to address these challenges by creating a scalable and visually appealing solution [29]. This latter point is overviewed in section Visualization.

Maoz and Harel [29] propose an approach based on two inputs: a scenario-based behavioral model, described using live sequence charts [30] - an extension of classical message sequence charts (MSC), and an execution trace of the system. The approach implemented in a prototype called Tracer allows one to visualize, zoom in/out, and explore, the progress of the scenarios during execution. Advanced visualization features are provided, including multiple instances of the same diagram, time-based and event-normalized tracing, various synchronous statistics.

Mellor and Balcer [31] designed Executable UML to produce a comprehensive model of a system independent of its implementation. They define three fundamental projections on the specification. The first model uses a UML class diagram. The second model represents how the objects may have lifecycles (behaviors over time) that are abstracted as state machines, using a UML statechart diagrams. Each state machine has a set of procedures, and the third model expresses the fact that the behavior of the system is driven by objects responding to events with the execution of a procedure, thus establishing the new state. These executable UML models can be executed to ensure that they produce the desired behavior. Although static verification can be applied, dynamic

[1] http://patterns.projects.cis.ksu.edu/

verification is performed through the process of running actual test cases against the models, just as we would write and run software tests.

Mayerhofer, Langer and Kappel [32] introduce extensions of the standardized fUML virtual machine in terms of a dedicated trace-model, an event-model, and a command API. fUml is an OMG standard defining the operational semantics of a subset of UML and the conforming virtual machine. Extensions enable runtime analysis and runtime adaptations of executable models.

F. Ontology-driven Diagnosis

To be useful, the information used for diagnosis must be divided into classes of data. Each data object within a class shares a set of properties chosen to enhance human ability to relate one piece of data with another. Unfortunately, the terms property and class are used in computer science with very different meaning, and we need to precise what do they mean from an ontological point of view. Guarino and Welty [33] point out that we shall consider ontological properties as the meanings (or intensions) of expressions, which corresponds to unary predicates in first-order logic. "*Given a possible world, we can associate with each property a class (its extension), which is the set of entities that exhibit that property in that particular world. The members of this class will be called instances of the property. Classes are therefore sets of entities that share a property in common; they are the extensional counterpart of properties.* [33]" Hence, ontologists do not focus on building single or multiple inheritance hierarchies: a property can apply to more than one class and may apply to classes that are not directly related by an inheritance path.

Once these differences stated, how do ontologies assist trace-based diagnosis? Failing or successful traces are witnesses of a system execution and are analogous to symptoms in medical diagnosis. The classification of diseases and the tests used for its diagnosis are central to medical practice and medical science pioneered the use of ontologies in general and especially for diagnosis purposes. The Systematized Nomenclature of Medicine – Clinical Terms (SNOMED CT) is a terminological resource designed to support electronic applications in health and medicine (http://www.ihtsdo.org/snomed-ct). Campbell [34] provided the basic evolutionary design principles upon which SNOMED development is still based. Spackman and Renoso [35] state that, as SNOMED development has continued, these broad principles have been operationalized using three fundamental criteria, abbreviated as URU for understandability, reproducibility, and usefulness. *Understandability* makes reference to whether a concept or feature can be fully and unambiguously comprehended by users of the terminology. Understandability is tested by checking to see whether users find that the concept is relevant or not relevant to a given patient or situation. This leads to the need for *Reproducibility* that indicates whether multiple users apply the concept to the same situations. *Usefulness* refers to the level of helpfulness and appropriateness conveyed in a concept or feature.

Applying these criteria will help to avoid the major pitfall of ontology-based diagnosis: each group of researchers defines its own set of diagnosis concepts and properties, arguing the specificities of its method and tool. Hence an effort has to be made to join ontological design efforts and to promote international adoption and use of diagnosis ontologies for model execution.

A major technical difficulty of ontological diagnosis is that we want diagnose software models with multiple levels of abstraction. Moreover, depending on the point of view, we may reason about classes or individuals (e.g. an assumption about all processes or a particular process). According to [36], ontological metamodeling is used to model a complex domain model where the distinction between classes and individuals is not clear-cut. Jekjantuk, Pan and Qu [36] have applied three different ontology metamodeling approaches to addressing requirements that demand modeling of a complex domain model at multiple levels of abstraction and concluded for an extended class-based approach that provide more configurable support for ontological metamodeling.

G.Model Transformation

Model transformation is the mechanical manipulation of a model, which by changing the model shape is expected to enable achieving a prescribed goal (be it analysis, reduction, compilation). Model transformation encompasses a broad spectrum of techniques based on the idea of using simple guard/action rules, with the understanding that the action-block of a rule is executed when the guard-block evaluates to true on a model segment. For a deeper understanding of model transformation, which is out of scope in our context, the reader is directed to a survey of rule-based transformations [37] and a feature-based presentation [38] of the domain.

Model transformation should be seen as a transversal tool that is used ubiquitously in the context of diagnosis. Model transformation can be used for model formulation, trace simplification, reformulation and ultimately understanding.

There is a large body of literature addressing the problem of transforming high-level formalisms toward specialized verification and analysis frameworks; for instance, Jouault and al. [39] introduce such an approach for transforming a subset of UML to the Fiacre formal language to enable model-checking.

For trace simplification, Kengne et al. [40] use a rewriting based approach to reduce the traces obtained from the execution of multimedia application. In this context trace reduction is essential due to the large amount of data produced by such application.

In Lieverse, Wolf and Deprettere [41], the authors consider the problem of mapping high-level application to computing architecture resources using trace transformation. Their approach consist of an expansion step during which an application trace is refined to a lower abstraction level matching the architectural primitives and a linearization step during which an execution path is selected among the ones available. Whereas this approach is not directly linked to diagnosis, it is interesting to note that during the expansion step the authors actually perform an embedding of the application semantics into the lower level (more detailed)

architecture semantics. This embedding step (sometimes referred as compilation) possesses a large number of issues in terms of diagnosis due to the potential semantic gap between the application and architecture. This problem can be formulated as: “how can we preserve the link between the application-level and the architecture-level traces so that, if a problem is detected at runtime, a witness trace can be produced and presented using the application semantics”. Simply put, a C application designer wants to reason about his model using the C semantics and not the processor semantics provided by the execution platform.

The use of model transformation, more specifically rewriting, for runtime system verification is another important use case. Rosu and Havelund [42] present a rewriting-based approach for evaluating temporal logic formulae on finite execution traces online at runtime.

H.Testing

Testing is probably the most known technique for detecting problems during system execution. The underlying principle is simple: “check the validity of a property during the execution of the system in a predefined scenario”. However, testing in general is a highly challenging task, mainly due to the complexity of scenario specification. To address this problem, a family of techniques, named “directed model-checking” [43], try to exploit model-checking algorithms, not in the traditional exhaustive fashion, but in a goal directed way relying of different directed search strategies.

Another popular technique is bounded-model checking [44], which rely on traditional model-checking algorithms but again, much like for directed-model checking, the state-space search is not exhaustive but bounded by a predefined exploration depth.

Context-aware Verification [45] is another technique that tries to close the system under test with a formally defined environment covering all environmental possibilities. Whereas this technique is geared towards exhaustive state-space exploration, the use of a dedicated formalism for environment specification makes it well adapted for testing if either the environment cannot be completely captured or if the exhaustive exploration fails due to the state-space explosion problem [15].

Lindstrom, Petterson and Offutt [46] introduce a criteria-driven test-set generation. This approach enables the generation of an exhaustive test harness through the use of a criterion-guided reachability strategy.

Whereas the previous testing techniques rely on the implicit traces induced by the model semantics for analysis, approaches, such as Counterexample-Guided Abstraction Refinement [47], use testing as an internal routine for verifying the accuracy of the abstraction used during the analysis. In the case of the counterexample-guided abstraction refinement the system undergoes a series of abstraction, model-checking, test cycles through which safety and liveness properties can be verified.

I. Debugging

Debugging is the process of finding and resolving the program defects that cause unwanted behaviors (crashes, erroneous results). Debugging techniques can be classified into two broad classes, interactive (the "standard" ones as found in IDEs like Eclipse) and automated [48, 49].

Rather than providing means for exhaustive analysis, interactive debugging provides the tools for observing, controlling, understanding and altering the system execution.

Due to the wide variety of programming paradigms addressed, and the way execution traces are handled debugging techniques vary enormously. Stack-based debuggers are probably the best known and are well adapted for sequential stack-based computation. In concurrent and distributed contexts the debuggers rely mostly on logical time abstractions [50] and make a heavy use of checkpoints (also known as distributed snapshots [51]) for building a view of the system. By storing the execution traces some debugging techniques enable back-in-time navigation features, post-mortem query processing, trace-analysis and reduction facilities, and execution replay [52, 53].

Goldszmidt, Yemini, and Katz [54] introduce an integrated system for distributed programs. This approach relies on the database recording of global (system-wide) and local (process-only) events, which can then be queried using a temporal assertion mechanism.

To address the language gap between the system and the query language, Lencevicius, Hölzle, and Singh [55] propose a query-based debugger where query expressions are expressions in the underlying programming language, this way the diagnostician can focus its attention on the problem at hand, and not loose focus by using two different formalisms.

Moreover, in the context of this study, it is interesting to note that most interactive debuggers keep traceability links (e.g. source-code line embedded in the DWARF debugging format [56]) between the system formalism concepts and the execution platform counter-parts. These links help to bridge the semantic gap between the different paradigms.

J. Trace Visualization

The analysis of large model execution is almost impossible without tool support; visualization tools exploit the human ability to quickly understand complex visual patterns. A large number of visualization tools exist for studying execution traces, they exploit a wide range of diagram structures ranging from waveforms (Figure 1) to large graph visualizations (Figure 6). Most of these visualization tools are focused at the visualization of single execution traces, and are tightly integrated in the development environment [57].

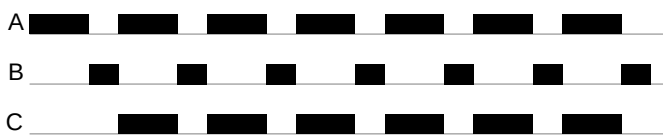

Fig. 1. Waveform visualization emphasizing a cyclic execution

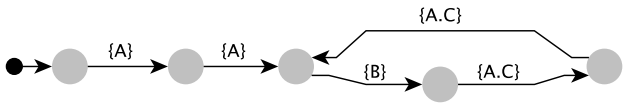

Fig. 2. Graph interpretation of a cyclic trace.

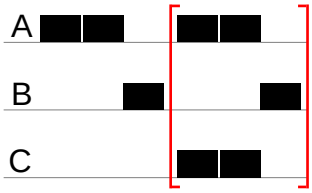

Fig. 3. Waveform visualization emphasizing a cyclic execution.

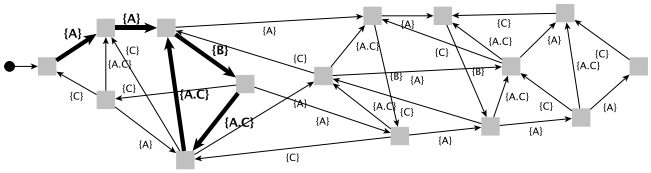

Fig. 4. State-space visualization emphasizing an execution cycle.

The Figure 1, Figure 2 and Figure 3 show three different interpretation of a cyclic trace (a typical counter-example trace for a liveness property), Figure 1 uses a traditional waveform diagram to represent the evolution of 3 Boolean variables (A, B, C). Figure 2 represents the trace as a graph (drawn with a classical hierarchical layout algorithm), while Figure 3 revisits the waveform representation explicitly showing the cyclic part between square brackets. Figure 4 places the previous trace in context, emphasizing its corresponding transitions in a graph visualization of the LTS representing the set of possible executions.

A very interesting visualization strategy was recently proposed by Issacs et al. [58], which suggest using a logical time abstraction for visualizing the causality relations in parallel application. Their approach equally promise scalable visualization, through the time discretization resulting from an event-based time interpretation, which induces a clustering of the local behaviors.

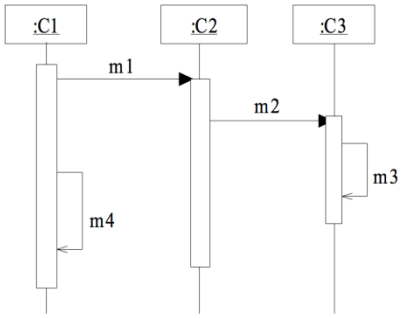

Fig. 5. State-space visualization emphasizing an execution cycle.

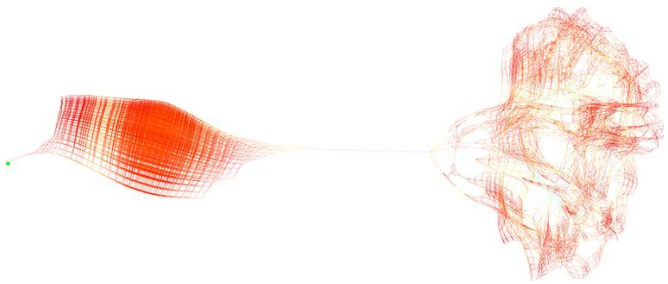

Fig. 6. Large state-space visualization, emphazing 2 operating modes.

However, in the context of model-checking most of the tools focus on counter-example visualization through sequence diagram-like interpretations (Figure 5), and rely on generic graph frameworks such as graphviz for the visualization of small state-spaces. However, for industrial systems, producing execution traces and LTSs with millions or billions of nodes most of these tools do not scale, or produce diagrams that are impossible to understand. Figure 6 shows, for example, the visualization of a small part of the state-space of a large system from the avionics domain. When corroborating this visualization with the model details the designer can identify the two clusters of states as being two execution modes of the system. In such cases it will be interesting to have a state-space exploration toolkit enabling: the navigation through such a graph, the inspection each state individually, the superposition of one of more execution or counter-example traces (like in Figure 5), the clustering of such a state-space according to user defined criteria, etc.

IV. Open Issues in Trace-based Diagnosis

The techniques presented in the last section provide the diagnostician with a broad set of powerful tools for understanding the behavior of the SUS. However, in some cases, such as model-checking, these techniques are either not available due to engineering constraints, or they do not adapt well to the constraints of this verification technique. Moreover, in some cases even the techniques that are currently integrated in model-checking toolkits do not always provide enough flexibility to enable the developer to easily identify and understand the causes of failures.

The remaining part of this section overview three important challenges that, from our perspective, should be addressed in order to ease the diagnosis process of concurrent systems.

A. Problem 1: The Semantic Gap

One of the main problems in the context of trace-based diagnosis is the semantic gap between the formalisms used to specify the model, the properties, the environment and the underlying transition system used for reasoning together with its conforming traces. While sometimes the model and its environment are expressed in a similar or an identical formalism, the properties are most of the time specified using temporal logics. There are some approaches trying to bridge the gap between the model and the properties by using automaton-based property specification languages (e.g. observers [45], temporal logic of action [59]) – at least for safety properties – or higher-level property pattern languages [17]. However, most of the time these approaches consider models expressed in model-checkers languages (such as Fiacre, TLA, Promela) well adapted for reasoning, but cumbersome and difficult to use in an industrial context. The emergence of Domain-Specific Languages (DSL) and Model-driven approaches gave rise to a plethora of literature on model-transformations targeting the generation of formal models from diverse modeling formalisms. Most of these work focus on the model-verification problem and, fail to offer the model designer the diagnostic toolbox really needed.

To better understand this problem, let us consider the case of a system specified with the statecharts formalism [60] using a tool like OBP[2] (any other tool like Tina or Spin will provide comparable features). Suppose for now that the system is modeled in the OBP modeling language (Fiacre). The OBP tool provides the user with features able to simulate, exhaustively execute the model, and verify a number of properties expressed either as predicates or observer automata. We could assume that in such an environment that she would be able to perform a number of diagnosis. However, there are still some important issues with the diagnosis environment:

- The system is specified with the statecharts formalism, and converted automatically to Fiacre. The designer should understand the details of this transformation in order to effectively use the tools and understand the result provided;
- Either the properties are described in a formalism very close to the specification (statecharts-like), in which case there is another transformation toward the predicates, and observers used by the toolkit; or the properties are described directly using predicates and observers in which case they are expressed with respect to the Fiacre code automatically generated.
- If a given property fails, the counter-example presented to the used is a path in the LTS, which however refers to the Fiacre model and not the expected statecharts model.

Therefore we can see that the heart of the problem is not the availability of diagnosis tools but the semantic gap between the modeling languages used by these tools, which are highly constrained (to ensure decidability – in the context of model-checking for example) and the domain-specific modeling languages used to specify the system.

Looking back to the techniques presented above, the semantic gap issue is addressed by several techniques as stated by research work surveyed in this paper: pattern analysis [18], [19], [21], [24], [26], model-based diagnosis [29], [31], [32], model-transformation [41], [42].

B. Problem 2: Multiple-level Trace Interpretation

The technological shift towards highly parallel computing architectures (multi-core, GPU, etc.) poses a large number of challenges to software development. Amongst these, traditional mature software diagnostic techniques focused on sequential stack-based execution cannot be applied in the context of highly concurrent applications, which exhibit emergent behavior due to the non-deterministic interleaving of execution threads. Whereas this issue is not new, being studied in the context of distributed system, it is important to note the paradigm shift needed when studying such systems. In this context the diagnostician steers its attention from the analysis of the execution stack to the analysis of execution traces. The execution traces record the history of the computation through a checkpoint mechanism, storing the state of each thread at given points during the execution. Amongst the tools focusing on the study of concurrent models, model-checking offers the possibility to analyze all the possible execution paths of the system to verify that a given system satisfy a predefined property.

In the context of highly concurrent applications, the use of execution traces for analysis solves the problem of capturing a global view of the emergent behavior system. While powerful, and well studied, this approach changes the sequential view of model execution towards the more natural view of concurrently-evolving computational objects, in which case the local-state evolution is abstracted away through atomic execution steps. While these atomic execution steps conceptually can be arbitrarily small, in the context of analyzing large systems the atomic steps tend to be as large as possible to remove the fine-grained interleavings (between functional components) to gain focus on the concurrent access to shared resources. Whereas such approaches render the analysis of large systems manageable, they also hide atomic sequential execution steps. These abstraction practices are largely employed especially in the case of asynchronously interacting components (which exhibit a large amount of interleavings). The diagnosis, in these cases, is limited at analyzing the trace configurations, without having a fine grain view of the functional steps. In terms of tool support, in some cases, such as SPIN [25], it is possible to use software debuggers to gain observability during the atomic execution steps. However, in this case, the debugger (gdb in occurrence) operates at an even lower level of abstraction (C language) than the verification language (Promela). In this case the diagnosis becomes almost impracticable, especially if a generational approach was used (as discussed in the previous section) to create the Promela verification model (there is a double semantic gap: DSL-Promela-C). It is worth mentioning however, that in this case part of the problem comes from the engineering complexity needed for developing an execution environment and debugging infrastructure, effort that was not invested in a research tools such as SPIN [25]. Nevertheless, debugging in the context of concurrent, parallel and/or distributed models is recognized as a very challenging problem, posing problems not only in terms of observation and witness generation (large traces) but also in terms of control during a supervised execution (what happens to the system while one process is stopped during debug?).

Obviously, the multiple-level trace interpretation issue is addressed by several techniques as shown by research work presented in this paper: counterexamples [3], trace analysis [7], [8], debugging [50], [51], [54], visualization [57], [58].

C. Problem 3: Inconsistency Robustness

The technological advances during the last decades made the parallel computing architectures ubiquitous today. This progress pushes for highly parallel computing paradigms composing millions if not billions of concurrently interacting components (ex. Asynchronous actors – Erlang) – mixing physical and computational entities (named cyber-physical systems). The diagnosis in this context becomes a highly challenging problem. As briefly stated in the last section, the problems of execution observation and of distributed control are even more exacerbated in such an environment.

However, the main problem, in this context, resides in the huge amount of data that should be gathered and processed during the analysis of such a system. Most of the traditional

[2] Available on http://www.obpcdl.org

automated techniques, such as model-checking, are expected to fail in these contexts principally due to the problem of state-space explosion [15]. To tame the diagnosis of such systems the community will need to focus on fundamentally different approaches, that are more appropriate than execution traces for the study of highly asynchronous systems. From our perspective, the main limitation of execution traces is that they impose a global view of the computation. The execution of the concurrent entities is check-pointed regularly (either periodically – in the case of back-in-time debugging – or systematically – in the case of model-checking) to build a coherent view of the system evolution.

Instead of focusing on this synchronous view of computation, a group of researchers, led by Carl Hewitt, started focusing on a fundamentally different view, which, under the name of "Inconsistency Robustness", strive to embrace the inconsistencies introduced by the emergent behavior of millions of components interacting in an unsupervised way [61]. As humans we are aware of the presence of massive inconsistencies in our understanding of the everyday life, however most of the computing systems we design operate under a coherence assumption. The challenges posed while diagnosing systems in the presence of inconsistencies dwarf the progress made by years of research in diagnosing systems based solely on execution traces.

We are not aware of research work about trace-based diagnosis that directly addresses the inconsistency robustness issue, but this issue appears in several research work presented in this paper: counterexamples [3], [4], trace analysis [7], data-mining [13], ontology-driven diagnosis [35], model-transformation [43], [44], [45]. Nevertheless, the diagnosis of complex inconsistent systems is at the core of other disciplines, such as medicine, biology, management, which can provide us, computer scientists, with valuable elements for managing the diagnosis of future cyber-physical systems.

V. Conclusion

A major advantage of model-checking is the production of a counterexample, a trace that provides a detailed witness of how the model violates the property. In this paper, we presented 10 families of techniques that may help the analysis of the counterexample and the refinement of the model. However, we identify at least three issues that should be addressed by an ideal diagnosis process.

The first issue, named "Semantic Gap", emphasizes the fact that traces are low-level in nature whereas the developer is reasoning at a much higher abstract level.

The second issue, named "Multi-level trace Interpretation", stems from the multiple viewpoints needed during the diagnosis process. The multiple viewpoints reflect the time and space distribution amongst the interacting components: each system component is evolving along different time scales and is topologically linked with a few other components.

The last issue, named "Inconsistency Robustness", is related to the tremendous progress toward massively parallel computing architectures, which calls for new diagnosis approaches.

It is interesting to note that the clustering of techniques presented in Section II, that is based on a proximity of usages, differs significantly from the grouping of techniques related to each issue presented in this section. It may indicate that each issue requires a transdisciplinarity of techniques and that requirements for each technique should be established in order to drive the research effort. Accordingly, further research on these issues will be our roadmap for the future.

References

[1] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. Cambridge, MA: The MIT Press, 2008.

[2] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *Proceedings of the 10th International Conference on Model Checking Software, ser. SPIN'03*. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 121–136.

[3] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: Localizing errors in counterexample traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '03*. New York, NY, USA: ACM, 2003, pp. 97–105.

[4] N. Sharygina and D. Peled, "A combined testing and verification approach for software reliability," in *FME 2001: Formal Methods for Increasing Software Productivity, ser. Lecture Notes in Computer Science, J. Oliveira and P. Zave, Eds.* Springer Berlin Heidelberg, 2001, vol. 2021, pp. 611–628

[5] H. Jin, K. Ravi, and F. Somenzi, "Fate and free will in error traces," in *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, ser. TACAS '02*. London, UK, UK: Springer-Verlag, 2002, pp. 445–459.

[6] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, California, USA, January 1977, 1977, pp. 238–252.

[7] C. Colby and P. Lee, "Trace-based program analysis," in *Conference Record of POPL'96: The 23rd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, USA, January 21-24, 1996, 1996, pp. 195–207.

[8] A. Groce and R. Joshi, "Exploiting traces in static program analysis: better model checking through printfs," *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 2, pp. 131–144, 2008.

[9] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: A program query language," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ser. OOPSLA '05*. New York, NY, USA: ACM, 2005, pp. 365–383.

[10] S. F. Goldsmith, R. O'Callahan, and A. Aiken, "Relational queries over program traces," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and*

Applications, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 385–402.

[11] I. Standards, "IEEE standard for property specification language (psl)," *IEEE Std 1850-2005*, pp. 0 1–143, 2005.

[12] L. Jiang and Z. Su, "Context-aware statistical debugging: From bug predictors to faulty control flow paths," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '07*. New York, NY, USA: ACM, 2007, pp. 184–193.

[13] S. Parsa, S. Naree, and N. Koopaei, "Software fault localization via mining execution graphs," in *Computational Science and Its Applications - ICCSA 2011, ser. Lecture Notes in Computer Science, B. Murgante, O. Gervasi, A. Iglesias, D. Taniar, and B. Apduhan, Eds*. Springer Berlin Heidelberg, 2011, vol. 6783, pp. 610–623.

[14] N. Ge, M. Pantel, and X. Crégut, "Automated failure analysis in model checking based on data mining," in *Model and Data Engineering, ser. Lecture Notes in Computer Science, Y. Ait Ameur, L. Bellatreche, and G. Papadopoulos, Eds*. Springer International Publishing, 2014, vol. 8748, pp. 13–28.

[15] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models, ser. Lecture Notes in Computer Science, W. Reisig and G. Rozenberg, Eds*. Springer Berlin Heidelberg, 1998, vol. 1491, pp. 429–528.

[16] W. Fan and A. Bifet, "Mining big data: Current status, and forecast to the future," *SIGKDD Explor. Newsl.*, vol. 14, no. 2, pp. 1–5, Apr. 2013.

[17] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International Conference on Software Engineering, ser. ICSE '99*. New York, NY, USA: ACM, 1999, pp. 411–420.

[18] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil, "Propel: An approach supporting property elucidation," in *Proceedings of the 24th International Conference on Software Engineering, ser. ICSE '02*. New York, NY, USA: ACM, 2002, pp. 11–21.

[19] S. Konrad and B. Cheng, "Facilitating the construction of specification pattern-based properties," in *Proceedings of the 13th IEEE International Conference on Requirements Engineering, 2005*, Aug 2005, pp. 329–338.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[21] S. Konrad, B. Cheng, and L. Campbell, "Object analysis patterns for embedded systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 970–992, Dec 2004.

[22] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Future of Software Engineering 2007, ser. FOSE '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–54.

[23] P. Cousot and R. Cousot, "Abstract interpretation and application to logic programs," *The Journal of Logic Programming*, vol. 13, no. 2ˆa3, pp. 103 – 179, 1992.

[24] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[25] G. Holzmann, "The model checker SPIN," *Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[26] W. Visser and P. Mehlitz, "Model checking programs with java pathfinder," in *Proceedings of the 12th International Conference on Model Checking Software, ser. SPIN '05*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 27–27.

[27] K. Havelund, "Using runtime analysis to guide model checking of java programs," in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. London, UK, UK: Springer-Verlag, 2000, pp. 245–264.

[28] S. Maoz, "Model-based traces," in *Models in Software Engineering, ser. Lecture Notes in Computer Science, M. Chaudron, Ed*. Springer Berlin Heidelberg, 2009, vol. 5421, pp. 109–119.

[29] S. Maoz and D. Harel, "On tracing reactive systems," *Software & Systems Modeling*, vol. 10, no. 4, pp. 447–468, 2011

[30] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *Formal Methods in System Design*, vol. 19, no. 1, pp. 45–80, 2001.

[31] S. J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architectures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[32] T. Mayerhofer, P. Langer, and G. Kappel, "A runtime model for fuml," in *Proceedings of the 7th Workshop on ModelsRun.Time, ser. MRT '12*. New York, NY, USA: ACM, 2012, pp. 53–58.

[33] N. Guarino and C. Welty, "An overview of ontoclean," in *Handbook on Ontologies, ser. International Handbooks on Information Systems, S. Staab and R. Studer, Eds*. Springer Berlin Heidelberg, 2009, pp. 201–220.

[34] K. E. Campbell, "Distributed development of a logic-based controlled medical terminology," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1997.

[35] K. A. Spackman and G. Reynoso, "Examining snomed from the perspective of formal ontological principles: Some preliminary analysis and observations," *Proc of the 1st Int Workshop on Formal Biomedical Knowledge Representation (KRMED 2004)*, pp. 72–80, 2004.

[36] N. Jekjantuk, J. Z. Pan, and Y. Qu, "Diagnosis of software models with multiple levels of abstraction using ontological metamodeling." in *COMPSAC*, 2011, pp. 239–244.

[37] E. Visser, "A survey of strategies in rule-based program transformationsystems," *J. Symb. Comput.*, vol. 40, no. 1, pp. 831–873, Jul. 2005.

[38] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Syst. J.*, vol. 45, no. 3, pp. 621–645, Jul. 2006.

[39] F. Jouault, C. Teodorov, J. Delatour, L. Le Roux, and P. Dhaussy, "Transformation de modèles UML vers Fiacre,

via les langages intermédiaires tUML et ABCD," *Génie logiciel*, vol. 109, Jun. 2014.

[40] C. Kamdem Kengne, L. C. Fopa, A. Termier, N. Ibrahim, M.-C. Rousset, T. Washio, and M. Santana, "Efficiently rewriting large multimedia application execution traces with few event sequences," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ser. KDD '13*. New York, NY, USA: ACM, 2013, pp. 1348–1356.

[41] P. Lieverse, P. v. d. Wolf, and E. Deprettere, "A trace transformation technique for communication refinement," in *Proceedings of the Ninth International Symposium on Hardware/Software Codesign, ser. CODES '01*. New York, NY, USA: ACM, 2001, pp. 134–139.

[42] G. Rosu and K. Havelund, "Rewriting-based techniques for runtime verification," *Automated Software Eng.*, vol. 12, no. 2, pp. 151–197, Apr. 2005.

[43] S. Edelkamp, V. Schuppan, D. Bosnacki, A. Wijs, A. Fehnker, and H. Aljazzar, "Survey on directed model checking," in *Model Checking and Artificial Intelligence, ser. Lecture Notes in Computer Science, D. Peled and M. Wooldridge, Eds*. Springer Berlin Heidelberg, 2009, vol. 5348, pp. 65–89.

[44] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[45] P. Dhaussy, F. Boniol, J.-C. Roger, and L. Leroux, "Improving model checking with context modelling," *Adv. Soft. Eng.*, vol. 2012, pp. 9, Jan. 2012.

[46] B. Lindström, P. Pettersson, and J. Offutt, "Generating tracesets for model-based testing," in *18th IEEE International Symposium on Software Reliability*, ISSRE 2007, Trollhättan, Sweden, 5-9 November 2007, 2007, pp. 171–180.

[47] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification, ser. Lecture Notes in Computer Science, E. Emerson and A. Sistla, Eds*. Springer Berlin Heidelberg, 2000, vol. 1855, pp. 154–169.

[48] M. Ducassé, "Coca: An automated debugger for C," in *Proceedings of the 1999 International Conference on Software Engineering*, ICSE '99, Los Angeles, CA, USA, May 16-22, 1999., 1999, pp. 504–513.

[49] H. Cleve and A. Zeller, "Locating causes of program failures," in *27th International Conference on Software Engineering*, ICSE 2005, 15-21 May 2005, St. Louis, Missouri, USA, 2005, pp. 342–351.

[50] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[51] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[52] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Comput. Surv.*, vol. 21, no. 4, pp. 593–622, Dec. 1989.

[53] G. Pothier, E. Tanter, and J. M. Piquer, "Scalable omniscient debugging," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada, 2007, pp. 535–552.

[54] G. S. Goldszmidt, S. Yemini, and S. Katz, "High-level language debugging for concurrent programs," *ACM Trans. Comput. Syst.*, vol. 8, no. 4, pp. 311–336, Nov. 1990.

[55] R. Lencevicius, U. Hölzle, and A. K. Singh, "Query-based debugging of object-oriented programs," in *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ser. OOPSLA '97*. New York, NY, USA: ACM, 1997, pp. 304–317.

[56] *Dwarf debugging information format version 4*, DWARF Debugging Information Format Committee, 2010.

[57] J. C. de Kergommeaux, B. Stein, and P. Bernard, "Pajé, an interactive visualization tool for tuning multi-threaded parallel applications," *Parallel Computing*, vol. 26, no. 10, pp. 1253 – 1274, 2000.

[58] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann, "Combing the Communication Hairball: Visualizing Parallel Execution Traces using Logical Time," in *IEEE Symposium on Information Visualization*, INFOVIS '14, Paris, France, November 2014.

[59] L. Lamport, "The temporal logic of actions," *ACM Trans. Program.Lang. Syst.*, vol. 16, no. 3, pp. 872–923, May 1994.

[60] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[61] C. Hewitt, "Inconsistency robustness in logic programs," *CoRR*, Ithaca, NY: Cornell University Library, 2014.

[62] A. Hamou-Lhadj and T. C. Lethbridge, "A survey of trace exploration tools and techniques," in *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research, ser. CASCON '04*. IBM Press, 2004, pp. 42–55.