A Systematic Review and Taxonomy of Runtime Invariance in Software Behaviour

Teemu Kanstrén
VTT Technical Research Centre of Finland
Oulu, Finland
teemu.kanstren@vtt.fi

Abstract— **Describing software runtime behaviour in terms of its invariant properties has gained increasing popularity and various tools and techniques to help in working with these invariants have been published. These typically take a specific view on the possible and supported properties. In many cases it is also useful to view these in a wider context to enable a deeper understanding of possible invariance and to provide more extensive support across different domains. This paper aims to identify different aspects of the runtime invariance based on a review of existing works, and to present these results in a taxonomy that positions the different aspects in relation to each other. The goal is to provide support for their use in practice and to help identify possible research directions. A systematic review has been performed to identify relevant works in the literature. From these, a set of relevant properties have been collected to form the taxonomy. The resulting taxonomy has been structured to describe the different properties of runtime invariance. One main axis gives an overview of usage domains. One describes process related properties that are further classified to specification and evaluation related properties. A third main axis describes properties of runtime invariance itself and is further classified to properties of measurements, patterns and scope. It is concluded that the taxonomy provides a representation of different properties of runtime invariance used in current works. It can be used as a basis for modelling and reasoning about software runtime behaviour generally or as a basis for specialization in different domains.**

Keywords-systematic review; software behaviour; runtime invariance; taxonomy

I. INTRODUCTION

This paper extends on previous work presented in [1]. Runtime invariance as discussed in this paper describes software behaviour in terms of its invariant properties as observed through dynamic analysis. Dynamic analysis uses as its basis information captured as observations from a (finite) set of program executions, such as test executions [2]. In line with these definitions, runtime invariance is defined here similar to [3] as a set of properties that hold at a certain point or points in a program execution. As a distinction from some uses of the term “invariant properties”, in this paper runtime invariance includes not only separate program points but also invariants over the overall behaviour of the observed system. That is, runtime invariance in this context refers to properties that are true for every observed execution. Recently the use of such invariants has become an increasingly popular technique in supporting different software engineering tasks (e.g., [4,5,6]).

Examples of runtime invariance include data-flow constraints (e.g., x always greater than 0 [3]), control-flow constraints (e.g., request always followed by a reply [7]), or their combinations (e.g., x is always greater than 0 when request is followed by a reply [8]). Runtime invariance can be specified manually as a model of expected behaviour for further processing with automated tools (e.g., [7]) or built (mined) based on observed behaviour (e.g., [3]). A model based on observed behaviour can also be referred to as describing likely invariance as it is based on observations made from a set of program executions, which typically do not cover the entire program behaviour state-space [3].

The idea of documenting and using invariants to reason about program behaviour at run-time can be seen to be as old as programming itself ([9,10]). Using invariants expressed in first-order logic to capture formal constrains on program behaviour was introduced as early as 1960's [9] by the pioneering work of Floyd [11] and Hoare [12].

Runtime invariance can be used in a variety of software engineering tasks and domains, such as helping in program comprehension [3], behaviour enforcement [13], test generation and oracle automation [6], or debugging [14]. Thus, when explicitly defined, a set of runtime invariants forms a basis for building automated support for many domains of software engineering.

There exist a number of tools to support the use of runtime invariance in different tasks (e.g., [3,6,15]). Many of these tools use a specific set of invariants for a specific domain. When applying runtime invariance in different domains, it is useful to also consider them in a wider context. When a set of invariants needs to be provided, either as manually defined input for evaluation by an automated processing tool, or as output (templates) from an automated specification mining tool, being able to generally reason about this invariance is needed for their effective use.

This paper describes a taxonomy of runtime invariance in software behaviour, describing different properties of these invariants. This is based on review of existing works on research and use of such invariants. The study is structured to describe how the invariants are specified and used, what kind of invariant patterns over software behaviour they capture, in which scope of behaviour they apply, and what information about the system behaviour is needed to be able to express and evaluate them.

The goal is to provide a systematic definition of the different properties of runtime invariance in software behaviour based on existing work, to facilitate their use in practice and to help form a basis for identifying future research directions.

This paper is structured as follows. Section II describes the overall approach taken to perform the survey and to create the taxonomy. Section III presents the taxonomy itself. Section IV provides two examples of applying the taxonomy. Finally, Section V provides discussion followed by the concluding remarks.

II. TAXONOMY BUILDING APPROACH

The taxonomy presented in this paper is based on performing a review of existing works in use and research on runtime invariance of runtime software behaviour. This review follows guidelines for performing systematic literature reviews (SLR) from [16]. SLR has been shown to be a robust and reliable research method in software engineering research [17]. In relation to the most common reasons for performing a SLR defined in [16], the intent here is to summarize existing works related to the use and research on the different properties of runtime invariance in software behaviour. Additionally, while not directly addressed by the resulting taxonomy, also two other most common reasons for a SLR defined in [16] can be observed as being supported by the provided taxonomy. Regarding these two, the taxonomy can be used as a framework to help position new research activities in the area, and to help identify gaps in current research.

A SLR can be defined in terms of the following features: a review protocol, a search strategy, selection criteria, and the definition of the information to be obtained from each included study [16]. Finally, data also needs to be synthesized to summarize the results [16]. The rest of this section discusses the approach taken in more detail relative to each of these features. The approach for the survey and taxonomy creation is also inspired by the taxonomy building approaches taken by Ducasse et al. [18] and Kagdi et al. [19], as well as the approaches for SLR taken by Cornelissen et al. [2] and Ali et al. [20].

A. *Development of a Review Protocol*

As noted before, the approach taken in this paper follows the guidelines for performing a SLR given in [16]. According to this, a review protocol should specify the methods that are used to perform the SLR. This includes defining the research questions, the search strategy, selection criteria, data extraction method, and synthesis approach [16]. Figure 1 illustrates the development process for the review protocol taken in the study presented in this paper. The first step is identifying the research questions that the survey is aiming to answer. The second step defines the search scope in terms of resources (journals, conferences, etc.) to be searched and the strategy for searching these resources. The selection criteria define what studies will be included in the SLR. A separate step of quality assessment is also possible at this point but in this paper this is embedded in the selection criteria as will be discussed in the following subsections. The data extraction strategy defines how the relevant information from each chosen study is extracted. Finally, the data needs to be synthesized to answer the research questions.

Piloting the different steps is also important in order to be able to identify any mistakes and problems in the procedures. For the study presented in this paper, a preliminary study was conducted and published as a conference paper [1]. For this pilot study, comments were requested from experts in the field and also received from the conference peer-review (identified in the acknowledgements). The feedback from these instances was incorporated into different phases of the review protocol, which was then applied to produce the extended version of the study presented in this paper.

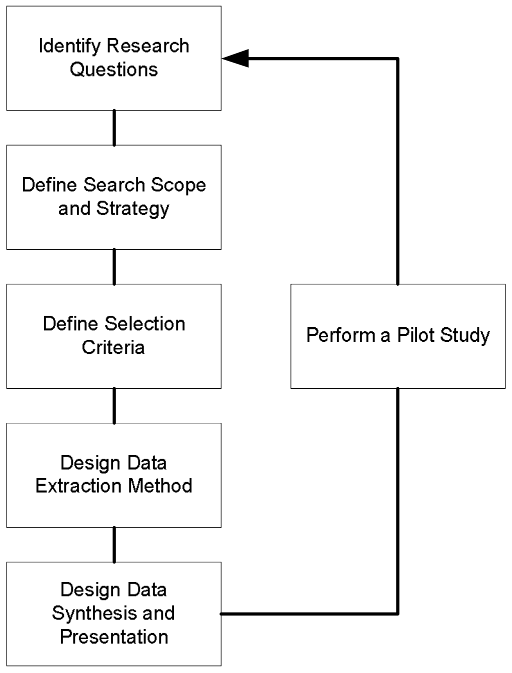
Identify Research Questions
Define Search Scope and Strategy
Define Selection Criteria
Design Data Extraction Method
Design Data Synthesis and Presentation
Perform a Pilot Study

Figure 1. Development process for the review protocol.

B. *Research Questions*

The research questions should support the goal of the study, which here has been stated as providing a systematic definition of the different properties of runtime invariance in software behaviour in order to facilitate their use in practice and to help in identifying future research directions. To support this goal, the following research questions are addressed:

RQ-1: What are the properties of the processes used in analyzing software behaviour in terms of its runtime invariance?

RQ-2: What are the properties used to describe runtime invariance in software behaviour?

The first question is related to how the invariants over software runtime behaviour are used, and the second one to how the invariants themselves are defined.

C. *Search Strategy*

For reasons similar to those presented in [2], the main approach for performing the search has been manually over the selected publication venues. These reasons include the lack of support in current software engineering libraries for the identification of relevant research and primary studies, and the lack of common keywords across different venues or any such standard in relation to runtime invariance.

Table 1. Venues for article selection.

Abbr.	Description
ASE	International Conference on Automated Software Engineering
CSMR	European Conference on Software Maintenance and Reengineering
FSE	European Software Engineering Conference / Symposium on the Foundations of Software Engineering
ICSE	International Conference on Software Engineering
ICSM	International Conference on Software Maintenance
ICST	International Conference on Software Testing
ISSTA	International Symposium on Software Testing and Analysis
WCRE	Working Conference on Reverse Engineering
IST	Information and Software Technology
JSME	Journal of Software Maintenance and Evolution
JSS	Journal of Systems and Software
STVR	Software Testing, Verification and Reliability
TOSEM	ACM Transactions of Software Engineering and Methodology
TSE	IEEE Transactions on Software Engineering

The search venues are described in Table 1. The first eight of these are conferences and the last five are journals. The timeframe for the initial paper selection is from January 2001 until October 2010. This timeframe is chosen in order to provide a reasonable scope for the survey similar to those of [2] and [20]. It also scopes the start of the survey around the publication of one of the seminal papers on analysis of runtime invariance by Ernst et al. [3]. The selection of venues is based on the selection of well-known conferences and journals in the area of general software engineering and runtime analysis, similar to those in [2] and [20]. Additionally, it was scoped by the results of the pilot study ([1]), which started by examining the related publications collected over time on the Daikon tool website [21] (the tool originally described in [3]) and by performing keyword searches over digital library databases (e.g., IEEE Xplore, ACM Digital Library). The search venues in this paper are a composition from these different inputs, focusing on those that were found to be most relevant in the pilot study.

Similar to [2], in addition to the initial article selection from the venues listed in Table 1, interesting references from the chosen papers in the initial selection were also checked and relevant ones included in the survey. Where the authors' expertise in the field allowed to identify additional relevant references (e.g., [22]), these were also included.

As the focus of the study is on runtime invariance from the dynamic analysis viewpoint, the selection of papers and venues is also focused on the domain of analysing software runtime behaviour via dynamic analysis. However, as this can be seen to share many relevant properties with domains such as formal specification in general, also relevant works in other domains as referenced from the main body of works have been included in the taxonomy. However, to provide a clear scope for the survey and to limit the scope of the study to a reasonable set, further exploration of the relations of the given taxonomy to other domains such as the formal methods community is left as a topic for future works.

D. Selection Criteria and Process

With regards to the research questions, two selection criteria for choosing the papers to be included were defined:

1. The selected papers directly discuss the use of invariants in relation to runtime software behaviour
2. The papers discuss modelling software behaviour in terms of properties that can be observed during runtime. When these models can be viewed in terms of invariance, they are considered relevant.

Table 2 lists the number of selected papers in relation to the selected venues. Due to the very large number of papers altogether (5817), it was not possible to read every paper fully. Instead, for each paper the title and abstract were first checked for relevance. If this provided no conclusion, the introduction and the conclusions were also reviewed for relevance. Finally, if needed, the full paper was read in order to define its suitability for inclusion. The total number of papers fully read is listed in the third column in Table 2 and is overall 348 papers.

The row titled "other" in Table 2 refers to papers that were included from venues other than the ones listed in Table 1. These come from the survey done in the pilot study, which included the papers listed on the Daikon website, digital library searches and from the reference checking in this extended study. Unfortunately the total number of such papers was not recorded during the pilot study and thus only the number of selected papers is given for these venues. The "selected" numbers in Table 2 reflect the references linked to the different axes of the taxonomy described in section III.

Table 2. Overview of study selection.

Venue	Papers	Read	Selected
ASE	406	49	12
CSMR	343	19	2
FSE	306	14	6
ICSE	481	25	20
ICSM	587	16	3
ICST	152	16	2
ISSTA	173	52	8
WCRE	277	17	3
IST	810	25	6
JSME	174	9	0
JSS	1270	29	3
STVR	101	11	2
TOSEM	132	8	3
TSE	605	58	7
Other	-	-	16
Total	5817	348	93

As mentioned, the process of SLR can also include a separate step of quality assessment after the paper selection step. Related to this, an exclusion criterion was also used. A paper was included only if it was observed as contributing something new to the building of the taxonomy. For example, when a paper presents some further research on specific properties of invariance already discussed in a previous paper, only the earlier paper is included. This is an approach similar to that taken by Ducasse et al. [18], and does not mean that other papers would not be interesting. It simply scopes the study from the research question perspective.

Thus the taxonomy does not aim to include all possible papers dealing with runtime invariance but rather those in the selected venues that contribute to the definition of a taxonomy observed as best capable of answering the research questions.

E. Data Extraction and Synthesis

In building the taxonomy, an initial version of the main axes and their classes was defined based on the seminal work of Ernst et al. [3] on analyzing the runtime invariance of software behaviour. This was then refined based on review of other works and how these contributed to evolving the taxonomy and its different properties. This resulted in a more advanced and more fully structured version of the taxonomy. At this point the pilot study was submitted for conference peer-review. As noted before, based on the received feedback a more systematic survey was conducted in terms of a SLR, which was then used to update the taxonomy similar to the way the initial approach is described above.

In building the taxonomy, the Protégé ontology editor tool was used to capture and describe the different aspects and classes of the taxonomy. An adapted version of Binder's "fishbone" diagram [23] is used in section III to describe the different aspects of the taxonomy, focusing on specific properties one at a time. For space reasons, the description of the different properties is kept at a general level and for specific (and possibly more formal) definitions the reader is referred to the original references given.

The descriptions of the fishbone show the high-level properties as **bold**, mid-level properties as ***italics bold***, and specific properties in *italics*. As noted before, given references are not intended to be all-inclusive but to give a set of examples. However, for clarity and space no "e.g." is repeated for all of the references.

III. Taxonomy of Runtime Invariance

This section presents the actual taxonomy of runtime invariance in software behaviour created based on the performed literature review. It is split into four subsections. The first subsection defines the main axes of the taxonomy and shows the overall picture. The second subsection describes different usage domains for such invariants. The third subsection describes the properties of the invariants themselves. Finally, the fourth subsection describes process-related properties for how such invariants are specified and evaluated. As noted before, the intent is to provide coverage of the different elements while including references observed as adding new elements to the taxonomy. Thus the intent is not to provide full coverage of all possible references for the described properties.

A. Main Axes

The generic process flow of using invariants in analysing software runtime behaviour, along with related properties for each step, is presented in Figure 2. This flow can be described as starting with *specification* of the *invariant information* that describes the software runtime behaviour of interest. Analysing the runtime behaviour requires capturing a set of observed measurements as a basis for the analysis, termed here as *measurement*. Depending on how the specification is done (automated mining vs. manual specification), it can also be interlinked with the measurement phase. Finally, in the *evaluation* phase the specified model of expected runtime invariance is compared against the observed model of actual runtime invariance.

The specification step is influenced by a set of *specification properties* that describe how the *invariant information* is formed. The invariant information describes the expected invariance and is formed as output from this step. This is then used as input for the measurement and evaluation steps. All steps in this process are related to the *usage domain* of the process. This means that the different phases of the process are impacted by the intended usage domain. The step of evaluation itself is described in this paper in terms of *evaluation properties*, which describes the general domain-independent properties of evaluation.

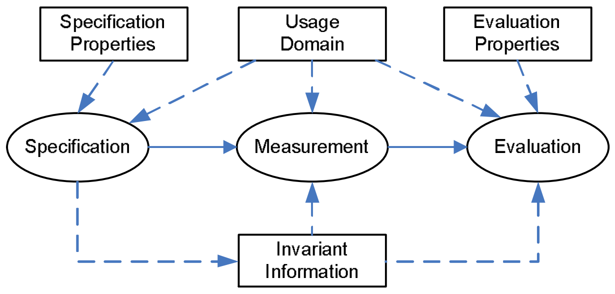

Figure 2. Flow of Elements.

The main axes of the taxonomy are divided into one axis describing the usage domains of runtime invariance, two for describing the process-related properties and three axes related to properties of runtime invariance itself. The invariance related axes embedded in the "invariant information" block in Figure 2 are *measurements*, *patterns* and *scope*.

Effectively making use of and reasoning about a concept requires thoroughly understanding it. Subsection III.B starts by describing a set of common *usage domains* for runtime invariance from the surveyed works. The properties of *invariant information* described in section III.C further describe the different elements of the runtime invariance itself. Finally, the properties of process described in section III.D provide more detailed insights into working with these invariants, related to their *specification* and *evaluation*.

B. Usage Domains

In order to use invariants as a basis to describe and analyse software runtime behaviour it is important to understand the context in which they can be applied. This includes both the targeted application domain (e.g., test automation or runtime adaptation) as well as the type of application (online or offline). The axis presented in this section aim to provide a basis for understanding what runtime invariance can be used for, how one might use them, and where one might be interested in their application. While this subsection synthesizes the surveyed related works, it should be noted that other application are also possible.

The usage properties refer to the context and type of application of the runtime invariance and are illustrated in Figure 3. This includes both the **usage domains** describing what the invariants are used for, and **usage types** describing if they are applied in an operational system or separately from it. Here the domains are split into five high-level usage domains, which can be observed from two viewpoints: online and offline use. These usage related properties can be observed to help answer questions such as *"How is runtime invariance applied and where?"*.

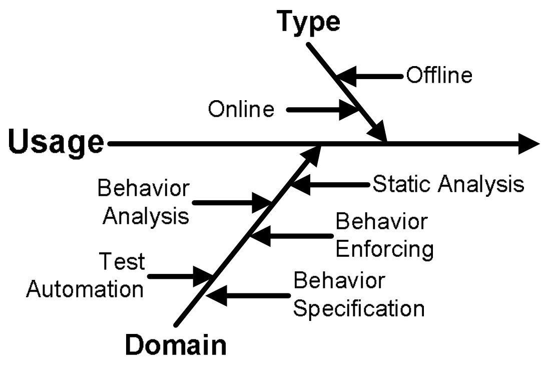

Figure 3. Usage Domains for Runtime Invariance.

Considering the **type** of use, in *offline* use, the invariants are applied separately from the execution of the analysed program. In this case, the observations (collected at runtime) about the runtime behaviour can be analysed external to the operational system. In *online* use the application of the invariants is linked to the executing program. In this case, the invariants are analysed while the system is operational and possibly the results are fed back to the operational system in some form.

Considering the five **usage domains** *behaviour enforcing* techniques guide the online operation of the observed system. *Static analysis* is focused on automated analysis of given static artifacts and thus mainly operates offline. Besides these two, the other domains can make equal use of both online and offline approaches. In the following, each of these usage domains is described in more detail.

Static analysis is not considered in this paper deeply as the focus is on runtime invariance in terms of dynamic analysis. However, some different usage relations can be identified. The invariants can be used as input for static analysis to check if they hold generally or only in specific cases ([24,25]). When information about the program structure is available, the correctness and accuracy of runtime invariants can also be checked and improved with combination of static analysis techniques such as symbolic execution ([26]). Invariants observed as useful and true in terms of runtime behaviour and dynamic analysis can be turned into more generic checks for static analysis ([14]). Although some specifications of invariants can be more suitable for dynamic and others for static analysis ([27]), many properties of static analysis can also be applied in the context of dynamic analysis and the other way around ([7,28,29]). Finally, as runtime invariants are defined in terms of some formalism, static analysis techniques can also be applied to check a chosen set of interesting properties in their specification, including their correctness and completeness in general and with regards to the observed runtime behaviour ([30]).

Behaviour specification is a basic concept for any application of runtime invariance, as the expected invariants need to be specified before they can be applied. Use cases for invariance in runtime behaviour specification include defining application programming interface constraints (e.g., `size() always >= 0` [25,31]), defining rules (constraints to be obeyed) for successful integration of a component with others ([32,33]), describing the valid (supported) input-space of a component ([32,34]), and defining error handling rules ([35,36]). A component can generally be anything from a method, a composition of classes, a service, or a complete software system.

Invariance also forms a basis for generic formal specification [7], which can be used to verify the actual behaviour against the specific expected behaviour [37]. This also includes constraints for executing a specific functionality [36]. The different properties related to specification of runtime invariance are described in more detail in section III.D.1). A core concept related to this is the requirement to be able to reason about the different properties of potential runtime invariance. A systematic definition for the properties of runtime invariance is needed in order to have a basis for reasoning about their composition and use. Such a definition is given by the taxonomy of invariant properties in section III.C. A specification can be produced either manually, or with the help of an automated specification mining tool.

Behaviour analysis of software runtime behaviour is a human-oriented process typically supported by automated tools. A set of runtime invariants is provided to the user as a basis for analyzing the system behaviour. This can be used for different types of tasks. In the software engineering domain, for example, failure cause location (debugging) can be supported by analyzing how the invariants change over time in an operational system and reporting any significant changes preceding an observed failure ([5,14,29]). This is possible as the runtime invariance of the program can be observed as changing over time. Similarly, debugging can be supported by comparing the invariants observed over both failing and non-failing program executions ([5,38]). Software evolution tasks can be supported by presenting any changes over given invariants when changes are made to a program to make the impacts of changes more explicit ([3,29,39]), such as changed interaction sequences and input-output transformation [39]. Another example in this domain is suggesting refactoring based on mined invariant specifications (e.g., to remove observed constant parameter)

[40], and by using given invariant specifications as contracts to define checks for properties that need to hold after refactoring [41]. Additionally, the invariants can support tasks such as program comprehension by providing a documentation that describes the software behaviour in terms of its important (invariant) behaviour ([3,4,32]). This can also take the form of asking queries to identify invariant sequences to find related sequences of behaviour ([42,43,44]).

When implementing automated software analysis features based on runtime invariance, different approaches for different domains can be taken. In security assurance observing a set of core invariants over specific variables, such as kernel data structures or session state variables, can be used to identify potential security attacks when the expected invariants are violated ([45,46,47]). In the same line, runtime invariance patterns over system interactions (such as library calls) can also be used to identify the origin of mutated code for purposes such as to protect against code theft [48] or to identify mutating malware [49]. Runtime monitoring in general can be implemented to check that the runtime behaviour conforms to the specified invariant specifications in all executions [50,51]. In case errors are observed, error correcting actions can be considered [51], which is a form of behaviour enforcing.

Behaviour enforcing mechanisms take as input the information from behaviour analysis and additionally take automated action to modify the behaviour based on differences in the actual observed runtime invariance vs the specified expected runtime invariance. Automatic adaptation mechanisms can use invariants to choose a new state for the software based on which specified invariants hold at different points in time [13]. Invariants can also be used to ensure that failure states specified in terms of invariants are avoided by modifying runtime behaviour that is observed to be outside the given set of invariants (the expected behaviour) to fit inside the expected invariants ([29,52,53,54,55]). For example, a specific case can be observed in disallowing saving of data or updating the state (of the user interface) when an invariant does not hold [56].

Test automation is basically a comparison of expected runtime behaviour to actual observed runtime behaviour. Defining a test case requires defining test inputs, expected outputs and their relations. Any automated test case basically requires defining the software runtime behaviour in terms of invariance for the automated test case to be able to produce any holding verdicts over the observed behaviour. Thus test automation is a fruitful application domain for runtime invariance.

A basic application is in defining the test oracle (the component that evaluates the test results) in terms of invariance. Such invariants typically define how a part of the system behavior is expected to work, in terms of properties such as determinism ([57,58]), data processing ([4,14,59]), message transitions ([6,60]), and their combinations ([22,61]). This is also related to the previously mentioned aspect of runtime monitoring for correctness in terms of using specified runtime invariance as a set of constantly running online test oracles ([29,35,50,62]). It is also related to the previously mentioned aspect of specifying how a component should work in relation to its use environment, in using runtime invariance specifications as a means to check how a new or updated component works in different environments ([31,39]).

In addition to test oracles, a test case requires producing valid input for the system under test. Here invariant models can be used to define valid data ranges, value relations and similar properties as a basis for a data model to be used to generate test data to cover these models ([61,63,64]). Another option is to generate test data to try to break previously defined invariants in order to further explore behaviour and to try to extend the model of invariance ([10,59,65,66]). Various approaches to generate test data from these models include search-based algorithms [64], random test generation [63], and generating test data to fulfill the different invariants defined [66].

The above mentioned uses of creating test coverage to cover or break invariants are in themselves also examples of using runtime invariance to reason about test coverage. A second aspect related to this is mutation analysis, which is used to change the SUT and to see how well a set of tests finds the mutations (with failing tests). Invariants can be used to also optimize SUT mutant coverage [10,67], but this can also be applied the other way around to evaluate the quality of the invariants themselves ([49,51]). In this case, the invariants are mutated and executions over these invariants are used to evaluate if the invariants are valid or if they catch useful changes in software behavior.

C. Properties of Invariance

This subsection describes the different aspects and properties related to describing runtime invariance itself. It starts with defining the properties of measurements for the actual runtime observations. Following this, it presents a set of different patterns of runtime invariance that are built from these observations. Finally, the different scopes of runtime invariance that allow for defining where these patterns of runtime invariance can be expected to hold are described.

1) Measurement Properties

As described in section III.A, any evaluation of runtime invariance requires first capturing the required information (observations) about the runtime behaviour. Similarly, as also noted in section III.A, this information can also be used as input in the specification phase. Thus, it can be said that any application of runtime invariance analysis requires also capturing a set of suitable information to describe this invariance. In this paper this information is referred to as measurements.

The basis for describing software behaviour in terms of its runtime invariance is the measurements used to observe this invariance. This in turn requires one to understand what kind of measurements can be made directly from the system, what kind of further measurements can be derived from these basic measurements, and how we may classify all these measurements. This information provides means to describe the system using the higher level invariant patterns presented in section III.C.2). It provides means to create more extensive patterns, and to evaluate the options available and needed to capture the required information as well as to understand what is needed to instrument the system to acquire these measurements. The axis presented in this section aims to support these goals by providing an overview of what types of measurements are used in existing works.

The properties of these measurements are illustrated in Figure 4. These properties can be observed as helping to answer questions such as "*What kind of basic measurements are used to observe runtime invariance?*", "*What other measures can we derive from these basic measures?*", and "*How can we characterize the different measurements?*".

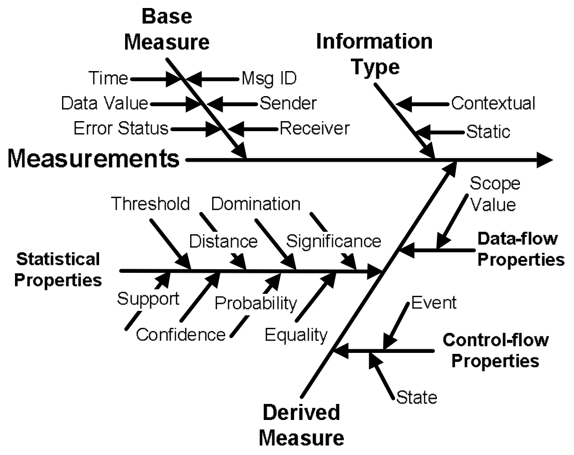

Figure 4. Measurement Properties.

The **information type** of the measurements can be classified into two different types of static and contextual information [28]. *Static information* in a dynamic runtime setting is information that is always the same for a given point of observation. For example, during a specific point of execution, a message passed can always be the same type of a message (e.g., method call named `publishData()`) and is thus static over different executions of this point. *Contextual information* describes dynamic information that changes over the executions of a single point depending on the context (e.g., test case or user session) of the observed information. For example, the time of observation, parameter values, and the thread of execution for a given message all can change over different executions of the same program point ([28,68]). The set of observations can also be grouped ("sliced") according to their contextual information, such as process (thread) id to produce a set of invariants over the scope represented by that slice ([15,28,68]). In this case, the scope (context) identifier becomes the basic measure (e.g., thread id [69] or a constant parameter value [28]).

The term **base measure** here refers to a type of measurement information that describes some basic value of runtime behaviour as it is observed. A basic value for any observation is the *time* when it occurred or was observed ([28,60,63,68]). For dataflow the base measures include variable *data values* and their basic data types such as Boolean values, integers, and character sequences (Strings) [3]. In the scope of object oriented programs the runtime type of an object can also be used as a base measure data value ([14,70]).

From the control-flow perspective, base measures are messages passed between different elements of the control-flow. Perhaps the most basic measure related to this is the *identifier of the message* passed, but also the *sender* and *receiver* objects ([71,72]). Examples of measurement targets include method invocations between components (such as classes or services) ([8,39,71,72]) or invocations on graphical user interface (GUI) operators ([6,61]).

A specific case of control-flow is error handling flows identified by an *error status*. Error scenarios can be classified to generic errors and application specific errors ([6,59]). Generic errors can be related to properties shared by different applications such as database access errors and user-interface (e.g., HTML or DOM tree for a web-application [6]) error codes. When represented in a uniform way (e.g., by programming language exception mechanisms [59]), these can be generally observed in the system behaviour (e.g., by an automated tool supporting a given domain). For example, all Java exceptions can be taken to describe a message that denotes erroneous behaviour being observed [59]. Application specific errors need to be described separately for each application in terms of application specific invariants. For example, one may expect a given error response to a message outside a given set of valid input [22].

A **derived measure** is something that is not directly observed in the system behaviour, but the value of which is rather derived from one or more base measures. To produce derived measures for ***data-flow***, the base measures for a system can be grouped based on invariant scopes [3]. For example, the values of variable x before and after a program point can be considered separately as variables `x1` and `x2`, to describe a pattern saying `x1>x2`. In this example, there are two derived measures `x1` and `x2`, both of which are scoped data values. The different scopes are discussed in section III.C.3).

Runtime **control-flows** are typically described in terms events and states ([46,61,68,73,74]). These are viewed here as derived measures for control-flow. From this viewpoint, an event can be described as an identifiable, instantaneous action in the observed software behaviour, such as passing a specific message or committing a transaction [69]. Similarly, a state can be described as values of properties that hold over time, such as over interactions between components. This information can be, for example, held in message parameters or inside components internal state variables [46]. A related property is branching, which defines how several different paths of events and states can be taken in the software behaviour. This can be described in terms of invariance of state when observing which paths are taken and which ones are not ([5,75]). In this case, the taking of a branch constitutes an invariant.

Statistical properties describe additional information for other base- or derived-measures. Support and confidence are two values commonly used together ([3,15,28]). *Support* defines the number of times a measure is observed in behaviour ([15,36]). *Confidence* can be used with the same definition [3] but also as a definition for how often another measure is observed in relation to support, meaning how often a precondition is followed by a post-condition ([15,28,76]).

Probability defines the threshold for a measure to be observed in a given scope, which can be used in different ways. A measure with low probability (support percentage) can be excluded from analysis to address anomalies ([3,13,15,69]). Different approaches are used for this depending on the target invariants, from low level *thresholds* (e.g., 1% or less [3]) to higher levels (e.g., 20% [15]). The probability can also refer to probabilities of a measurement value inside a range of allowed values [13,77]. Deviations from the expected values are typically given a probability which can then define the significance of the deviation ([13,14,46]). This threshold can be used for different purposes such as identifying probable failure causes [14], security attacks [46], possible state transitions [77], and to decide new states for automated adaptation [13].

Equality of objects can be defined in different ways. Besides expecting measures to be exactly equal, semantic equality can also be considered (e.g., two lists can be considered semantically equal while focusing on contained objects and ignoring their ordering [57]).

Distance defines the window inside which two events are observed and considered for evaluating a pattern of invariance. Thus it can be seen as related to the scope of invariance as described in section III.C.3). Events and correlations observed inside a shorter time-window are seen to represent more likely "true" invariance [58]. *Dominance* is a measure used to remove overlapping patterns where one includes the other as a sub-pattern ([78,79]).

Significance defines the importance of an invariant violation or of the measured variable. With regards to invariant violations, different approaches to significance can be taken where the latter observed violations are given higher priority as they are seen to be closer to a failure [14], or earlier violations as they are expected to have more impact on later behaviour [53]. When a variable is observed as having no correlation with other variables it can be considered as irrelevant for analysis purposes ([31,40]). The significance of observations and invariants can also be defined according to the number of observations ([13,58]).

2) *Patterns*

Having a set of measurements available in itself is not useful alone. They provide a basis for analysing the runtime invariance of the system but do not tell much about the invariance itself. A statistical derived measure can sometimes be useful in terms of considering basic runtime invariance in terms of single measurements at a single point of execution. However, more complex patterns describing relations between the measurements in different scopes (e.g., over time, described in more detail in section III.C.3)) are also needed. This includes considering their relation to the overall control-flow in terms of their occurrence in the given scope, their concurrent interleaving and sequential ordering. It also includes considering how the values interact in terms of data-flow and whether a specified invariance should be expected as normal or exceptional behavior of the system. Knowing these more advanced patterns enables describing and analysing the runtime invariance more effectively. The axis presented in this section aims to support these goals by providing an overview of what types of patterns are used in existing works.

Analysis of runtime invariance is basically about analysing patterns of invariance over the observed behaviour. The set of such patterns identified in this paper is shown in Figure 5. In relation to the different types of measurements described in section III.C.1), control-flow related patterns describe ordering of events or states in the observed system ([7,63]). Data-flow related patterns describe the data-flow of the observed software, such as what values a given variable takes during the software execution ([3,74]). These patterns can be observed as helping to answer questions such as "*How are the different measurements grouped?*", "*What are their relations to each other?*", and "*What type of behaviour do they describe?*".

Together these can be combined to represent the overall behaviour of the software in terms of the control-flow combined with the data-flow. A basic way to describe these combinations is in terms of conditional dependence; a control-flow event can only be followed by one of many (branches) depending on a given condition ([8,27,73,76,78,79]). A natural way to express these conditions is then in terms of invariants related to the data-flow in the context of that control-flow. For example, event `P1` can be followed by event `P2` when `x<0` or by `P3` when `x>=0` ([8,22,78]).

Overall, these are referred to here as behavioral invariants, where the constraints for a given control-flow pattern are defined in terms of its data-flow invariants. These can be described in terms of models at different abstraction levels as discussed in section III.D.1).

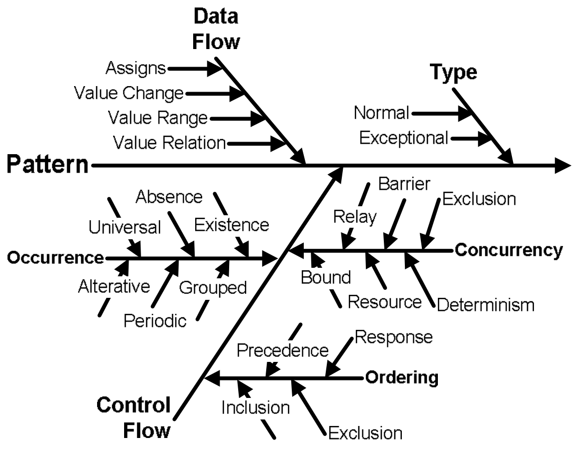

Figure 5. Patterns of Runtime Invariance.

Each pattern can further be related to describing different **type of behaviour**, which can be generally classified as *exceptional* (error) or *normal* (correct) behaviour of the observed system ([6,25]). Errors can be classified as either persistent, in being possible to identify them at different points, or as transient, in which case they are not observable after some set of events [80]. Exceptional states can also refer to more than just error situations, such as behavior deviation and need for adaptation [13]. A basic approach to detection of exceptional states is to look for deviations of a model describing the "correct" behavior, and to act on the observed exceptions with a predefined strategy ([6,13,61]).

Patterns related to runtime invariance of **control flow** can be defined as describing the sequential dependencies between a programs events and states ([7,76]). In the following discussion the term "event" is used to refer to both events and states. Here, the control-flow patterns are classified to three main categories related to ***concurrency***, ***occurrence*** and ***ordering***.

Concurrency in runtime analysis can be categorized as an event stream produced from several sources at a time [76]. *Determinism* defines that a set of operations always produces the same result, even if performed with different parallel schedules, as long as the start state is the same [57]. *Bounded* refers to allowing at most N threads to execute at one time in a single block [81]. This can also be referred to in terms or overlapping or sequential blocks [69]. *Exclusion* is a pattern only allowing one thread (from a set of threads) in a single block (or a set of blocks) [81], and can also be referred to as mutual exclusion ([61,69]). *Resource* is a pattern where a thread can execute only if enough resources from a resource pool shared by a group of threads are available [81]. A *barrier* pattern relates to two or more threads having to wait at the edge of a synchronization block to exit at the same time [81]. *Relay* is similar to barrier but allows exit of thread t1 from region r1 after thread t2 enters r1 [81]. Both relay and barrier can also be generalized to groups of threads (or event sources) arriving and leaving [81]. When these constraints of invariance are not met, problems can occur. For example, if two or more events affect the same state asynchronously, they typically will corrupt the overall application state if executed concurrently [82].

Occurrence related patterns describe properties related to observing an event or a state [7]. The *grouped* pattern describes two or more events always appearing together regardless of their ordering ([58,76,83]). For example, the methods `setHost()` and `setPort()` can be called to set up properties for a connection in any order but must always appear together [58]. The grouping can be seen in relation to context, coupling an event to its context in allowing only certain events in a certain context (state) [83]. *Absence* defines an expectation that the measure does not exist in the defined scope [7]. *Existence* denotes that the measure exists in a scope [7], and can be extended as bounded existence defining that the measure exists N time in a scope, where N denotes either exact, minimum or maximum number ([7,78]). *Universal* defines an expectation that the measure applies to the whole scope ([7,29,84]). *Periodicity* describes a measure repeating over a given cycle (scope) ([76,85]). The *alternative* pattern defines a set of events out of which one should be observed in a given scope ([86,87]).

Ordering describes the patterns of order between the different events ([7,88]). Precedence describes a specific event `P` always occurring before another specific event `Q` [7], also referred to as a precondition ([27,29]). Any number of events can also be observed between the two events ([15,58,63]), and it is possible to define the number of allowed events in between [58,84]. A specific case of this is chain precedence, which specifies that a sequence of events (`Q1,Q2,Q3,`…) is always preceded by another sequence of events (`P1,P2,P3,`…) [7]. This can also be related to an event enabling or disabling another on in the future [89].

The opposite of precedence is *response* which defines that event `P` is always followed by event `Q` ([27,29]). Similar to precedence, also here any number of events can be observed between the two events ([15,58,63]), and it is possible to define the number of allowed events in between ([58,84]). The scope for response can be defined in different terms such as inside a given time duration [85]. This is again a specific case of chain response, which defines that a sequence of events (`P1,P2,P3,`…) is always followed by another sequence of events (`Q1,Q2,Q3,`…) [7]. This can also be related to relation of objects to events, such as a created object always being passed as a parameter or an object that is related to event `A` also being related to event `B` [90].

Related to the precedence and response patterns, and also to the grouped occurrence pattern, two or more events can also be grouped together as an alternating sequence such as `ABABAB` ([76,78]). Further, it is also possible to define other more specialized cases such as events in the alternating sequence repeating multiple times, `AB*C`, where `B` is repeated 1-N times between A and C [78], or a cutoff in the end of the sequence (`ABABA`) [15].

Inclusion can be used to define an event always appearing inside another, where the dominating event must then be defined in terms of a larger scope ([63,85,88]). For example, an event A can hold while a specific state B holds [88], or when one is observed, another must also hold for a given scope [85]. *Exclusion* is the opposite of inclusion and defines that when a dominating event holds, a specific other event cannot hold. For example, when state `B` holds, event `A` is now allowed ([61,83]). Related to this and also the alternation pattern, it is also possible to define that when event `A` holds, event `B` does not, but when A no longer holds, `B` must hold [88]. Some basic examples are disallowing communication with a thread that is not started [91], or a model dialog disallowing communication with other dialogs (states) [61].

Patterns related to runtime invariance of **data flow** describe properties and relations over variable values during program execution ([3,74]). The *assigns* pattern defines that in a defined scope, values of specific variables are assigned to (modified) ([3,25]). This can also be described in terms of values that are not modified [3]. *Value change* is an evolutionary pattern that describes how a value changes over time in a given context ([14,27,92]). This pattern defines the expected scale of change for a variable in a given scope. For example, the expectation can be that change in value is always small (within a given threshold such as `change<5`) ([14,92]). Examples of specific case are a variable that is never set (`null` value) ([3,27]), and a value that is generally constant in the given scope [27].

A *value range* describes a variable always having a value inside a defined range in a given scope (e.g., [3,14,27,33,56]). Examples include value always being constant, one of a set of possible values (e.g., one of `1,2,4`) and a value between given boundaries (e.g., `1<x<4`) ([3,14,86]). Common constants such as zero or one can also be considered a specific case in itself ([3,14]). Optimizing for performance a subset can also be selected such as looking for positive (`x>0`) or negative values (`x<0`) [14]. Another example is that the contents of a character string are expected to be a human readable character ([27,64,93]). This can be further extended with a probability distribution describing how often each character is expected to be observed [46]. For user interfaces, it is also possible to define a possible set of UI widgets and their values ([56,61]).

A *value relation* pattern describes how one variable is related to another ([3,31]). These can be basic mathematical operations (e.g., `x<y` or `x=y+1`), or their combinations [3]. Relations can also be described in terms of the relation of one variable to several others, such as the relation of program output to its inputs [31], or as a variable always belonging to a larger set (member of another array variable) [3,27]. In the case of larger sets of values (e.g., arrays), the same relations can be described internally between the elements of the set [3]. Additionally, a set of specific relations can be considered such as one set (array) reversing another or matching a subset of a bigger set [3]. Additionally, a single value (e.g., a given variable or a constant) can be described to always be included in a given set [3]. The evolution of different variables can also be linked so that when the value of one changes, the other must change also [94].

3) *Scope of Invariance*

While the patterns described in section III.C.2) describe the basic important properties of runtime invariance, simply defining these patterns without defining when they are expected or observed to hold is not sufficient. To effectively describe the runtime invariance of a system, it is also important to be able to define when this invariance is expected to hold. This scope can be defined in terms of events or time specifying the constraints for when the pattern should hold. The axis presented in this section aims to support these goals by providing an overview of what types of scopes for different patterns are used in existing works.

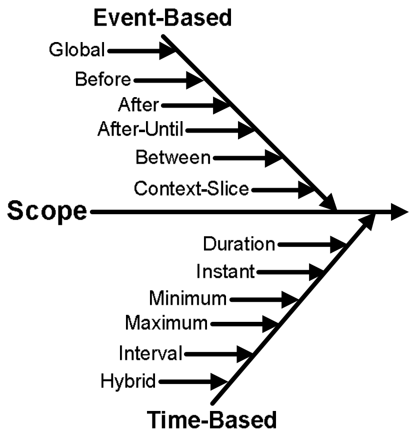

Figure 6. Scope of Invariance.

The scope of an invariant defines when and where a specific pattern of runtime invariance is expected to hold. This scope element of the taxonomy is shown in Figure 6. This part of the taxonomy can be observed to help answer questions such as "*Where and when do the patterns of runtime invariance apply?*" and "*What defines these scopes of runtime invariance?*".

In the following descriptions, the term event is used to refer to both control-flow events and states and data-flow measures. The scope is split into two main categories of time-based scope and event-based scope, which are discussed next.

An **event-based** scope defines the scope in terms of relations between events and observations. An invariant may define that it should hold *after* a given event [7]. Specific cases of this are defining the tail of an event set, where on N last events are considered [3], or when an event is never expected to occur after a given observation [88]. For example, using the tail scope, the relations between the last 2 observations can define how a value in a set increments [3]. More specifically than after an event generally, another event may be defined as the end condition in which case the invariant should hold after the observed start event until the observed end event (termed as the *after-until* scope) [7]. This can also be defined as the state holding true until a specific event is observed or forever if the end condition is never met [84]. A similar scope is *between*, which defines two events in between which the invariant pattern should hold [7]. However, the difference is that this holds only once both the start and end events have been observed, and after-until holds from the first observation of the start event [7]. Specific examples of these are the start and end of a method invocation on a component ([3,46]).

As opposed to the after scope, an invariant pattern can also be defined to hold only *before* a given event is observed [7]. Similar to the tail for the after scope, here the first N observations of a set can be considered with the term of head [3]. A *global* invariant pattern should hold for all observed behaviour during the program execution [7]. The scope can also be defined in combination with a specific slice of the program behaviour, such as a thread ([28,68]) or a specific web application session [46]. In this case the scope becomes a combination of the *context slice* and one of the other scope definitions discussed above.

A **time-based** scope defines the scope in terms of relation of the observations to the passage of time. Different aspects of the *duration* of time can be used to define the scope in terms of time. The basic form can be defining the start time and length [63], or defining an event as *instant* without specific timing constraints [69]. The duration can also be specified as an *interval* with a *minimum* or *maximum* time [88]. It is also possible to define a *minimum* or a *maximum* time (duration) for a pattern to hold without specifying the other bound [85]. Additionally, the time duration can be combined with an event-based scope to produce a *hybrid* scope. For example, an invariant pattern can be defined to hold after 10 time units (according to choice of time unit) until a specific event is observed [88].

D. Process Properties

In addition to the properties of invariance itself, it is useful to consider the properties of the process used to work with these invariants. The properties of the process in this section are described in terms of two main axes as noted in section III.A. These are the properties related to specification and evaluation of runtime invariance. The third step of measurement as described in section III.A is most closely related to the invariant information described in section III.C and thus not covered in this section.

1) Specification Properties

In addition to understanding how and where the runtime invariants are applied, it is also important to understand how runtime invariance can be expressed and where the information to describe a system in terms of this runtime invariance comes from. This helps to choose effective means both to specify the expected runtime invariance for a system, and to express it in suitable and effective terms for analysis. The axis presented in this section aims to support these goals. This provides a basis for specifying invariants using the detailed properties of runtime invariance presented in section III.C.

The different aspects related to the specification of runtime invariance are illustrated in Figure 7. The **expression** axis describes the different aspects relevant to how the invariants are expressed in specifications, including the expression ***language*** and the ***abstraction level*** of the expression. The specification ***source*** describes the different sources of information for the specification. The ***method*** of specification defines how the specification is created. These different properties can be observed to help answer questions such as "*Where does the information to define runtime invariance come from?*", "*How is the runtime invariance defined and reasoned about?*", and "*How is runtime invariance expressed?*".

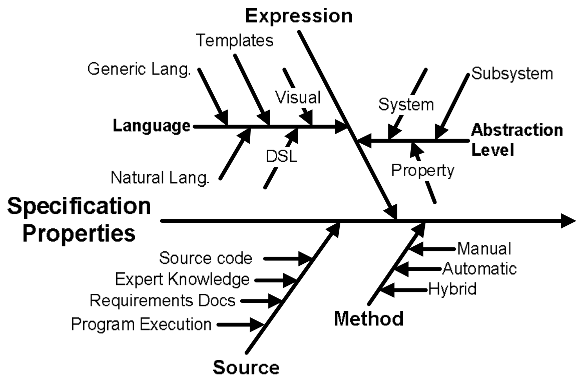

Figure 7. Specification Properties.

Different types of **methods** can be used to obtain the information for specifying the invariants. One is to fully (*automatically*) reverse engineer these from observing program behaviour ([3,38,80]), commonly referred to as specification mining. The opposite of this is describing the invariants *manually* based on specifications or expert knowledge ([6,29,80]). Finally, a *hybrid* approach can be taken, where an initial specification is first generated with tools similar to the automated approach, and this reverse-engineered information is manually augmented with information from specifications and as feedback from continuously evaluating this preliminary model ([22,37,42,71]).

As noted above, the method of specification is closely related to the **source** of the information used for the specification. Natural language documents such as *requirements specifications* can be manually analysed to find a set of relevant invariants [22]. *Expert knowledge* about the system behaviour can also be used to specify runtime invariance at different abstraction levels, for example, specified by domain experts as describing high-level system behaviour [56], or as lower level invariant properties specified by developers with detailed knowledge about the implementation [29].

Source code and program execution are two sources of information most suited for automated analysis. *Source code* is a static artefact, but where available can also be used as an additional input for dynamic analysis such as providing interface definitions ([22,44,95]), or to provide additional information for assisting in dynamic analysis [26]. *Program execution* is observed in terms of dynamic analysis to capture how the observed system behaves in a given context such as a test case [2].

From the different sources of information, one needs to capture a set of invariants covering the relevant properties of dynamic behaviour in the software. Experiments have shown that combining both manual and automated sources of information gives the best results, where both provide useful invariants not identified by the other approach [96].

Expression needs to define the invariance using a suitable expression language, and at the chosen abstraction level. The ***abstraction level*** of the specification can be described as defining the overall execution of the system or focusing on specific parts of execution ([88,97]). This definition at different levels is also described in terms of different types of ***languages***, such as automata for overall behaviour and assertion style specifications for specific properties ([37,98]). A relevant concept for definition of runtime invariance for a component is the hierarchical relations of the different components, where a subtype can be viewed to also inherit the invariant definitions of its supertype ([29,34]).

In terms of the ***abstraction level***, different focuses for partitioning the modeling can be taken. For example, models can be classified at the implementation, design, and domain level [98]. This can be translated to the internal implementation of components (e.g., embedded checks for specific *properties* [27,29]), the external interfaces of components as *subsystems* ([35,50]), or the overall *system* (domain) behaviour ([8,61]). Different types of approaches can be taken that combine different viewpoints from these. For example, some approaches define small-scale state-machines for specific parts of execution ([60,62,99]), which are then combined to form larger wholes to be checked, with the expectation that the larger whole needs to be covered ([18,60,78,99]).

A benefit of a small-scale specification can be seen in making it easier to produce a suitable formal specification for a specific property ([51,57]). Small (more specific) invariant definitions are seen as more suitable for specific low-level checks, but for system-level checks they can become too complicated and interleaved ([51,99]). In this sense, a different type of an approach such as specifying of larger-scale state-machines is seen as more appropriate ([51,99]). To reduce the complexity of managing specific checks some recommend defining fewer checks, and focusing on effective specifications that embed the important elements of behaviour ([27,51]). In this way, the larger specification can also be composed of several smaller ones [99].

For a higher-level specification a common model is different forms of a state-machine ([8,61,89,93]). Guards can be defined in terms of invariance over data values ([8,33]). Similarly, the possible transitions can be viewed as invariants in possible actions in a specific state ([8,61,89,93]). Many approaches in analyzing the state-machines are similar to how low-level invariants are combined to form state-machines. In this case, high-level state-machines are decomposed into smaller invariants for various analysis purposes, such as using expected transitions of chosen length as test oracles ([89,93]).

To support the different types of specification methods, the ***expression language*** needs to be both formal to allow for automated tools to effectively process them but also understandable to a human user in order to also support their manual review and analysis where needed. *Domain specific languages (DSL)* can be used to describe the invariants specifically for a chosen domain, such as in form of test oracles for web-applications ([6,56,98]). Other application domains include analysis support for refactoring [41], synchronization [81], determinism [57], general temporal properties [88], and test definitions in terms of runtime invariance [100]. Besides direct support, specific domains from more generic models can also be supported through model transformation (e.g., to monitoring code for runtime correctness [83]). Many of these languages are specifically designed to support modeling of the chosen set of properties for runtime invariance ([41,56,57,60,63,81,88,100]).

Besides these specific expression languages, also *generic languages* can be used. For example, the object constraint language from the unified modeling language can be used to express invariants ([101,102]), or live sequence charts to express invariance in event ordering [43]. Logical expressions can be used as a basis for defining invariance in themselves ([42,84,85]), or as part of the specific languages [41]. Regular expressions are generic expressions intended to express patterns over strings, and thus form a natural basis for expressing also patterns of runtime invariance. In this case, the event stream is expressed as a string of events, and regular expressions are used to express patterns over these events ([58,100]). Perhaps the most expressive means to define invariant properties is in terms of programming languages, which allows using their full expressiveness to define the invariance. This can take different forms, such as interleaving with the implementation code to be compiled with the program itself ([25,29]), writing them separately as a basis for a separate monitoring program [35], or as "model programs" for how parts are expected to behave allowing to execute the invariant definition separately with or with the implementation to perform different analysis and checks [50].

A common approach to support the definition of invariants is in using *pattern templates*. This is especially true in the specification mining approach, where a set of templates are defined and reflected against the actual observed behaviour to report observed invariants matching the template definitions ([3,14,58]). Templates are also used in manual specification ([41,103]). In both cases, the templates describe a predefined set of patterns, which are then parameterized according to the expected or observed runtime behaviour. The set of patterns of runtime invariance is described in the section III.C.2) of this paper.

As mentioned previously, an important aspect of specifying runtime invariance is that the specification should support both manual analysis and processing by automated tools in relation to the methods of working with these invariants. The languages described above are mainly focused on effective description from the automated processing perspective. One approach to address this is to produce specifications in a programming language when targeting developers in order to provide a familiar language to reason in [37]. Along with the different language transformations discussed before (in relation to [83]), it is also possible to provide transformations into *natural language* to support easier comprehension of the specifications (e.g., into structured English [85]). A related specification approach is also that of grammar-based specification, which aims to describe the invariance in terms of sentences of events [99].

Formal textual specification of complex behaviour has also been shown to be very hard for humans [103]. To address this, besides using textual languages and transformation between them, *visual* representations can also be seen as a more natural way of expressing invariance for humans ([56,103]). In this case, the transformation is done from the visual representation to a more machine processable form.

2) *Evaluation Properties*

In relation to this overall process flow described in section III.A, the overall process of runtime invariance evaluation is also related to the step of measurement and typically consists of three distinct steps: Collecting information (a set of events) from the runtime execution of the program, building a model of runtime invariance based on these observations, and comparing this model against the specified reference model of runtime invariance ([88,93,104]).

Specific aspects to consider in evaluating the runtime invariance include how extensive evaluation is done in terms of depth and frequency, the cost-effectiveness of these choices, and how the evaluation check is performed. Effective evaluation requires considering these different aspects together when building the evaluation for the case at hand. The axis presented in this section aims to support these goals by providing an overview of the different properties related to evaluation of runtime invariance in existing works.

The evaluation properties are illustrated in Figure 8. This is described in terms of three main properties. The **depth** of evaluation defines how extensively evaluation is performed. **Triggers** are events or states that trigger the evaluation of a specific runtime invariant pattern. Finally an evaluation function needs to be defined to **check** each defined invariant property. These properties can be observed to help answer questions such as "*How is runtime invariance evaluated?*", "*How extensively is it evaluated?*", and "*What is the impact of different properties of evaluation?*".

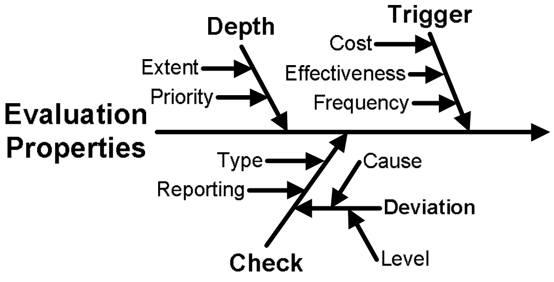

Figure 8. Evaluation Properties.

The **depth of evaluation** defines how extensively evaluation is performed. One approach is to define a *priority* to each defined check and to choose when an evaluation is performed to evaluate only checks of a given priority or higher [27]. Different properties for different types of components can be evaluated at different levels, such as checking a library while in development thoroughly and after release, focusing in more detail how the clients interact with it [29]. The evaluation level can also be defined in terms of evaluated data, such as if input is given, if it is of correct type and if it meets the domain-specific checks defined for it [56]. The breadth of evaluation can also be defined in terms of which components are checked, such as checking only properties of the active component, all components in the scope, or all components accessible [105].

The **check** defines the evaluation function for each specified property of runtime invariance. Similar to the level of evaluation, it is also possible to define the severity *level of deviations* from expected invariance, such as informative, warning and error [56]. When a deviation is observed, the *cause of deviation* can be classified to one of three options: the original specification was lacking due to insufficient input, the observing a feature in a specific context or environment that was not previously considered, or that there is a real failure observed in the runtime behavior [67]. The checks can be classified to two main *types*: the property must never be violated, or the property must always hold [84]. Different approaches to *reporting* the status of violations can be taken, such as reporting only violations [58], reporting also passing checks with chosen filtering criteria [28], or reporting the status of all checks. The threshold for when a deviation from the specification is considered significant enough to be reported can also vary according to the properties of specified runtime invariance (as discussed in more detail in section III.C), such as reporting only violations for a property when a certain number (threshold) of violations for that property have been observed [58].

The evaluation **trigger** defines when the evaluation is performed. The *cost effectiveness* of evaluating invariants is related to the frequency and depth of their evaluation ([60,97]). Different options for *frequency* can be, for example, to check a single property for a single component or for all components after each event, to check all properties after each event, or to check all properties after larger execution points (e.g., test case [97]). Some trade-offs to consider include checking a single property not always revealing errors [97], performing checks often helping in finding "transient" errors that are no longer visible in later states [80], and the increased cost and power of fully performing all possible checks at all times [97]. One approach is to focus the checks on a specific set of chosen events and properties [56].

IV. EXAMPLES

This section illustrated mapping the taxonomy presented in the previous section to practical concepts in terms of two different types of runtime behaviour modelling approaches. The first one describes modelling specific aspects of runtime invariance in a distributed sensor data collection system. The behaviour of interest in this case is event-focused, where patterns over sequences of interactions are important and should be observed to hold and are observed while the system is operational (online). A second example is provided in terms of a data visualization application. In this case, the data-flow aspects are more important and the invariance is analysed separately from its operation (off-line).

The examples presented here are intended to illustrate how different properties of runtime invariance in software behaviour can be defined for different systems. From this, different approaches can be taken to build required support for different usage domains. The details for this support are left for specific works in those domains, while we illustrate a set of specific invariant properties for each.

A. Case 1: Sensor Data Collection

This case example describes a mapping of the taxonomy for describing the runtime behaviour of a system in terms of a model-based testing tool called OSMOTester [106]. The target system is a sensor-platform server-node that manages a set of sensor-nodes. This section uses as an example a single feature related to keeping track of the dynamic non-persistent runtime state of the node.

As one of the main properties of upholding this state, the server needs to keep track of all available sensor nodes that register to the system. The sensor's registration is by sending a specific message. After this, the constant keep-alive messages need to be provided to uphold the registration. If one is not received for duration of 10 seconds from a registered sensor, it is removed from the list of connected sensors and a matching event is generated to inform any interested clients of the sensor platform.

Considering the process perspective of the taxonomy, this is mainly in the test automation usage domain. The tool we use applies a generic programming language (Java) to model the behaviour so this sets the expression language. For abstraction level, our focus with this example is on a specific behavioural property, which could be extended in following iterations to include further properties separately or integrated into a subsystem model. The specification approach in this case is manual specification with the help of expert knowledge and documentation, although tools such as Daikon could be used to provide invariants over observed executions as input as well. From the evaluation perspective, in the case of test automation, we wish to evaluate all relevant aspects for the property under analysis, report only the deviations and their cause. These aspects define the process related properties for this example in relation to the taxonomy.

Considering the properties of invariance perspective, we can define as basic measurements the registration messages, keep-alive messages, the sender of the messages (the sensor node) and the time of receiving any message. Of these, the messages themselves are static information, whereas the timestamps and the sender are contextual information. As an example pattern of invariance, we can define an expectation as "Keep-alive observed continuously after registration with a minimum of 10s interval". Several other patterns could also be formulated, including the timeout and first registration itself. These aspects define the invariant information related properties for this example in relation to the taxonomy.

B. Case 2: Data Visualization Tool

This second case example describes a mapping of the taxonomy for a tool used to help visualize and analyse behavioural data collected from the execution of a system. This section uses as an example a single feature of observing historical data over time. This feature allows taking several different measurement values and comparing their evolution over time independently and in relation to each other using graphical visualizations.

Considering the usage domain and process perspectives, in this case, the analysed data is not captured and used in real-time but rather the tool is used to load data from a log file, making this practically an offline approach. The application domain is specifically behaviour analysis, and the applied representation language in this case is a visual language in terms of graphical representations. The abstraction level depends on which part of the system is described in the log file, but is typically the overall system behaviour. The trigger and depth of evaluation are in this case related to the information captured during the runtime logging step, and depends on how the system in instrumented. The evaluation checks are performed by the human user based on the visualizations and their manipulations.

Considering the properties of invariance perspective, the basic measurements include time and data values. The relevant patterns are mainly data-flow oriented due to analysing data value evolution over time. This includes value change as observed over time for a single variable, and value relations as observed across the different visualized values. As the visualization produces a separate graph for each value, each of these is a contextual slice and the scope of these graphs is always time-based. We can further define derived measures as statistical properties of the observed data and use those to define events that can be used to define further scopes for patterns and comparison.

V. Discussion

The taxonomy and its classes presented above are based on the existing work in the literature. In this sense it limits itself to discuss properties only relevant to those in the chosen works. Additionally, it is possible to use and explore other possible relations. For example, many of the described control-flow patterns also apply to data flow patterns. For example, a value may be defined to precede another value (relating to the precedence control-flow pattern). Similarly, the set of data-flow patterns can be considered to apply in the context of control-flow. For example, the range of possible control-flow options following one control-flow event can be in a given range of possible defined control-flow events or states (related to value-range data-flow pattern).

It is also possible to take different viewpoints on the different properties discussed categorized in the taxonomy presented in this paper. For example, the taxonomy lists a set of time-based scopes, and a set of patterns related to the ordering of events. In other cases, for example, Konrad and Cheng define a set of patterns such as bounded response where a reply is expected in a given timeframe [85]. This is an example where a set of properties presented in the taxonomy in this paper are combined to form a specific set of patterns in a given domain. Specifically, the taxonomy presented in this paper aims to decompose these into their constituting parts that can be composed in different ways. While this is more generic and provides a basis for wider application, in practical application in different domains, it may be more suitable to specify a set of more specific patterns such as those defined in [85]. In this case the taxonomy can be used as an aid to create the set of suitable patterns.

Overall, the discussion in this paper is from the generic viewpoint of using runtime invariance. When a set of invariants are defined for a system, one important aspect to consider is how representative these are in describing the relevant properties of software behaviour. When defined manually by an expert, the invariants can be expected to describe relevant and important properties. However, even in these cases important invariants can be missing and in many cases no invariants are defined at all. In these cases, automated inference techniques can be used to assist in finding invariants. Both of these cases have been shown to be valid as also discussed in section III.D.1). Improving the means to help manually define invariants and to automatically mine for relevant ones thus is an interesting research question. Potential approaches to investigate include using a set of chosen invariants known to be interesting in the given domain, using combined information from static analysis, relying on statistical values to report the more interesting ones, and providing more advanced support for combining both the manual and automated approaches as also discussed in section III.D.1).

Discussion on the statistical properties of different patterns and measures highlight differences in the applied approaches. For example, in many cases the invariant patterns that have only low support level (i.e., there are few cases) are not reported. In the extraction phase, this can be useful in removing patterns observed merely due to chance that may be incorrect in themselves due to interleaving of concurrent behaviour, or completely irrelevant in the general context [3]. On the other hand, sometimes all observed behaviour is important regardless of their probability. This can be, for example, behaviour that is only rarely observed in the observed executions but is still equally important for the overall system behaviour (e.g., error handling or corner cases) [22].

Use of invariants in different domains as discussed here is not limited to those aspects discussed. In fact, many systems use invariants for various purposes but these are not always discussed in terms of invariance. For example, as discussed in section III.D, in test automation the test oracle practically always needs to be described in terms of invariance, where the input is expected to produce a given output (the relation of input to output should be invariant). In this sense, defining invariants as discussed here can be beneficial in a wider context of how people think about the behaviour of programs. However, presenting a meaningful language to describe the invariants and use them in different contexts, as well as furthering people's understanding of their relation to different domains, can be required for adopting them as a concept more widely. Use of domain-specific languages and related tools is an option for this.

Understanding and using invariants generally requires specific considerations for specific usage purposes. For example, one may refactor code based on suggestion from invariant analysis [40] but this also needs to consider the part where the human user needs to read the code and understand it. If the refactoring reduces this understanding by hiding information, this refactoring may be more harmful for the overall software maintenance. Similar needs for understanding the invariants in general ought to be considered.

Application of different properties depends on the target domain as well as the process it is used to support. For example, in testing it may be more appropriate to report all results of invariant evaluation. On the other hand, in runtime (online) monitoring of an operational system it is more useful to optimize the evaluation approach to minimize the intrusiveness on the system.

The case examples presented in section IV are from actual applications of the taxonomy, where different people have applied the taxonomy with the help of the author. In these cases, the typical approach has been to start with some specific properties and progressing from those iteratively to include more functionality. Thus, an iterative approach of adoption from the taxonomy can be seen as a useful process for its application. However, more extensive case studies in software behaviour analysis and application of the taxonomy are left as a topic for future work.

These case examples also illustrate how the taxonomy could be used as a basis for building domain-specific languages to support analysing different topics of runtime invariance. For example, in modelling system behaviour in terms of runtime invariance for test automation as presented in the first case example, a domain-specific language could allow composing modelling components based on the different types of measurements, patterns and scopes into test models. Similarly, in the second case example, the properties could be used to provide more generic base for features to manipulate and visualize the measurement data in terms of derived measures and patterns. These studies are left as a topic for future works.

VI. CONCLUSIONS

Today, runtime invariance is used in the context of many aspects of software design and analysis. The invariants for different systems are as different as their behaviour, but this paper has collected a set of common properties from existing works and presented a taxonomy describing these common properties. This should help give a more common understanding of runtime invariance in software behaviour and help in using invariants to describe it in different domains.

The presented taxonomy is based on six main facets, one describing the usage domains, two related to processes of using the invariants and three related to the information describing the invariants themselves. Overall the focus can be defined as describing the invariant information in the context of the process.

The main contribution of this paper is presenting the underpinning of a classification overview for understanding the space of runtime invariance. This provides a basis for more thorough reasoning about invariants, building tool support and identifying future research questions. Some specific questions identified include possibilities of providing more focused domain-specific invariants on top of the taxonomy and providing more extensive tool support for using the invariants according to the taxonomy presented, as existing tools only consider parts of it.

ACKNOWLEDGMENT

The author wishes to thank Arie van Deursen, Ali Mesbah, and the anonymous reviewers in the PATTERNS 2010 conference for their useful comments and discussions on the pilot study. The author also wishes to thank Lars Ebrecht for the interesting discussion and comments on the extended version of the taxonomy.

REFERENCES

[1] T. Kanstrén, "Towards a Taxonomy of Dynamic Invariants in Software Behaviour," in *2nd Int'l. Conf. on Pervasive Patterns and Applications (PATTERNS 2010)*, Lisbon, Portugal, 2010, pp. 20-27.

[2] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transaction on Software Eng.*, vol. 35, no. 5, pp. 684-702, 2009.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin,

"Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Eng.*, vol. 27, no. 2, pp. 99-123, Feb. 2001.

[4] M. Boshernitsan, R. Doong, and A. Savoia, "From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing," in *Int'l. Symposium on Software Testing and Analysis (ISSTA 2006)*, Portland, Maine, 2006, pp. 169-179.

[5] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective Statistical Debugging via Efficient Path Profiling," in *Int'l. Conf. on Software Eng. (ICSE 2009)*, Vancouver, Canada, 2009, pp. 34-44.

[6] A. Mesbah and A. van Deursen, "Invariant-Based Automatic Testing of Ajax User Interfaces," in *Int'l. Conf. on Software Eng. (ICSE 2009)*, Vancouver, Canada, 2009, pp. 210-220.

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-State Verification," in *Int'l. Conf. on Software Eng. (ICSE 1999)*, Los Angeles, CA, USA, 1999, pp. 411-420.

[8] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic Generation of Software Behavioral Models," in *Int'l. Conf. on Software Eng. (ICSE 2008)*, Leipzig, Germany, 2008, pp. 501-510.

[9] L. A. Clarke and D. S. Rosenblum, "A Historical Perspective on Runtime Assertion Checking in Software Development," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 25-37, 2006.

[10] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient Mutation Testing by Checking Invariant Violations," in *Int'l. Symposium on Software Testing and Analysis (ISSTA 2009)*, Chicago, USA, 2009, pp. 69-80.

[11] R. Floyd, "Assigning Meaning to Programs," in *Symposium on Applied Mathematics*, 1967, pp. 19-32.

[12] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576-580, 1969.

[13] L. Lin and M. D. Ernst, "Improving the Adaptability of Multi-Mode Systems via Program Steering," in *Int'l. Symposium on Software Testing and Analysis (ISSTA 2004)*, Boston, Massachusetts, USA, 2004, pp. 206-216.

[14] S. Hangal and M. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," in *Int'l. Conf. on Software Eng. (ICSE 2002)*, Orlando, Florida, USA, 2002, pp. 291-301.

[15] D. Lo and S. Khoo, "SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner," in *Int'l. Symposium on Foundations of Software Eng. (FSE 2006)*, Portland, Oregon, USA, 2006, pp. 265-275.

[16] B. Kitchenham, "Guidelines for Performing Systematic Literature Reviews in Software Engineering," Keele University, Keele, Staffs, EBSE Technical Report 2007.

[17] S. MacDonell, M. Shepperd, B. Kitchenham, and E. Mendes, "How Reliable are Systematic Reviews in Empirical Software Engineering?," *IEEE Transaction on Software Eng.*, vol. 36, no. 5, pp. 676-687, Sept./Oct. 2010.

[18] S. Ducasse and D. Pollet, "Software Architecture Reconstruction: A Process-Oriented Taxonomy," *IEEE Transactions on Software Eng.*, vol. 35, no. 4, pp. 573-591, 2009.

[19] H. Kagdi, M. L. Collard, and J. I. Maletic, "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution," *Journal of Software Maintenance and Evolution*, vol. 19, no. 2, pp. 77-131, 2007.

[20] M. S. Ali, M. A. Babar, L. Chen, and K-J. Stol, "A Systematic Review of Comparative Evidence of Aspect-Oriented Programming," *Information and Software Technology*, vol. 52, no. 9, pp. 871-887, 2010.

[21] MIT Program Analysis Group. (2012, January) Daikon-related invariant detection publications. [Online]. http://groups.csail.mit.edu/pag/daikon/pubs/

[22] T. Kanstrén, *A Framework for Observation-Based Modelling in Model-Based Testing*. Oulu, Finland: VTT, 2010.

[23] R. V. Binder, "Design for Testability in Object-Oriented Systems," *Communications of the ACM*, vol. 37, no. 9, pp. 87-101, Sept. 1994.

[24] J. W. Nimmer and M. D. Ernst, "Invariant Inference for Static Checking: An Empirical Evaluation," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, pp. 11-20, 2002.

[25] L. Burdy et al., "An Overview of JML Tools and Applications," *Int'l. Journal in Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212-232, June 2005.

[26] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic Symbolic Execution for Invariant Inference," in *Intl. Conf. on Software Eng. (ICSE 2008)*, Leipzig, Germany, 2008, pp. 281-290.

[27] D. S. Rosenblum, "Towards a Method of Programming with Assertions," in *Int'l. Conf. on Software Eng. (ICSE 1992)*, Melbourne, Australia, 1992, pp. 92-104.

[28] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining Temporal API Rules from Imperfect Traces," in *Int'l. Conf. on Software Eng. (ICSE 2006)*, Shanghai, China, 2006, pp. 282-291.

[29] B. Meyer, "Applying Design by Contract," *Computer*, vol. 25, no. 10, pp. 40-51, 1992.

[30] S. Sims, R. Cleaveland, K. Butts, and S. Ranville, "Automated Validation of Software Models," in *Int'l. Conf. on Automated Software Eng. (ASE 2001)*, San Diego, USA, 2001, pp. 91-96.

[31] S. McCamant and M. Ernst, "Early Identification of Incompatibilities in Multi-Component Upgrades," in *European Conf. on Object-Oriented Programming (ECOOP 2004)*, Oslo, Norway, 2004, pp. 440-464.

[32] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic Extraction of Object-Oriented Component Interfaces," in *Int'l. Symposium on Software Testing and Analysis (ISSTA 2002)*, Roma, Italy, 2002, pp. 218-228.

[33] A. Coronato, A. d'Acierno, and G. De Pietro, "Automatic Implementation of Constraints in Component based Applications," *Information and Software Technology*, vol. 47, no. 7, pp. 497-509, 2005.

[34] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-Crasher: A Hybrid Analysis Tool for Bug Finding," *ACM Transactions on Software Eng. and Methodology*, vol. 17, no. 2, pp. 1-37, 2008.

[35] J-M. Jézéquel, D. Deveaux, and Y. Le Traon, "Reliable Objects: Lightweight Testing for OO Languages," *IEEE Software*, vol. 18, no. 4, pp. 76-83, 2001.

[36] S. Thummalapenta and T. Xie, "Mining Exception-Handling Rules as Sequence Association Rules," in *Int'l. Conf. on Software Eng. (ICSE 2009)*, Vancouver, Canada, 2009, pp. 496-506.

[37] J. Henkel, C. Reichenbach, and A. Diwan, "Discovering Documentation for Java Container Classes," *IEEE Transactions on Software Eng.*, vol. 33, no. 8, pp. 526-543, 2007.

[38] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical Debugging: A Hypothesis Testing-Based Approach," *IEEE Transactions on Software Eng.*, vol. 32, no. 10, pp. 831-848, Oct. 2006.

[39] L. Mariani, S. Papagiannakis, and M. Pezzé, "Compatibility and Regression Testing of COTS-Component-Based Software," in *Int'l. Conf. on Software Eng. (ICSE 2007)*, Minneapolis, USA, 2007, pp. 85-95.

[40] Y. Kataoka, M. Ernst, W. Griswold, and D. Notkin, "Automated Support for Program Refactoring Using Invariants," in *Int'l. Conf. on Software Maintenance (ICSM 2001)*, Florence, Italy, 2001, pp. 736-743.

[41] N. Ubayashi, J. Piao, S. Shinotsuka, and T. Tamai, "Contract-Based Verification for Aspect-Oriented Programming," in *Int'l. Conf. on Software Testing, Verification, and Validation (ICST 2008)*, Lillehammer, Norway, 2008, pp. 180-189.

[42] S. Ducasse, T. Gîrba, and R. Wuyts, "Object-Oriented Legacy System Trace-Based Logic Testing," in *European Conf. on Software Maintenance and Reeng. (CSMR 2006)*, Bari, Italy, 2006, pp. 37-46.

[43] David Lo and Shahar Maoz, "Mining Scenario-Based Triggers and Effects," in *23rd Int'l.l Conf. on Automated Software Engineering (ASE 2008)*, L'Aquila, Italy, 2008, pp. 109-118.

[44] D. Ganesan et al., "Architectural Analysis of Systems based on the Publisher-Subscriber Style," in *Working Conference on Reverse Engineering (WCRE 2010)*, Boston, USA, 2010, pp. 173-182.

[45] M. Christodorescu, S. Jha, and C. Kruegel, "Mining Specifications of Malicious Behaviour," in *Joint meeting of the European Software Eng. Conf. and the Symposium on the Foundations of Software Eng. (ESEC/FSE 2007)*, Dubrovnik, Croatia, 2007, pp. 5-14.

[46] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An Approach for the Anomaly-Based Detection of State Violations in Web Applications," in *Int'l. Symposium on Recent Advances in Intrusion Detection (RAID 2007)*, Queensland, Australia, 2007, pp. 63-86.

[47] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic Inference and Enforcement of Kernel Data Structure Invariants," in *24th Annual Computer Security Applications Conf. (ACSAC 2008)*, 2008, pp. 77-86.

[48] D. Schuler, V. Dallmeier, and C. Lindig, "A Dynamic Birthmark for Java," in *Int'l. Conf. on Automated Software Eng. (ASE 2007)*, Atlanta, Georgia, USA, 2007, pp. 274-282.

[49] M. Feng and R. Gupta, "Detecting Virus Mutations via Dynamic Matching," in *Int'l. Conf. on Software Maintenance (ICSM 2009)*, Edmonton, Canada, 2009, pp. 105-114.

[50] M. Barnett and W. Schulte, "Runtime Verification of.NET Contracts," *Journal of Systems and Software*, vol. 65, no. 3, pp. 199-208, 2003.

[51] Y. Le Traon, B. Baudry, and J-M. Jézéquel, "Design by Contract to Improve Software Vigilance," *IEEE Transactions on Software Eng.*, vol. 32, no. 8, pp. 571-586, Aug. 2006.

[52] B. Demsky et al., "Inference and Enforcement of Data Structure Consistency Specifications," in *Int'l. Symposium on Software Testing and Analysis (ISSTA 2006)*, Portland, Maine, 2006, pp. 233-243.

[53] D. Lorenzoli, L. Mariani, and M. Pezze, "Towards Self-Protecting Enterprise Applications," in *Int'l. Symposium on Software Reliability (ISSRE 2007)*, Trollhättan, Sweden, 2007, pp. 39-48.

[54] J. H. Perkins et al., "Automatically Patching Errors in Deployed Software," in *Symposium on Operating System Principles (SOSP 2009)*, Big Sky, USA, 2009, pp. 87-102.

[55] Y. Wei et al., "Automatex Fixing of Programs with Contracts," in *Int'l. Symposium on Software Testing and Analysis (ISSTA 2010)*, Trento, Italy, 2010, pp. 61-71.

[56] M. Book, T. Brückmann, V. Gruhn, and M. Hülder, "Specification and Control of Interface Responses to User Input in Rich Internet Applications," in *Int'l. Conf. on Automated Software Eng. (ASE 2009)*, Auckland, New Zealand, 2009, pp. 321-331.

[57] J. Bumim and K. Sen, "Asserting and Checking Determinism for Multithreaded Programs," in *Joint meeting of the European Software Eng. Conf. and the Symposium on the Foundations of Software Eng. (ESEC/FSE 2009)*, Amsterdam, The Netherlands, 2009, pp. 3-12.

[58] M. Gabel and Zhendong. Su, "Online Inference and Enforcement of Temporal Properties," in *Int'l. Conf. on Software Eng. (ICSE 2010)*, Cape Town, South Africa, 2010, pp. 15-24.

[59] C. Pacheso and M. D. Ernst, "Eclat: Automatic Generation and Classification of Test Inputs," in *European Conf. on Object-Oriented Programming (ECOOP 2005)*, Glasgow, UK, 2005, pp. 504-527.

[60] J. H. Andrews and Y. Zhang, "General Test Result Checking with Log File Analysis," *IEEE Transaction on Software Eng.*, vol. 29, no. 7, pp. 634-648, July 2003.

[61] A. M. Memon, "An Event-Flow Model of GUI-based Applications for Testing," *Journal of Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137-157, 2007.

[62] L. Ebrecht and K. Lemmer, "Highlighting the Essentials of the Behaviour of Reactive Systems in Test Descriptions Using the Behavioural Atomic Element," in *2nd Int'l. Conf. on Pervasive Patterns and Applications (PATTERNS 2010)*, Lisbon, Portugal, 2010, pp. 53-59.

[63] M. Auguston, J. B. Michael, and M-T. Shin, "Environment Behavior Models for Automation of Testing and Assessment of System Safety," *Information and Software Technology*, vol. 48, no. 10, pp. 971-980, 2006.

[64] M. Alshraideh and L. Bottaci, "Search-Based Software Test Data Generation for String Data Using Program-Specific Search Operators," *Software Testing, Verification and Reliability*, vol. 16, no. 3, pp. 175-203, 2006.

[65] B. Korel and A. M. Al-Yami, "Assertion-Oriented Automated Test Data Generation," in *Int'l. Conf. on Software Eng. (ICSE 1996)*, Berlin, Germany, 1996, pp. 71-80.

[66] T. Xie and D. Notkin, "Tool-Assisted Unit-Test Generation and Selection Based on Operational Abstractions," *Journal of Automated Software Eng.*, vol. 13, no. 3, pp. 345-371, July 2006.

[67] M. Harder, J. Mellen, and M. D. Ernst, "Improving Test Suites via Operational Abstraction," in *Int'l.Conf. on Sofware Eng. (ICSE 2003)*, Portland, Oregon, USA, 2003, pp. 60-71.

[68] J. E. Cook and A. L. Wolf, "Discovering Models of Software Processes from Event-Based Data," *ACM Transactions on Software Eng. and Methodology*, vol. 7, no. 3, pp. 215-249, 1998.

[69] J. E. Cook and Z. Du, "Discovering Thread Interactions in a Concurrent System," *Journal of Systems and Software*, vol. 77, no. 3, pp. 285-297, Sept. 2005.

[70] C. Csallner and Y. Smaragdakis, "Dynamically Discovering Likely Interface Invariants," in *Int'l. Conf. on Software Eng. (ICSE 2006)*, Shanghai, China, 2006.

[71] J. Huselius and J. Andersson, "Model Synthesis for Real-Time Systems," in *9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, Manchester, UK, 2005, pp. 52-60.

[72] S. Ali et al., "A State-Based Approach to Integration Testing based on UML Models," *Information and Software Technology*, vol. 49, no. 11-12, pp. 1087-1106, 2007.

[73] R. Allen and D. Garlan, "Formalizing Architectural Connection," in *Int'l. Conf. on Software Eng. (ICSE 1994)*, Sorrento, Italy, 1994, pp. 71-80.

[74] J. Yang and D. Evans, "Automatically Inferring Temporal Properties for Program Evolution," in *Int'l. Symposium on Software Reliability Eng. (ISSRE 2004)*, Saint-Malo, Bretagne, France, 2004, pp. 340-351.

[75] N. Kuzmina, J. Paul, R. Gamboa, and J. Caldwell, "Extending Dynamic Constraint Detection with Disjunctive Constraints," in *Int'l. Workshop on Dynamic Analysis (WODA 2008)*, Seattle, Washington, 2008, pp. 57-63.

[76] J. E. Cook and A. L. Wolf, "Event-Based Detection of Concurrency," in *6th International Symposium on Foundations of Software Engineering (FSE 1998)*, Paris, France, 1998, pp. 35-45.

[77] M. Pradel and T. R. Gross, "Automatic Generation of Object Usage Specifications from Large Method Traces," in *Int'l. Conf. on Automated Software Eng. (ASE 2009)*, Auckland, New Zealand, 2009, pp. 371-382.

[78] M. Gabel and Z. Su, "Javert: Fully Automatic Mining of Temporal Properties from Dynamic Traces," in *16th International Symposium on Foundations of Software Engineering (FSE 2008)*, Atlanta, USA, 2008, pp. 339-349.

[79] D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani, "Mining Quantified Temporal Rules: Formalism, Algorithms, and Evaluation," in *Working Conference on Reverse Engineering (WCRE 2009)*, Lille, France, 2009, pp. 62-71.

[80] A. Memon and Q. Xie, "Using Transient/Persistent Errors to Develop Automated Test Oracles for Event-Driven Software," in *Int'l. Conf. on Automated Software Eng. (ASE 2004)*, Linz, Austria, 2004, pp. 186-195.

[81] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno, "Invariant-Based Specification, Synthesis and of Synchronization in Concurrent Programs," in *Int'l.l Conf. on Software Eng. (ICSE 2002)*, Orlando, Florida, 2002, pp. 442-452.

[82] A. Machetto, P. Tonella, and F. Ricca, "State-Based Testing of Ajax Web Application," in *Int'l. Conf. on Software Testing, Verification and Validation (ICST 2008)*, Lillehammer, Norway, 2008, pp. 121-130.

[83] P. O. Meredith, D. Jin, F. Chen, and G. Rosu, "Efficient Monitoring of Parametric Context-Free Patterns," in *Int'l. Conf. on Automated Software Eng. (ASE 2008)*, L'Aquila, Italy, 2008, pp. 148-157.

[84] N. Walkinshaw and K. Bogdanov, "Inferring Finite-State Models with Temporal Constraints," in *Int'l. Conf. on Automated Software Eng. (ASE 2008)*, L'Aquila, Italy, 2008, pp. 248-257.

[85] S. Konrad and B. H.C. Cheng, "Real-Time Specification Patterns," in *Int'l. Conf. on Software Eng. (ICSE 2005)*, St. Louis, Missouri, USA, 2005, pp. 372-381.

[86] C. Ackermann, M. Lindvall, and R. Cleaveland, "Recovering Views of Inter-System Interaction Behaviors," in *Working Conf. on Reverse Engineering (WCRE 2009)*, Lille, France, 2009, pp. 53-61.

[87] S. Thummalapenta and T. Xie, "Alattin: Mining Alternative Patterns for Detecting Neglected Conditions," in *Int'l. Conf. on Automated Software Eng. (ASE 2009)*, Auckland, New Zealand, 2009, pp. 283-294.

[88] P. Bellini, P. Nesi, and D. Rogai, "Expressing and Organizing Real-Time Specification Patterns via Temporal Logics," *Journal of Systems and Software*, vol. 82, no. 2, pp. 183-196, 2009.

[89] X. Yuan and A. M. Memon, "Iterative Execution-Feedback Model-Directed GUI Testing," *Information and Software Technology*, vol. 52, no. 5, pp. 559-575, 2010.

[90] A. Wasylkowski and A. Zeller, "Mining Temporal Specifications from Object Usage," in *Int'l. Conf. on Automated Software Eng. (ASE 2009)*, Auckland, New Zealand, 2009, pp. 295-306.

[91] K. Saleh, A. A. Boujarwah, and J. Al-Dallal, "Anomaly Detection in Concurrent Java Programs Using Dynamic Data Flow Analysis," *Information and Software Technology*, vol. 43, no. 15, pp. 973-981, 2001.

[92] G. Marceau, G. H. Cooper, S. Krishnamurthi, and S. P. Reiss, "A Dataflow Language for Scriptable Debugging," in *Int'l. Conf. on Automated Software Engineering (ASE 2004)*, Linz, Austria, 2004, pp. 218-227.

[93] A. Cavalli, C. Gervy, and S. Prokopenko, "New Approaches for Passive Testing Using an Extended Finite State Machine Specification," *Information and Software Technology*, vol.

45, no. 12, pp. 837-852, 2003.

[94] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst, "Dynamic Inference of Abstract Types," in *Int'l. Symposium on Software Testing and Analysis (ISSTA 2006)*, Portland, Maine, USA, 2006, pp. 255-265.

[95] J. Henkel and A. Diwan, "Discovering Algebraic Specifications from Java Classes," in *European Conf. on Object-Oriented Programming (ECOOP 2003)*, Darmstadt, Germany, 2003, pp. 431-456.

[96] N. Polikarpova, I. Ciupa, and B. Meyer, "A Comparative Study of Programmer-Written and Automatically Written Contracts," in *Int'l. Symposium on Software Testing and Analysis (ISSTA 2009)*, Chicago, USA, 2009, pp. 93-104.

[97] Q. Xie and A. M. Memon, "Designing and Comparing Automated Test Oracles for GUI-Based Software Applications," *ACM Transactions of Software Eng. and Methodology*, vol. 16, no. 1, pp. 1-36, Feb. 2007.

[98] N. Delgado, A. Q. Gates, and S. Roach, "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools," *IEEE Transactions on Software Eng.*, vol. 12, no. 30, pp. 859-872, Dec. 2004.

[99] C. Zhao, J. Kong, and K. Zhang, "Program Behavior Discovery and Verification: A Graph Grammar Approach," *IEEE Transactions on Software Eng.*, vol. 36, no. 3, pp. 431-448, May/June 2010.

[100] Y. J. Ren and F. Chang, "ATTEST: A Testing Toolkit for Validating Software Properties," in *Int'l. Conf. on Software Maintenance (ICSM 2007)*, Paris, France, 2007, pp. 469-472.

[101] T. H. Gibbs, B. A. Malloy, and J. F. Power, "Automated Validation of Class Invariants in C++ Applications," in *Int'l. Conf. on Automated Software Eng. (ASE 2002)*, Edinburgh, UK, 2002, pp. 205-214.

[102] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka, "Overview and Evaluation of Constraint Validation Approaches in Java," in *Int'l. Conf. on Software Eng. (ICSE 2007)*, Minneapolis, USA, 2007, pp. 313-322.

[103] G. J. Holzmann, "The Logic of Bugs," in *Int'l. Symposium on the Foundations of Software Eng. (FSE 2002)*, Charleston, South Carolina, USA, 2002, pp. 81-87.

[104] J. Burnim and K. Sen, "DETERMIN: Inferring Likely Deterministic Specifications of Multithreaded Programs," in *Int'l. Conf. on Software Eng. (ICSE 2010)*, Cape Town, South Africe, 2010, pp. 415-424.

[105] A. Memon, I. Banerjee, and Adithya. Nagarajan, "What Test Oracle Should I Use for Effective GUI Testing," in *Int'l. Conf. on Software Eng. (ICSE 2003)*, Portland, Oregon, USA, 2003, pp. 164-173.

[106] T. Kanstrén. (2012, January) OSMOTester - Simple Model-Based Testing Tool. [Online]. http://code.google.com/p/osmo/