Towards A Taxonomy of Dynamic Invariants in Software Behaviour

Teemu Kanstrén
VTT Technical Research Centre of Finland
Kaitoväylä 1, 90570 Oulu, Finland
teemu.kanstren@vtt.fi

Abstract— **The use of dynamic invariants to describe software behaviour has gained increasing popularity and various tools and techniques for mining and using these invariants have been published. Typically, these invariants are used to support various software engineering tasks, such as testing and debugging, which require one to understand and be able to reason about the system behaviour in terms of these invariants. However, the existing works are generally focused on a specific set of invariants for a specific purpose. In many cases it is also useful to view these in a wider context to enable a wider understanding of the invariants and to provide more extensive support across different domains. This paper presents work towards a general taxonomy describing the properties of dynamic invariants based on a review of existing work in their use, providing a basis for a wider adoption of different invariant features in different domains.**

Keywords-dynamic invariants; taxonomy; software behaviour

I. INTRODUCTION

Dynamic invariants are used to describe invariant properties of software behavior in terms of dynamic analysis. Dynamic analysis uses as basis information captured as observations from a (finite) set of program executions, such as test executions [1]. In line with these definitions, a dynamic invariant is defined here as a property that holds at a certain point or points in a program execution [2]. Recently the use of such dynamic invariants has become an increasingly popular technique in supporting different software engineering tasks (e.g., [3,4,5]).

Examples of dynamic invariants include data-flow constraints (e.g., `x` always greater than `0`) [2], control-flow constraints (e.g., `request` always followed by a `reply`) [6], or their combinations (e.g., `x` is always greater than `0` when `request` is followed by a `reply`) [7]. Invariants defined in terms of dynamic analysis can also be referred to as likely invariants as they are based on observations made from a set of program executions, which typically do not cover the entire program behavior state-space [2].

Dynamic invariants can be mined with automated tools or specified manually for further processing with automated tools. The idea of documenting and using invariants to reason about program behavior at run-time can be seen to be as old as programming itself [8,9]. Using invariants expressed in first-order logic to capture formal constrains on program behavior was introduced as early as 1960's [8] by the pioneering work of Floyd [10] and Hoare [11].

Dynamic invariants can be used in a variety of software engineering tasks and domains, such as helping in program comprehension [2,12], behavior enforcement [13], test generation and oracle automation [5], or debugging [14]. Thus, when explicitly defined, a set of invariants forms a basis for building automated support for many different purposes.

There exist a number of tools to support the use of dynamic invariants in different tasks [2,5,12]. Many of these tools use a specific set of invariants for a specific domain. When applying dynamic invariants in different domains, it is useful to also consider them in a wider context. Also, when a set of invariants needs to be provided, either as manually defined input for a tool to use as a basis for automated processing, or as output by an automated specification mining tool, being able to generally reason about them is needed for their effective use.

This paper describes a taxonomy for dynamic invariants. The taxonomy describes a set of common properties for invariants describing the dynamic properties of software behavior. As a basis, a set of invariants and their use have been reviewed from existing works. The study is structured to describe how the invariants are specified and used, what kind of invariant patterns over software behavior they capture, in which scope of behavior they apply, and what information about the system behavior is needed to be able to express and evaluate them.

The goal of this paper is to provide a starting point for a `road map' of the work accomplished so far on dynamic invariants, to provide help software engineers identify open research questions and new branches of discoveries, and to facilitate the use of dynamic invariants by a systematic definition of their different properties.

This paper is structured as follows. Section II describes the overall approach taken to create the taxonomy. Section III presents the taxonomy, its axes, and the individual categories. Finally, section IV provides discussion followed by concluding remarks.

II. TAXONOMY BUILDING APPROACH

Following guidelines from [15] for performing reviews, the works selected in this paper have been chosen where they describe or use some form of invariants over dynamic software behavior. This includes how these invariants are (manually) defined, and how automated specification mining approaches are used to produce them. For the sake of space and focus, this selection is focused on the originality of the work (in terms of adding to the taxonomy), its excellence (study process), and observed impacts (citation). Papers that take specific approaches to use and define invariants are also considered to provide a wider view. This approach is in-

 ISBN: 978-1-61208-111-3

spired and follows the taxonomy building approaches taken by Ducasse et al. [16] and Kagdi et al. [17].

The information presented in the taxonomy is based on information from publicly available works such as research papers, PhD theses and technical reports. Different publication databases were used as a basis for the search of related work. The focus of this paper is on properties related to invariants over dynamic behavior of software and thus the selection is focused on invariants over software runtime behavior. Work in the more static formal methods domain is reviewed when referenced from the works on dynamic behavior analysis but otherwise is not considered more deeply. An adapted version of Binder's ``fishbone" diagram [18] is used to describe the different aspects of the taxonomy.

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. [2] on defining dynamic invariants in terms of the program data-flow. 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. This was presented to experts in the field (identified in the acknowledgements section). After this, additional refinement was done based on the comments received.

III. TAXONOMY

This section describes the taxonomy that is the core contribution of this paper. The presentation starts with showing the main axes of the taxonomy, and proceeds to describe each axis in more detail in the following subsections.

A. Axes of the taxonomy

The main facets of the taxonomy of dynamic invariant properties of software behavior as discussed in this paper are presented in

Figure 1. This taxonomy is divided in six main facets, which are further divided to three process related ones and three facets describing information about the invariants themselves. The process-based facets describe various aspects of working with the invariants and include invariant specification, extraction, and usage. The invariant information facets are defining the invariants and include measurements, behavioral patterns, and scope. These six facets will be briefly presented here and discussed in more detail in the following subsections.

The generic flow of using invariants is presented in Figure 2. To make use of invariants, a set of invariants describing the aspects of interest in the software behaviour needs to be defined. This can be done either manually or with the use of an automated mining tool. In case a mining tool is used, a set of invariant templates describing a potential set of useful invariants is needed (e.g., [2]). An extensive basis for providing such templates this is provided by the invariant information properties of the taxonomy. The same applies for manual specification, as the properties of invariant information enable effectively reasoning about possible invariants. Analyzing software behaviour is typically based on large sets of observations (trace data), automated tools to help project the specified invariants over the captured observations is needed. This is again based on a similar tools and invariant templates as when using automated mining tools in the specification phase. For this reason, the specification and extraction phases are described in terms of shared properties in the following subsections of the taxonomy. However, from the process perspective, it should be noted that typically two phases of the process follow where a step of invariant specification is done and another step of extraction is done in order to form a basis for the final step of invariant usage, where these two are compared against each other.

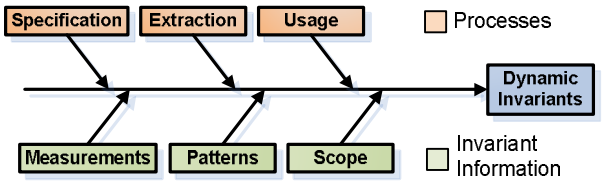

Figure 1. The taxonomy: Main facets.

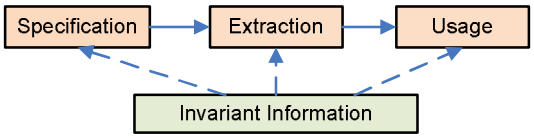

Figure 2. The taxonomy: Flow of elements.

Invariant information includes the measurements that describe the actual data that one needs to observe in order to build or evaluate an invariant, the behavioral patterns that the invariants describe over the measurements, and the scope of program behaviour when the invariant is expected to hold.

B. Specification and Extraction

The different aspects of invariant specification and extraction are illustrated in Figure 3. As mentioned before, these two phases share many properties and are thus described here in terms of common properties.

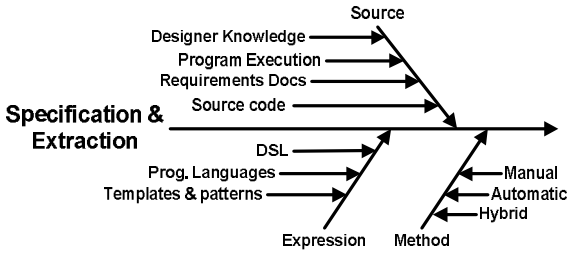

Figure 3. Invariant specification.

Different approaches to obtain the information for specifying the invariants include fully (**automatically**) reverse engineering these from observing program behaviour [2], describing them **manually** based on specifications or designer knowledge [19,5], or taking a **hybrid** approach where the reverse-engineered information is manually augmented with information from specifications [20]. When extracting a set of invariants based on dynamic analysis, the set of observed **program executions** is defined by what is available (e.g., test suite) and what is the goal of the analysis (e.g., analysis of a subset of the entire test suite) [1].

 ISBN: 978-1-61208-111-3

The type 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 [20]. Additional **designer knowledge** gained when working on the system provides insights into its invariant behaviour, allowing them to design additional invariants to describe the implementation [19]. Source code and program execution are more suited for automated analysis. As **source code** is mainly useful in terms of static analysis, it can be used mainly as an additional input for dynamic analysis such as providing interface definitions [21,20]. From these sources, one needs to capture a set of suitable invariants describing the relevant properties of dynamic behaviour in the software. Experiments have shown that combining both types of sources gives the best results, where both provide useful invariants not identified by the other approach [22].

When the invariants have been defined, they need to be expressed in order to allow the user to process them effectively. 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 [5]. **Templates and patterns** can be created to express a chosen set of invariants generally over different programs and used for automated invariant mining [2,14]. **Programming languages** provide powerful constructs that can be used to describe invariants as they allow for full specification of all properties expressable in a full programming language [19].

C. Invariant Usage

The usage domain of an invariant refers to the context and goal to which it is applied. This describes 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. The invariant usage aspects of the taxonomy are illustrated in Figure 4.

In **offline use**, the invariants are used separately from the execution of the analysed program. In **online use** the usage of the invariants is linked to the executing program. 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 operate offline. Besides these two, the other domains make equal use of both online and offline approaches. Since this paper is focusing on dynamic analysis, static analysis is not considered further here other than to note that dynamic invariants can also be used as input for it [23].

Behaviour specification is the basic relevant concept for any application of invariants described in this paper, as the invariants need to be specified before they can be used. In itself, this does not constitute as a usage domain but rather as a basis for the other approaches. From the specification perspective, these invariants can also be used as the basis for application of many formal methods such as model checking and other forms of static analysis [6,24]. For example, a data-flow invariant can specify that a return value should always be greater than zero [24], or that the value returned by get() should always match the last given parameter of set() [25]. Similarly, control-flow related invariants can be used to define constraints such as always closing opened database connections [26].

A specific area in the domain of behaviour specification is the automated mining of specifications based on dynamic invariants. Various tools that work with dynamic invariants are in fact aimed at automatically mining specifications in terms of invariants for the user to process [2,4,7,12,14,26,27,28,29,30]. In this sense, dynamic invariants are also used to assist in the process of specifying the software itself. However, the taxonomy described in this paper takes no stand on how the invariants are obtained. The taxonomy is intended to support the process of using and creating the invariants, whether through automated mining techniques or by manual specification. In both case, a systematic description provided by the taxonomy should help in creating and using them.

Behaviour analysis supports either automated or manual analysis of software runtime behaviour. A set of specified invariants are given and used in each case to analyse how the system behaves. This information is presented to the human user for analysis. Failure cause location can be supported by analyzing how the invariants change over time and reporting any significant changes before a failure is observed [4,14] and by comparing the invariants observed over both failing and non-failing program executions [4]. 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 [2,31], such as changed interaction sequences and input-output transformation [31]. Another example in this domain is suggesting refactoring based on invariants holding over values (e.g., parameter always constant) that can be used to simplify the program [32].

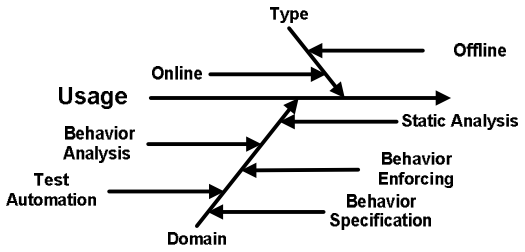

Figure 4. Invariant usage.

Further, 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 [28,33]. 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 [2].

Behaviour enforcing mechanisms analyse the behaviour of the observed software based on a given set of invariants similar to the domain of behaviour analysis. However, they additionally take automated action to modify the behaviour based on differences observed with regards to the given (expected) invariants. For example, automatic adaptation mechanisms can use invariants to choose a new state for the

 ISBN: 978-1-61208-111-3

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 [34,35].

Software **test automation** is basically a comparison of the expected behaviour of the software to its actual behaviour. This comparison is done by a test oracle that needs a representation of the expected behaviour of the software in terms of input-output transitions. As this needs to be described in terms of invariant behaviour, dynamic invariants can be used to encode this information as a basis for the test evaluation, where test results are expected to conform to these invariants [5,14,36,37].

When the invariants describe meaningful (important) properties of the software behaviour, they also make good candidates for evaluating which parts of the software behaviour should be covered in testing. Invariants can then be used to assess test coverage in terms of invariants covered by the test suite [9,38]. This can further be improved by automatically generating test inputs that aim to increase the set of covered invariants [37,39].

Component upgrade checking is a special type of test automation. In this context it is important to verify that an update of a component works with the rest of the system. Invariants can be used to describe how the component behaves with the other components, and to assess the relations of the invariants of the different components against each other. These invariants can describe, for example, the inputs and outputs of the different components in terms of control- and data-flow [25,31]. The comparison is then an evaluation of these invariants over the different versions.

D. Measurements

In order to apply dynamic invariants one needs to collect the required information to either assess that they hold or to infer (mine) them, depending on the intended usage domain. In any case, one needs to be able to define and collect the required information from the observed system. The process of extracting this information is referred to here as information extraction, similar to [40]. This aspect of the taxonomy is shown in Figure 5.

The **information type** of the measurements can be classified to two different types of static and contextual information [27]. **Static information** in a dynamic 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** described dynamic information that changes over the program executions over a single point depending on the context (e.g., test case) 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 [27,41]. 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 [12,27,41]. In this case, the scope identifier becomes the basic measure (e.g., thread id [29] or constant parameter value [27]).

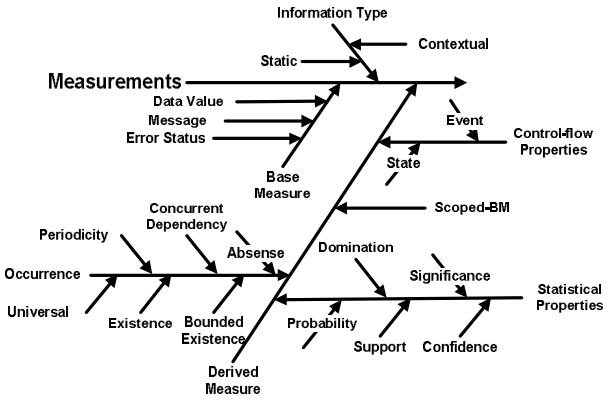

Figure 5. Measurements for invariants.

The term **base measure** is used here to refer to a type of measurement information that describes some basic value of program behaviour as it is observed. For dataflow variables this includes the **data values** with basic data types such as Boolean values, integers, and text strings (character sequences) stored in different variable and parameter values [2]. In the scope of object oriented programs the runtime type of an object can also be used as a base measure [14].

From the control-flow perspective the base measures are the **messages** passed between different elements of the control-flow. For example, method invocations between components (such as classes or services) [7,31] or invocations on graphical user interface (GUI) operators [5,36].

A specific case of control-flow is error handling flows identified by some **error status**. Error scenarios can be classified to generic errors and application specific errors [5]. 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[5]) error codes. When represented in a uniform way (e.g., by programming language exception mechanisms[37]), these can be generally observed and described 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 [37]. 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 input [20].

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 [2]. 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`. These produce **scoped**

base measures. The different scopes are discussed in subsection F.

A specific case of this is software **control-flow properties** that can be described in terms events and states. From the control-flow point of view, an **event** can be described as an identifiable, instantaneous action in the observed software behaviour, such as passing a message or committing a transaction [29]. 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 interaction parameters or inside components internal state variables [33]. 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 invariants when observing which paths are taken and which ones are not [4,42].

Two properties related to both data- and control-flow measures are those describing their occurrence and statistical properties. Derived measures related to **occurrence** describe how data- or control-flow measures are expected to occur in a given scope. Of these, **absence** defines an expectation that the measure does not exist in the defined scope [6]. **Existence** denotes that the measure exists in a scope, and **bounded existence** that the measure exists N time in a scope, where N denotes either exact, minimum or maximum number [6]. **Universal** defines an expectation that the measure applies to the whole scope [6]. **Concurrent dependence** is related to observed fork and join points in execution [43]. A fork expects one measure to be followed by several measures of a given type and a join expects several measures of a given type to be followed by a single specific measure [43]. **Periodicity** describes a measure repeating over a given cycle (scope) [43].

Statistical properties describe additional information for other base- or derived-measures. Support and confidence are two values commonly used together. **Support** defines the number of times a measure is observed in behaviour [26,44]. **Confidence** can be used with the same definition [2] but also as a definition of how often another measure is observed in relation to support, meaning how often a precondition is followed by a post-condition [27,44].

Probability defines the threshold for a measure to be observed in a given scope. This can be used in different ways. A measure with low probability (support percentage) can be excluded from analysis to address anomalies [2,12,29]. Different approaches are used for this depending on the target invariants, from low level (1% or less) [2] to 20% [12]. The probability can also refer to probabilities of a measurement value inside a range of allowed values [13]. Deviations from the expected values are typically given a probability, which can then define the significance of the deviation [13,14,33]. This threshold can be used for different purposes such as identifying probable failure causes [14], security attacks [33], and to decide new states for automated adaptation [13].

Significance defines the importance of an invariant violation or of the measured variable. 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 latter behaviour [34]. When a variable is observed as having no correlation with other variables it can be considered irrelevant [32]. A generic derived measure used for these is the number of measurements. **Dominance** is a measure used to remove overlapping patterns where one includes the other as a sub-pattern [30].

E. Behavioral Patterns

A dynamic invariant in software behaviour basically describes a pattern over the observed behaviour. This aspect of the taxonomy is shown in Figure 6. Control-flow related patterns describe ordering of events or states in the observed system [6]. Data-flow related patterns describe the data-flow of the observed software, such as what values a given variable takes during the software execution [2].

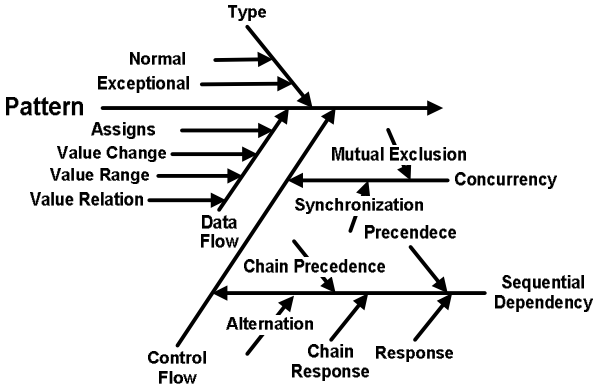

Figure 6. Behavioral patterns.

Together these can be combined to represent the complete 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 [43]. 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 ex-ample, event `P1` can be followed by event `P2` when `x<0` and by `P3` when `x>=0`. Together 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. For example, a stack allowing three pop operations after having three push operations performed on it [30]. These can be further combined to form a more complete model such as an extended finite state machine, where states represent the control-flow and the transitions between states are defined in terms of data-flow [20,34].

Each pattern can further be related to describing different types of behaviour, which can be generally classified as **exceptional** (error) or **normal** (correct) behaviour of the observed system [5]. For example, a transaction may complete or fail due to its parameters and environment state. As described in subsection D, different base measures related to errors can be used to identify them.

Control flow patterns basically describe the sequential dependencies between a programs events and states [43]. In

the following discussion the term ``event'' is used to refer to both events and states.

Alteration describes two or more events being grouped together and always appearing as an alternating sequence such as ABABAB [30,41]. Specialized cases can also be defined such as events in the alternating sequence repeating multiple times, AB*C, where B is repeated 1-N times between A and C [30], or a cutoff in the end of the sequence (ABABA) [12].

Precedence describes a specific event P always occurring before another specific event Q [6]. This can also be referred to as a precondition [24] and is a specific case of **chain precedence**, which defines that a sequence of events (Q1,Q2,Q3,…) is always preceded by another sequence of events (P1,P2,P3,…) [6].

The opposite of precedence is **response,** which defines that event P is always followed by event Q. 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,…) [6].

Specific cases of alteration are the patterns related to **concurrency**. These can be classified to two main patterns of mutual exclusion and synchronization [29]. **Mutual exclusion** occurs when no two measures are observed at the same time. **Synchronization** has two specific cases, where two measures are always observed together (overlapping) or where one starts as another ends.

Data flow patterns describe properties and relations over variable values during program execution. The **assigns** pattern defines that in a defined scope, values of specific variables are assigned to (modified) [24]. This can also be described in terms of values that are not modified [2]. **Value change** is an evolutionary pattern that describes how a value changes over time in a given context. This pattern defines a reference value for the expected value distribution of the observed variable in the given scope. For example, the expectation can be that change in value is always small (within a given threshold such as change<5) [14]. A specific case can be a variable that is never set [2].

A **value range** describes a variable always having a range of values in a given scope [2,14]. 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) [2]. Common constants such as zero or one can also be considered a specific case in itself [2,14]. Additionally, the maximum and minimum can be considered [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 with a given probability distribution in how often each character is expected to be observed [33].

These can be seen as a special subset of the **value relation** pattern. A value relation describes how one variable is related to another [2,25]. These can be basic mathematical operations (e.g., x<y or x=y+1), or more complex mathematical functions [2]. Relations can also be described in terms of the relation of one variable to several others [25]. One example of this is the relation of program output to all of its (several) inputs [25]. In the case of larger sets of values (e.g., arrays), the same relations can be described internally between the elements of the set [2]. Additionally, a set of specific relations can be considered such as one set reversing another or matching a subset of a bigger set [2]. Additionally, a single value (e.g., a given variable or a constant) can be described to always be included in a given set [2].

F. Invariant Scope

The scope of an invariant defines where this invariant is expected to hold. The scope element of the taxonomy is shown in Figure 7. In the following descriptions, the term event is used to refer to both control-flow events and states and data-flow measures.

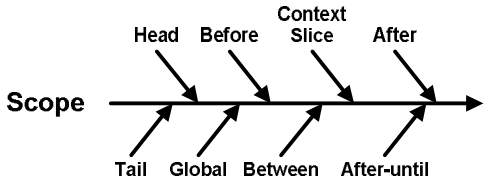

Figure 7. Invariant scope.

An invariant may define that it should hold **after** a given event [6]. Additionally, 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 (**after-until**) [6]. This is similar to the scope **between**, which defines two events in between which the invariant pattern should hold [6]. However, the difference is that this only holds once both the start and end events have been observed, and after-until holds from the first observation of the start event.

As opposed to the after scope, an invariant pattern can also be defined to only hold **before** a given event is observed [6]. A **global** invariant pattern should hold for all observed behaviour during the program execution [6]. Considering only the first N (**head**) or the last N (**tail**) observations of a set can also define a meaningful scope [2]. For example, the relations between the last 2 observations can define how a value in a set increments [2].

The scope can also be defined in combination with a specific slice of the program behaviour, such as a thread [41,27] or a specific web application session [33]. In this case the scope becomes a combination of the context **slice** and one of the other scope definitions discussed above.

IV. 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

 ISBN: 978-1-61208-111-3

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).

The discussion in this paper is from the generic viewpoint of using dynamic invariants. One important aspect to consider is how representative the available invariants 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.B. 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.B.

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 only 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 behavior, or completely irrelevant in the general context [2]. On the other hand, sometimes all observed behavior is important regardless of their probability. This can be, for example, behavior 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) [20].

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 called invariants. For example, in test automation the test oracle practically always needs to be described in terms of an invariant, 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 is required for adopting them as a concept more widely as many are not used to thinking in these terms.

Understanding and using invariants generally requires specific considerations for specific usage purposes. For example, one may refactor code based on suggestion from invariant analysis [32] 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 need to be considered.

V. CONCLUSIONS AND FUTURE WORK

Today, dynamic invariants are used for many points in software design and analysis. The invariants for different system 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 dynamic invariants in software behaviour and help in using them in different domains.

The presented taxonomy is based on six main facets, three related to processes of using the invariants and three related to the information describing the invariants themselves. The main focus was on describing the properties of the invariants themselves, and thus on the parts 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 dynamic invariants. 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.

Topics for future work include further exploring the different aspects of dynamic invariants and their relations to each other, such as scopes, patterns and measurements. Similarly, a deeper investigation of their relation to other formalizations of software behavior, such as those used in the formal methods community is seen as interesting. Applications of the taxonomy along with the further investigations are also needed for practical validation and evolution.

ACKNOWLEDGMENT

The author wishes to thank Ali Mesbah and Arie van Deursen for their helpful comments on this paper and on the taxonomy.

REFERENCES

[1] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transactions on Software Eng.*, 2009.

[2] 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.

[3] 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*, 2006, pp. 169-179.

[4] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective Statistical Debugging via Efficient Path Profiling," in *31st International Conference on Software Engineering*, 2009, pp. 34-44.

[5] A. Mesbah and A. van Deursen, "Invariant-Based Testing of Ajax User Interfaces," in *31st International Conference on Software Engineering*, 2009.

[6] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-State Verification," in *21st International*

 ISBN: 978-1-61208-111-3

Conference on Software Engineering, 1999, pp. 411-420.

[7] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic Generation of Software Behavioral Models," in *30th International Conference on Software Engineering*, 2008, pp. 501-510.

[8] 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.

[9] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient Mutation Testing by Checking Invariant Violations," in *18th Int'l. Symposium on Software Testing and Analysis*, 2009, pp. 69-80.

[10] R. Floyd, "Assessing Meaning to Programs," in *Symposium on Applied Mathematics, American Mathematical Society*, 1967, pp. 19-32.

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

[12] D. Lo and S. Khoo, "SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner," in *14th Int'l. Symposium on Foundations of Software Engineering*, 2006.

[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*, 2004.

[14] S. Hangal and M. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," in *24th International Conference on Software Engineering*, 2002.

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

[16] 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.

[17] 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.

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

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

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

[21] Johannes Henkel and Amer Diwan, "Discovering Algebraic Specifications from Java Classes," in *17th European Conference on Object-Oriented Programming*, 2003.

[22] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer, "A Comparative Study of Programmer-Written and Automatically Written Contracts," in *18th Int'l. Symposium on Software Testing and Analysis*, 2009.

[23] Jeremy W. Nimmer and Michael D. Ernst, "Invariant Inference for Static Checking: An Empirical Evaluation," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, 2002.

[24] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. Leino, and E. Poll, "An Overview of JML Tools and Applications," *International Journal in Software Tools for Technology Transfer*, vol. 7, pp. 212-232, 2004.

[25] S. McCamant and M. Ernst, "Early Identification of Incompatibilities in Multi-Component Upgrades," in *18th European Conference on Object-Oriented Programming*, 2004, pp. 440-464.

[26] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code," in *7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2009.

[27] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining Temporal API Rules from Imperfect Traces," in *28th International Conference on Software Engineering*, 2006.

[28] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic Inference and Enforcement of Kernel Data Structure Invariants," in *Annual Computer Security Applications Conference*, 2008, pp. 77-86.

[29] 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.

[30] M. Gabel and Z. Su, "Javert: Fully Automatic Mining of Temporal Properties from Dynamic Traces," in *16th Int'l. Symposium on Foundations of Software Engineering*, 2008.

[31] L. Mariani, S. Papagiannakis, and M. Pezzé, "Compatibility and Regression Testing of COTS-Component-Based Software," in *29th International Conference on Software Engineering*, 2007.

[32] Y. Kataoka, M. Ernst, W. Griswold, and D. Notkin, "Automated Support for Program Refactoring Using Invariants," in *International Conference on Software Maintenance*, 2001, pp. 736-743.

[33] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An Approach for the Anomaly-Based Detection of State Violations in Web Applications," in *10th Int'l. Symposium on Recent Advances in Intrusion Detection*, 2007.

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

[35] J. H. Perkins, G. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, M. Rinard, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, f. Sherwood, and S. Sidiroglou, "Automatically Patching Errors in Deployed Software," in *ACM SIGOPS 22nd Symposium on Operating System Principles*, 2009.

[36] A. M. Memon, "An Event-Flow Model of GUI-based Applications for Testing," *Journal of Software Testing, Verification and Reliability*, vol. 17, pp. 137-157, 2007.

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

[38] M. Harder, J. Mellen, and M. D. Ernst, "Improving Test Suites via Operational Abstraction," in *International Conference on Sofware Engineering*, 2003, pp. 60-71.

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

[40] C. Ackermann, M. Lindvall, and R. Cleaveland, "Recovering Views of Inter-System Interaction Behaviors," in *16th Working Conference on Reverse Engineering*, 2009, pp. 53-61.

[41] J. E. Cook and A. L. Wolf, "Discovering Models of Software Processes from Event-Based Data," *ACM Transactions on Software Engineering and Methodology*, vol. 7, pp. 215-249, 1998.

[42] N. Kuzmina, J. Paul, R. Gamboa, and J. Caldwell, "Extending Dynamic Constraint Detection with Disjunctive Constraints," in *Int'l. Workshop on Dynamic Analysis*, 2008.

[43] J. E. Cook and A. L. Wolf, "Event-Based Detection of Concurrency," in *6th Int'l. Symposium on Foundations of Software Engineering*, 1998, pp. 35-45.

[44] D. Lo, L. Mariani, and M. Pezze, "Automatic Steering of Behavioral Model Inference," , 2009, p. 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering.