SSAT DATCPI (FACTS)

Theoretical Soundness Issues Report

Deliverable 2

for

Contract NNL14AA07C

September 30, 2015

Revised: November 18, 2015

Revised: October 18, 2016

1. Introduction

From Exhibit A: Statement of Work, p 3:

> *4.1 Theoretical Soundness Issues*
>
> *The Contractor shall compile use cases for different classes of formal methods tools, identify the potential sources of errors in these tools, quantify these errors if possible, characterize the implementation with respect to the fundamental algorithms, and assess the impact of errors propagating downstream in the lifecycle. Because the tools have input and output interfaces where ambiguity and uncertainty may play a role in interpreting performance, the Contractor shall characterize the input and output interpretations and both static and dynamic metrics if reasonable. Where it is not reasonable or feasible to characterize interpretations or metrics, the Contractor shall provide a rationale. The Contractor shall deliver an informal report inclusive of the requirements of this paragraph and subparagraphs 4.1.1, 4.1.2 and 4.1.3 (Del 2).*

2. Theoretical vs Actual Soundness

In accordance with DO-333, the soundness criterion we will be examining in this report is the following:

> *A characteristic of a logical argument or method that holds if and only if it is impossible to reach a false conclusion from true statements. A sound method is one that never permits a property to be declared true when it is not true. (p. 58)*

Because they are formal artifacts with denotational semantics, and suitably small and abstract, systems of logic can often be formally proven to be sound (or not). The computer programs that implement formal analysis tools are also formal artifacts, but they are very large and concrete instances of programming languages with imperative semantics that may not be completely defined or predictable when compiled into a particular machine architecture and operating system. For this reason, it is usually infeasible to formally prove these tools sound. Instead, we must rely on informal proofs by the authors of the algorithms used, and trust that the implementation process is faithful to the description of the algorithms.

There is no way to conclusively verify these assumptions. Testing, by the implementers or the users of these tools, can only refute presumptions of soundness, by exposing concrete counter-examples. Because soundness is a property about what will always or never happen, on any possible execution of the tool, positive test results will never be conclusive. For this reason, vendor representations of soundness (with accompanying informal arguments) are the only practical evidence of actual soundness. In general, this is good evidence because vendors that make soundness claims will usually back their

claims with formal arguments, and will have suitably formal reputations. Vendors without such credentials will typically not make soundness claims.

Given the intervening human implementation process, it is unlikely that any such "theoretically" sound formal analysis tool is actually sound (as unlikely as that any large program has no bugs). So as a practical matter, it can be useful to rate analysis tools relative to a more informal notion of soundness that comes in degrees. One tool can be *more* sound than another if it yields fewer incorrect results. If there is some metric available to quantify the analysis domain, such as total number of statements or total number of memory references, then this relative notion of soundness could possibly be assessed against maximum possible correctness. Even here, though, such a metric would only be theoretical. To actually score an analysis tool, you would need an independent assessment of what is true, independent of the findings of the tool. And there is no practical way to obtain this except from another analysis tool.

Perhaps the best practical model for vetting the soundness of formal analysis tools comes from our experience with compilers. Compilers are programs, typically written in imperative languages, of similar size, complexity and architecture as analysis tools. They are written by similarly formally inclined developers who draw on decades of published research in formal language processing. Any given compiler probably has a few bugs, but in general, once it has been used for some time for production programs, it is one of the most trusted and reliable pieces of software out there. Only rookie programmers blame the failure of their own programs on compiler bugs. Seasoned professionals learn early on that this is the hypothesis of last resort. It is much more productive (and accurate) to assume that the compiler is never wrong. It is feasible, though very expensive and time consuming, to produce a provably correct compiler. The fact that no commercial organization has been willing to foot the bill for such an effort speaks to the lack of a market for such assurance (at such a cost). Compilers with good track records are judged to be good enough. Consumers have confidence in the low likelihood of compilers producing incorrect results. Formal analysis tools share this same formal provenance. What they typically don't share is the long track record of use on many millions of programs.

3. Soundness Issues for Model Checking

In this section we provide an overview of model checking approaches, tools, and the soundness issues that arise in using model checkers. The selection of model checker to qualify for certification requires care as the uses to which one can put those tools are limited by the underlying techniques employed. Therefore we begin in Section 3.1 with a review of model checking concepts and approaches, and continue in Section 3.2 with a discussion of typical properties that model checking is used to establish. While these sections attempt to provide a concise overview of model checking, they are also rather detailed owing to the breadth of model checking research and the many disparate model checking techniques and tools. Section 3.3 gives an overview of Java Pathfinder, the exemplar tool we are focusing on in the case-study portion of this effort. We then discuss in Section 3.4 the many possible sources of error that arise when using model checkers.

We conclude in Section 3.5 with a description of how these possible errors contribute to the soundness issues for model checking, and we discuss their potential downstream impact.

3.1 Model Checking

The concept of *model checking* as a method for automated verification has foundations in temporal logic and theorem proving. Fundamentally, model checking is algorithmic means of determining whether a (finite-state, abstract) model, typically representing a hardware or software system design, satisfies a formal specification of a desired (often temporal) property of behavior or state. Moreover, if the property is proven false (is reachable), most model-checking algorithms can identify a counterexample that shows a path of execution through the model to the error, often exposing the root of the problem. Model checking algorithms do this by systematically checking whether this property holds for that model by enumerating or symbolically executing all possible transition sequences through states of the model. However, even early experiments with automated model checkers showed that the number of states critically limited the scale of systems that could be analyzed usefully – "the state explosion problem". Moreover, Turing's and others work on the halting problem explains that we cannot have both sound *and* complete algorithmic solutions for many interesting systems. Even so, early model checking techniques, originally intended for verification of finite-state concurrent *software* systems, were adapted in other domains. Some of first widely practicable uses of model checking were applications in *hardware* verification. In hardware verification, the finite state restriction comes naturally. Regardless of the state-space problems, careful use of model checking techniques still provide tangible results in both hardware and software verification.

Heavily influenced by complementary research areas, modern *model checkers* bear little resemblance to the original concepts for verifying finite-state concurrent systems. Contemporaneously developed, model checking is attributed largely to Clarke and Emerson [1, 2], and Quielle and Sifakis [6], and their seminal work provided the foundations upon which all model-checkers are built. Extended and adapted to different domains over years of extensive research, software model checking use includes a panoply of techniques to address the fundamental state explosion problem: *symbolic model checking*, *partial-order reduction, data abstraction, symmetry reduction, bounded model checking and induction, compositional reasoning*, and even *direct or concolic execution exploration*. Individually, these techniques have found benefits in various domains. Combinations of these techniques offer a wide variety of model-checkers tuned for their specific domain, and expand the scope of automated techniques, both in terms of the scale of programs handled as well as the complexity of properties that can be checked. Additionally, more recent adaptations of "model checkers" include use of abstract interpretation and decision procedures, especially in the form of Satisfiability Modulo Theories (SMT) solvers. Taken as a whole, the field of *Software Model Checking* is a very broad umbrella covering a wide variety of tools and techniques.

Developing out of the logic and theorem proving community, most model checkers aim for both *completeness* and *soundness*, and excepting errors in design or implementation

of the tools themselves, both aims are generally achieved, at least for simple models. As the scale of the model increases, some tools and techniques compromise on soundness but maintain completeness, and conversely, some focus more on sound execution of the model and acquiesce on being able to explore the entire model. Additionally, different model checkers address the state explosion problem differently, with some geared towards property *falsification* ("bug-finding") and others towards *verification*. In the case of the former, a "positive" claim proves the existence of an error, often with a counter-example, but proves nothing if no claim can be made; conversely, verification approaches explore a superset of actual model executions, with positive claims about the model holding for all actual expansions of the model, but a violation of the property proves nothing about the original model. We will expand on different types of model-checkers later in this section.

With these restrictions in mind, it should be clear that proper use of model checkers requires not only a clear understanding of **what** questions one is trying to answer, but also of **how** the choice of model-checking tool and techniques affects the interpretation of the output of the model checker. This is especially true in the context of verification of safety-critical systems, including those to be certified under DO-178 or related guidelines. In all cases, the most obvious limitation in any claims that can be made from the results of a model checker is the ***model*** itself. Excepting limited recent research in binary-level model checkers, most tools which have roots in model checking do not operate on either source code or binary artifacts, but on code-fragments, abstractions of behavior, or design-level artifacts. Therefore as one applies model-checkers for certification of safety-critical systems, one always has to consider not only errors in the model itself, but also differences between the more abstract model being analyzed and the code- or machine-level representation. Conversely, this ability to evaluate a partial or abstract specification of the system behavior can sometimes be an advantage, helping to alleviate the state explosion problem, most often when analyzing design-level artifacts against requirements specifications.

While we cannot reasonably expect to cover even most interesting features of all model checkers in this overview, we will attempt to outline the broad classes of model checkers, typical use cases for model checking of software systems, as well as limitations of model checking with special emphasis on the usefulness in the realm of certification of safety-critical systems. As an exemplar, we will refer to Java PathFinder (JPF) and related extensions to JPF. Briefly, JPF started as a (executable) software model checker for Java bytecode, but unlike a normal Java VM, JPF identifies points in the bytecode from where execution could proceed differently (i.e. branch points), then JPF systematically explores all possible execution paths. Typical branching points include conditional (if-then) branching, scheduling sequences, and random values, but JPF has been augmented with different execution modes and extensions that include a variety of model-checking and symbolic-execution modules.

3.1.1 Model Checking Properties

The primary goal of any software model checker is to prove *properties* with respect to a model of execution. The model may come from many sources, but is generally a finite-state model refined from a higher-level design artifact or abstracted from lower-level

(implementation or) executable representation. Properties are typically (a) simple assertions, that state that a predicate over a finite-set of model variables holds whenever the execution of the model reaches a particular location (e.g., "*'x <= y' whenever in model state 's'*"), or (b) global invariants, where certain predicates will hold for all reachable states (e.g., "*mutex 'm' is never held by more than one thread simultaneously*"), or (c) reachability properties, that state that there exists some execution trace that will eventually reach a given state (e.g., "*for all possible inputs, eventually we reach state 't'*"). Model-checker properties are generally classified as either *safety*- or *liveness-properties*. Informally, safety properties claim that "bad things" *never* happen during any program execution, whereas liveness properties show that "good things" always *eventually* occur. With both the model and property, the simplest form of verification is performed by exhaustive state-space search of the *combined* model and (negation of the) property specification. Thereby the verification problem becomes a search problem, with the aim of finding reachability of any state satisfying the negated property; if found, the resulting path forms a counter-example to the property. However, other strategies exist, which we will elaborate upon in subsequent sections.

3.1.2 Explicit-State Model Checking

The simplest forms of model checkers effectively search the program space or model states and transitions, unified with a state-representation of the verification properties, using various graph traversal techniques. This method effectively enumerates all possible states and transitions of the program, thus the term *explicit-state model checking* is often applied. Other terms such as *enumerative* or *concrete* are applied as well, both as opposed to *symbolic*, with the latter manipulating sets of states simultaneously. Typical explicit state model checkers construct the state-space *on-the-fly*, hashing the set of states already visited for efficiency, and checking properties of interest after each state transition. Such techniques exploit the fact that the set of reachable states (for given inputs) is often much smaller than the entire state space of the model. Moreover, if a violation of a property is found, essentially a "bug" in the model, many model checkers will terminate immediately and produce a counter-example to the property. Notable exemplars of explicit-state model checkers include SPIN [12] and MURPHI [9].

The most fundamental flaw in explicit-state model checkers is that the expanded state space of a model can be exponentially larger than the description of that model. Known as the *state explosion* problem, it is the most tangible problem barring practical application of model checking. Given that most software is more dynamic and flexible than more structured hardware, software verification poses particular problems for model checking. Therefore, ameliorating state explosion has been a major direction of research in model checking, yielding many techniques, each with its own advantages and disadvantages.

3.1.3 Abstraction

Abstraction is the most common and essential technique for making verification assessments tractable, especially those that might otherwise contain unbounded data types or infinite state spaces. In this way, (abstract) model checking is still a reachability search problem, but the analysis is performed on an abstract domain that captures some, but not all, of the behaviors or state representation of the 'real' system. A well-chosen

abstraction is required to ensure that the abstract domain and semantics can be relied upon to produce sound results. That is, that a proof (or violation) in the abstract domain implies the verification (or bug) of the desired safety property in the original domain.

Abstractions are often made by observing the fact that design or specifications of systems usually involve much simpler relationships among state variables than is theoretically possible at the implementation level, and yet the most interesting cases for verification occur at the design level; for example, verifying concurrent systems to ensure the absence of deadlocks. Most abstractions are made to correspond to the design, using system requirements as the properties of interest. Moreover, one can often exclude elements of the design (or implementation) that are contextually irrelevant to the properties (requirements) that one is seeking to verify. In other words the context of execution matters a great deal for proving certain properties of models. In some situations, abstractions might be realized by a mapping between data values in the implementation and a smaller set of abstract data values that characterize the essence of the system in question. By extending the mapping to states and transitions, it is possible to produce a much smaller, abstract version of the models, rendering verification by model checking tractable.

While abstractions remain the most widely used technique to overcome the state-space problem for model checking, it can also be the largest source of errors. Specifically, as noted earlier, it is imperative that the abstraction retains the necessary elements to correctly represent the desired or eventual behavior. Moreover, any abstraction needs to be made conservatively such that all possible behaviors in the lower-level artifact are represented in the abstract domain, particularly if one is performing verification. Conversely, if one is only interested in falsification to find bugs, then the abstraction needs to retain sufficient resolution of the implementation details to render counter-examples useful. Either way, soundness of the analysis is a primary concern when utilizing abstractions, often with the burden falling to the human author of the model.

3.1.4 Reductionist Techniques

One of the most common uses of model checking is in the domain of concurrent software systems, where parallel threads of execution operate largely independently of one another. Errors in managing concurrency occur often in software systems, accounting for a non-trivial number of bugs in implementation; finding errors in such systems was a primary driver for the development of model checking itself. Reductionist techniques are a particularly effective, largely automated means to increase the scale of model checking.

By exploiting the independence of potential state transitions over non-intersecting state variables, *partial-order reduction* [7, 8] can ignore the order of these independent transitions. For example, if two transitions t_1 and t_2 can be executed in parallel threads and share no state variables, the final state reached after executing t_1 then t_2, is the same as that reached after executing first t_2 and then t_1. Only when the state-transition matrix overlaps in common variables does the order of execution matter. Partial-order reduction techniques maintain soundness while significantly reducing the size of the completely unrolled state-transition space.

Sy*mmetry reduction* [3,9,10] is another technique that may be employed against the state explosion problem. Similar to partial-order reduction, symmetry reduction is applicable most often in finite state concurrent systems which often contain repeated (sub-) components. Symmetry reduction finds the symmetries in the model and explores the state space of only one example of each class. Often the syntax of the modeling language can be used to determine the symmetries; e.g. one might define a model construct of an identical network process, that is then explicitly replicated, each communicating with the other. This sort of explicit symmetry can define an equivalence relation over the state space that retains the transition graph of each component while again reducing the size of the expanded state-space.

3.1.5 Unsound or Incomplete Approximations

Whereas most uses of reductionist techniques maintain completeness and soundness of the analysis, some uses of model-checking for *falsification* give up on either soundness or completeness of the search. Often this is achieved by placing artificial bounds on the amount of time or memory dedicated to the search, or by bounding the depth of the search. *Bitstate hashing* is a common technique in which the *hash* of each state is stored, rather than the actual state itself. The hashing function and size of the hash table determines the scale of the system the model checker can analyze, albeit unsoundly and incompletely. Bitstate hashing is both unsound and incomplete because two distinct states may hash to the same hash value (a hash collision). SPIN and other tools include options to turn on bitstate hashing, with appropriate warnings about soundness issues.

Bounded model checking [17] is another technique applied in symbolic model checking (see section 3.1.7) that trades completeness for effective checking of safety properties and bug finding. In bounded model checking, a (Boolean) formula is checked for satisfiability with respect to a finite-sequence of state transitions of length k. If reachability (satisfiability) cannot be proven, the search is continued for a larger k. This method remains symbolic in that a check for paths of length k covers all possible values of the variables encoded in the underlying representation of the model. If a solution from the solver is found, that assignment of values to the variables forms the basis for a counter-example. Typical searching is completed in a breadth-first manner, thus the counter example is guaranteed to be of minimal length.

3.1.6 Compositional Reasoning

Similarly to symmetry reduction, *compositional reasoning* exploits the ability to decompose the problem thereby reducing the scale of the state-space explored during verification of each of the subcomponents. In the case of compositional reasoning, we reduce the verification problem by divide-and-conquer, decomposing the original larger model into smaller sub-models such that the results of model checking sub-models can be combined to prove properties about the original model. Simplistically, if each of the sub-model properties are provable and the conjunction of those lower-level properties implies a higher-level specification, then the complete system must also satisfy that higher-level specification. Unfortunately, this simplistic reasoning may not be possible due to interdependencies between the components. For example, when attempting a verification of a property on one model, M_1, it may depend upon correctness assumptions about another model, M_2, and vice-versa. To satisfy these situations, *assume-guarantee*

reasoning [14,15,16] provides a framework in which, along with each model, M_1 and M_2, assumptions and guarantees ($\{A_1 , G_1\}$ and $\{A_2 , G_2\}$) about each model are provided such that, globally, the ($M_1 \, U \, M_2$) is satisfies a global property (guarantee) G. This requires a carefully defined assumption, A, about the environment of the $M_1 + M_2$ union such that ($A + G_2 + M_1$) ➔ A_2 and similarly, ($A + G_1 + M_2$) ➔ A_1. Then, the global property G can be shown to logically follow from (A, G_1, and G_2).

In theory these techniques may make tractable otherwise intractable model-checking problems. However, in practice, both compositional and assume-guarantee reasoning rely heavily upon a logical division of the larger model, as well as careful thought into how the results of each lower-level model verification may be soundly combined, if at all. While there is an active community of research into automatically generating or learning assumptions, current research has not yet simplified the use of A-G reasoning sufficiently to be regularly used in the context of safety critical systems for industry.

3.1.7 Symbolic Model Checking

While explicit-state techniques are the archetype of software model checking, *symbolic model checking* [21, 22], sometimes called *implicit model checking,* developed as an alternative to address the state-space explosion problem. In symbolic model checking, model-checkers manipulate representations of *sets* of states rather than *individual* states, and perform the state-space search by the transformation of these symbolic representations. Symbolic representations can be much more succinct than an explicit enumeration of all states, and can represent an infinite state-space. Moreover, checking of verification properties can often be done in polynomial time in the size of the BDD using underlying constraint solvers.

One common representation technique employs Binary Decision Diagrams (BDDs), an efficient means for representing and manipulating Boolean functions. BDDs are directed, acyclic graph structures that compactly represent the truth table of a Boolean function. BDDs may be a canonical form for representing a Boolean function given a fixed ordering for the variables of the function, and BDDs may be efficiently combined by Boolean operations into a new BDD. Moreover, transformation of the source model into BDDs may be performed **once**, and subsequent multiple property checks may be performed in constant time. However, building of the BDD becomes the bottleneck. It is provably exponential both in terms of time required to build, as well as in the size of the resulting BDD, both dependent on the number of variables to be represented, as well as the ordering of the variables.

While the worst-case time and size issues of BDDs may be stumbling blocks, BDDs have been key in scaling some model-checking problems, as well as representing infinite state spaces. The canonical tool using BDDs for symbolic model checking is SMV.

3.1.8 Execution Exploration

As a special case of explicit-state verification, *execution exploration* uses the runtime system of an executable to explore the state-space of the model. Java PathFinder is an example in this category. In this case, the model is typically source code, virtual machine byte code, or even (compiled) machine code. The execution environment is a specially

built "runtime" for (a subset of) the source language, a modified virtual machine, or a (modified) OS scheduler. In all cases, the non-determinism comes from two sources: (user) inputs from the environment, and branch-points and scheduler context switches from the code/model. In some cases the user inputs are held fixed (e.g. Verisoft [18]), and only the scheduler choices need be considered. In others, the user/environmental inputs may be considered symbolically (e.g. Java PathFinder [19, 20]), and a wider set of possible execution paths may be considered.

One of the primary benefits of such an approach is that the semantics of the modeling language *are* the semantics of the programming language itself; there is no translation gap between the design and what is being executed in the form of code. Additionally, when a counter-example is generated, a concrete execution demonstrating the failed property (i.e. a bug) can be directly provided to the user. In some cases, the user may be a developer of the code rather than a verification expert, thus bringing the power of model checking to a wider audience.

However, because we are using the execution environment for state space exploration, now the state of execution may necessarily include additional variables, such as machine registers, the heap and stack, as well as other aspects of machine execution that can be avoided when performing verification using a special purpose modeling language with unique semantics.

Some tools (e.g. Verisoft) make a space-time tradeoff to avoid the overhead of storing this additional state information by performing the execution in a stateless manner; that is, they do not keep track of the set of visited states. Rather, each time a choice-point due to branching or non-determinism is reached, a fixed scheduler is called to provide the next value to be used to determine the path to be examined. Since the scheduler controls all context switches, all non-determinism can be made deterministic in that the modified scheduler can ensure that all possible choices are eventually explored. However, this can come at the expense of re-executing the same path twice, thus taking more time in the verification.

In contrast, Java PathFinder (JPF) stores visited states, which means it more closely resembles an explicit-state model checker, and can take advantage of other reductionist-based techniques to counter the state-space problem. Additionally adaptations built on top of JPF allow one to perform *symbolic execution* to control the search to expand the coverage, while others allow the driver to be specially crafted so that one may analyze subsets of the code or code fragments.

In addition to Verisoft and Java PathFinder, other tools in this category include CMC, MaceMC, and Chess. While the (modified) execution environment is usually considered a sound implementation, except for the smallest of code samples, many tools of this category are most often used in testing frameworks. Some tools in this category, JPF included, include sufficient features to be used both for bug finding and verification.

3.1.9 k-Induction

Model checking by k-induction [23] seeks to establish that invariants or safety properties hold for infinite transition systems. The idea is based upon the realization that all infinite paths on finite state transition systems must eventually revisit a state (i.e., there is a loop). If the model checker can establish that the property holds for each state along a k state path and the k+1 state is already on the path, then the property will hold forever. Model checkers using k-induction work by incrementing k until no further loop free k state paths can be found. If the property holds for each loop free path of length k or less, then it is guaranteed to hold forever. In this way, model checkers can prove safety for infinite transition systems with a finite amount of search.

Like most other approaches to model checking, k-induction falls victim to the state explosion problem, and can be incomplete. As k increases, there are more possible paths of length k. If the model checker cannot increase k to the point where it identifies all loop free paths, then it cannot establish the safety property for the system.

3.2 Typical Use Cases

The typical use case for model checking is to prove temporal properties about models of system behavior. Most often applied to concurrent systems models, the properties of interest frequently include detection of deadlocks, buffer overflows, data access or race conditions, and termination or reachability properties. The input models are typically refined from a higher-level design artifact, but can also be abstracted from lower-level (implementation or) executable representation, and properties to be proven are usually derived from the software system requirements.

All uses cases of model checking can be broken into two categories: invariant ("safety") and reachability ("liveness") properties. These properties are typically represented via a temporal logic expression. In addition to each model-checking tool having its own unique language and semantics, the temporal logic specifications come in different flavors as well. Most are based upon either Linear Temporal Logic (LTL) or Computational Tree Logic (CTL), both of which have been shown to be a subset of CTL*. LTL specifies properties of paths, CTL specifies properties that quantify over paths, and CTL* combines the two. While there are some expressions that are not representable in LTL but can be stated in CTL and vice-versa, in general LTL is easier for less versed users to specify and understand. Regardless, most property specification languages include the ability to specify propositional logical expressions augmented with temporal operators including Globally/Always, Finally/Exists (eventually), Next-State, Until, Weak-Until/Release. Some logics augment these with bounded-time operations, probabilistic operators, or other operators.

3.2.1 "Safety" Properties

Informally, safety properties prove that "nothing bad ever happens." Concurrent systems examples of safety properties include mutual exclusion (e.g., at most one process is executing with a critical section), freedom from deadlocks (e.g., it is never the case that all processes are waiting for each other to provide/release a resource), or a specified data-state (e.g., it is never the case that the buffer is full, or the value of a variable is less than

zero). All of these are *invariants*: they mean that in every state of the model, it should always be the case that the property holds. Other forms of safety properties are not invariants, and include causal or temporally causal relationships (e.g., the flow will remain above "x" until the valve is closed).

Typically, model checkers require the property be specified in some form of temporal logic. As stated earlier, most model checkers use a form of LTL or CTL to specify properties. Simple example of LTL safety properties:

`G ( "x" >= 0 )` – it is always true that "x" remains positive.

`G ( ~ s )` – it never the case that we reach state "s".

This temporal logic expressions are typically negated and merged with the model specification, where reachability of the "property" state is sufficient to show that the original property is violated, resulting in a direct counter example to the property.

3.2.2 "Liveness" Properties

In contrast to safety properties, liveness properties require some progress, or reachability of an (eventual) state, such as termination. Intuitively, they state that "something good" will always, eventually happen in some future state of the model. Whereas safety properties can be shown to be violated by finite traces (by a finite state counter-example), liveness properties may be infinite. Eventual reachability is the standard example of a liveness property (e.g.it is always the case that once a mutex is acquired, eventually it is released), with execution termination reachability being a special case.

Other more troublesome liveness properties involve infinite or repeated executions (e.g., each process will acquire the mutex infinitely often) or fairness (e.g., eventually every process will acquire the mutex). The difficulty with proving liveness properties with model checkers comes when one is unable to reduce the (negation of) property specification to (infinite) reachability of a 'bad' state from an initial state. However, in practice some form of temporal logic can still represent most interesting liveness properties, and is thereby used by the model checker to prove the property or to provide a (infinite state, often recursive) counter example that shows the inability to make progress or "reach" the desired state.

Two LTL-style properties that exemplify liveness properties include:

`GF ( t )` – always we eventually terminate (reach state "t").

`G ( x -> F y )` – after reaching state "x", we eventually reach state "y".

3.3 Java PathFinder and Symbolic PathFinder

Before proceeding to potential sources of errors, we should briefly expand our earlier description of our exemplar tool-set: Java PathFinder. Java PathFinder (JPF) is an extensible framework for verification of Java bytecode. While JPF began as an explicit-state model checker, it has evolved until it is now more of a framework within which

multiple plug-ins exist to perform different forms of model-checking related analyses. JPF-core is still an explicit-state model checker capable of automated detection of deadlocks, data race conditions, and assertion violations (equivalent to safety properties). The core also uses typical explicit-state scalability techniques including on-the-fly exploration and partial order reductions. JPF has been extended to include a symbolic execution plug-in, Symbolic PathFinder (SPF), that combines symbolic execution, model checking and constraint solving to better handle dynamic inputs, loops, and recursion. In addition, SPF has been used to generate test cases that can guarantee coverage of all possible paths through the code relative to a set of (potentially symbolic) inputs. We selected JPF as our exemplar because of the broad support for different model checking techniques, independent of the JPF's reliance on models written in Java. Even though Java is not the language of choice for many safety-critical systems, the feature set of JPF is broad enough to provide an overview of different model checking options.

Model checkers fall into a taxonomy that can be divided along several dimensions:

- System Model Type: JPF and many other model checkers implicitly represent the model as a set of rules that can be applied to a system state to determine the next state. Explicit state model checkers explicitly represent the model as a state transition system. In comparison to explicit representations, implicit representations require more reasoning to explore execution traces, but can be more compact and easy to model. For example, JPF models are, arguably, easy to create because they use a high-level programming language (Java) instead of a explicit representation of each JVM state for a particular program. However, JPF requires a JVM to generate the state transitions.
- Property Specification Type: Property specification types align with the major languages, including linear temporal logic (LTL), and computation tree logic (CTL). LTL encodes state sequence properties, and CTL encodes properties of sets of state sequences. JPF natively supports a very simple LTL safety property that no uncaught exception is thrown. However, JPF can be extended with custom property listeners that will in principle support any LTL and CTL properties. Another aspect of properties is whether they are over finite or infinite state sequences. Many algorithms require finite properties, which can also improve performance because the bound on the length of the state sequence limits search. While JPF is not specifically limited to finite properties, its search uses a depth limit and can only make completeness claims with respect to a finite property.
- Model Checking Algorithm: Nearly all algorithms are search-based, but differ in the format of their search space. Symbolic model checkers based upon SAT or SMT typically search in the space of assignments to logical variables. BDD-based model checkers perform breadth-first search in the space of symbolic states. Explicit state and execution exploration model checkers (including JPF) search in the space of states. SPF combines explicit state search with search over logical variable assignments.
- Output Types: Most model checkers will state which properties are satisfied and provide counter examples for violated properties. Model checkers vary in the

depth of the reporting. Counter examples can range from a single raw trace to a set of traces to a summarization of the traces as a regular expression or fault tree. Local model checkers will report whether properties are satisfied for a single state and global model checkers will report which of the states satisfy the property. Model checkers can report just the output or explanations of how the algorithm or parameters effect its soundness and completeness. They can also be anytime, reporting counter-examples as they are found. JPF reports where properties are violated in a program, along with stack traces (where applicable) and execution traces. SPF creates unit tests and path conditions for the Java program that test each path of execution.

3.4 Potential Sources of Errors and Errors of Interpretation

While model checking is typically slower but more precise than static analysis, the single largest source of error in model checking are the model and properties themselves. Regardless of the source of error, we can classify the types of errors into two camps: *False Negatives* (real bugs in the model that are not reported) and *False Positives* (reported bugs that do not actually exist). The latter are usually deemed somewhat "acceptable" in that reporting of an error that does not exist encourages further examination, often resulting in modification of the model or property to exclude false behaviors. On the other hand, too many false positives and the tool may become useless. In contrast, false negatives are the most important class of error in the use of a model checker, in that they represent true errors in the model that are undetected and unreported.

Earlier in this text, we described potential sources of errors in the context of each type of model checking technique. Below we briefly describe the most common sources of false positives in model checking.

3.4.1 Modeling Errors

As stated earlier, the modeling language is often unique to the model checker, and these languages have their own specification and semantics that are necessarily different from machine-level specifications. Because the models are often written by hand, either extracted from a lower-level representation (code) or higher-level (design) artifacts, one may inadvertently exclude potential behaviors. The challenge is always to assure oneself that the model being analyzed is a true and accurate representation of the desired system behavior. However, since many analyses performed using model checkers are performed on system *design* artifacts, the simplicity of the design relative to code is often sufficient to render the models "correct by observation". While JPF operates on Java bytecode, it is still the case that the Java "model" under assessment is not usually the actual executable code due to abstractions that must be made. Therefore it is subject to the same concerns about "model" correctness. Model checking that is performed on the actual binary to be executed can help avoid this error source. However, even then the model checker must rely on a model of the underlying computational engine, which can itself be flawed.

3.4.2 Testing Environment

Closely related to errors of modeling, errors in the testing environment can result in false negatives. Since the model itself may be a partial specification, it is often necessary to

define a "test harness" or stubs for other behaviors. This is often required to provide (partial) behavior for "code" that either does not exist in the model, or if it were included, the model checker would be unable to complete exhaustive verification due to the expanded state-space of the model. These stubs necessarily provide only partial implementation of the true behavior of the eventual system. Therefore it may be possible that not all paths through the model being analyzed are properly checked, again leading to false negatives. In JPF, stubbed behaviors are often used in place of calls to system and library functions. This is also a major challenge for SPF because it must somehow represent the stubbed behavior as constraints (whereas the constraints needed to express executed byte-code are already defined by SPF's encoding of the byte code instruction semantics).

3.4.3 Property Specification Errors

While the modeling languages for most model checkers are usually fairly similar to other programming languages, property specifications are much more complex, especially for the naïve user of a model checker. Both LTL and CTL, common languages for property specifications, require a great deal of care to ensure that properties written are both a correct representation of the requirement being checked and not trivially true given the model being analyzed. A very common error in use of temporal logic is to mistake logical implication with causal implication; i.e. the propositional implies (→) is not a temporal operator. Consider the simple $p \rightarrow q$ expression, which is *logically* equivalent to $(\sim p \,||\, q)$, which can be satisfied simply by never "reaching" state p. Clearly, this is unlikely to be the intended behavior, and could lead to the tool reporting no errors when in fact one may exist. Commonly used formalisms for temporal logic often challenge the human analyst's intuition, which can adversely impact the validity of verification efforts.

Many tools have default or "basic" properties pre-encoded (e.g. deadlock detection), and others, JPF included, allow one to directly represent some classes of safety properties directly into the model by means of much more intuitive assertion statements. However, these can be insufficient to fully capture all potential behaviors of interest, so we must still rely on more complicated temporal logic languages.

Finally, it is up to the user of the model checker to assure they have a comprehensive set of requirements and that they have been faithfully represented in property specifications. Failure to do so will again result in false-negatives and likely a false sense of correctness of the model.

3.4.4 Incomplete Results and Tool Configuration Errors

While not strictly a case of false negatives, it is often the case that insufficient resources are provided to the model checker to fully analyze the model. It may be impossible to provide sufficient time or memory to exhaustively explore the entire space of the model. In other cases, configuration of the tool may allow one to place arbitrary bounds on the analysis (e.g., maximum search depth), or enable unsound abstractions (e.g., bitstate hashing). For instance, JPF includes bounds on its search depth and program stack size. SPF also requires bounds on each numeric program variable because it encodes and solves the program path constraints with an SMT solver. In general, SPF is often limited by the capabilities of its SMT solver. While most tools will often report when results are

incomplete or unsound analysis options have been enabled, the potential lack of error reports (false negatives) can lead to an incorrect belief that the model is correct.

3.4.5 Sub-Solver Limitations

Many model checkers use specialized sub-solvers as subroutines in their model checking algorithms. It is frequently the case that the sub-solvers have limitations. For example, as previously noted, SPF uses an SMT solver as part of symbolic execution. Depending on the specific SMT solver, SPF is not capable of reasoning about non-linear branch conditions, some string operations, or complex data types (e.g., trees, heaps, or queues). When the sub-solver cannot deal with the full complexity of the model, common approaches will either abstract or ignore these aspects the model. In other cases, the model checker might use a more capable, but incomplete or unsound sub-solver and thus inherit incompleteness and unsoundness.

3.5 Downstream Impacts

As described above, model checking is often an exercise in analysis of design-level artifacts. Numerous studies have shown that errors found and corrected at design time result in many fewer bugs and less rework when it comes to implementation. Unfortunately, model checking is not a silver bullet in this regard as the model and properties are as subject to "bugs" as the eventual implementation. Automated model extraction or using a model checker that operates on a low-level implementation can reduce the risk of errors in modeling. Similarly, providing pre-defined properties or tools that assist the analyst in specifying (temporal) properties can reduce that source of errors as well. Finally, overcoming the state-space problem via abstractions and test harnesses is a common source of errors.

Secondarily, the choice of model checker tool and which features to enable in the tool are highly contextual to the analysis to be performed. The old adage, "a fool with a tool is still a fool" is no less apt in this regard. Misinterpretation of the lack of an error report (false negative) with a misconfigured tool is common.

While modeling and property specifications are somewhat challenging, model checking is still one of the best automated techniques for finding errors, especially those involving concurrency and inter-process communications.

4. Soundness Issues for Static Analysis

4.1 Static Analysis

The term *static analysis* is often used to classify analyzers in contrast to *dynamic analysis*. Informally, dynamic analyzers examine the runtime behavior of a program as it is executing (typically with instrumentation). Statics analyzers, it is said, don't execute the program, but rather infer properties of interest from the syntactic form of the program alone. This is perhaps not the best choice of terms because some static analyzers, e.g.,

model checkers and abstract interpreters, achieve their results by *symbolic* execution or interpretation. For these tools, it is not a static/dynamic distinction (execute/don't execute), but an abstract/concrete distinction in how the execution is performed. A more inclusive criterion for static analyzers would be that they analyze programs in the absence of actual inputs. Such analyzers necessarily must consider all execution paths.

The vast majority of static analysis tools do not claim to be sound, because their analyses are based on recognition of syntactic patterns in the program that are correlated with errors. This is somewhat like statistical studies in science that find significant correlations between two classes of events. The correlation doesn't establish that either event is the *cause* of the other, only that they are likely to co-occur. One class of events can be seen as a "risk factor" for the other. And because the underlying mechanism of cause is not known, diagnoses based on these factors can be expected to yield some incorrect results. This is not a flaw in the technique; it is level of precision that is known not to be obtainable. These can still be useful diagnoses, on average, depending on the strength of the correlation, even though the method is known not to be sound.

Formal methods analyzers, by contract, usually base their conclusions on the formal semantics of a program – a rigorous theory of the underlying causes – and reach their conclusions using some form of formal proof over these semantics. This is where the confidence for a claim of soundness originates. There is, in general, no practical ambiguity in the expectation of soundness. The authors of the tools know whether or not they have a formal basis for a soundness claim. Mistakes can be made in implementation, but no one supposes a tool *might* be sound without some upstream proof of this. There is no such thing as lucky.

We will be concerned with static analyzers in this last category, the ones with a credible claim of soundness. Sound analyzers are of particular value to developers of safety critical systems because they can definitively establish that some event will or will not occur under any possible execution of the system. If evaluators could truly rely on this soundness, they would not bother testing for these events. It would be unnecessary, redundant. But because theoretically sound analyzers are fallibly implemented by humans, and play only point roles in an otherwise unsound lifecycle of development and verification, this report examines issues that evaluators should be aware of in the practical application of these tools, even if we grant their theoretically soundness.

4.2 Abstract Interpretation

Perhaps the best-known category of sound static analysis is *abstract interpretation*. The theory was originally developed by Patrick and Radia Cousot in the 1970s [4]. This method attempts to get around the infeasibility of proving properties over all possible (potentially infinite) executions of a program by finding a suitably abstract, over-approximation of all program behaviors (that *is* feasible to compute) over which the property also holds. Since the over-approximation of behaviors contains all actual behaviors as a subset, the proof extends to all actual behaviors.

The over-approximation is necessarily imprecise, so for some abstract results, a property that holds for all actual behaviors may not hold for the over-approximation. There is typically a trade-off between precision and computing resources. More precision (better chance of containing proofs) requires more resources. The trick is to find an abstraction precise enough for a proof that it is still feasible to compute.

Instead of using actual input values, the method interprets the program over sets of its possible input values. Abstract data value domains are used to constrain the possible values in these sets. So a program that takes integers as inputs, for examples, might be interpreted over the domain of integer *intervals* instead. Each instruction in the program that operates on integers will be abstracted to one that operates on intervals. A subtraction instruction ($A := B - C$), for instance, would yield $A = [-4,4]$ if $B = [1,7]$ and $C = [3,5]$. The interpreter walks the braches in the program's control flow graph breadth-first, propagating all data flows in parallel. Predicates on conditional branches *narrow* the population of the abstract data sets by constraining the possible values that flow through to their true and false branches. Cross edges and back edges that join multiple paths *widen* these abstract sets by forming the union of sets of the incoming branches. Loops must be iterated to a fixpoint, which typically widens the sets due to back edges.

The aim is to use an abstract domain that is precise enough to prove the property of interest. After the interpretation is finished, each <variable, location> pair of interest will have an abstract invariant expressing an over-approximation of the actual values at that point (for all executions). If the goal is to prove that an array reference can never be out of bounds, for instance, we will attempt to prove that the interval invariant on the index variable is contained within the declared upper and lower bounds of the array. It doesn't matter that the invariant may contain values that will never occur, as long as they are all contained within the bounds.

4.3 CodeHawk Technology

We will be using Kestrel Technology's CodeHawk abstract interpretation technology as our reference model for this study. At present, CodeHawk is a technology, not a particular tool. The core of the technology consists of a number of abstract domains and iterators that can be plugged into a graph propagation engine. It operates on an abstract program defined in CHIF, a customizable intermediate representation of typical imperative programming language operators. The general technology has a modular architecture that is designed to be specialized to a particular concrete programming language, a particular set of conjectures to be proved, and a set of domains over which to attempt the proofs. It becomes a particular tool by adding a front-end translator from a concrete programming language into CHIF, a specialized translator for embedding the conjectures to be proved into the CHIF translation, and the selection of domains and interpretation strategies.

Kestrel currently has front-ends for C, Java, and x86 binary. We have built several specializations of these language-specific analyzers to solve problems in formal methods research contracts. It may take some initial experimentation to find the best mix of precision and computing resources to solve a particular problem. For example, we

typically start with very coarse-grained domains such as arithmetic intervals, which are inexpensive to compute, to see how many of the proof obligations can be discharged at low cost. If these are too imprecise to discharge some proofs of interest, we move up to more expensive, precise domains. Sometimes very computationally expensive domains that are not feasible to be run on the entire program can be brought to bear on just the specific parts of the program that require more precision.

4.4 Typical Use Cases

The generic use case for abstract interpretation is to prove a conjecture about whether a module of software will never, will always, or may execute some behavior of interest. This can be a conjecture about behavior or intermediate data values at specific program locations, or about all possible locations. The typical use cases for CodeHawk, so far, have revolved around verifying cyber security properties of software.

4.4.1 Memory Safety

Our most substantial use case for the C version of CodeHawk, so far, has been a DHS project for proving the memory safety of C applications. This was a comprehensive study covering every location in the software at which an undefined memory access is possible according to the C language standard. Undefined memory access covers all of the categories of overflow, underflow, array bounds, null pointers, undefined pointers, string buffer overlaps – for reading and writing. Since the C compiler cannot guarantee such accesses will not occur for certain types of references, C programs with such exposures are potentially vulnerable to exploitation and attack.

Abstract interpretation is used, in the first instance, to attempt to prove that a program has no such vulnerabilities, by attempting to prove that the computed abstract invariants on data values, at every reference in the program, satisfy all declared bounds and initialization preconditions. Using computationally efficient, yet reasonably precise domains, it is typically possible to automatically prove around 80% of these conjectures – i.e., the computed over-approximation is proved safe, so the actual values must be as well. In a large portion of the remaining cases, the unsafe ranges in the over-approximation are not contained in the subset of actual behaviors, but the approximation is too imprecise to determine this. By adding lemmas, and using more precise domains selectively, a human analyst can usually prove that the remaining 20% of references are definitely safe or definitely vulnerable.

Because of the human requirement, the cost of proving this last 20% is relatively high. The goal of the DHS project, called Gold Standard, was to create an objective benchmark against which to score arbitrary C static analyzers by performing this exhaustive, human assisted analysis on 6 large, open source C applications that represent a cross-section of styles, features, and architectures. Since these 6 exhaustive analyses would contain no false positives and no false negatives, other analyzers could be objectively scored by comparing their results on the 6 applications against the standard results, for both soundness and completeness.

Proving memory safety in C programs is a perhaps the most typical use case for all C abstract interpreters. The Gold Standard project illustrates two variations on this theme: automatically proving safety conjectures, at low cost, for large portions of many programs (a large gain in coverage and confidence over other verification methods); and exhaustively proving these conjectures, at great expense, over a few programs (achieving soundness and completeness for a reference standard).

4.4.2 Value Flow

A use case for the Java version of CodeHawk is illustrated by its role in the DARPA Stone Soup program. In this case, it was paired with a dynamic analyzer for finding and fixing exposures in Java programs to SQL injection attacks. The 'taint' domain was used to exhaustively trace the dataflow from external sources (sources of possibly tainted, or malicious, SQL strings) to internal SQL calls. Such call locations whose abstract input invariants include the taint value need to be remediated by editing or type checking guards to ensure that SQL will not be called with pernicious strings.

In this case, the taint domain represents any binary domain (taint, no-taint; *A*, not *A*), so the abstract interpreter is just exploiting its breadth first propagation of values, rather than its abstract computation of derived abstract values (as would be the case with numeric intervals, for instance). Taint never narrows to no-taint. The only abstract value derivation is widening no-taint to taint upon convergence of at least one tainted path.

4.4.3 Binary Analysis

Use cases for the binary (x86) version of CodeHawk are interestingly different than those for higher-level languages. In 3GLs, like C and Java, variables are defined and referenced symbolically, as are program locations that are the targets of control flow constructs (such as **do** and **goto**). This allows the compilers for these languages to restrict programmer access to the actual binary addresses of the resulting machine-language program (for reading, writing and branching). The compiler provides a reasonable guarantee that the translated binary-level accesses will conform to the symbolic declarations of the input program. For this reason, the front-ends that translate 3GL programs into CodeHawk's abstract CHIF form can rely on the symbolic indirection of the programming language semantics to produce an equivalent abstract program (modulo abstraction). A reasonably well-defined concrete program is translated into an equivalently well-defined abstract one.

In the case of binary programs, all bets are off. Variables and non-contiguous control transfers may have been implemented or changed by programmers, even if first compiled. The CHIF front-end must first *discover* variables and branch targets and distinguish between bytes that implement instructions vs. data vs. padding. For this reason, the first, essential use case for binary abstract interpretation is reverse engineering – discovering the semblance of a concrete program with stack and heap variables in addition to the registers, that can be turned into an abstract one. So the CHIF translation process itself is an iterated series of abstract interpretations that progressively discover the location addresses of variables and branch targets, by propagating address and offset constants through the known variables and targets from the previous iteration. When this process

converges, we are in a position to do something like the memory vulnerability analyses of the higher-level languages.

An important caveat, however, is that proof is almost always unattainable. This is because we are rarely in a position where all data and control addresses have been resolved to constant locations. If the resulting CHIF program has just *one* unknown read, write or transfer target remaining, this could possibly be *any* location in the program (or out of the program). So we are always dealing with an approximation of a program rather than an actual one. For this to be a safe approximation, we would have to assume the worst case (that these unknown targets could be all possible addresses), which would make all of the computed abstract invariants too coarse-grained to be useful. To get meaningful vulnerability results, we have to make unprovable assumptions, so this use of the abstract interpreter cannot be sound.

4.4.4 Range Bounds for Digital Filters

A use case of CodeHawk that is directly relevant to the aerospace domain is its role in a AFRL project to verify the output range bounds of digital filters. A specialized version of the C CodeHawk analyzer has been implemented to analyze C functions implementing such filters, given some additional constraints about the filter implementation and intended numerical constraints on its inputs. These input assumptions, and the C filter code, are generated from formal models of the filters by Honeywell's HiLiTE tools. A custom integration pathway allows these input constraints, and desired conjectures from the model about the filter's output ranges, to be piped directly into CodeHawk. A new domain of reals specialized to their floating point implementations has been added to CodeHawk so that floating-point rounding effects can be accounted for in the analysis. CodeHawk then computes abstract invariants over the output values and attempts to prove the output range conjectures.

This is a typical case where the generic CodeHawk technology is specialized to a particular analysis problem, but represents an atypical implementation. Usually, the conjectures to be proven, the abstract domains and iteration strategies to be used, and any custom assertions about the input programs are hard coded into a new CodeHawk front-end by Kestrel. In this case, we parameterized a more generic front-end to take custom constraints and conjectures from Honeywell in a special language for the filter domain, implemented as a JSON input file.

4.5 Potential Sources of Errors under Use Cases

With the exception of the reverse engineering of binaries use case, CodeHawk performs a (theoretically) sound analysis, which means we expect no errors in the reported results. But since each use case specializes the CodeHawk framework to a specific analysis problem, tool users are not always aware of the assumptions and boundaries of the specific analysis. This can lead to erroneous assumptions and expectations about the results.

4.5.1 Memory Safety

In the memory safety use case for C programs, *all* program locations involving memory access are subjected to a proof of memory safety using the computed (abstract) invariants at each location. Because the analysis is sound, all locations proved safe, are safe. The remaining locations, however, are not necessarily unsafe. Many of them may be safe as well. They just cannot be proven to be safe with the precision of the abstract domains used for analysis. If this unproven class of locations is large, it can create the erroneous expectation that the analyzed program has lots of bugs that must be fixed. This can lead to a lot of unproductive human attention. Such a large class may be due to usage by the program of characteristics that require more precise domains. A more productive approach might be to focus the analyzer on just the remaining cases using a more expensive, precise domain. The unproven locations are not incorrect results; they are simply unknowns, reflecting the incompleteness of the analyzer with respect to safety.

From the converse perspective, the analyzer will be complete with respect to *unsafe* locations (all unsafe locations are included in the set of unproven locations). A user of the analyzer can rely on its soundness to exclude all the safe locations from further scrutiny for *safe memory access*. But failure to appreciate how the class of safe memory access is related to all possible classes of program errors can lead to overconfidence in the results – assuming that no errors (of any kind) remain in the safe memory locations.

4.5.2 Value Flow

In this use case, the abstract interpreter has been paired with a separate, dynamic analysis tool to identify and remediate all vulnerabilities to SQL injection attacks in a program. The dynamic analyzer determines the possible sources and sinks for such attacks within the program. The abstract interpreter statically computes the flow of taint from source locations to sink locations. The dynamic tool then remediates the sink locations that may receive tainted data flow. The soundness of the abstract interpretation analysis does not carry over to the dynamic analyzer, so the original source and sink candidates may possibly have been over-estimated or under-estimated. Under-estimation will result in not all vulnerabilities being fixed. Over-estimation will lead to some non-vulnerable locations being fixed. The abstract interpreter can also contribute to the over-estimation problem because the taint domain is essentially Boolean (tainted, not-tainted). Widening at joining branched in the control graph will always take the union of the abstract values on its incoming branches. So taint and no-taint widen to taint. This safe approximation is always vulnerable to worst-case results.

4.5.3 Binary Analysis

In the analysis of binary programs use case, there is always an exposure to unsound results due to the front-end translation of the binary program. Sound analysis is predicated on a well-defined input program, but an initial disassembly of a binary program can only approximate these semantics. Even in well behaved programs, a given binary machine word can be interpreted as either data or an instruction. A first approximation of this difference must be used to refine subsequent interpretations. So the abstract interpretation may ultimately be carried out over an abstract program that does not fully correspond to the input program.

4.5.4 Range Bounds for Digital Filters

The specialized CodeHawk analyzer for digital filters is an atypical use case in that it uses assumptions from an external formal tool (Honeywell's HiLiTE) as an essential part of the abstract interpretation process. Filters are a class of numerical algorithms for which the normal widening process of abstract interpretation is not particularly useful. Generic, conservative bounds used in widening will generally yield bounds on the filter's output that are much wider than known analytic results that can be derived mathematically from the formal specification of the filter's class (the recurrence relation and coefficients). Accordingly, the analysis is specialized to particular classes of filters, and exploits the known analytic results for the class during the widening and convergence process.

A sound analysis, in this case, is thus predicated on correct information being supplied from the outside. The collection of filter class characteristics are provided only once, and become part of the implementation when the analyzer is built, so these are generally well vetted. At runtime, however, the call to analyze a particular filter instance will be accompanied by instance-specific filter parameters (including its algorithmic class) and presumed bounds on the inputs to the filter. CodeHawk will assume these values to be true, and the correctness of the computed (abstract) output bounds on the filter will be conditional on this correctness assumption. Since, in this particular use case, the input bounds and parameters are automatically generated by the upstream HiLiTE tool, the chance of these being incorrect is acceptably low.

A more pervasive source of error in the abstract interpretation of numerical algorithms comes from the analyses typically being performed over the mathematically ideal domains of real or rational numbers. The C code that implements the algorithms will be employing floating-point numbers, which will have the capacity for precision and round-off errors not present in the reals. So analytic results computed over the reals may not always hold for their corresponding floats. To cover this possibility, this particular project also does an abstract interpretation of the filters over an abstract domain of floating-point intervals. The computed bounds from the real analysis are then compared to those for the floating-point analysis to expose possible error bounds.

4.6 Implementation and Interpretation Issues

4.6.1 Interpretation

As we have described above, CodeHawk is an abstract interpretation *technology* rather than a specific analysis tool. The core technology is agnostic about the language of the programs it analyzes and the sorts of conjectures one attempts to prove. It accepts an abstract program, and computes abstract invariants over variables at every program location requested. To use this technology to prove particular program properties, the technology must have a front-end translator to parse the program's language and abstract relevant executable statements into CodeHawk's internal operations (CHIF) that expose the variables of the conjectured property, a specification of iterators and domains to be used, and a back-end module that attempts to prove the conjectured property using the computed invariants. The soundness of the over-approximation of the variables' invariants that abstract interpretation provides is separate from the soundness of the front-end translator and the back-end prover.

Parts of the front and back ends can be implemented once and reused -- for instance, the parser for each programming language, and the general-purpose proof procedures. These modules can contain errors, but because they implement known, standard algorithms, and are used (and thus tested) many times, confidence in their soundness is high. But other portions of these surrounding modules are custom-written for each analysis problem. These are the portions that interpret the parsed input program to choose the parts that are relevant for abstract interpretation and set up the properties to be proved. The soundness of these interpretations is a function of the skill of the analyst who programs the particular analysis. Flaws in this interpretation, e.g. incorrect assumptions about operational language semantics, omitting portions of the program that bear on the invariants, incorrect design of custom abstract operations, can produce an incorrect result, even if the implementation of the more permanent portions of the analyzer are sound.

4.6.2 Implementation

A CodeHawk-derived analyzer, like any computer program, is originally written by humans, and thus subject to the standard exposure to implementation errors – either errors that cause it produce incorrect results, or errors that cause the analyzer to abnormally terminate or fail. Its exposure to such implementation errors, however, is minimized by several characteristics. CodeHawk is written in Ocaml, a very high level, strongly typed, functional language that minimizes the conceptual distance between algorithm specification and executable code. The central abstract interpretation engine, and the permanent parts of the front and back ends, are implemented according to well-known algorithms from the literature. These algorithms generally have their own proofs of correctness, so the remaining exposure is to flaws in their representation as Ocaml programs.

The literature on abstract interpretation, in particular, is very formal, and has been vetted over several decades by the research community. Its fundamental algorithm is to interpret a program over an abstract domain of variables that is precise enough to prove conjectures of interest (such as numeric intervals, or linear equalities). The concrete program is projected onto an abstract program that preserves only the operations on variables that affect the conjecture, and translates them to operations on corresponding abstract values from the abstract domain. Abstract values are then iteratively propagated over all paths in the flow graph until a fixpoint is reached. Paths that pass through branch predicates and computations *narrow* the set of values that variables can hold by adopting the constraints implied by the predicates and computations. Paths that converge to a common point *widen* the set of possible values by taking the union of the sets on each of the convergent branches. The sets of values for variables that remain at each program location after propagation reaches a fixpoint represent invariants on the variables at those locations – each set contains all actual values that a variable could possibly have on all paths reaching that point. These sets will usually be supersets of the actual values, representing safe, over-approximations of those values. The precision of the over-approximation varies with the precision of the abstract domain, and the precision of the abstract domain varies inversely with the cost to compute it.

As with the exposure to interpretation errors in the writing of the non-permanent parts of a complete analyzer detailed above, these non-permanent parts are also more exposed to

implementation errors, both because they are custom designed (not following published algorithms), and because they get less exposure to use and testing due to their ad hoc nature.

4.7 Downstream Error Impact

Internal implementation errors that may cause abstract interpreters, such as CodeHawk, to fail at runtime, or to fail to terminate at runtime, have no specific effects on other tools downstream in the software lifecycle. In these cases, there is no analysis result produced. So there is no result about which to be wrong. The situation is the same as if the tool had not been run at all. Internal implementation errors that may compromise the abstract interpreter's soundness, however, can be expected to have a downstream impact. One runs an abstract interpreter to prove that a program has certain properties, or the absence of certain properties. The claim of soundness can lead tool users to place extraordinary trust in the results, and thus more likely to omit other downstream verification steps that would be theoretically redundant.

Perhaps the greatest downstream exposure to soundness errors is faced by testing. It would be rare (and unwise) to find an organization so trusting of soundness claims that they forego testing of safety-critical software after a favorable analysis result. If the analyzer were sound, then a later test for the same property just proved would be redundant. But the defeasible nature of tool implementations should qualify this confidence. An organization may, however, perform less testing of non safety-critical software, or allocate limited testing resources to other properties and programs, in the wake of a favorable upstream result from abstract interpretation. These are reasonable, practical choices that nevertheless expose an organization to the possibility of unsoundness.

Another downstream exposure to unsound abstract interpretation can result from removing runtime checking code from a performance-sensitive application because such checks are deemed unnecessary. Code that checks every reference to an array index, for example, to ensure that the index is within the declared bounds of the array may add an unacceptable performance degradation to an application. The runtime checks may be used during development and testing to catch possible out-of-bounds references when the performance overhead is not relevant. But a result from an unsound abstract interpreter that "proves" every index reference to be within the declared bounds may be used as justification for removing the runtime checks from the production code.

A different kind of downstream error impact can result not from analyzer errors per se, but from an abstraction gap between real (or rational) numbers and their implementation as floating-point numbers. Abstract interpretation is often performed on real or rational numbered domains that effectively have infinite precision. Computer approximations of reals and rationals as floating-point numbers necessarily have a machine-word imposed, finite precision, and thus will eventually produce round-off errors when this precision is exceeded. So the abstract interpretation may be sound, producing an error-free result over the reals/rationals that does not carryover to the floats. In certain cases, the production application may accumulate these undiagnosed round-off errors in such a way that affects

the application's function. When this is a possibility, either downstream testers need to stress-test for critical round-off error accumulation, or the abstract interpreter must perform an analysis using an abstract floating-point domain which is sensitive to the round-off error.

Because it deals in over-approximations, upstream use of abstract interpretation can lead to downstream interpretation errors, even though the upstream analysis is sound. If the abstract interpretation is overly coarse-grained, catching many acceptable program behaviors in its safe approximation, unnecessary testing and code reviews may be deemed necessary downstream to probe this over-exposure. With a customizable abstract interpretation technology, such as CodeHawk, where an analysis can be run on different combinations of abstract domains, it is usually advisable to revisit what appear to be too large over-approximations using more precise domains. There is always a trade-off, of course, between the precision of the domains and the performance of the analyzer, so an organization needs to tailor the computationally expensive runs to the (parts of) applications where there is more safety risk for runtime errors. A good general strategy is to do an initial analysis of all applications with coarse-grained, inexpensive domains, then re-analyze those with large over-approximations, or exceptional safety risks, with more expensive domains.

4.8 Characterization of Input and Output Interpretations

As noted above, CodeHawk is an abstract interpretation technology rather than a particular tool. It becomes a specific tool when specialized to a language, a set of conjectures to be proved, and a set of abstract domains over which to perform the analysis. To date, it has been used exclusively in research contracts, which determine the languages, conjectures and domains of interest. Because of this, it does not have standard inputs and outputs at the analyst level and has, as yet, no end-user documentation. To provide an example characterization of its possible input and output interpretations in this section, we will reference a series of questions that have been posed about these interpretations in a recent project to productize the C memory-safety analyzer from the DHS Gold Standard project.

When CodeHawk uses library function templates as input, what is implied by the use of these templates (why not just provide the source code?)

A typical C application references many library functions, both from system libraries used by the compiler/linker system, and from user or open source libraries. The C source for these functions is typically not part of the application, and the functions are linked in from their pre-compiled, binary form. Some of these library functions may have been compiled or assembled from languages other than C. To follow the dataflow through the entire application, CodeHawk needs some sort of characterization of the semantics of these library functions. If C source code for *all* of the library functions were available (unlikely), there would be no need to distinguish between them and the normal application functions. But beside the fact that the sources typically aren't available (or not in C), there is an advantage in treating library functions differently. Such functions, typically from the system libraries (such as **strcmp**), are known to have well-tested,

stable implementations. They typically have some form of semi-formal documentation, defining the preconditions that the functions expect when called, and the post conditions that will hold on return from the call (if the preconditions are true at entry). These conditions could, in principle, be discovered by pre-analyzing the source code with CodeHawk, but often the conditions provided by the compiler developers are significantly stronger than what CodeHawk could easily infer with generic domains. So the CodeHawk strategy is to use formal "summaries" of these functions, pre-written by the CodeHawk developers from the formal documentation, as black-box analyses available at their application call sites. If the formal preconditions to such a library function can be satisfied by CodeHawk's computed abstract invariants on the C code at function entry, the formal post conditions can be assumed true at function return. From CodeHawk's point of view, these function calls are treated as single instructions (such as assignments) whose dataflow semantics are predefined by the language standard.

When CodeHawk indicates there are N verification conditions in a function, what does that mean?

The verification conditions are formal proof obligations (conjectures needing to be proved) that CodeHawk inserts at various places in the code to establish that the properties of interest hold. The number and placement of these depends on what CodeHawk is attempting to prove. In the case of C memory safety, the conditions represent conjectures that must be true for safe memory access that cannot be guaranteed by the C language standard at compile time. *Every* instruction in the C program being analyzed is vetted at translation time by CodeHawk for memory safety. Those that are guaranteed safe by the C syntax alone, either because they use an inherently safe access method that can be enforced by the compiler, or because they don't access memory at all, need no conditions. All others accrue the additional conditions that must be true for their memory access to be safe. After CodeHawk computes the abstract invariants at every location, these will be used to attempt to discharge the remaining proof obligations.

How can the analyst determine what properties CH examined for a particular output report?

This depends, of course, on the particular tool specialization. The Gold Standard analyzer is looking to prove the safety of every memory reference. The meaning of this generic "memory safety" property is given by the C language standard itself, and represents approximately 37 CWEs in the MITRE Common Weakness Enumeration taxonomy. So in this case, the final guarantee of memory safety comes from both the proof obligations successfully discharged during analysis, and from the front-end translator's formal interpretation of the C language standard.

What does the buffer underflow result status='safe' [|'error'|'warning'] imply?

'safe' implies either that the compiler can guarantee, or CodeHawk was able to prove, that the buffer operation cannot access an address below the buffer's lower bound. 'error' implies that at least one execution path in the program can access below the lower bound. 'warning' implies that the compiler cannot guarantee, and CodeHawk cannot prove, that

the buffer won't be accessed below the lower bound. There may be no actual execution path that accesses memory below the bound, but the abstract domains used for analysis were not precise enough for CodeHawk to prove this. The over-approximation of the access range includes addresses below the bound.

How can we associate an output product with the input set that generated that product?

In the case of Gold Standard CodeHawk, the output product is always an assessment of memory safety for the entire call graph rooted in some function. Typically this is the main function, in which case the call graph encompasses the entire application. The unit of input is essentially a C compilation unit, and libraries will be searched to find all of the referenced modules by the same processed used to build (compile and link) the application. Individual assessments of memory safety in the output will reference corresponding line numbers from the input source files.

When CH reports an error on a particular line, how can we back out the evidence for the proof?

The evidence for a proof, or a failed proof, is not generally visible to the end user. Proof obligations are generated at the front-end according to what cannot be guaranteed by the C language standard. The general strategy for this generation is documented in a formal paper, by the CodeHawk developers, that describes the method. The abstract invariants used to prove (or that failed to prove) a given memory safety conjecture can be viewed by line number in some implementation of CodeHawk.

What proof method does the tool use?

CodeHawk uses general purpose, first-order logic methods to discharge proof obligations. These generally involve the simplification of expressions. The proof methods are local to CodeHawk and are neither complete, nor as strong as one might obtain from a dedicated theorem prover. So a memory safety proof may fail either because the computed invariants aren't sufficiently precise, or because the proof method is too weak.

Which analysis domain?

Gold Standard CodeHawk uses the intervals, value-sets, and linear equalities domains by default.

Is that the only possible evidence?

No, it is the default evidence that is easy to automate. In the Gold Standard project, these default methods were able to discharge as much as 80% of the proof obligations automatically. Discharging the remaining 20% required human intervention by adding in safe preconditions for functions and data structure invariants.

What are the limitations and assumptions behind the analysis domain (e.g. does the implementation of the polygon domain have any limitations?)

Abstract domains have two forms of limitations. The primary one is precision. The more coarse-grained a domain (more imprecise), the fewer computing resources it takes to compute it. The intervals and value-set domains are relatively inexpensive to compute, and are useful in assessing memory access bounds. Linear equalities can capture (equality) relations between variables, which can support more bounds proofs. Linear inequalities can prove more precise relations, but are relatively expensive (in memory size) to use in the default case. In general, as the precision of relational domains increases, the memory requirements become exponential. The other limitation of abstract domains is a "theory vs practice" one. By default, real and rational numbers are used. These have infinite arithmetic precision, which may not precisely model the actual imprecision of their machine-limited, floating-point implementations.

When the CH run log reports errors or warnings occurred during the analysis (e.g. it reports an unknown library function), does this affect the trustworthiness of the results?

CodeHawk analysis is sound. What it proves -- memory safety in the case of the Gold Strandard-derived CodeHawk – is correct. The warnings relate to instances where it is unable to prove things. For instance, the presence of an unknown library function means that CodeHawk was not able to follow the data flow through this function. This will typically result in a widening of the return values to some safe approximation of what *is* known. This makes the final invariants more imprecise. This may make it impossible to prove the safety of some actually safe operation. But it will never lead to a proof that an unsafe operation is safe.

Some selected samples of input and output forms from this version of CodeHawk are presented below in Appendix A.

5. Soundness Issues for Intermediate Representations

5.1 What are Intermediate Representations?

A tool is used within the context of a system development to automate verification objectives as per the guidance provided in DO-178C. Figure 1 illustrates a typical system development process that starts with system requirements, generates intermediate development artifacts including software high-level requirements low-level requirements (e.g., design models), to finally produce executable object code that runs on the target processors in the airborne system.

The verification objectives typically include the following types of verification:

- system development artifacts are complete, consistent, and accurate
- successive development artifacts comply with previous ones
- there is absence of anomalous behavior

Tools are used in the verification process to automate parts of verification activities. The *intermediate representations* are models and other representations (tool inputs/outputs, configuration) derived/created from the development artifacts, as required by the tool.

The DO-333 soundness criteria apply to all interactions of intermediate results and abstract models among a chain of integrated tools including inputs/outputs interactions with human analysts and interpretation of results. This implies that sound (conservative) abstractions must be employed in the underlying notations of these representations with respect to the property being proven by the tool – one never permits a property (relating to the target system) to be declared true when it is not true. This includes limitations with respect to input models and transformations, limitations for language constructs or properties to be proven, scalability, usage errors, internal tool errors, and non-graceful response to abnormal inputs.

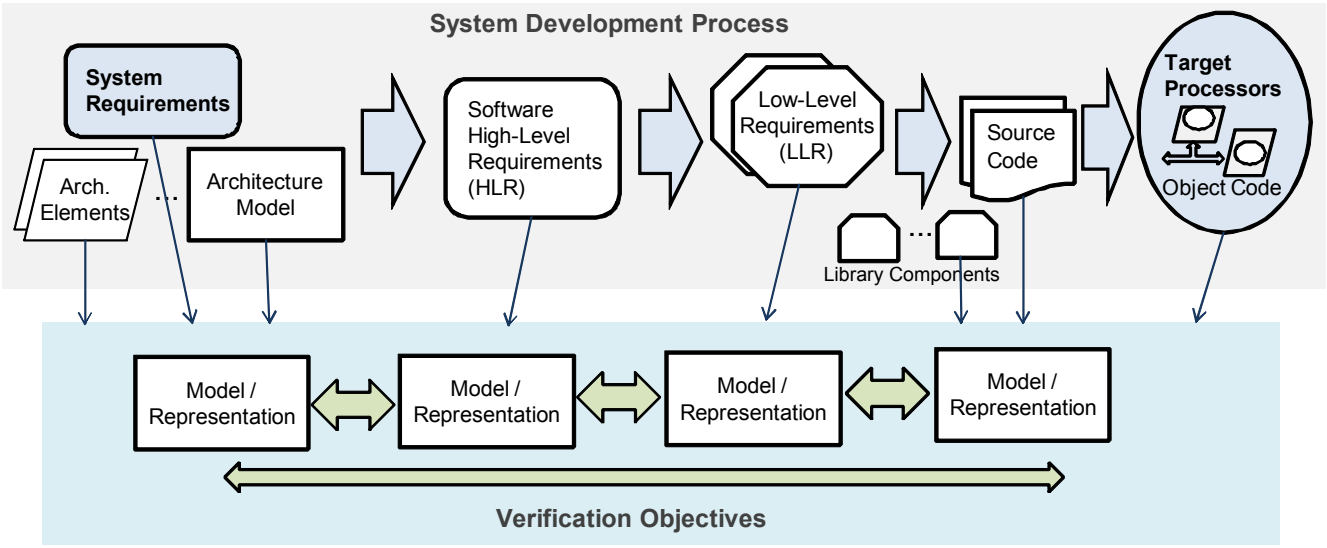


Figure 1. System Development Process and Verification Objectives

Model checking technology is often used to formally verify liveness and safety properties of safety relevant system algorithms. However, often model checking technology and related tooling are developed by university and research groups that may lack experience with formal design assurance development methods, such as the guidelines of DO-178C. Thus, a model checker documentation and usage examples may not explicitly state how the tool fits in the certification process: specifically which parts of DO-178C objectives are automated by the tool and what intermediate representation considerations the user must adhere to when relating tool inputs/output to the development artifacts of the airborne software. Figure 2 depicts the use of a model checker for automating parts of DO-178C objective (A-4.1) that low-level requirements (e.g., design model) comply with high-level requirements. All the artifacts shown in the green color in this figure are intermediate representations. Thus, in addition to the soundness of the core model checker tool implementation, the soundness of the intermediate artifacts is also a concern.

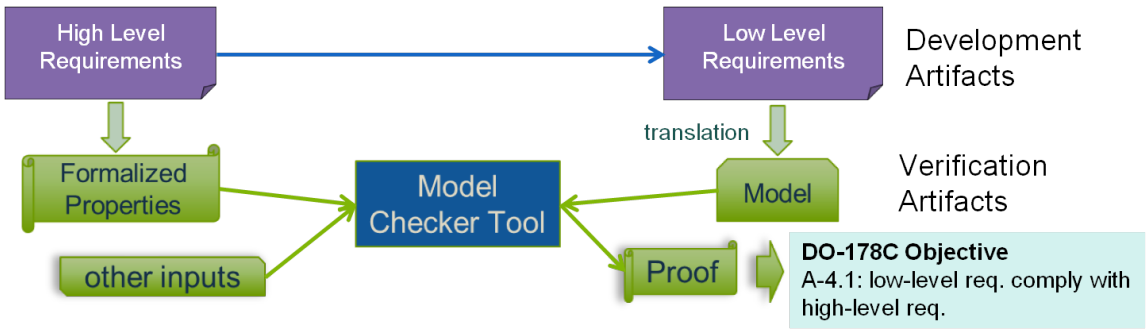


Figure 2. Example of Intermediate Representations when using a Model Checking Tool

The following subsections provide several examples of theoretical soundness issues relating to intermediate representations.

5.2 Example: Soundness of the Model under Exploration

When using a model checking tool, in addition to the soundness of the core model checker implementation, the soundness of the model under exploration with respects to the tool's abstraction is also an essential concern. In many usages, the "model" provided to the tool is created from development artifacts by a combination of automated/manual transformations. If errors in the modeled system abstraction can impact the model-checker's state exploration, invalid proofs may be indicated.

The following example shows that a mistake in the typing enabled an invalid proof by k-induction. The tools output is shown below for this example.

```
sal-inf-bmc brain_invalid{6} brain_strong_validity -v 1 -d 0
-i -ice
importing context "brain_invalid"...
parsing SAL file "brain_invalid.sal"...
creating abstract syntax tree for context "brain_invalid"...
type checking context "brain_invalid"...
flattening modules in the assertion located at [Context:
scratch, line(1), column(1)]
simplifying abstract syntax tree...
expanding function applications...
unfolding quantifiers...
eliminating common subexpressions in an assertion...
eliminating common subexpressions in a flat module...
flattening data structure in flat module...
flattening data structures in the property...
simplifying abstract syntax tree...
substituting simple definitions...
simplifying abstract syntax tree...
number of system variables: 320, number of auxiliary
variables: 0
executing BMC from depth 0 to 0...
executing k-induction with k=0
proved.
total execution time: 0.96 secs
```

In this instance the erroneous proof was quite problematic. Prior to the proof, the model checker had correctly identified counter examples over a period of a few weeks, which led the tool users erroneously to increase their trust of the tool's performance. The actual mistake in the model was relatively simple. It related to the declaration of the data type on line 58

```
DATA: TYPE = [1 .. 9];
```

With the erroneous assignment of 0 to the data filed in line 356

```
out[i] = (# data := 0, integrity := false, present := false
#)
```

Interestingly, the model passed the model-checker's well-formed checks as indicated below.

```
sal-wfc brain_invalid.sal  -v 2
importing context "brain_invalid"...
parsing SAL file "brain_invalid.sal"...
creating abstract syntax tree for context
"brain_invalid"...
  ast generation time: 0.0 secs
type checking context "brain_invalid"...
  type-checker time: 0.02 secs
Ok.
total execution time: 0.02 secs
```

In other technologies, such as PVS, Type-Correctness Conditions (or TCCs) are generated as part of the proof strategy. These require the soundness of the types to be proven before the proof Integrating such technology with the model checking technology may be an interesting direction to address this soundness vulnerability above. Extending and linking these proofs using an Evidential Tool Bus [5], may also be a valuable direction, providing a mechanism for such proofs to be linked, audited and replayed as necessary.

5.3 Internal Tool Abstractions for Models

Tools that directly read the actual development artifacts, such as source code and the design models from which the code is generated, would seem to avoid the problems mentioned in the previous example with the transformation of abstractions. In practice, however, the same issues remain; they are just hidden within the tool internals or the interpretation of model semantics. This applies not only to the internals of verification tools but also the internal representations in modeling and code generation tools used in the development process.

For exploration of these issues, we use model analysis, test generation, and source code verification tools used with models created using MATLAB Simulink/Stateflow modeling tools. Figure 3 shows the modeling and verification tool set where a model represents low-level requirements. Note that MATLAB Simulink/Stateflow tool suite does not define formal semantics or a single underlying notation for the model specification. Thus the *model* is considered to be the *in-memory representation* within the tool that is influenced by the contents of the *.mdl file*, *MATLAB environment/configuration*, *workspace parameters*, and *defaults*. Each of the SCV and HiLiTE verification tools must recreate the *model* semantics within the tool by either directly reading all the artifacts that impact the model or by querying the in-memory representation within MATLAB. This internal view of semantics in the verification tool must be accurate with respect to the MATLAB internal semantics view as well as the behavior semantics of the object code on the target processor.

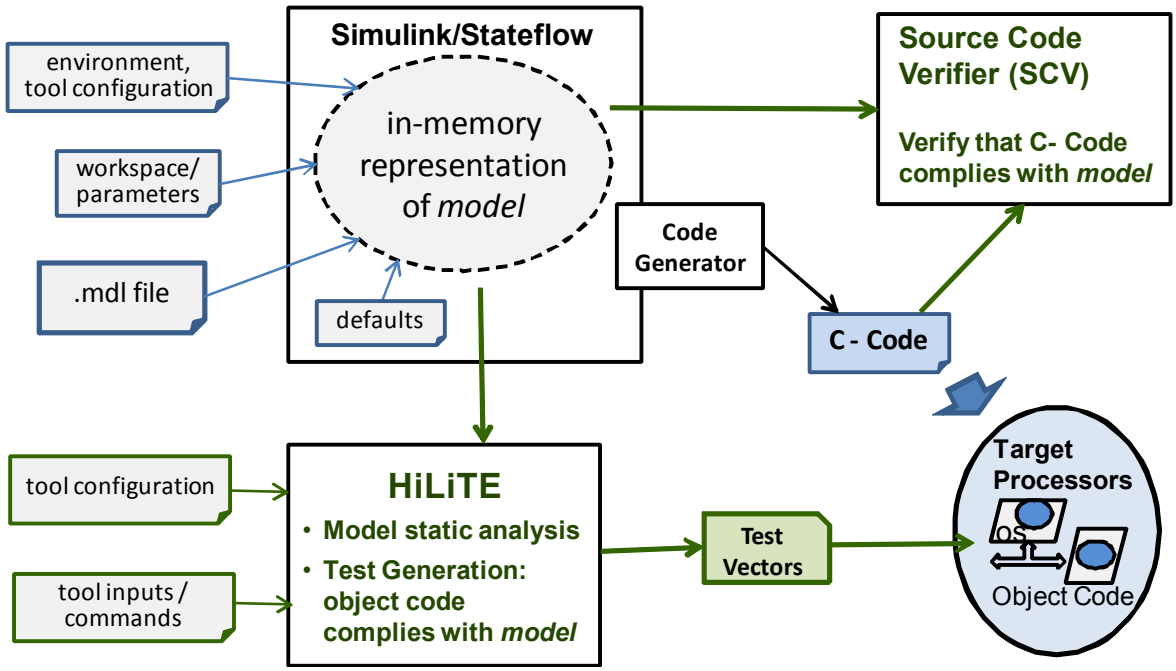


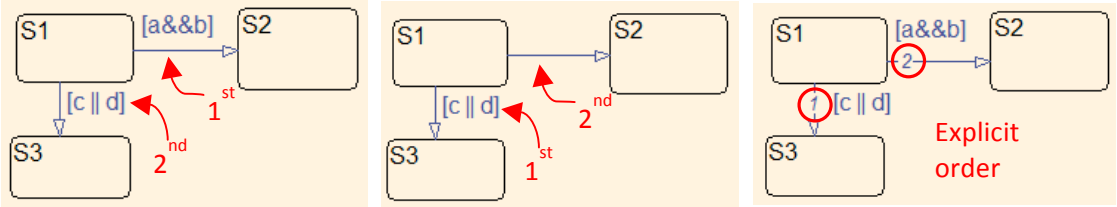
Figure 3. Example of Internal Representation of Model, in the Context of Simulink/Stateflow

The Honeywell Integrated Lifecycle Tools and Environment (HiLiTE) takes Simulink/Stateflow design models as an input, analyzes them, and generates tests for them. The detailed semantics of the models are independently defined/recreated within HiLiTE, based upon the MATLAB user manuals. That addresses common mode failure concerns, but opens the door to the possibility that the HiLiTE semantics are not consistent with MATLAB, especially given that the semantics defined in MATLAB user manuals are incomplete and may be unsound in certain places.

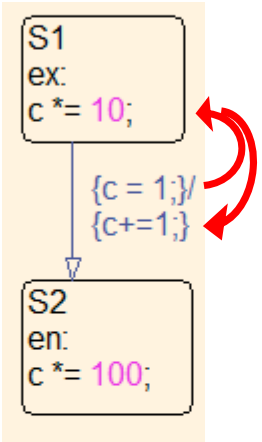
5.3.1 Example: priority inversion in Stateflow semantics.

HiLiTE's origins date back far enough that there have actually been changes to those model semantics (or the documentation thereof) that impact the interpretation of models and translations to other abstract intermediate representations for use by analysis tools. The HiLiTE Stateflow test generator has run into multiple kinds of priority inversion:

- Older versions of Stateflow used a clockwise order of priority of transition arcs emanating from a state. When a state had multiple outgoing transitions, they were evaluated starting at the 12:00 position going around the state in the clockwise direction (with the exception that an unguarded transition is evaluated last). Subsequent versions of Stateflow made the execution order explicit. The default order was set with clockwise priority, but users may override. This created a order of execution violation of the model semantics that could impact expected results during test generation.



- The model of the execution of actions around a transition inverted the priority of exit and condition actions. This could have allowed a code generation or compilation error escape the testing process.



- At the time we started the Stateflow test generation capability, there was an error in the MATLAB documentation. Since Stateflow overloads the '^' operator to mean either exponent or bitwise XOR based on the "Enable C-bit operations" checkbox. The precedence table explicitly listed the power operation as having the same precedence as bitwise XOR—much lower than one would expect. It turns out that, since the code generator used the power function (e.g., `pow(x, y) + z`), the generated code actually places the power operator at highest precedence. MathWorks ultimately corrected the documentation, rendering HiLiTE's semantics incorrect. HiLiTE has since raised the priority of the '^' exponent operator, and it is no longer qualified to support bitwise-XOR.

To make sure such errors in internal semantics do not escape qualification, tests were developed that are deliberately dependent on execution order. If HiLiTE is not executing actions, or evaluating expressions, in precisely the correct order, the qualification tests will fail.

5.3.2 Example: Changes in Model Semantics due to Block-Library Changes

It is a common practice to create a library of higher-level building blocks (e.g., timers, digital filters) out of low-level primitive operators provided by a modeling tool like Simulink. Specific behavior semantics abstractions are associated with these library blocks, with the expectation that certain analyses and results provided by the formal methods tools are in terms of these abstractions – e.g., the range bounds at the outputs of a filter are within limits across time, given the transfer function of the filter. Such higher-

level behavior abstractions must be translated through intermediate representations to the abstractions provided by a formal methods analysis tool. Our observations over several certification programs have been that the library block semantics can change during the lifecycle due to desired improvements or bug fixes; for example the semantics of the timer blocks in Honeywell's HAM block library was corrected to provide a deterministic behavior in face of floating-point representation errors across different processing platforms. Consequently, tool changes were required over the lifecycle of individual certification programs and even more changes have been required over the lifecycle of the HiLiTE tool. From the tool qualification perspective, it is necessary to make the soundness criteria explicit for all such higher-level behavior abstractions and the corresponding properties that a tool analyzes. Qualification tests are then designed around these criteria to provide confidence in the tool implementation.

Automatic code verification tools likewise take the development model as an input to compare against the generated code. Because these tools must account for every model block and every expression in the code, any error in the tool's interpretation of the design model should be easily exposed during qualification. As a third tool with a presumably independent implementation of the semantics, this is an opportunity to use diversity to strengthen the case for soundness of the overall process.

5.4 Intermediate Artifacts: Using SMT Solver as a Back-End Tool

The success of HiLiTE to generate over 90 percent of low-level requirements tests for Simulink design models has left only the most challenging model constructs beyond reach. To address those, Honeywell has turned to using a SMT solver as a back-end tool for HiLiTE. For test generation, this involves generating constraints from model subgraphs that are difficult to resolve usually because of complex relationships between the signals. If the SMT solver is able to find a solution to the constraints, HiLiTE can automatically verify the solution by simulating it. Therefore the SMT solver and the generation of SMT constraints need not be qualified independently because we have already qualified HiLiTE's capability to generate and evaluate tests for correctness and requirements coverage. However, the input model (in SMT language) for the SMT Solver is an intermediate artifact that is generated by HiLiTE, excerpting it from the complete model. The soundness of abstraction transformation must be maintained across the intermediate artifacts; the qualification testing must define the exact subset of the SMT construct used (as part of tool requirements) and develop qualification tests to the verify correct abstraction transformation for this subset.

5.5 Tool Independence

Every advancement in software engineering has involved raising the level of abstraction to leave out details that do not require the attention of the system developer. Whether it was processor instruction codes, register selection, or loop un-rolling, we have let our compilers and assemblers make decisions that are rooted in the specifics of the target processor architecture instead of the system being implemented. With model-based development, this includes fully specifying the order of operations, resolving data types, or even determining vector dimensions. The verification process therefore is about

establishing that the implementation is among the set of correct implementations of the rules defined during design. Given the requirements have been determined to be complete, all the implementations of a design that satisfy all requirements must be valid.

Simulink models establish the data flow for a system. Technically this amounts to a partial ordering; where *a* happens before *b* and *c* before *d*, but the relative ordering of *a* and *c* do not matter. The generated code selects an ordering based on arbitrary rules that may change from one version to the next.

Additionally many blocks in a Simulink model do not specify a data type. They "inherit" the type of the input signal. This is convenient so the blocks do not all have to be altered when a signal's data type is changed. In this case there is an algorithm to derive each data type given an input data type to ground the analysis.

Likewise, the signals may be scalars or vectors of unspecified dimension for most blocks. The semantics of the blocks define the output signal dimensions relative to the input dimensions; so these can also be computed given starting dimensions of input signals.

It has been argued that the simulation semantics and in-memory representation constitute the low-level requirements; therefore the complete ordering, data types, and dimensions should be read from the model environment rather than computed from the incomplete model specification file. That would suggest that these details that have been abstracted out are salient and must be defined by the developer, not the modeling tool. The information specified by the developer and read during model reviews must be sufficient to define a correct implementation; thus all implementations that conform to the specification (e.g., all complete orderings that satisfy the partial ordering) are correct.

5.6 Completeness of the Intermediate Artifacts

In order for an assurance argument to be repeatable, all the data that informs the behavior of a model must be controlled. In effect "verification" is a mapping of specific artifacts with verification results such as test cases or formal proofs guaranteed for those artifacts. Real world development models are often not contained in a single file or even a single type of file, the complete behavior of the simulation and the full content of generated code depend on data from outside the model file. For example workspace parameters may be referenced by a model, but the values of those parameters are loaded separately to support reuse of the model. Direct integration with the model development environment to verify against the in-memory representation conflates all this information, and it can be impossible to positively determine that all data incorporated into that verification has been properly controlled.

It is standard practice that artifacts that haven't changed need not be re-verified. However, if the simulation behavior of a model is based on uncontrolled parameter values, then reusing a model with its source and object code, and their certification artifacts could result in an escape. An independent tool that reads all the data files can verify that all the data is present and controlled, and further ensure that nothing is assigned multiple values.

5.7 Completeness in the Enumeration of Low-level Properties

An important aspect of intermediate representations is the completeness of the properties being verified using a tool with respect to the actual system. This is especially important when a tool, such as static analysis using abstract interpretation, verifies several low-level properties of the system design or source code. An essential part of the intermediate representation of the system is the complete enumeration of all relevant properties that are present in the actual system. For example, if a tool detects "division-by-zero", it is essential that the tool's representation of the properties it checks includes not only the explicit division operations in the design or source code, but all possible mathematical operations that can result in a division by zero such as reciprocals, negative exponent, trigonometric or transcendental operations such as tangent. Several currently available static analysis tools report only the "division-by-zero" instances the tool could find; it remains unclear whether the tool enumerated all possible explicit and implicit cases and analyzed each one. Our general recommendation in this area is that such a tool should attempt to prove a safety property such as "absence of overflow" and as part of the analysis/proof, produce a list of all instances of "overflow" enumerated by the tool so that another tool in the chain can check the list for completeness. In addition, as part of the tool qualification, the Tool Requirements should specifically list all types of overflow instances enumerated by the tool so that a determination of completeness with respect to the actual system can be made. If there is an intermediate abstraction in between the system design/code and the tool analysis, the completeness of the intermediate abstraction should be assessed against the actual system.

5.8 Soundness and Completeness of Requirements-Based Coverage Metrics

In the DO-178C certification process, there are certain verification objectives where coverage metrics are computed on the design or code elements with respect to higher level requirements. The specific objectives seek to analyze which elements of a design model or source code are exercised by requirements-based tests. This analysis serves the purpose to rule out unintended functionality (in the design models or code) as well as provide a metric of the efficacy of requirements-based tests. In some certification arguments, the coverage metrics are even used to solely define the entire criteria of completeness of the requirements-based tests, and thus skip other downstream testing objectives. A common method is to execute requirements-based tests on the model or code and use a tool to measure coverage achieved on the model or code elements by these tests. Therefore, tool qualification must address the soundness and completeness of the coverage metrics reported by the tool.

Two major issues have been observed with the current tools and approaches for computation of test coverage metrics. The first is the issue of *observability* : namely if the unique impact of the computation of a model or code *element* is not observable in the forward path to an output value being tested in a requirement-based test, then the *coverage* of that element cannot be claimed. The CAST-6 paper [24] mentions this as a fundamental underlying concept of testing in the following paragraph quoted below:

"Do the requirements-based test cases adequately exercise the structure of the source code? Two factors in exercising any structural element of the source code are: (a) the ability to test that element by setting the values of the element's inputs (this is the concept of controllability), and (b) the ability to propagate the output of that element to some observable point (this is the concept of observability). Controllability and observability are fundamental concepts used in testing logic circuits, and also apply well to testing software."

Note that even though the CAST-6 paper mentions "source code" elements, the same criteria are applicable to test cases on model elements. Figure 4 shows a simple example of a MATLAB Simulink model and a requirement-based test case where the input and output values of the test case are shown in green color. The value of the test case at the 3rd input (control) of the switch block selects the path of the 1st input of the switch to the measured output. Therefore the blocks in gray, even though they may be "executed" in the simulation, are not covered by the test case because their results are not observable at the measured output. In general, the observability criterion applies to various types of blocks (and their computation) that are in the forward path to the output.

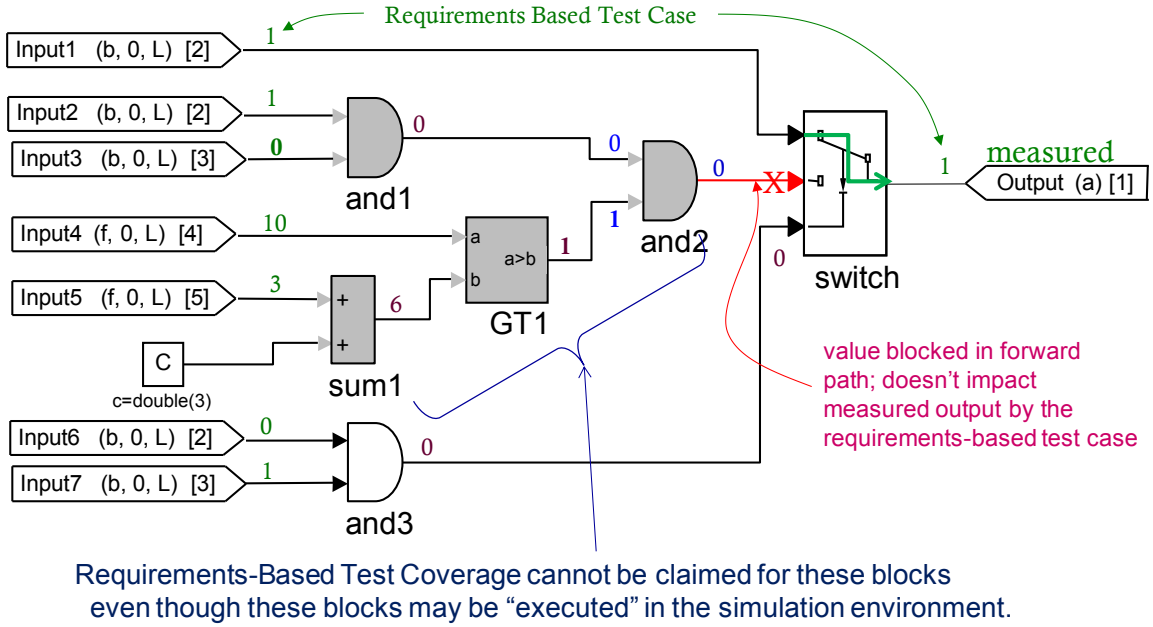


Figure 4. Example of Model Coverage Soundness Criterion

A recent paper in the 2015 NASA Formal Methods (NFM) conference [25] illustrated via case studies that if a coverage tool doesn't implement observability, there can be a significant over reporting (up to a factor of 5) of the coverage by the tool over coverage truly achieved by the tests. It is important for a tool, that claims any credit against these objectives, to describe the mathematical basis for how it determines observability in claiming coverage.

The second problem is: Does a coverage tool *completely* enumerate the test coverage requirements (behavior equivalence classes) for all computational elements in the model? For example, the Model Coverage Analysis activity of DO-331 requires a computation of coverage achieved on the model elements identified as low-level requirements (LLR) in terms of numeric, Boolean, and time-dependent function equivalence classes. As part of the intermediate abstraction, a coverage analysis tool must completely enumerate all such equivalence classes and then report coverage of which ones were exercised by the tests. Many available model coverage analysis tools, however, do not do such complete enumeration. These tools require the user to place specific "instrumentation" constructs in the model and report coverage only on those instrumentation constructs; thus a tool can report 100% model coverage even if the user forgot to put instrumentation in the model and only ½ of the model element are actually covered by the tests. A tool vendor can argue that this is the user's and not the tool's responsibility. But the fact remains that the lack on completeness on the tool's part introduces opportunities for errors in the process of verification of DO-178C objectives; the tool's report of "100% coverage achieved" can give a false sense of completeness. In such cases, tools should indicate the coverage is over the user-defined test requirements not all LLR. The qualification plan should also indicate at best partial credit for the objective with additional activity to verify completeness of the test requirements supplied to the tool. In the use of tools in DO-178C verification, the entire process of humans and tools to achieve a DO-178C objective must be shown to be sound and complete. The tool qualification needs to ensure that the intermediate representations and the tool results are sound and complete as part of this process.

References

1. Clarke, Edmund M., and E. Allen Emerson. "*Design and synthesis of synchronization skeletons using branching time temporal logic.*" Springer Berlin Heidelberg, 1982.
2. Clarke, Edmund M., E. Allen Emerson, and A. Prasad Sistla. "*Automatic verification of finite-state concurrent systems using temporal logic specifications.*" ACM Transactions on Programming Languages and Systems (TOPLAS) 8.2 (1986): 244-263.
3. Clarke, Edmund M., et al. "*Exploiting symmetry in temporal logic model checking.*" Formal Methods in System Design 9.1-2 (1996): 77-104.
4. Cousot, Patrick and Cousot, Radhia. "*Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints.*" POPL (1977) , 238-252.
5. Cruanes, Simon, et al. "*Tool integration with the evidential tool bus*", in Verification, Model Checking, and Abstract Interpretation. Springer Berlin Heidelberg, 2013.
6. Queille, Jean-Pierre, and Joseph Sifakis. "*Specification and verification of concurrent systems in CESAR.*" International Symposium on Programming. Springer Berlin Heidelberg, 1982.

7. Katz, Shmuel, and Doron Peled. "*Verification of distributed programs using representative interleaving sequences.*" Distributed Computing 6.2 (1992): 107-120.
8. Godefroid, Patrice, et al. "*Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*." Vol. 1032. Heidelberg: Springer, 1996.
9. Dill, David L. "*The Murphi Verification System.*" Computer Aided Verification. Springer Berlin Heidelberg, 1996.
10. Emerson, E. Allen, and A. Prasad Sistla. "*Symmetry and model checking*." Formal methods in System Design 9.1-2 (1996): 105-131.
11. Ip, C. Norris, and David L. Dill. "*Better verification through symmetry.*" Formal methods in system design 9.1-2 (1996): 41-75.
12. Holzmann, Gerard J. "*The Model Checker SPIN*." IEEE Transactions on software engineering 5 (1997): 279-295.
13. Havelund, Klaus, and Thomas Pressburger. "*Model Checking Java Programs using Java PathFinder.*" International Journal on Software Tools for Technology Transfer 2.4 (2000): 366-381.
14. Misra, Jayadev, and K. Mani Chandy. "*Proofs of networks of processes.*" Software Engineering, IEEE Transactions on 4 (1981): 417-426.
15. Jones, Cliff B. "*Tentative steps toward a development method for interfering programs.*" ACM Transactions on Programming Languages and Systems (TOPLAS) 5.4 (1983): 596-619.
16. Grumberg, Orna, and David E. Long. "*Model checking and modular verification.*" ACM Transactions on Programming Languages and Systems (TOPLAS) 16.3 (1994): 843-871.
17. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. "*Symbolic Model Checking without BDDs*". Volume 1579 of LNCS, pages 193–207, March 1999.
18. Godefroid, Patrice. "*Model checking for programming languages using VeriSoft*." Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1997.
19. Havelund, Klaus, and Thomas Pressburger. "*Model checking Java programs using Java PathFinder*." International Journal on Software Tools for Technology Transfer 2.4 (2000): 366-381.
20. Visser, Willem, et al. "*Model checking programs*." Automated Software Engineering 10.2 (2003): 203-232.
21. Burch, Jerry R., et al. "*Symbolic model checking: 10^20 states and beyond.*" Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e. IEEE, 1990.
22. K. L. McMillan. "*Symbolic Model Checking: An Approach to the State Explosion Problem*". Kluwer Academic Publishers, 1993.
23. L. Pike. "Real-Time System Verification by k-Induction", NASA/TM-2005-213751, 2005.
24. Certification Authorities Software Team (CAST), *"Rationale for Accepting Masking MC/DC in Certification Projects."* Position Paper CAST-6, August 2001.

25. Murugesan, Anitha, et al. “Are We There Yet? Determining the Adequacy of Formalized Requirements and Test Suites.” Proceedings of the 2015 NASA Formal Methods Conference, 2015.

Appendix A

This version of CodeHawk attempts to prove safe memory access (according to the C standard) for every statement in a program. It does this by first placing a series of proof obligations, at every location in the program, which must be true for safe access to be guaranteed by the compiler. It then computes abstract invariants (the abstract interpretation, proper) about the values of variables at every such location. Finally, it uses these computed invariants as lemmas to attempt to discharge all of the proof obligations. A memory reference, at a given location, is guaranteed safe under all possible execution paths if all of the proof obligations assigned to that reference can be discharged. If one or more proof obligations remain open, the memory access may still be safe, but that fact cannot be proven with the current precision of the abstract domains used.

Below are some samples of discharged (proved safe) and open (don’t know) proof obligations from the function activeExpireCycle() in the open source in-memory database application Redis. In the source listing of this function (from the source file redis.c) we have highlighted two lines, one in green (line 771) for which all obligations were discharged, and one in yellow (line 814) for which some proof obligations remain open. The raw CodeHawk output is produced as XML files, which will be interpreted by the user interface in which the CodeHawk analysis engine is embedded according to its own custom format.

```
void activeExpireCycle(int type) {
    /* This function has some global state in order to continue the work
     * incrementally across calls. */
    static unsigned int current_db = 0; /* Last DB tested. */
    static int timelimit_exit = 0;      /* Time limit hit in previous call? */
    static long long last_fast_cycle = 0; /* When last fast cycle ran. */

    int j, iteration = 0;
    int dbs_per_call = REDIS_DBCRON_DBS_PER_CALL;
    long long start = ustime(), timelimit;

    if (type == ACTIVE_EXPIRE_CYCLE_FAST) {
        /* Don't start a fast cycle if the previous cycle did not exited
         * for time limt. Also don't repeat a fast cycle for the same period
         * as the fast cycle total duration itself. */
        if (!timelimit_exit) return;
        if (start < last_fast_cycle + ACTIVE_EXPIRE_CYCLE_FAST_DURATION*2)
              return;
        last_fast_cycle = start;
    }

    /* We usually should test REDIS_DBCRON_DBS_PER_CALL per iteration, with
```

```
 * two exceptions:
 *
 * 1) Don't test more DBs than we have.
 * 2) If last time we hit the time limit, we want to scan all DBs
 * in this iteration, as there is work to do in some DB and we don't want
 * expired keys to use memory for too much time. */
if (dbs_per_call > server.dbnum || timelimit_exit)
    dbs_per_call = server.dbnum;

/* We can use at max ACTIVE_EXPIRE_CYCLE_SLOW_TIME_PERC percentage of CPU
   time
 * per iteration. Since this function gets called with a frequency of
 * server.hz times per second, the following is the max amount of
 * microseconds we can spend in this function. */
timelimit = 1000000*ACTIVE_EXPIRE_CYCLE_SLOW_TIME_PERC/server.hz/100;
timelimit_exit = 0;
if (timelimit <= 0) timelimit = 1;

if (type == ACTIVE_EXPIRE_CYCLE_FAST)
    timelimit = ACTIVE_EXPIRE_CYCLE_FAST_DURATION; /* in microseconds. */

for (j = 0; j < dbs_per_call; j++) {
    int expired;
    redisDb *db = server.db+(current_db % server.dbnum);

    /* Increment the DB now so we are sure if we run out of time
     * in the current DB we'll restart from the next. This allows to
     * distribute the time evenly across DBs. */
    current_db++;

    /* Continue to expire if at the end of the cycle more than 25%
     * of the keys were expired. */
    do {
        unsigned long num, slots;
        long long now, ttl_sum;
        int ttl_samples;

        /* If there is nothing to expire try next DB ASAP. */
        if ((num = dictSize(db->expires)) == 0) {
            db->avg_ttl = 0;
            break;
        }
        slots = dictSlots(db->expires);
        now = mstime();

        /* When there are less than 1% filled slots getting random
         * keys is expensive, so stop here waiting for better times...
         * The dictionary will be resized asap. */
        if (num && slots > DICT_HT_INITIAL_SIZE &&
            (num*100/slots < 1)) break;

        /* The main collection cycle. Sample random keys among keys
         * with an expire set, checking for expired ones. */
        expired = 0;
        ttl_sum = 0;
        ttl_samples = 0;

        if (num > ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP)
            num = ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP;

        while (num--) {
```

```
                dictEntry *de;
                long long ttl;

                if ((de = dictGetRandomKey(db->expires)) == NULL) break;
                ttl = dictGetSignedIntegerVal(de)-now;
                if (activeExpireCycleTryExpire(db,de,now)) expired++;
                if (ttl < 0) ttl = 0;
                ttl_sum += ttl;
                ttl_samples++;
            }

            /* Update the average TTL stats for this database. */
            if (ttl_samples) {
                long long avg_ttl = ttl_sum/ttl_samples;

                if (db->avg_ttl == 0) db->avg_ttl = avg_ttl;
                /* Smooth the value averaging with the previous one. */
                db->avg_ttl = (db->avg_ttl+avg_ttl)/2;
            }

            /* We can't block forever here even if there are many keys to
             * expire. So after a given amount of milliseconds return to the
             * caller waiting for the other active expire cycle. */
            iteration++;
            if ((iteration & 0xf) == 0) { /* check once every 16 iterations. */
                long long elapsed = ustime()-start;

                latencyAddSampleIfNeeded("expire-cycle",elapsed/1000);
                if (elapsed > timelimit) timelimit_exit = 1;
            }
            if (timelimit_exit) return;
            /* We don't repeat the cycle if there are less than 25% of keys
             * found expired in the current DB. */
        } while (expired > ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP/4);
    }
}
```

This is a section from the XML file containing the final results:

```
<?xml version="1.0" encoding="UTF-8"?>
 <codehawk-c-analysis>
  <log>
   <log-entry time="11/06/2014 22:22:55"/>
   <log-entry time="11/06/2014 22:02:45"/>
   <log-entry time="11/06/2014 21:43:43"/>
   <log-entry time="11/06/2014 21:27:08"/>
   <log-entry delta-checkvalid="59" delta-invariant="60" delta-lifted="15"
time="11/06/2014 21:04:00"/>
  </log>
  <function name="activeExpireCycle">
   <statistics checkvalid="59" invariant="60" invariant_with_api="15"
total="223" total-proven="134"/>
   <open-proof-obligations>
       .
       .
       .
    <open id="68" line="814" predicate="not-null"/>
    <open id="69" line="814" predicate="valid-mem"/>
    <open id="70" line="814" predicate="lower-bound"/>
```

```
    <open id="71" line="814" predicate="upper-bound"/>
       .
       .
       .
   </open-proof-obligations>
   <proof-obligations-discharged>
       .
       .
       .
    <discharged domain="symbolic sets" id="4" method="invariants"
time="11/06/2014 21:04:00" type="initialized">
     <evidence comment="assignedAt#764"/>
    </discharged>
    <discharged id="5" method="check-valid" time="11/06/2014 21:04:00"
type="initialized">
     <evidence comment="last_fast_cycle is global"/>
    </discharged>
    <discharged id="6" method="check-valid" time="11/06/2014 21:04:00"
type="integer-underflow">
     <evidence comment="add non-negative number: value is 2000"/>
    </discharged>
    <discharged domain="none" id="7" method="invariants" time="11/06/2014
21:04:00" type="int-overflow">
     <evidence comment="predicate depends on global variables last_fast_cycle,
which is delegated to global analysis"/>
     <assumptions>
      <uses a-id="11" a-type="global"/>
     </assumptions>
    </discharged>
       .
       .
       .
  </proof-obligations-discharged>
  </function>
  <header time="11/06/2014 22:22:55">
   <application file="src/redis.xml" name="application"/>
  </header>
 </codehawk-c-analysis>
```

This is the xml (from a separate file) for one of the discharged proof obligation (id=”6”) above:

```
    <proof-obligation c-complexity="1" id="6" origin="prim-op" p-
complexity="2">
     <location byte="281641" file="src/redis.c" line="771"/>
     <predicate op="plusa" size="ilonglong" tag="int-underflow">
      <exp1 etag="lval">
       <lval>
        <lhost>
         <var vid="10255" vname="last_fast_cycle"/>
        </lhost>
       </lval>
      </exp1>
      <exp2 etag="const">
       <constant ctag="cint64" ikind="ilonglong" intValue="2000"/>
      </exp2>
     </predicate>
     <context>
```

```
      <cfg-context>
       <node name="if-expr"/>
       <node name="stmt" num="153"/>
       <node name="if-then"/>
       <node name="stmt" num="150"/>
      </cfg-context>
      <exp-context>
       <node name="2op" num="2"/>
      </exp-context>
     </context>
    </proof-obligation>
```

and this for one of the open obligations (id=”68”):

```
    </proof-obligation>
    <proof-obligation c-complexity="11" id="68" origin="prim-lval" p-
complexity="1">
     <location byte="282363" file="src/redis.c" line="814"/>
     <predicate tag="not-null">
      <exp etag="lval">
       <lval>
        <lhost>
         <var vid="10263" vname="db"/>
        </lhost>
       </lval>
      </exp>
     </predicate>
     <context>
      <cfg-context>
       <node name="instr" num="0"/>
       <node name="stmt" num="173"/>
       <node name="if-then"/>
       <node name="stmt" num="172"/>
       <node name="loop"/>
       <node name="stmt" num="170"/>
       <node name="loop"/>
       <node name="stmt" num="166"/>
      </cfg-context>
      <exp-context>
       <node name="mem"/>
       <node name="lhs"/>
      </exp-context>
     </context>
    </proof-obligation>
```