CONFORMIQ

A Conformiq Technology Brief

The Conformiq Approach to Algorithmic Test Generation

Conformiq Designer is a commercial tool for model driven testing. It derives tests automatically from behavioral system models. The derived tests are black-box tests by nature, which means that they depend on the model and the interfaces of the system under test, but not on the internal structure (e.g. the source code) of the implementation. This whitepaper explores the technical implementation of Conformiq Designer, historical and experimental rationale for it as well as highlighting key technical capabilities we believe are important for successful, real commercial deployments.

The Conformiq Challenge

Conformiq Designer is an ongoing attempt to provide an industrially applicable solution to the following technical problem: given an artifact (a "model") M that describes the external behavior of an open system, construct computationally (without human interaction) a strategy to test real-world implementations of M in order to find out if they have the same external behavior as M or not.

This is a very difficult problem, for various reasons, but before elaborating on that, the first question that needs to be posed is what would be the commercial relevance of a solution to this "automated test design challenge".

Test design (not necessarily automated) is ubiquitous in software engineering. Every company that designs and develops software, in any form, has to test software also (or outsource it to someone else), and testing involves always test design, whether it is explicitly recognized as a separate process step or not. Test design is the task of deciding how a system is to be tested, which data parameters to send in, at which time, how to validate the results and so on. Test execution is then the task of actually executing them against the real system under test and performing the validation checks.

In a totally manual process, these two steps can be so intermingled that they seemingly cannot be separated; e.g. "exploratory testing" means that a user interacts with a system and decides on the fly what to try next, and in this case test design and test execution could be seen as a one process of "exploration". But that is mostly a matter of perception, as the human operator is constantly making decisions about how to interact with the system next and is thus designing tests "online" (on the spot) all the time.

During the last few decades, there has much emphasis on "test automation", but that has really meant the automation of test execution, not of test design. Even today it is a common belief that test design cannot be automated, because it is such a creative process that only humans can

carry it out well.

Regardless of whether test design can be automated or not, claiming that it cannot because it is seen as a creative process is clearly premature. Chess playing was considered in the second half of the 20th century as a prime example of "human intelligence", so that *Popular Mechanics* wrote in 1979 that "teaching a computer to play chess... is more than a game. Many consider such research and development to be the first step toward realization of machine intelligence".

But by today, playing chess algorithmically has been solved to the extent that the best programs beat all grandmasters; yet, the computer has not become intelligent in any biological sense but the problem has become so well understood and hardware so fast that a mathematical deconstruction of the problem has become feasible and successful. There is not a fundamental reason why test design could not be similarly deconstructed.

"Many consider such research and development to be the first step toward realization of machine intelligence"

In addition to being a universally prevalent part of any software engineering process, test design is expensive and difficult. It can easily consume 50% of the total quality assurance budget (when it is identified as a separate line item), and has a commanding, primary impact on the quality of the released products.

The question is then that if the "automatic test design challenge" can be solved in a satisfactory manner, would creating models and deriving tests automatically from them be economically more efficient than designing tests by hand?

To see why that might be the case, it is useful to factor the test design task (when carried out by humans) into two parts: understanding how the system under test is expected to operate, and deciding how to best validate that it works as expected. Every tester who tests a system and is, in some form, responsible for test design, needs to understand some part of the system's expected behavior as otherwise it is impossible for the tester to pass judgement on whether the system operates correctly or not. Also, because most computer systems have infinitely many different ways to interact with them, at least on a detailed level, every tester needs to make a (conscious or not) decision on which scenarios to use for testing the system and which not; which data parameters to use on the selected scenarios and so on.

Once it is recognized that testers need to understand how the systems they test are expected to work, it is not a long step to postulate that the testers have a partial, cognitive model of the behavior of those systems in their mind, in the same way as humans create mental models of other objects they interact with. But this mental model in itself is not a set of tests and the tester needs to design the tests separately from the model, but yet based on it, as the model implies what can be done with the system and what not, and what are the expected, correct responses from the system when it is interacted with.

Therefore test design is a separate mental task from the task of constructing a mental model of the system under test, a division that is clearly visible in a usual manual test design process. On one hand, testers need to learn how the system under test operates and how it should operate from user manuals, functional specifications, previous experience and other similar sources of information; on another hand, they need to jot down test plans and test cases where they select individual scenarios to be executed against the system, and in order to argue that the selected set of scenarios is sufficient for ensuring a low-risk release, they need to post good traceability and coverage metrics.

Now, if the mental model could be transferred to a computer, and the computer could carry out the second task of test design, it would clearly save money and human effort if the cost of getting the model to a computer-understandable form would be low enough. That is, if a computer could do at least as good of a job as a human in selecting test cases from the perspective of bringing the risk of unfound defects down, and the cost of modeling would be lower than the cost of test design by a tester, the net result would be at least as good testing as before at a lower cost. This is the premise of model driven testing.

It is then an experimental question whether a computer can derive good tests from a computer-readable system model, and whether a computer-readable system model can be produced cost-efficiently enough. Conformiq has benchmarked computerized test generation against manual test design in the context of Conformiq Designer since 2007, with a consistent benchmark of around 80% cost savings, meaning that it takes roughly 1/5 of the time spent in manual test case design to computerize the system model

and use it as a basis for algorithm test case generation that generates at least as good tests as the manual process. Of course, as computers become more powerful and the state of the art progresses, these benchmark figures get better over time (leading to our vision of "100X productivity improvement in test design" that we want to achieve).

Test Generation is Difficult

There are two fundamental reasons why (good) test generation from system models is difficult: (1) the algorithmic complexity of any realistic attempt on that, and (2) the fact that the model is independent from the implementation.

Kurt Gödel

Just generating input sequences that cover all the statements, say of a system model, is a theoretically undecidable problem (meaning that it can be never solved completely). This is a simple corollary of the fundamental incompleteness theorems proved first by Kurt Gödel and later by Alan Turing. This does not mean that there could not be an algorithm that handles most of the industrially relevant problem instances, but it gives a hint that it is a hard engineering problem to produce even that algorithm.

The fact that the model is independent from the implementation based on it implies that it is impossible to produce a test suite from a model that would provably catch all functional defects in the actual implementation as long as the test suite is of any reasonable size. Indeed, an underpinning of the model driven testing theory is that the system implementation is assumed to resemble, somehow, the model, which then justifies the idea that achieving high *model-level* coverage indicates a lower risk of residual defects in the implementation. This idea is very hard to express in an exact mathematical form, though, but because it works in practice it is in general accepted.

Conformiq System Models

In Conformiq Designer, systems are modeled as possibly multi-threaded computer programs that interact with an external, unspecified environment by message passing. These model programs are written in a combination of Java code and UML statecharts. Java code is used to describe how data works in the system, to declare data types and classes, express arithmetics and conditional rules and so on, whereas the UML statecharts are used to capture high-level control flow and life cycle of objects as well as event loop structures and high-level logic of message, event and time-out processing. Together object-oriented Java code and graphical UML statecharts form a powerful modeling language for the purposes of test generation.

Though not immediately visible to the user, it may be interesting to know that the models are internally compiled into a variant of LISP that is then processed by the test generation algorithm. This highlights how the whole test generation process is driven by semantics: even though there can be statecharts in a Conformiq system model, the graphical structure of the statechart is not used in any fashion to guide test generation, but only the logical meaning of the model. This is in stark contrast with simpler state machine driven test generation approaches where the structure of a (typically only one) state machine is used to generate a sequence of tests that correspond to different paths through the state machine; and this is important because only the fully semantics driven approach can tolerate models where the high-level control flow is deeply dependent on data values.

The purpose of the models is to describe the expected external behavior of the system under test. What this means is that the models do not need to be structurally equivalent to any implementation, they just need to exhibit an external behavior that corresponds to that of the system's on the chosen level of abstraction, and this makes the whole approach feasible as the models are multiple orders of magnitude smaller than the real implementations. (This complexity disparity is driven by multiple factors, the most important of them being the ability to abstract implementation details away, the ability to model only some aspects of the system's behavior instead of modeling it all, and the ability to use simple data structures and simplistic code as the model is never executed in real time and thus does not need to be efficient.)

Creating Models

Humans need to create and maintain the system models, and that is indeed the main task of a model driven test engineer. We typically recommend that the models would be created without knowledge of the real implementation but based on documentation such as functional specifications alone. In other words, the actual implementation is considered a black box whose internals cannot be observed, which is where the term "black-box testing" comes from. Sometimes documentation is not available and then other information sources need to be used, including experimentation with the system that is then validated by review and only afterwards encoded into the model.

In contrast, a model that is created based on an existing implementation is not independent and tests generated from it cannot really find functional defects in the system, as the tests will in fact only verify that the system works as it works.

Because model driven test engineers spend the majority of their time working with models, modeling and model maintenance should be as efficient as possible. One aspect is the expressivity of the modeling language: the more expressive the language is, the easier it is to capture the intended behavior in models. Another aspect is practical support for modeling, including for example model analysis and debugging as well as automatic code completion.

As our goal has always been to create an industrially applicable model driven test generation solution, the modeling language in Conformiq Designer is very expressive and driven by industry standards. All the usual (object-oriented) programming devices of Java are available, including classes, inheritance, threads and exceptions, and a model can consist of multiple objects described as UML statecharts that interact with each other by internal message passing.

Conformiq Designer also has support for model analysis including an interactive, graphical model and test debugger, graphical mapping from tests to models (useful also in post-execution analysis after tests have been run), and a built-in model profiler for analyzing test generation performance bottlenecks. These features are critical to quickly creating good models.

The core of Conformiq Designer 4.X is its semantics driven, symbolic execution based test generation algorithm.

Correctness of Models

Conformiq Designer cannot know if the model a user has created is functionally correct or not in any external sense, as there is no further standard available to compare the model to. However, because the model is (ideally) created without reference to the actual implementation, the risk of a systematic error that is present both in the implementation and in the model is significantly lower than the risk of having the error in one of the two artifacts (implementation and model) only. In practice, this means that when tests are generated and executed, an erroneous model usually manifests itself by resulting in failing tests.

In addition to this automatic feedback loop, the correctness of models is usually enforced also through peer reviews. Teams review both the models as well as the generated tests, the latter usually in a human-readable format such as message sequence charts (produced automatically by Conformiq Designer).

Test Generation

The core of Conformiq Designer 4.X is its semantics driven, symbolic execution based test generation algorithm. The algorithm traverses a part of the (usually infinite) state space of the system model. The explored part in itself is infinite also, but is yet only a part of the whole state space. Conformiq Designer searches this state space part for "checkpoints", which can be thought of as testing goals. The number of checkpoints generated by a system model depends on the testing heuristics selected by the user from the set of available ones. For example, selecting "transition coverage" as one of the test generation heuristics causes every arrow in every statechart to become a checkpoint, whereas selecting "atomic condition coverage" causes the evaluation of any atomic Boolean expression in the model to true or false to become a checkpoint (two checkpoints per any atomic expression).

Conformiq Designer then constructs test from the explored part of the state space by selecting paths from it that lead to checkpoints, and converting those paths to tests. Every input on an execution path (to the system model) becomes a test stimulus (to be sent to the real

system), and every output on an execution path becomes an expected output (to be verified during test execution).

Because there are often many different ways to put together a set of test cases that covers every reached checkpoint, the tool uses a combinatorial optimization method known as "set cover optimization" to minimize the total cost of the generated test collection. The cost function has been selected so that it provides a balance between test case length, number of test cases, number of test steps, and independence between different tests. Set cover is an NP-complete optimization problem (meaning finding an exact solution is exponentially hard in the worst case) and the tool reverts to approximated solutions when finding the exact solution would take too much time.

Because the models are open, meaning that they communicate with an unspecified environment, the models cannot be directly explored or simulated as simulating an open system requires some model of the environment also. Concretely, a straightforward simulation of a system model would need to be stopped at the moment the model waits for the first input, as the input comes from an unspecified environment and is thus not fixed. One could generate the inputs "randomly", but this is not a scalable approach which is very easy to demonstrate. This is why Conformiq Designer uses an algorithmic approach known as constraint solving to efficiently handle the unspecified input messages; this is why the approach is also known as symbolic execution as the input messages are represented internally as variables whose values are fixed only later by the constraint solving process.

Non-Redundancy

Every test case generated by Conformiq Designer has a reason why it belongs to the published set of tests: it either covers a checkpoint that no other test case covers (so it is the only test for that particular item), or it covers a checkpoint earlier than any other test case (meaning that it covers it after a shorter input and output sequence than any other test case). Our tool enforces the testing of every reachable testing goal with as short test sequence as possible so that if something goes wrong in the actual implementation it is easy to isolate the problem.

This means that all the generated test sets are "non-redundant". Every test case has a reason to be in the test set, and the reason is easily documented from the traceability data available to the user.

Test Generation Heuristics

A Conformiq Designer user can combine any subset of the available test generation heuristics to guide test set generation. The heuristics available are:

- Requirements coverage, converting every reference to an (external) requirement in the model into a checkpoint
- Transition coverage, converting the execution of any individual UML transition into a checkpoint
- State coverage, converting the entry into any individual UML state into a checkpoint
- 2-Transition (switch) coverage, converting the execution of any sequence of two individual UML transitions into a checkpoint
- Condition coverage, converting the evaluation of any Boolean condition that controls an if-, while- or for-statement into two checkpoints, one for Boolean true and one for Boolean false
- Atomic condition coverage, converting the evaluation of any Boolean condition that appears as a subexpression of the test of an if-, while- or for-statement into two checkpoints, one for Boolean true and one for Boolean false (for short-circuiting languages such as Java this has been shown to supersede in strength MC/DC, modified-condition-decision-coverage, which is popular in many areas of the embedded software industry)
- Boundary value analysis, converting the limit cases of all integral arithmetic comparisons in the model into multiple checkpoints, depending on the comparison operator used
- Parallel transition coverage, converting any execution sequence of two transitions within two distinct statecharts into a checkpoint
- All transition paths coverage, converting the execution of any transition sequence into a checkpoint of its own (can be only used on models whose execution

always terminates)

- All control paths coverage, converting any control flow path into a checkpoint of its own (can be only used on models whose execution always terminates)

- Statement coverage, converting every statement into a checkpoint of its own

- Method coverage, converting entry into any method in the model into a checkpoint

In addition to the basic test generation heuristics, it is possible to create more complex test generation goals. Combining multiple testing goals into compound goals is possible by using all-pairs and all-combinations operators within a model. It is also possible to force the same testing goals to be covered multiple times under different contexts, e.g. different system configurations, by using a feature known as coverage regions that is available in the tool.

State Space Explosion

Algorithms based on state space enumeration typically suffer from a problem known as state space explosion. This means that the state space generated by a computer program can be astronomically large compared to the size of the program itself. For example, a simple program that just increments a 32-bit counter sequentially has a state space of 4.3 billion states. A symbolic state space exploration procedure alleviates part of that problem but does not solve it fundamentally. Even worse, the actual state space of a Conformiq Designer system model is typically infinite, even though in practice it does not matter whether it is mathematically truly infinite or only extremely large.

Because of this, Conformiq Designer can never explore the whole state space, but explores only a prefix of it. The algorithm by which Conformiq Designer selects which paths to expand and to which extent has been carefully tuned during the last eight years and its details are a trade secret. Conformiq Designer provides the user with tools to modify the state space exploration algorithm if necessary, both through global, project-level settings as well as through annotations in the model, to eliminate the issue of missed checkpoints (all checkpoints not covered during test case generation are always reported to the user).

Probabilities and Priorities

There are tools on the market that focus on what is known as "stochastic" or "Markov-chain" test case generation. Typically this means that they generate paths through a finite state machine according to some path enumeration heuristics that takes into account transition probabilities, meaning annotations provided by the user that attach numeric probability values to the transitions (arcs or arrows) in the finite state machine model. Conformiq Designer also supports probabilities, but they can be used anywhere in the model and not just on transitions.

Their use is completely optional, and using them does not turn Conformiq Designer into a simple Markov-chain enumerator. When probabilities are used in a model, two things happen: (1) the test generation algorithm is modified so that it searches for highest-probability test cases for the different checkpoints, overriding the default heuristics of finding shortest test cases to the checkpoints; and (2) when test cases are reported, their compound probabilities are also reported, helping users to execute test cases in their priority order or to execute only the most important ones.

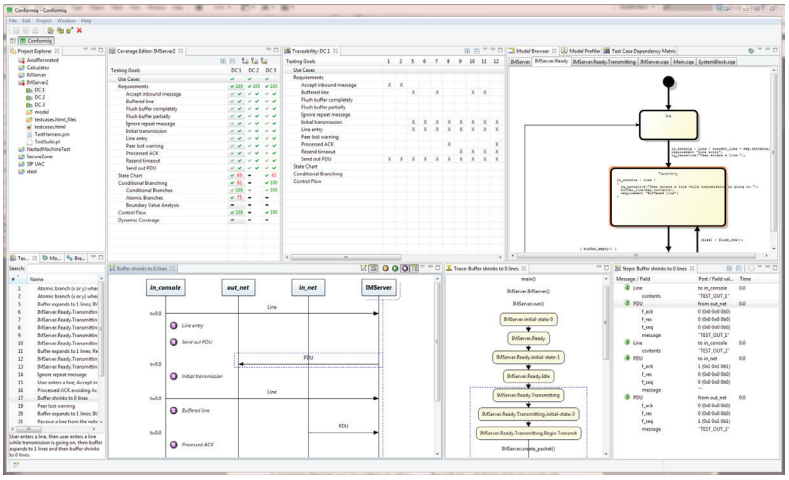
As an alternative to using probability values (floating point numbers from 0 to 1), it is also possible to use priority values. The only difference is that probability values are multiplied together and priority values added together; in both cases, the test case with the numerically highest total probability or priority value is considered the most important.

Incremental Test Generation

Typically a system model goes through multiple edits and revisions. Conformiq Designer has an incremental test generation algorithm that, upon a model revision, does not generate a test set from scratch, but first checks which of the tests already available are still valid for the revised model. After this check, the tool then generates test cases to fill the coverage gaps that have been caused by changed or new features in the edited model. Reusing the existing test cases to as large as an extent possible when a model changes helps users with test set version management, and enables them to execute the newly generated test cases first. This also means that only currently valid test cases are included for regression testing, which can dramatically reduce the size of the regression test suite.

Automatic Test Case Naming

When our tool generates a test case it also generates a name for it. The algorithm that does this takes different available potential name tags for the test case, including requirements covered in it, narratives and scenario tags attached to it and transitions and states traversed, and then selects combinatorially a name that is at the same time unique among the tests, short and descriptive. In practice, this leads to much better test case naming than only using for instance sequence numbers as test case names. Users can change the computer generated names manually, and it also possible to revert back to computer-generated names for individual test cases if desired.



Conformiq Designer

Automatic Preambles, Postambles and Dependencies

In the OSI testing methodology, a test case is commonly divided into three parts: a preamble, a test body, and a postamble. The idea is that the preamble is a "setup phase", the test body is the "actual test" and postamble is a "cleanup phase". This can be interpreted so that the test case is not assumed to fail in the setup phase; if the system fails in the test body it is a real fault, and if the system fails in the postamble it does not matter because there should be another test testing the postamble features in its test body. The purpose of this structure is to make it easier to map system faults (observed defects) to individual tests, and to map tests to individual requirements.

Conformiq Designer has support for this type of test generation as an option that can be activated in any Conformiq Designer project. The OSI test generation support automatically identifies preambles, test bodies and postambles in all generated test cases, and also automatically produces a test case dependency matrix which shows how test cases depend on each other. It also causes the test cases to be published in their dependency order so that dependent test cases come later than the test cases they depend on. (On a detailed level the dependency relationships form normally a partial ordering, and the publication order is obtained by topological sorting. Sometimes there can be cycles in the dependency order; they are reported in the user interface and ignored during topological sorting.)

Test Data Variation

Normally Conformiq Designer produces a minimum number of set cases to cover all the testing checkpoints defined by the user's choices of testing heuristics. It is possible to get more test cases and more data variation simply by enabling the tool's automatic data variation feature ("data perturbation"). It causes every test case to be generated in multiple copies, every copy following the same general logic but having different input (and typically output) data values. We have taken special care to make sure that the generated data variants are, in addition to being correct, generated deterministically. This means that even the varied test cases do not change from one test case generation run to another, and even if you delete the test cases and regenerate them, you still get the same variants back. Naturally, if you explicitly want to change the data variation pattern, there is support for that.

Use Cases

It is possible to augment the system model with use cases. Use cases are sequences of patterns of input and output events, and they can be organized hierarchically in the tool. Every use case can be used a testing goal, causing Conformiq Designer to generate at least one test case that implements and fulfills the use case.

This feature has multiple uses. One of them is to employ use cases to represent complex requirements and thus use them as a device for enhancing requirements

traceability. The other is to force the tool to generate a specific testing scenario among all the other computer-selected test cases, for example, to reproduce a known bug or to make sure that the basic scenarios and use stories a business analyst or software architect has thought of get tested straightforwardly.

Test Case End Conditions

Normally Conformiq Designer generates the shortes test cases possible, but sometimes this can lead to a situation where a test case ends, undesireably, in the "midst of an action", leaving the system in an unclean or unfinished state. This can be prevented by tagging parts of the model as "incomplete" for the purposes of test generation; the tool then only accepts tests that end the model outside any of the areas tagged "incomplete" and thus avoids producing tests that would leave the system under test in an unclean state (assuming that the test pass; when a test fails, the state of the system under test is generally assumed unknown).

In addition to supporting multi-core computers, we have also a cluster based solution that distributes test generation across multiple computers.

Exact Rational Arithmetics

The tool supports rational number arithmetics in addition to integer arithmetics and uses exact rational arithmetics internally. The benefit of using exact rational arithmetics is that there are no rounding errors during test generation. Indeed, if you have rational numbers in your model you can get a test case where the system under test should respond with value 1/3, a value not representable in ieee floating point arithmetics! The exact rationals are typically rounded into ieee doubles or other floating-point formats when test cases are published, and when used as criteria for validating system outputs, should be augmented with a tolerance interval.

Test Generation on Multi-Core Computers

Our tool has a carefully crafted multi-threaded test generation algorithm that generates the same test sets deterministically regardless of the number of processor cores available on the computer, and the multi-threaded algorithm is used automatically to take advantage of all the processor cores available.

The test generation algorithm parallelizes relatively efficiently, leading typically to roughly saving 90% of test generation time by scaling from one to sixteen processor cores. This is important because it improves the productivity of model driven test engineers as they need to wait less before they can see their newly generated tests.

Test Generation on Clusters

In addition to supporting multi-core computers, we have also made available a deterministic cluster based solution that distributes test generation across multiple computers also over standard 1GB Ethernet communication links. This technology enables serious users to create computational clusters for rapid test generation that can be shared across the users.

Comparison with Other Approaches

Other model based testing approaches have also been implemented in the industry, the two prominent ones being (1) finite state machine driven test generation and (2) test model driven test generation.

Test generation from finite state machines is easy to implement and solutions in this area date back to the 1970's. One typical approach is to run a path generation procedure on a finite state machine model that goes through all the arcs of the machine; one algorithm to achieve that is the "Chinese Postman Walk". Another approach is to sample the paths randomly. This, when augmented with probabilities, leads to stochastic or "Markov-chain" path generation.

The problem with finite state path generation approaches is that they do not usually solve the test data generation and test output verification problems in a productive manner, and generating inputs and verifying outputs ("test oracle") is left completely to the user. For example, the user may need to augment transitions in the finite state machines with programming code to generate inputs, and with programming code to verify the outputs. In addition to being resource intensive, this approach makes maintaining finite state machine models and

creating modular models very difficult. All these issues are solved with Conformiq Designer.

The idea of test model driven test generation is that instead of modeling the intended behavior of the system under test, one creates a "model" of a tester, and then straightforward execution of the model (often in a random fashion) constitutes test generation. Even though this can be well called model based testing, it is basically equivalent to creating hand-coded test procedures in any programming language, an approach as old as software engineering itsef. The tester models need to contain procedures both for input generation as well as for output verification, and suffer from the same maintenance and modularity problems as the finite state machine approaches.

Implementation

The test generation algorithms of Conformiq Designer are implemented in C++ and the distributed architecture for supporting multi-core and grid platforms is based on corba. The tightly coupled, proprietary test generation algorithm ensures maximum scalability and efficiency to the user, unlike solutions based on off-the-shelf third-party components.

About Conformiq

Originally established in 1998, Conformiq is a leading provider solutions for automated test design and advanced model-based testing, dedicated to improving test design processes within software-intensive product companies operating in business-, mission- and life-critical industry segments.

Conformiq Designer™ is the company's fourth-generation test design tool, built upon a decade of advanced basic and applied research as well as testing and test design experience.

Privately held, independent and known for extraordinarily responsive customer service, Conformiq is the partner of choice for companies who are ready to step ahead of the curve.

For more information about Conformiq and the company's software and services, please visit www.conformiq.com.