

Functional Verification Technology and Methodology Backgrounder

Tom Fitzpatrick
Mentor Graphics

Introduction

The advent of new technologies—such as constrained-random data generation, assertion-based verification, coverage-driven verification, and formal model checking—has changed the way we see functional verification productivity. Simulation speed is no longer the primary benchmark on which a verification process should be evaluated. Although simulation continues to be the heart of any realistic verification methodology, an advanced verification process enables users to manage the application of the aforementioned technologies in a complementary way, providing confidence that the myriad corner cases of today’s increasingly complex designs have been covered.

This advanced verification process is made possible by the application of testbench automation and formal property checking to functional verification. Therefore this paper discusses basic concepts, values, and methodologies in order to foster an understanding of what these advanced functional verification technologies can do and how to most effectively apply them.

Defining a New Verification Strategy

Functional verification answers the question “did I build the right *thing*?” as opposed to implementation verification, which answers the question “did I build the thing *right*?” Until recently, verification engineers only wanted to keep doing what they were doing, but with a faster simulator. Therefore, the typical tool evaluation consisted of running an HDL model with an HDL testbench (usually just a directed test) on different simulators. Whichever simulator got to the end quickest with the “right answer” was the one they’d choose.

However, it is no longer sufficient to keep doing the same thing only faster. Designs have become way too large and complex for users to predict all of the corner cases successfully, so verification strategies must change. In determining how they must change, it is necessary to keep in mind some basic goals of any comprehensive verification process:

1. It must verify that the design does everything it is supposed to do.
2. It must verify that the design does *not* do anything it is *not* supposed to do.
3. It must show when the first two goals have been met.

Because Verilog and VHDL, and the simulators that supported them, did not provide the features necessary to satisfy these goals, constrained-random data generation, assertion-based verification, coverage-driven verification, and formal model checking were born.

As designs have kept up with Moore’s Law, there is obviously more and more functionality to be verified. With time-to-market pressures also increasing, it is imperative to find a way to perform the necessary verification, which can take as much as 70 percent of the design cycle, in less time. Since every feature needs to be tested, the only ways to meet schedule demands are:

1. Eliminate features from the design so there are fewer tests to write. Presumably the

features were designed to meet the market demand, so removing some of them may make the product non-viable. Therefore, this is not really an option.

2. Add more people to the verification team so they can write more tests. This also is not really an option due to shrinking budgets and also to the fact that adding people doesn't necessarily scale linearly.
3. In fact, the only viable option is for verification teams to be able to produce more tests with less redundancy to cover the desired features more quickly.

The optimal verification solution helps engineering teams travel this third path to verification productivity via the application of advanced verification technologies and methodologies. As we will see, the technologies themselves have strengths and weaknesses, so the important thing is to apply them in a way that uses the technologies to their best advantage in a complementary way.

Constrained-Random Data Generation

Simulation has long been the “workhorse” technology for verification. Verification engineers write tests to provide stimuli to the design and check that it responds correctly. Each of these tests takes time to write, debug, and run. Once a test is working and confirmed to cover a particular feature, it becomes part of a regression suite and the user moves on to write a new test, which also takes time to write, debug, and run. Such an approach requires the test writer to create an explicit test for each feature of the design to be covered (the first goal of verification), which requires more tests and more time as more features are added to the design.

In a simulation-based flow, before any test can be written, the verifier must assemble the infrastructure necessary to drive stimulus into the design and record and check the results. This testbench development can take a substantial portion of the verification time and is usually on the critical path of the overall project. The term *testbench automation* refers to tools and methodologies that assist in assembling such environments and that allow the verifier to check the greatest number of features in the shortest amount of time. The quality of a testbench automation solution, therefore, is determined by how quickly the environment can get up and running and how quickly tests can be developed that actually verify the desired functionality of the design.

One can picture the testbench as a large machine with a number of knobs and switches on it, and a test as the commands used to turn the knobs and flip the switches in a certain order. The more knobs and switches there are, and the more settings each has, the more flexibility the test writer has to describe a greater number of complex scenarios. These scenarios explore different paths through the state space of the design and, thereby, check specific features. The productivity gain is achieved by allowing the verification tools to automatically control the knobs and switches to generate the specific scenarios to test.

This aspect of testbench automation is achieved through the use of *constrained-random* (CR) simulation, in which each “test” actually describes a set of possible scenarios and the simulator

itself chooses a specific scenario for each invocation. To assist in actually controlling the knobs and switches, verification languages such as SystemVerilog and SystemC allow the user to describe stimulus scenarios in terms of *constraints*, which limit the set of legal values for signals or transactions that drive the design. The simulator thus generates random values for stimuli, with the constraints ensuring that the generated scenarios are valid. A new scenario is run simply by re-running the simulator with a different random seed, causing different, but equally valid stimuli to be generated, thereby checking a new feature.

Consider a bus-based design in which the protocol supports both single-word and burst accesses. A single test may be written in terms of the *transactions* that occur on the bus, allowing both the transaction type (read/write) and the bus mode to be randomized. Using CR, this *one* test could verify that *all* bus modes are implemented correctly, including uncovering corner cases such as a burst-read followed immediately by a single write. Now suppose that multiple devices sit on the bus and respond to different address ranges, where some devices do not support all of the bus modes. We simply refine the original constraints to generate the addresses randomly and constrain the bus mode to one that is supported for that address range. Now this single test can exercise each of the devices on the bus in every mode that it is intended to support (and none that it does not).

Coverage-Driven Verification

In this example, you can't tell exactly which addresses and modes will be exercised until you actually run the test, since they are generated randomly. It is therefore necessary to keep track of some information during the simulation to ascertain which scenarios actually were run. This activity is usually referred to as *functional coverage*, the general metric that tells you how well you're doing in your verification process. Using functional coverage metrics to ascertain whether (and how effectively) a particular test verified a given feature and feeding that information back into the process to determine what to do next is called *coverage-driven verification* (CDV). In the context of today's advanced verification, CDV means the automatic recording and analysis of information that indicates whether scenarios have been exercised and the ability to use that information to target additional verification efforts more effectively.

CDV relies on the random generation of stimuli to produce multiple scenarios automatically from a single test. The inherent randomness of the process allows the test to uncover corner case bugs that the designer may not have considered. As coverage data are collected, new tests are created (either automatically or manually) that modify the constraints in an attempt to target previously uncovered features or scenarios. Proper execution of CDV requires the user to specify, via assertions or other mechanisms, the *coverage points*, or specific behaviors, which must be exercised.

The term *functional coverage* has different meanings, depending on context and source, so clarification is worthwhile. In its most basic form, functional coverage is simply a metric to track whether all required scenarios have been tested. Even an Excel spreadsheet with check-

boxes to indicate that specific tests have been completed is a form of functional coverage. The unsophisticated user might run a bus-based test and look at waveforms to determine which addresses and modes were actually run and check them off on the spreadsheet.

Just as there are different types of code coverage (line, toggle, path, expression, etc.) so too there are various sub-classes of functional coverage, depending on what exactly is being measured. Terms such as *test coverage* and *specification coverage* measure whether features have been tested, based on the verification plan or the functional specification, respectively. Note that these types of coverage readily lend themselves to the spreadsheet model mentioned above. The key is to build into the verification process the ability to record and report on these metrics.

Structural coverage is an advanced sub-class of functional coverage initially developed by 0-In[®]. There are certain design components, such as arbiters, FIFOs, clock-domain crossings and the like, which are notoriously difficult to verify. However, because components of these types are relatively well-understood, we know what is required to ensure that they are properly covered. Structural coverage is the practice of identifying these components in the design and automatically inserting the proper functional coverage metrics to report on their verification progress.

In addition to the *analysis* of functional coverage as a metric in itself, it is useful to discuss the specific methods for *collecting* the information in the first place. Unfortunately, the data collection itself is also referred to as functional coverage, so one must be careful to make sure of the context of the discussion. In the context of data *collection* there are two basic types of functional coverage recording, which we will refer to as *data-oriented coverage* and *control-oriented coverage*.

Data-oriented coverage is the recording of sets of variable values at a particular point during the simulation. In a network application, it might be recording the contents of network packets as they enter and exit a model. In the bus-based example above, it might be sampling the address and bus mode at the start of each bus cycle. Such recording is usually done during the stimulus generation process to ensure that you've actually created all transaction types to all address spaces or other analogous scenarios. It is also often done during the results checking process to record that the device responded with the correct results. In either case, data-oriented coverage recording is typically done in the testbench, not in the device under test (DUT). The result is typically a matrix view of how many times each variable achieved a particular value when other variables had specific values. Each unique combination of values is assigned a *bin* (counter), and the tool reports the number of occurrences of each unique combination.

Control-oriented coverage is the recording of specific temporal behaviors that occur either on a bus or inside the DUT. Typically, control-oriented coverage is specified using assertions, where the tool records that specific protocol modes have been executed, or specific scenarios have occurred; for example, a retry occurring during the Nth packet of a frame transfer or an interrupt while the CPU is executing a jump instruction. The typical result of temporal, control-oriented coverage is a set of counts of how many times each sequence occurred. Also keep in mind that sequences specified by assertions can be flexible, such as “b occurs from 1 to 3

clocks after **a**.” It is often useful to know how many times the sequence was satisfied by **b** coming one clock after **a** and how many times with **b** coming 3 clocks after **a**. This particular sub-class of coverage is typically referred to as *assertion coverage*.

Formal Model Checking

Also referred to as *property checking*, formal model checking (FMC) is distinct from its formal verification cousin, equivalence checking. While both rely on detailed mathematical analysis of the design, equivalence checking refers to comparing two different implementations of a design to see if they are functionally equivalent. In contrast, FMC exhaustively examines all of the possible states of a design to determine if any of them violate a specified set of properties.

The properties themselves, often expressed as assertions, represent a precise description of sequential (“this happens after that”) or invariant (“this will never happen” or “that will always happen”) behaviors about the design, with each property considered a “piece of the specification.” The properties themselves can describe design behaviors, or they can describe limitations on the behavior of inputs to the design, in which case they are called constraints. The use of the term *constraint* in this context is somewhat consistent with the notion of constraints in CR simulation (see page 3), although the mechanisms for describing the two types of constraints may differ.

In theory, it is possible to use FMC to fully verify the functionality of an arbitrarily complex design. In practice, however, the complexity of the mathematical analysis involved typically limits the application of FMC to the sub-blocks of a design. However, because FMC can analyze all of the possible states the design can reach, it can verify specific functionality exhaustively, without requiring the user to write a testbench. Instead, the properties serve as targets for the verification with the constraints defining the environment in which the block will operate. A further advantage of using FMC at the block level becomes apparent when the block is inserted into the larger system, as will be seen in the following section.

Formal Model Checking and Simulation

Formal model checking is a complementary technology to simulation and an integral part of a CDV methodology. FMC exhaustively evaluates all possible states of the design to ensure that they conform to a set of assertions and that all coverage points have been reached successfully. When the exhaustive analysis of FMC is coupled with simulation—using a technique pioneered by 0-In called *dynamic formal verification* (DFV)—the CDV methodology is greatly enhanced.

It is important to be aware of some basic facts regarding FMC and simulation. Most simulators, including ModelSim®, VCS (Synopsys), and NCSim (Cadence) rely on event-based semantics to evaluate the behavior of a design. Formal tools, on the other hand, evaluate the design in terms of cycle-based semantics. These two ways of evaluating a design have subtle differences, so unless you’re careful it’s possible to have a formal tool and a simulator get different results

about a particular property. For that reason, 0-In Formal Verification automatically re-runs any FMC counter-examples in the simulator to ensure consistency between the two interpretations.

A key goal in standardizing both SystemVerilog and PSL as assertion languages was to establish a common semantic for the properties that would guarantee a consistent answer between simulation and FMC. Such consistency allows the properties that are formally verified at the block level to be used as monitors in simulation, particularly when simulating larger systems. If a block is formally verified to behave a certain way given a set of constraints on its input, it is necessary to verify that the rest of the system in which the block operates also conforms to those same constraints. With the assertion monitor automatically checking the block inputs during system simulation, we can verify that the input constraints were correct and therefore that our FMC analysis is valid.

If FMC can formally prove that the block behaves correctly for a given set of input behaviors, and the system only drives the block in a manner consistent with those behaviors, then the block is guaranteed always to work correctly in the given system. Therefore, it is not necessary, in full-chip simulation, to try to create overly complex stimulus scenarios that are aimed at getting the block into a particular state. The simulation can simply focus on the end-to-end behavior of the full chip, secure in the knowledge that the block will always behave correctly.

Assertion-Based Verification

While the term *assertion-based verification* (ABV) is often used to refer to the use of assertions as monitors in simulation, ABV more correctly refers to the use of assertions in both simulation and formal verification. Assertions are a central part both of simulation and formal model checking. They are concise mathematically precise descriptions of behavior that must hold or that constrain the operating environment of the block. In addition, they can also describe *coverage points*, which are scenarios that must be exercised by the verification process, regardless of the specific technology used.

Assertions themselves are often categorized as either *white-box* assertions or *black-box* assertions. White-box assertions provide the user visibility into the internals of the design, and often reflect some specific implementation choice made by the designer (e.g., “this fsm is one-hot”). Black-box assertions describe end-to-end behavior of the block and are independent of the actual implementation choice of the designer (e.g., “all packets that come in will go out”). One major benefit of assertions is that they give the designer the ability to participate actively in the verification process by capturing his or her *intent* about what the design (or the environment) should do. The question often asked when using assertions is “how do I know I have enough?”

If the purpose of assertions is to capture pieces of the specification against which to verify the design behavior, then there are two ways to look at the question of having “enough” assertions. The first is to ensure that you have assertions for all of the critical behaviors you want to verify. This is typically part of the verification plan. The second is to make sure that all parts of the

design are adequately related to one or more assertions. This is referred to as *assertion density*, which is measured by the number of clock cycles required for register values to propagate to an assertion. The fewer clock cycles required for this propagation, the easier it is to perform the formal analysis to evaluate whether or not the assertion is violated. When every part of the design is checked by an assertion within 2–3 clock cycles (an empirical measurement which experience has shown to be a good rule of thumb), then you have “enough” assertions.

Technology Value Statements

Coverage-Driven Verification Value Statement

To understand the value of a CDV solution, there are two aspects of the methodology to consider. The first is how well the proposed solution can shorten the development time of the testbench. The second is how effective and efficient the related tests will be at exercising and verifying the targeted features of the design.

In assembling the testbench, the level of value comes from a combination of:

- the programming features supported by the verification tools
- the quantity and quality of IP provided to reduce the amount of coding required
- the quality of guidelines on how to structure a reusable environment and support multiple levels of abstraction (see below).

Once the testbench is assembled and is working properly, the greatest advantage lies in applying CR simulation in test development. To be used successfully, CR simulation requires three fundamental characteristics. First, the simulator must support a language that is able to specify constraints for randomizing values. Second, the simulator must include a *constraint solver*; that is, the ability to arrive at a set of random values consistent with the set of constraints specified. Obviously, the better the constraint solver is at maximizing the size of these value sets, the more varied and meaningful scenarios it will generate. Third, because it is not possible to predict *a priori* what scenarios will actually wind up being generated by constrained random simulation, functional coverage is required to determine that desired scenarios were actually exercised. From a methodology standpoint, CDV also requires the tests to be self-checking.

A good CDV methodology supports the first goal of verification (verifying the design does everything it is supposed to do) in that it makes it easier to assemble the verification environment and create a set of tests that will cover specific features that the design is intended to support. Keep in mind that there may be some features that are better tested simply by writing a directed test where the test writer explicitly exercises a given scenario directly³. As an example, consider an interrupt mask register test. The likelihood is exceedingly small that random stimulus would be able to, in succession, set each mask bit, generate the corresponding interrupt, and then clear the

mask bit. However, for the vast majority of features, CDV will most effectively rely on constrained random tests to generate many different valid scenarios to prove specific features.

CDV also supports the second goal of verification quite well (verifying the design does *not* do anything it is *not* supposed to do). The value in randomizing stimulus is that the unpredictability of the process makes it more likely that it will uncover corner cases where the designer may not have sufficiently accounted for all of the ways in which different subsystems of the design will interact.

Formal Model Checking Value Statement

We stated above that the value of CDV is that it uncovers scenarios in simulation that the designer may not have considered up-front. By definition, the exhaustive nature of formal analysis guarantees that *all* possible scenarios will be evaluated. Therefore, if we can use FMC to verify formally that a block will always function correctly for a given set of input constraints, it is not necessary to simulate at the block level, whether using constrained-random or directed tests, and the simulation activities will be reserved only for verifying that the blocks play well together.

There are those who claim you can actually use FMC to verify an entire chip hierarchically, eliminating the need for simulation. The theory is that if you can prove blockA functions correctly, assuming a set of input behaviors, and you can prove that blockB, which drives those inputs, cannot violate these behaviors, then everything is guaranteed to be correct. This is referred to as the *assume-guarantee* paradigm of formal verification, and it is a powerful concept, in theory. Of course, you then have to prove that the block driving blockB's inputs conforms to those assumptions, and so on. In reality, the complexity of managing the assume-guarantee relationships between properties and blocks makes this a much more difficult prospect than it might appear in theory.

However, there is a substantial (and growing) segment of the user community who recognize the potential benefits of FMC, particularly for mission-critical components (arbiters, clock-domain crossings, FIFOs, etc.) The real value of FMC is that its exhaustive analysis is very useful in finding bugs, both in the design and in the specification. When formal analysis shows that a problem occurred, debugging involves determining whether the problem is actually in the design or in the assertion. It may be that the assertion writer misinterpreted the specification, in which case the assertion must be changed, or it may be that there is an actual bug in the design that must be fixed. Because FMC is exhaustive, all such cases will be discovered and therefore fixed. When these verified assertions are used in simulation, as discussed above, there is an added level of confidence that the assertions are correct since the simulation and the test-bench provide yet another objective comparison of the assertions against the desired behavior, as specified by the test.

Coverage-Driven Verification and Formal Model Checking Value Statement

CDV relies on the random generation of stimulus to automatically produce multiple scenarios from a single test. The inherent randomness of the process allows the test to uncover corner case bugs that the designer may not have considered. FMC exhaustively evaluates all possible states of the design to ensure conformance with a specification, expressed as a set of properties or assertions, thus guaranteeing that corner case bugs will be detected. It is accurate then to describe constrained random simulation as an approximation of FMC, using the simulator in place of formal analysis.

One problem, as mentioned above, is that FMC is limited to static analysis of the sub-blocks. This limitation comes mostly from the memory and computational resources necessary to do the exhaustive analysis. However, dynamic formal verification *amplifies* a given simulation trace, enhancing the CDV methodology.

Given a set of target behaviors, expressed as assertions or coverage points, DFV uses formal analysis of “interesting states” along the simulation trace to determine if it is possible to reach any of the coverage points and/or violate any assertion. For functional verification, an “interesting state” is one where a new or rare design behavior has occurred.

To illustrate the concept, consider a bug that occurs many cycles deep into a simulation. A particular bug might occur only if a particular FIFO in the design is full and a particular FSM is in state FOO. With simulation, it is possible that a test will manage to fill up the FIFO, while the same or another test can also get the FSM into state FOO. But to get exactly the right combination of random stimuli for both events to occur together may be a low probability event. Constrained-random stimulus may require the testbench to contain *a large number* of knobs and switches to allow such fine-grained control to exercise all such corner cases. With DFV, from a simulation state where the FIFO is full, formal analysis examines all possible states leading up to and beyond this point in time, enabling it to identify and report the particular sequence that leads to the state that manifests the bug. Thus, DFV coupled with CR uncovers a bug that would have required additional testbench code to detect using CR alone.

Without DFV, the CDV user would have to know that a potential bug is lurking and modify the stimulus constraints in an attempt to guide the test to the right set of states—typically a difficult exercise. In actuality, DFV can amplify any single scenario into a substantial number (10,000s or more) of effective tests. With this in mind, the actual set of constraints for the testbench can be much simpler because they do not need to accommodate the fine-grained tweaking discussed above, saving the substantial time it takes to conceive and code such elaborate constraint environments.

Methodology Discussion

The preceding discussion has focused mainly on the application of testbench automation and formal property checking technologies to the verification problem. In order to be most effective, however, it is as necessary to discuss *how* to apply these technologies as it is to discuss *what* they can actually (or potentially) do. Such a conversation requires thinking about the functional verification process from a slightly higher altitude.

The initial specification of a design is usually in the form of stating *what* the design will do, with little or no specification of exactly *how* it will do it. The *how* is up to the designer. The job of verification is to ensure that the implemented *how* matches the specified *what*. It's important to remember that in moving from specification to gates, there are often several layers of implementation at different levels of abstraction. One of the keys to a good verification methodology is to make it easy to migrate across different abstraction levels while maintaining functional consistency between them.

Implementation verification is the process of ensuring consistency between the RTL and gate-level implementations of the design. Because of the well-understood nature of the synthesis process, checking RTL-gate consistency is a relatively straightforward process. In this context, the RTL is the “specification” against which the gate-level model is compared. To all intents and purposes, the functional verification process is to ensure that the RTL model properly reflects the desired functionality and is thus a valid target against which the gate-level functionality is compared. So, in the context of functional verification, the RTL is the “implementation” which must be compared to the specification.

Whether discussing implementation verification or functional verification then, the fundamental job of verification is to compare two representations of behavior. There are multiple ways of describing the expected behavior. One way is to use properties or assertions, which FMC can compare to the RTL to exhaustively verify the functionality of sub-blocks of the design. Assertions can also be used in simulation to monitor the executed behavior of the design. Of course, simulation also requires another representation of the expected behavior: the testbench.

Transaction-Level Modeling

Because it is always helpful to think of the verification process in the proper context, it is most useful to keep the testbench focused on *what* the design is supposed to do. The most convenient way to do this is for the testbench to be written at the *transaction-level*. A *transaction* is a representation of an arbitrary activity of the design, and can be represented at various levels of abstraction (e.g. untimed, cycle-accurate, pin-accurate, etc.). Thus, a transaction-level test is one that describes the transactions that should execute. The verification environment is responsible for transforming these transactions into specific behaviors that correspond to the proper abstraction level of the model being simulated. This is most often accomplished through the use of *transactors*, or *bus-functional models* (BFM), whose job it is to perform these transformations.

A proper verification environment is, therefore, one that allows tests to be written at the transaction level and provides the proper infrastructure so that the tests can be used with multiple implementations of the design at various levels of abstraction.

First, let us consider simulation. Regardless of the abstraction level at which they are implemented, transactions always have certain information associated with them (e.g. address, data, etc.). The test is responsible for *describing* the transactions, while the transactors are responsible for *implementing* the transactions (e.g., wiggling the pins, etc.). All of the benefits of CDV discussed earlier can be applied to the generation of the *transaction descriptors*, which tell the transactors what to do and which random constraints may be specified. By establishing a standard set of interfaces to govern the communication between the test, the transactors, and the design, lower-level portions of the infrastructure can be swapped in and out without requiring changes at the higher levels (especially the test). Thus, the same test can drive a transaction-level model of the design through one set of transactors as well as driving an RTL model through a different but compatible set of transactors.

When simulating a transaction-level model, black-box assertions can be used to specify expected behaviors at this level. As the design gets refined down to RTL, the black-box assertions can be similarly refined. Therefore, when the design and the assertions get down to the RTL, these same assertions (as refined) can serve as targets for dynamic formal analysis in order to enhance the automation of the transaction-level test.

Conclusion

Applying dynamic formal verification in conjunction with a constrained-random testbench enhances the value of CDV by allowing substantially more scenarios to be evaluated from the same stimulus. In addition, the ability of DFV to analyze multiple scenarios around a particular simulation trace means that the testbench itself requires fewer knobs and switches, since DFV takes care of the tweaking that the user would otherwise have to code directly. The testbench therefore becomes simpler and easier to create in the first place, and each test achieves greater functional coverage by combining constrained random simulation with dynamic formal analysis. As a result, it becomes more likely that corner case bugs, even in extremely complex designs, will be caught and fixed.

For more information, call us or visit: www.mentor.com/fv

Copyright © 2005 Mentor Graphics Corporation. This document contains information that is proprietary to Mentor Graphics Corporation and may be duplicated in whole or in part by the original recipient for internal business purposes only, provided that this entire notice appears in all copies. In accepting this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use of this information. 0-In, ModelSim, and Mentor Graphics are registered trademarks of Mentor Graphics Corporation. All other trademarks are the property of their respective owners.

Corporate Headquarters
Mentor Graphics Corporation
8005 S.W. Boeckman Road
Wilsonville, Oregon 97070 USA
Phone: 503-685-7000
North American Support Center
Phone: 800-547-4303
Fax: 800-684-1795

Silicon Valley
Mentor Graphics Corporation
1001 Ridder Park Drive
San Jose, California 95131 USA
Phone: 408-436-1500
Fax: 408-436-1501

Europe
Mentor Graphics
Deutschland GmbH
Arnulfstrasse 201
80634 Munich
Germany
Phone: +49.89.57096.0
Fax: +49.89.57096.400

Pacific Rim
Mentor Graphics Taiwan
Room 1603, 16F,
International Trade Building
No. 333, Section 1, Keelung Road
Taipei, Taiwan, ROC
Phone: 886-2-27576020
Fax: 886-2-27576027

Japan
Mentor Graphics Japan Co., Ltd.
Gotenyama Hills
7-35, Kita-Shinagawa 4-chome
Shinagawa-Ku, Tokyo 140
Japan
Phone: 81-3-5488-3030
Fax: 81-3-5488-3031

