Case Study - Using STIL as Test Pattern Language

Daniel Fan, Steve Roehling, Rusty Carruth
NPTest, Inc. LLC
150 Baytech Drive
San Jose, CA. 95134

Abstract

This paper describes the implementation of a test pattern language using STIL [1], the IEEE Standard Test Interface Language (1450-1999), in a next generation, open architecture Automated Test Equipment (ATE) platform. The advantages of this approach in extensibility and easy interface with Electronic Design Automation (EDA) tools for the ATE user are presented. Some challenges of using STIL as a general purpose ATE test pattern language are also presented. The overall EDA and ATE strategy and pattern system architecture based upon STIL are discussed.

1 Introduction

The increasing functional complexity of electronic circuits and the high integration of system-on-chip make testing a challenging task, especially with short time-to-volume, high quality test coverage and low total test cost. These demands are translating into higher volume of test pattern data, shorter device debugging time and tighter design-and-test link. Major ATE vendors typically provide their users a proprietary pattern syntax for implementing test pattern language. A proprietary pattern syntax makes the whole test pattern environment coupled tightly with the closed ATE hardware architecture. This environment also enforces an extra translation process for the test pattern from any EDA's ATPG tool to the ATE test pattern. In the translation process, sometimes some of the pattern information is lost and other times the mismatched resources can not be resolved. Furthermore, the usage of a proprietary pattern language makes it very difficult to support the failure analysis process interface to the corresponding ATPG tool because the ATPG tool produces a different simulation output than the ATE's test results.

The ATE industry is also working very hard to address the needs of reducing total test cost [2]. To address the total test cost, the test life cycle must be considered, including data collection, test generation, test debug, diagnosis, and production [3]. The total test cost can be reduced, if there is no ATE/EDA barrier contributing to the first three elements of the test life cycle. Pattern data can always transfer from EDA to ATE without losing any information. EDA's ATPG tool can also retrieve the test pattern data from the ATE with the fail information to perform failure analysis for diagnosis. The other effort to achieve the reduction of the total test cost is to prolong the life time of the testers. One of the approaches is the migration toward an open architecture platform. To have an open architecture test system, ATE vendors must also support an open pattern language, such that it can be extended by third party vendors.

STIL is being adopted by most companies in the EDA and ATE industries. Some DFT-based ATE companies have adopted STIL as their native language, while supporting their target application very well. However, most of the other major ATE companies are taking the approach of translating STIL to their own pattern language[4].

This paper focuses on the following aspects of adopting STIL as the native pattern language:

- the key reasons for using STIL;
- the challenges to support general purpose ATE;
- the overall architecture of EDA and ATE model; and,
- the test pattern environment.

2 Adoption of STIL

Adopting STIL as a native pattern language is easier for a new company than it is for a company with a legacy of proprietary pattern languages. If the scope of the language matches the tester's application, acceptance is very straightforward. For a general purpose ATE vendor that needs to support broader base applications, such as mixed signal testing and Algorithm Pattern Generation (APG), the trade-off between a proprietary pattern language and a standard language must be carefully evaluated.

The main obstacle to switching to a new pattern language (a proprietary one or a standard one) from an existing pattern language is the wide acceptance of the current legacy language. Introducing a new pattern language creates an additional burden for existing users, since some products need to be tested on both a legacy test system and a new system based upon the new language. The other consideration is the capability of the new language - can it support all the requirements of the target applications? The acceptance of a new pattern language must overcome the above obstacles. To adopt a standard language as the new language is a more complex task than to have a proprietary one because there is no full control of the syntax.

The realization of STIL's strength is the major factor for the commitment to use this standard as our next generation pattern language. Its three most important character-

0-7803-8106-8/03 $17.00 Copyright 2003 IEEE

istics are:

- easy transportability from our legacy proprietary pattern language (M9K) to STIL;
- leveraging STIL to support a broader design and test capability than the current pattern language does; and,
- an extensible standard to support an open architecture.

2.1 Compatibility between Legacy Pattern Language and STIL

STIL is constructed with ease of understandability and transportability in mind [5]. The current STIL standard, STIL 1450-1999, supports our legacy pattern language (M9K) very well. A current M9K pattern file consists of three major components:

- Pindef Table - defines all the pindefs or pindef buses used by the current pattern;
- Vector Definition - defines the mapping of all the timing sequences of each signal or signal bus for each vector; and,
- Pattern Data - the test vector data sequence.

As shown in Table 1, STIL has the equivalent blocks to support each of these M9K components.

M9K	STIL
Pindef Table	Signal Block
pindef	signal
Vector Definition	WaveformTable
timing sequence	event list
Pattern Data	Pattern
INC / RPT	vector / loop

Table 1: M9K to STIL Mapping

The one-to-one mapping between M9K's components and STIL's block constructs eases the uncertainty of moving to a new language.

2.2 STIL Supports Design and Test Flow

STIL is not only a standard pattern data format, it covers the test generation flow ([6],[7]) and ATPG diagnostic (failure analysis) flow [8] as well. Various active working groups of the STIL are expanding the scope of the current definition to encompass design environments [9] and tester targeting [10]. With the adoption of STIL, the goal of eliminating the barrier between design and test is achievable.

2.3 Open Architecture and STIL

The other major driving force for the acceptance of STIL is to support the open architecture tester platform. In an open architecture environment, the host ATE vendor must not define its own proprietary test language (both test program and pattern). The system must allow any third party vendor to have the capability to define its extension to support its plug-in instrument(s). A standardized syntax and extension support via the UserKeywords syntax makes STIL very compatible with open architecture test platforms.

3 Consideration of STIL Extensions

STIL is an ideal pattern language for a DFT-based tester because of the perfect match between the strength of the standard and the corresponding ATE application. For a general purpose SOC ATE, there are some applications that are beyond STIL's scope. Some needed STIL extensions for the next generation tester are:

- Algorithm Pattern Generator (APG) - a general purpose ATE often needs to provide an APG option to support embedded memory testing. STIL does not address the APG syntax because it is ATE hardware implementation-specific. Extension is needed to support the APG instruction set for the new tester platform.
- Analog Pattern Support - a SOC tester will have various analog instruments for different applications. These options generate the need to have pattern syntax to support analog sequencing and analog sourcing data. Extended pattern syntax and usage model will need to be established to support these analog pattern applications [11].
- Subroutine Memory - STIL does have the Procedure statement to support similar capability, but ATE only has a fixed size of subroutine memory. STIL's procedure can be used in some applications that can not map to the subroutine in a tester, such as scan procedure. ATE's subroutine also has some special capability (such as Continuous Loop). Two different extensions are needed: a.) UserKeyword to identify which STIL's procedure(s) should be mapped into subroutine memory, and b.) UserKeywords to support hardware capabilities.
- Synchronization and Trigger between Instrument - The new tester platform provides various synchronization and trigger capabilities between different instruments. This is the key capability to support mixed-signal applications. STIL extensions are provided to support triggers in pattern syntax.
- KeepAlive Vector - a UserKeyword is defined to identify a specific vector that will be used as the KeepAlive vector. KeepAlive is needed when a DUT must keep its clock running from test to test, if the DUT is powered.

All these extensions will use STIL's UserKeyword statement and follow STIL's syntax rules. The intent is to ensure that the extended STIL pattern can be parsed by any EDA tool's STIL reader. It is easy to add UserKeywords to support various hardware capabilities without consideration of the design and test flow. The goal is to allow any EDA tool's STIL reader to skip all these UserKeywords, but the EDA tool still can produce the correct pattern data.

Some of the extensions do have the freedom without considering EDA's STIL reader, such as APG and Analog Pattern support. The other extensions that may coexist with a digital only pattern must support the EDA

feedback loop. This will be ensured by the dual path capability of the validation loop.

The summary of the challenges of STIL extensions are:

- allow the user to take full advantage of proprietary hardware capabilities
- ensure design-and-test link between EDA tools and ATE test environment is supported

4 Design-and-Test Model

With the adoption of STIL as our primary pattern language for the next generation tester platform, the formulation of a STIL-based test pattern environment was driven by the goal of reducing the total test cost. Test generation, test debugging and diagnosis are all part of the test cost equation [3]. The design and test flow model of our tester environment takes full advantage of STIL standard and its extensions. The three major process steps are design-to-test pattern imports, validation loop, and ATPG diagnosis support.

4.1 EDA to ATE Data Flow

The first important step to improve test generation productivity is to eliminate the barrier between design and test pattern translation. The utilization of STIL as the native pattern language not only skips the translation process, it also eliminates the possible information loss during the translation. Furthermore, using the same language for EDA and ATE helps the test debugging process because the user does not need to deal with two different pattern syntaxes (simulation and ATE).

This data flow model takes full advantage of the current P1450.3 (Tester Targeting [8],[10]) working group's effort (Figure 1).

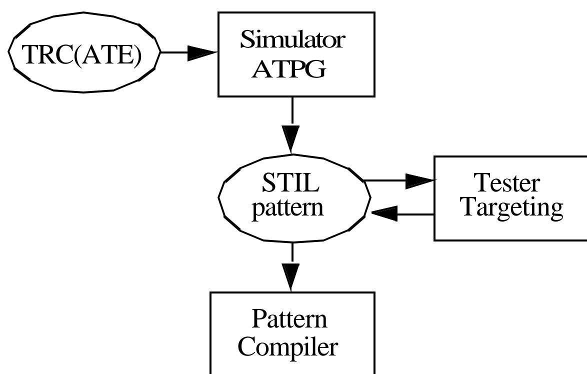


Figure 1. Front End Process

In the early stage of the pattern generation phase, the usage of a tester resource checking (TRC) file in the EDA's ATPG tool will allow the user to produce test patterns that fit into the targeted tester environment. The resultant STIL pattern files will contain the tester resource usage report for test planning purposes. The tester targeting process adds additional information into the STIL patterns that specify how the resources of a specific tester or tester type are to be assigned. The following information will be applied to the EDA's STIL output pattern:

- resource tags for period and event list in the WaveformTable - these tags allow the optimization of tester timing resource usage. There is the possibility that the timing equation and values in a simulation model do not match with the final test specification. There is a need to allow mapping event lists with identical timing values to different timing sequences in the test program or vice versa.
- tag the specific STIL procedure to utilize a tester's subroutine memory - The pattern compiler will expand the procedure or macro into straight line pattern data, but map these specific procedures into subroutine memory.
- tag a specific vector as the KeepAlive vector - If the device requires its clock running during entire test flow, then the specific vector should be identified in each STIL pattern file.

The resultant STIL pattern can directly feed into the pattern compiler to generate a pattern object file for the tester. The EDA simulator will be able to read the resultant STIL pattern and produce the same simulation waveform result because these extensions are either defined as UserKeywords or utilize P1450.3's resource tag.

4.2 The Validation Loop

The second step of the Design-and-Test model is the validation loop. It provides the bi-directional link between EDA and ATE to ensure pattern data fidelity. In the test debugging phase, there has always been the need to modify or generate some patterns on the tester. These modified patterns need to have the option to feed back to EDA for validating with the device model. The validation loop model is shown in Figure 2.

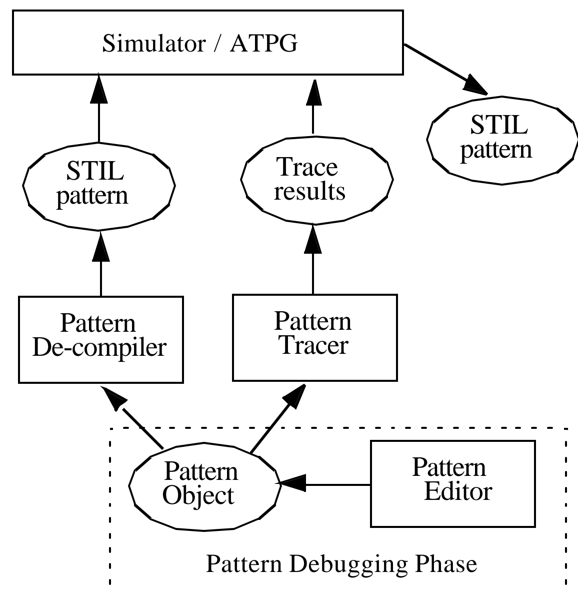


Figure 2. Validation Loop

In the pattern debugging phase, the user modifies the pattern on the tester through a pattern editing facility in the

ATE environment. Two different paths are provided -

- pattern de-compiler generates STIL source file from the modified pattern object file; and,
- pattern tracer performs a tester-level emulation to produce pattern data states that can be formatted as flat pattern data for a EDA tool.

The need for pattern tracer is to support some ATE specific extension in the STIL syntax that may not be handled by the EDA tool, such as the APG instruction set. In this case, the flattened pattern data will be provided to the EDA tool to simulate the device behavior through the pattern tracer path.

4.3 Diagnosis Support

The third step of the design and test model is the ATPG's failure analysis support for fast turn around of device diagnosis. The test fail information on a tester must feedback to ATPG's failure analysis tool with the corresponding failure pattern, so the diagnosis can be performed to locate the failing flip-flop. There are two essential types of data in this step, 1) The corresponding pattern data that can be provided by the validation loop, and 2) The failure information associated with the pattern that is covered by the simulator feedback model of P1450.1 ([8],[9]).

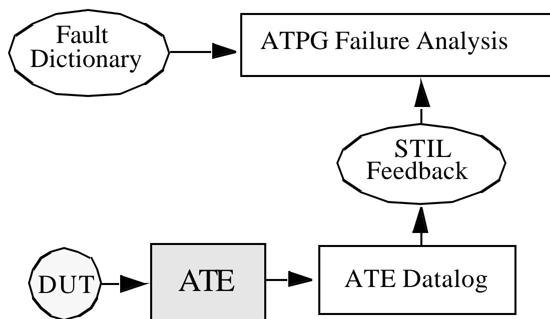


Figure 3. ATPG Diagnosis Support

5 Test Pattern Environment

With the adoption of STIL as our primary pattern language and the design-and-test model, the formulation of a STIL-based test pattern environment was driven by several important goals.

In particular, not unlike user-defined keywords in the STIL language, each component in the environment must have an open architecture, such that the environment can be extended by third party vendors.

Moreover, round-trip EDA to ATE linkage must be preserved throughout every layer of the pattern environment. The system as a whole must not only maintain front-end interface compatibility with EDA tools via the STIL standard, but traceability must be preserved all the way through the back-end of the system (e.g., pattern debugging tools must show traceability back to STIL).

5.1 Subsystems Overview

With STIL as a pattern language, and the goals above in mind, the creation of a modular pattern system consisting of multiple layers and subsystems is possible. In general, the core patterns environment consists of the following subsystems:

- STIL Parser - parses STIL source files to create a virtual pattern representation (VPR) intermediate output.
- Loadable Image Generator (LIG) - takes a VPR as input, then creates a loadable pattern image targeted to specific instrument hardware.
- Pattern De-compiler - takes a VPR as input, creates a STIL source file.
- Pattern Editor - provides pattern data editing capability for pattern objects.
- Pattern Tracer - performs a tester-level emulator of pattern execution.
- M9K-to-STIL Converter - converts a legacy M9K pattern to a STIL pattern for easy migration path

The three different pattern objects in the subsystem are:

- Virtual Pattern Representation (VPR) - hardware-independent intermediate output from the STIL parser.
- Loadable Pattern Image (LPI) - binary image suitable for direct loading on the target instrument hardware.
- Runtime Pattern Repository (RPR) - stores pattern related mapping and meta-data generated by the LIG.

5.2 Features and Benefits

Decomposing a pattern test environment as described in section 5.1 contributes to the overall functionality, performance, open architecture, and extensibility of the test pattern environment.

5.2.1 Two Pass Compilation

In general, the pattern compiler is a two-pass compiler, as shown in Figure 4 below.

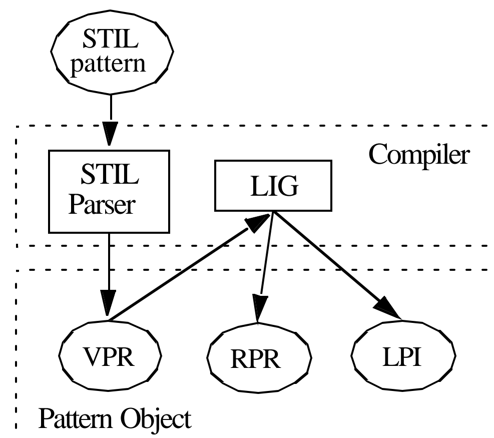


Figure 4. Two Pass Pattern Compilation

In the first pass, a STIL source file is parsed, and an intermediate format VPR file is created.

In the second pass, a "loadable image generator" (LIG), with specific knowledge of the target instrument hardware is executed.The output from the second compilation pass is a binary pattern image suitable for direct, high-performance loading on the target instrument hardware.

The choice of a two-pass STIL compiler system achieves modularity and a separation of concerns in the test pattern environment. In particular, the first pass is only concerned with parsing STIL and storing an intermediate pattern representation, while the second pass generates a pattern image specific to the target instrument hardware. A two pass compiler facilitates an open architecture, because any instrument hardware-specific loadable pattern image generation pass, that is accomplished via an open-architecture LIG plug-in, can reuse a common STIL parser.

5.2.2 Retarget Capability

Having a two pass compiler means the second pass, or LIG compiler phase, can be executed stand-alone with respect to the STIL parser. Therefore, a user can retarget a pattern to a different type of hardware or different hardware configuration (e.g., different instrument type in the same slot among different testers) without the performance penalty of re-parsing the entire STIL source.

5.2.3 Multi-part Pattern Object Decomposition

As shown in Figure 4, the VPR, RPR, and LPI collectively represent the pattern object output from the compiler. In general, there is a one-to-one correspondence between information in the VPR and STIL source. In this regard, the VPR is a hardware-independent intermediate representation of a STIL pattern.

The goal of the LPI is to be directly loadable to the target instrument hardware, thus minimizing load times in production test environments. To this end, the LPI has a very low-level binary format, and is not suitable for storing pattern "meta-data", such as label tables or STIL pattern statement to instrument instruction mapping information. In this regard, the RPR sits between the VPR and LPI, and serves as a place for the LIG to store possibly hardware-specific pattern meta-data.

5.2.4 Interactive Pattern Edit Support

The choice of a two pass compiler and multi-part pattern object format indirectly impacts system capabilities beyond the core compiler and pattern tracer.

In particular, with a binary representation and its one-to-one correspondence with STIL, the VPR is stored in a format to facilitate interactive, context sensitive pattern edit and display within graphical user interface (GUI) tools.

For example, in the VPR, a signal name is only stored once, but all references to that signal are stored as pointers. Therefore, during interactive pattern edit, a user can change the name of a signal in one place, for example, and all references to that signal will be automatically updated.

With the VPR and interactive pattern edit tools, incremental pattern development is also possible. For example, a user can insert a vector in the middle of a pattern block, without having to re-compile the whole pattern.

In general, the VPR and RPR collectively support incremental pattern development and an extension of ATE to EDA traceability into interactive pattern edit and display tools.

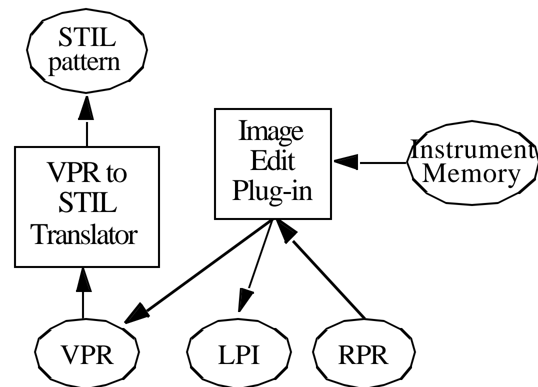
5.2.5 Decompilation

Decompilation is supported primarily in the context of interactive pattern editing. Several components work together to collectively provide decompilation support, including;

- since the VPR is basically a STIL file in binary object format, a generic translator to convert a VPR back into STIL is provided;
- during compilation, a plug-in is expected to instrument the RPR with enough context information to support keeping edits to the LPI file, pattern image in instrument memory, and VPR in sync; this could include, for example, a listing of which loops where unrolled during compilation; and,
- a instrument specific image edit plug-in is needed to coordinate changes to pattern data in instrument memory or the LPI file, while performing the appropriate pattern image to VPR/STIL translations to keep the VPR in sync.

Leveraging the components described above, Figure 5 below summarizes the overall decompilation process. An image edit plug-in is expected to read compiler generated context information from the RPR. When changes to the pattern image are requested, the image edit plug-in coordinates those changes, such that instrument memory, VPR, and LPI always remain in sync. With everything in sync, the VPR to STIL translator can be invoked at any time to regenerate the STIL source pattern.

Figure 5.Decompilation Flow



5.2.6 Loadable Pattern Image

The LPI not only needs to have EDA linkage preserved, but also needs to provide optimized performance for pattern loading. One challenge to reduce production test cost is to reduce the test setup time. With the increasing complexity of the circuit and the high demand of test quality, the number of patterns and the number of vectors

per pattern have been increased. The LPI must support block transfer mode to produce the highest performance.

5.2.7 Legacy Pattern Migration Support

An M9K to STIL pattern converter is provided to support our customers who own our legacy 9000 or EXA3000 testers. All the resource mapping is preserved through the conversion process, and there is no need to go through the tester targeting phase.

5.3 Open Compiler Architecture

The STIL language is most prominently represented in the test pattern environment in the pattern compiler. The open architecture concept is embodied in the pattern compiler via LIG plug-ins for specific target testers. Plug-ins rely upon a foundation of compiler software to provide generic, reusable support to parse and process the STIL language.

To the extent STIL is a well-defined source language, with a reasonably extensive set of rules governing how the pattern data is interpreted and processed (e.g., incremental data substitution, rules for waveform character resolution), a compiler can be formulated around the STIL language which has a large foundation, or core, of generic and reusable capabilities. However, appropriate plug-in points must be provided to support pattern generation for specific tester hardware; to this end, each subsystem in the pattern environment contributes to the open architecture concept. Where appropriate, this includes support to extend the compiler, customize compiler behavior, or enforce target tester specific resource constraints.

5.3.1 Generic STIL Parser

Pattern compilation begins by parsing a STIL source file and creating a VPR. During this phase, the STIL source is parsed per the STIL language definition. The overall STIL syntax does not change from one target tester to the next. Plug-ins can reuse the STIL parser as a whole.

There is utility in splitting the parse phase itself into front-end and back-end components. In particular, the parser front-end only parses and validates STIL syntax. Typically, the back-end consists of processing this syntax and generating a VPR for subsequent loadable image generation. However, it is possible to use a different parser back-end under this front-end/back-end configuration. For example, a back-end which reliably splits very large patterns from EDA tools is possible; this facilitates splitting out and make changes to the timing block, for example, while leaving the main body of the pattern untouched.

5.3.2 Generic LIG

The foundation of the compiler features a generic loadable image generation component, or "generic LIG." The generic LIG and tester specific plug-in(s) work together to accomplish the loadable image generation compilation phase. The generic LIG governs overall compilation flow, starting with a PatternExec, and stepping through the pattern from there. During compilation, the generic LIG reads intermediate data from the VPR, processes it according to STIL rules, while feeding processed pattern data to the LIG plug-in (s). This model greatly reduces the amount of processing each LIG plug-in needs to perform.

For example, the generic LIG translates based data into waveform characters, and resolves the waveform character for each signal, given cyclized data possibly consisting of the \r, \w, \h, or \d syntax; a plug-in is then given a simple list of signals and their associated waveform character(s). For example, cyclized data like \r5 a is fully expanded into five waveform characters, and each of these five waveform characters is matched with an individual signal, before the LIG plug-in sees the waveform character. After resolving a waveform character for each signal, the generic LIG also resolves the current waveform associated with each signal.

Where appropriate, the generic LIG also delegates certain decisions to the plug-in, or makes default decisions. For example, the decision to unroll or not unroll a loop is given to a plug-in, but the default is to unroll each loop. If the loop is not unrolled, the generic LIG presents the pattern data once to the plug-in; otherwise, it is repeatedly presented to the plug-in as many times as the loop statement specifies.

A plug-in can also override certain generic LIG behaviors, such as handling of pattern data inside a procedure. The default behavior is to inline procedures, but a plug-in can override this behavior if the target tester supports a native subroutine feature (see section 3).

5.3.3 Extensions via UserKeywords

STIL user keywords provide the sole, consistent mechanism to extend the pattern language in the test pattern environment. The compiler is organized around the UserKeyword syntax to carry this extensibility through to the entire test pattern environment.

UserKeyword's are passed through the STIL parser untouched. Their name, contents, and location within the pattern are captured in the VPR, but no further processing takes place during the parse phase.

However, based upon the position of a user defined keyword in a pattern, the generic LIG makes some basic assumptions regarding the use of these keywords by a plug-in, and presents these user defined keywords to the underlying plug-in(s) in a specific context. For example, the list of globally defined user keywords is presented prior to image generation; the assumption here is a plug-in will use these globally defined user keywords to configure image generation as a whole. As the generic LIG steps through the pattern, user keywords appearing next to pattern statements are presented to the plug-in in order with the other pattern statements; the assumption is these user keywords will be used to generate special opcodes on the target tester.

A plug-in can also ignore unknown keywords, or raise an exception if an unsupported keyword is given.

5.4 LIG Plug-in Requirements

A new LIG plug-in is defined for each new target instrument requiring pattern support. The LIG has a plug-in architecture, such that both NPTest and third party hardware can be supported without requiring a new STIL parser or generic LIG processing. This capability pro-

vides for an open compiler architecture, and greatly reduces the amount of effort and cost to support compilation to new target instrument hardware.

Building upon a set of foundation compiler components, such as the generic STIL parser and LIG, a LIG plug-in is responsible for all aspects of target-instrument loadable pattern image generation. Principally, the plug-in is responsible for sequencer opcode generation. It takes as input from the generic LIG STIL pattern statements, and translates them into appropriate sequencer opcodes, operands, and loadable pattern image data.

A plug-in is required to enforce instrument specific resource constraints, such as the maximum number of waveforms per signal, or the maximum procedure nesting depth. If a resource is exhausted, the plug-in raises an exception, so the compilation process as a whole can exit gracefully.

Where the RPR is an extendable database-oriented file to store meta-data related to instrument-specific features (see sections 5.1, 5.2.3), a plug-in may insert records into the RPR where appropriate. For example, a plug-in may insert signal and timing setup information needed for pattern load and instrument configuration. Separate from the loadable pattern image data, which needs to be in a low-level binary format suitable for direct loading on the target instrument, the RPR is understood throughout the test pattern environment as the common database for pattern related meta-data that is collected during compilation.

Finally, to extend the STIL language to support native tester features, such as instrument synchronization, subroutine, APG, or analog (see section 3), plug-ins support different STIL UserKeyword statements. Plug-ins are expected to handle UserKeyword statements in the context defined by the generic LIG (see section 5.3.3); for example:

- UserKeywords for instrument specific opcode generation are presented to the plug-in in order with the other pattern statements during the flow of compilation;
- UserKeywords used to configure the plug-in for special features, such as setting which trigger to use for instrument synchronization (see section 3), are also processed in order with respect to other pattern statements;
- UserKeywords inserted into the Procedures STIL block identify which procedures are mapped to native subroutine memory, and the plug-in processes the VPR representation of the procedures block prior to image generation to build up a list of procedures mapped to native subroutine; or,
- during instrument development, keywords are a convenient mechanism to enable diagnostic features for image generation.

6 Conclusion

This paper presents a specific example of adopting STIL as a native pattern language for a general purpose SOC ATE. We discussed the benefit of adopting STIL instead of using a legacy pattern language, M9K, for the new tester platform. We emphasize the design-and-test data flow in addressing the bottleneck in the EDA and ATE link issue. Our goal of reducing total test cost is extended to consider our customers' entire test life-cycle. The adoption of the STIL standard enables an open environment and facilitates our open architecture platform.

7 Future Work

In the future paper, we will provide the information regarding to the size of pattern object files, the speed of compilation, and the matrices with the current compiler. We will also continue work on design-and-test to support more upcoming standards, such as the Design Environment extension (P1450.1), Tester Targeting (P1450.3) and CoreTest Language (P1450.6). It is just a beginning, and we will continue to work with the EDA industry to eliminate the barrier between design and test.

8 Acknowledgments

The authors would like to acknowledge the valuable inputs from Dave Grant. Thanks also go to Rudy Garcia for his feedback on the paper.

References

[1] IEEE Standards Association, Standard Test Interface Language (STIL) for Digital Test Vectors, IEEE std. 1450-1999

[2] Erik H. Volkerink, Ajay Khoche, Linda A. Kamas, Jochen Rivoir, Hans G. Kerkhoff, "Tackling Test Trade-offs from Design, Manufacturing to Market using Economic Modeling", ITC 2001, paper 40.1

[3] Gordon D Robinson, "DFT, Test Lifecycles and the Product Lifecycle", ITC 1999, paper 27.4

[4] Bruce R. Parnas, "Doing it in STIL: Intelligent Conversion From STIL to an ATE Format", ITC 2000, paper 3.2

[5] Gregory A. Maston, "Structuring STIL for Incremental Test Development", ITC 1997, paper 40.1

[6] Peter Wohl, Nathan Biggs, "P1450.1: STIL for the Simulation Environment", VLSI Test Symposium 2000, paper 17.1

[7] Gregory A. Maston, "Consideration for STIL Data Application", ITC 2002, paper 10.3

[8] Tony Taylor, "Standard Test Interface Language (STIL), Extending the Standard", ITC 1998, paper 38.1

[9] IEEE Standards Association, Extensions to STIL for Semiconductor Design Environments, Project Authorization Request (PAR) for P1450.1

[10] IEEE Standards Association, Extensions to STIL for Tester Target Specification, Project Authorization Request (PAR) for P1450.3

[11] Marc Loranger, "Is there a STIL for Mixed Signal Testing?", ITC 1999, paper P10.3