2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Complete Formal Verification of a Family of Automotive DSPs

Rafal Baranowski, Marco Trunzer
Robert Bosch GmbH, Reutlingen, Germany
{rafal.baranowski, marco.trunzer}@de.bosch.com

***Abstract*— Formal verification becomes the method of choice for designs with stringent quality requirements. For complex architectures with many implementation alternatives, however, the development and maintenance of complete formal specifications remains a challenge. In this work, we present an efficient semi-formal specification approach for processor designs with a large number of architectural variants. The semi-formal specification serves as a reference to implementation and facilitates automatic generation of formal properties. We show an application of this method to complete formal verification of a family of automotive digital signal processors (DSP), report on the verification effort, and discuss the lessons learned.**

Keywords—complete formal verification, processor verification, semi-formal specification

I. INTRODUCTION

Custom-designed DSP architectures deliver the required safety and performance for automotive appliances at minimum area and power consumption. Their instruction set architecture (ISA) and implementation details strongly depend on the target use. Due to the large spectrum of possible applications, the design and verification of reusable automotive DSPs poses a challenge.

The architectural diversity of automotive DSPs is managed by adopting a high-level synthesis approach. Instead of manually developing and maintaining each processor design, a code generator covering a family of DSP architectures is developed. The generator produces processor implementations from architectural parameters such as word width, set of supported instructions, amount of registers, or type of safety mechanisms. This increases reusability, reduces maintenance effort, and facilitates agile development with frequent requirement changes.

To fully benefit from the push-button high-level synthesis and to make the intellectual property (IP) truly reusable, the entire DSP family must be covered by verification, including architectures that may be generated in the future. Due to stringent safety requirements of automotive applications, the verification must be complete—the functional sign-off must assure that the generated implementations adhere to the specification and that the specification itself is complete and unambiguous.

A. *Related Work*

High-level synthesis of application-specific instruction set processors (ASIP) is well covered in the literature and widely applied in the industry [10]. To make sure that generated processor implementations match high-level architectural descriptions, ASIP development methodologies incorporate methods for automatic generation of test benches and formal properties [10,3]. To our best knowledge, however, no existing ASIP framework can guarantee verification completeness. Thus, a problem in the synthesis algorithm or a programming bug may still result in an implementation error that escapes the verification process.

The progress in model checking algorithms has enabled complete formal verification of complex digital systems. To deal with high design complexity, assume-guarantee reasoning [12] and compositional verification methods such as [9] can be applied. Specification completeness can be verified by showing that any two implementations that adhere to the formal specification are functionally equivalent [1,4].

In the complete verification methodology of [1], formal specifications are developed manually in an iterative process which requires a relatively mature design [2]. Extensions of this approach include automatic generation of formal specifications from intermediate models of processor ISA [7,8] or bus protocols [14]. This reduces the overall verification effort but still requires the verification engineer to develop the specification model and provide a mapping between the model and the actual implementation.

B. Contributions

In this work, we present a novel approach to processor specification that significantly reduces the effort of complete formal verification. Instead of manually deriving properties or models from informal specifications, as in [1,2,7,8,14], we develop a *semi-formal specification* as a reference for both implementation and verification. The semi-formal specification consists of a formal description of instruction commitments and an informal specification of auxiliary functionality. This hybrid specification is (i) sufficiently precise to allow completeness analysis of the main functionality even before the implementation process starts, (ii) abstract enough to withstand major architectural changes, (iii) easy to develop for engineers with little or no background in formal methods, and (iv) concise and easy to understand and review. This approach proved very successful in the verification of several DSP architectures. The resulting verification framework features full functional coverage and is flexible enough to verify any future architecture of the DSP family.

After a short overview in Section II, we describe the semi-formal specification approach in Section III. Section IV presents the verification flow. In Section V we report on the application of this approach to the DSP family. Section VI concludes the paper.

II. Basic Concepts

In the following, we give a brief overview over the state-of-the-art complete verification methodology, present the basics of our semi-formal specification approach, and discuss the verification flow.

A. Operational Properties and Completeness

Our verification approach is based on the verification methodology of [1,2], which is marketed under the name GapFreeVerification™ by OneSpin Solutions[1]. This approach uses so called *operational properties* to construct complete formal specifications and includes methods to verify specification completeness.

The concept of operational properties derives from interval properties [11]. An operational property is in principle an implication defined over a time frame. Both the antecedent and the consequent of an operational property specify a sequence of conditions over a finite but not necessarily fixed number of clock cycles. The property holds if the sequence of events specified in the antecedent is always followed by the sequence specified in the consequent. For instance, the following behavior can be conveniently captured with an operational property: "*If the content of the instruction register in cycle 0 corresponds to a branch, and the branch condition evaluates to true in cycle 1, then the program counter must be set to the target address in cycle 3*".

To enable completeness analysis, an operational property must have a specified *start cycle* and an *end cycle* (called *hooks* in [1]) which may be different from the time span covered by the property's antecedent and consequent. For a set of operational properties to be complete, every execution trace must seamlessly match to a sequence of properties, such that (i) the end cycle of every property in the sequence overlaps with the start cycle of its successor property, (ii) around every start cycle, the antecedent of *exactly one* property that may start in this cycle is satisfied, and (iii) in every clock cycle, the state of all relevant primary outputs and architectural registers is unambiguously *determined* (specified) by the consequents of consecutive properties.

Apart from operational properties, the completeness analysis requires a so called *completeness plan* which lists, among others, primary design inputs, environment constraints, determination requirements for architectural registers and primary outputs, and a property graph specifying the expected property sequencing. For a more detailed overview please refer to [2].

B. Specification Approach

Our specification process begins with the development of implications that describe the execution of processor instructions. One implication covers one cycle of instruction execution and captures the functional and temporal relations between primary inputs, architectural registers, and primary outputs. The implications for the full instruction set are written in a tabular form that is henceforth called *tabular specification*. The functionality covered by the tabular specification is called *core functionality*.

In order to improve specification readability and make the verification process manageable, fragments of processor functionality that are shared by many instructions are black-boxed in the tabular specification. A black-boxed part of functionality with a clearly defined interface is called *auxiliary cluster*. Each auxiliary cluster is specified in an informal

[1] http://www.onespin-solutions.com/

way and verified separately. For instance, stacks and queues are good candidates for auxiliary clusters since their interface is clear and their functionality is same for all instructions that use them.

The structure of the resulting semi-formal specification is depicted Figure 1. The direction of arrows denotes which specification is responsible for determining the target signals, i.e., specifying their expected value for all possible input scenarios. The tabular specification is responsible for *determining* a subset of primary outputs, architectural registers, and all inputs to auxiliary clusters. The informal specification determines all cluster outputs as well as the remaining primary outputs and architectural registers.

To facilitate sound assume-guarantee reasoning, we avoid circular dependencies between specifications of the core functionality and auxiliary clusters. To this end, the outputs of auxiliary clusters are treated as free (unconstrained) variables in the tabular specification. Likewise, the specification of an auxiliary cluster includes no assumptions about the core functionality or any other cluster—all cluster inputs are treated as primary inputs. The resulting semi-formal specification is explained in more detail in Section III.

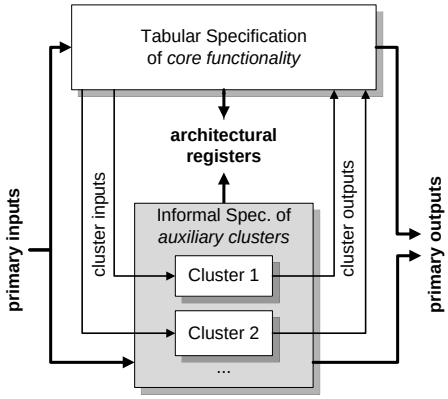

Figure 1. Structure of the semi-formal specification

C. *Verification Flow*

Figure 2 presents the general verification flow. The tabular specification (formal model of core functionality) is automatically translated into a set of operational properties. The ISA-specific constraints and the completeness plan are also generated automatically. In order to support a wide spectrum of design variants, the automatic translators are kept flexible enough to deal with major changes of the tabular specification (e.g. removal and addition of instructions).

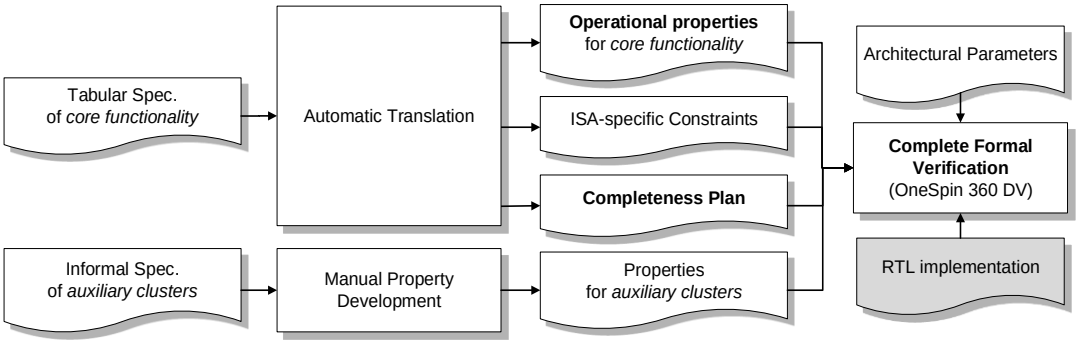

Figure 2. Verification flow

Formal properties of auxiliary clusters are inferred manually from the informal cluster specifications. Where possible, the cluster properties specify the expected value of cluster outputs for all execution traces and hence are complete. Architectural features that are subject to change (word width, number of registers, optional functionality, etc.) are parameterized so that the property set is valid for all architectural variants.

The completeness and soundness of the tabular specification is verified early in the design process, possibly even before any implementation is started, by analyzing the set of generated operational properties. It is formally verified that the properties of core functionality determine all relevant outputs in every possible execution trace, thus guaranteeing verification completeness. The verification flow is described in more detail in Section IV.

III. SEMI-FORMAL SPECIFICATION

In this section, by taking the DSP example, we explain the tabular representation and the decomposition into core functionality and auxiliary clusters.

A. *Tabular Representation*

The automotive DSPs considered in this paper feature a single instruction, multiple data (SIMD) architecture with an arbitrary number of data paths and a configurable number of general-purpose registers per data path. The three-stage pipeline with in-order execution supports about 80 single- and multi-cycle instructions, fixed-point arithmetic of configurable width, interrupts, and a sleep mode. The DSPs also include dedicated (non-memory-mapped) stacks, configurable address generators for RAM and ROM, and several error detection mechanisms. For a given set of architectural parameters, the register transfer level (RTL) processor implementation is generated automatically. A typical implementation consists of 20k lines of VHDL code.

The instruction set of every processor in the DSP family is described with a tabular specification that completely covers the core functionality and enables completeness analysis in the sense of [1]. Figure 3 shows a simple example that conveys the basic principles and syntax of the tabular representation. Each row of a tabular specification describes the expected execution of either one single-cycle instruction, one cycle of a multi-cycle instruction, or one cycle of any behavior that is not triggered by software (startup sequence, interrupt handling, etc.). Instructions can be added or removed simply by adding and removing rows in this specification. The *start cycle* of a row is called n-th cycle; the *end cycle* of a row is cycle n+1. The cycle n+1 of a row *overlaps* with the n-th cycle of another row that describes either the first cycle of a successor instruction or the next cycle of a multi-cycle instruction. For pipelines with no stall mechanism, the cycles refer to the actual pipeline clock. For pipelines with a stall mechanism, the cycles refer to a virtual clock that ticks only when no stalling takes place.

The columns are divided in two sets: the left-hand side columns describe *instruction triggers*, while the right-hand side columns describe *instruction commitments*. Each column describes the state of exactly one signal over cycle n+m, where m∈ℤ is a constant. The content of a row is understood as follows: if all conditions specified in the instruction trigger are fulfilled, all expectations posed in the instruction commitment must hold. In case of multi-cycle instructions specified in more than one row, the instruction trigger is understood as the conjunction of triggers of all the corresponding rows; the same applies to commitments.

Configurable architectural features, such as processor word width or number of registers, are represented by variables. Signal names are prefixed with PI_/PO_ for primary inputs/outputs, AR_ for architectural registers, and CI_/CO_ for cluster inputs/outputs. Time points (cycles) are denoted in parentheses, e.g. AR_PC(n+1) represents the state of the program counter in cycle n+1. Trigger and commitment cells may contain immediate values and expressions in SystemVerilog syntax. An empty cell denotes a *don't care*. Triggers for the instruction register are specified using bit patterns (see column AR_INSTR(n) in Figure 3): zeroes and ones stand for fixed opcode bits, whereas letters denote the fields of operand specifiers. For instance, the expression AR_A[a](n+1)+AR_D[ddd](n+1) denotes the sum of the content of the chosen accumulator and data register in cycle n+1, whereby these registers are selected by the content of the ddda field of AR_INSTR(n) (i.e., content of the instruction register in the n-th cycle).

Figure 3. Example of a tabular specification for two single-cycle and two multi-cycle instructions

	Instruction Triggers			Instruction Commitments				
Mnemonic	AR_INSTR (n)	AR_INSTR (n-1)	AR_INSTR (n-2)	AR_PC (n+1)	AR_A[a] (n+2)	CI_STK_OP (n)	CI_STK_DIN (n)	PO_RAM_ADDR (n+1)
NOP	00110000 00100000			AR_PC(n)+1	Stable	nop		
ADD D<0÷7> A<0÷1>	001000ii ii00ddda			AR_PC(n)+1	AR_A[a](n+1)+ AR_D[ddd](n+1)	nop		
LDR <addr> A<0÷1>	11001000 00000a11			AR_PC(n)+1	Stable	nop		
(cycle 2)	bbbbbbbb bbbbbbbb	11001000 00000a11		AR_PC(n)+1	PI_RAM_RDATA(n+1)	nop		bbbbbbbb bbbbbbbb
JSR <addr>	101000ii iiiiiccc			AR_PC(n)+1	Stable	nop		
(cycle 2)	bbbbbbbb bbbbbbbb	101000ii iiiiiccc		cccbbbbb bbbbbbbbbb	Stable	**push**	AR_PC(n)+1	
(cycle 3)	00000000 mmmmmmmm	bbbbbbbb bbbbbbbb	101000ii iiiiiccc	AR_PC(n)+1	Stable	nop		
RTS	10101000 00000000			CO_STK_DOUT(n)	Stable	**pop**		
(cycle 2)		10101000 00000000		AR_PC(n)+1	Stable	nop		

For conditional instructions or instructions with multiple execution modes, several rows may be used to describe the commitments in each mode. This, however, may bloat the specification and lead to data duplication. Instead, we introduce so called *overlay rows* for each execution mode (not shown in the example). Just one overlay row is used, for instance, to describe how the behavior of all arithmetic-logic instructions changes when their condition is not met.

B. *Auxiliary Clusters*

A fragment of functionality is classified as auxiliary when it either appears in the description of many instructions or is too complex to capture in the tabular specification. (Note that auxiliary clusters do not necessarily correspond to actual modules of the implementation.) The functionality of auxiliary clusters is specified informally and verified separately. Clustering reduces redundancy in the tabular specification and eases the sign-off review. Moreover, it improves verification efficiency since auxiliary clusters need not be verified multiple times (for each instruction that uses their functionality) but just once, possibly at module-level.

Auxiliary clusters are black-boxed in the tabular specification: the tabular specification does not describe any relation between cluster inputs and outputs but merely determines the value of cluster inputs. To avoid circular dependencies, the tabular specification poses no assumptions on cluster outputs and uses them as if they were primary inputs (e.g. to determine the values of other signals).

In the example of Figure 3, the return stack is black-boxed. The tabular specification determines only the stack inputs (CI_STK_OP and CI_STK_DIN) but does not specify how the stack works. The stack head (CO_STK_DOUT) is used to determine the state of the program counter (AR_PC) in the second cycle of the RTS instruction as if it were a primary input.

In the semi-formal specification of the DSP family, the following functionality is handled with auxiliary clusters: multiply-accumulate operations, configurable address generation, I/O control, stacks, and interrupt queues. This functionality is active during the execution of many instructions, which justifies separate verification. The clusters have a relatively simple input/output behavior and require no assumptions that would need to hold of core functionality or other clusters.

IV. VERIFICATION FLOW

In this section, we describe the generation of properties from tabular specifications, underline the benefits of early completeness checks, and explain the verification process.

A. *Translation of the Tabular Specification*

The tabular representation is automatically translated into a set of operational properties. Since the tabular representation is derived from the concept of operational properties, the translation is fairly straightforward. Each generated property corresponds to one instruction specified in one or more rows of the tabular specification. For conditional instructions and instructions with several execution modes, the properties are augmented with the content of overlay rows. The rows describing the startup sequence, interrupt handling, and other externally triggered operations are treated in the same way as rows of regular instructions. Below we give an example of the property generated for the instruction ADD from Figure 3 (language: SystemVerilog with Timing Diagram Assertion Library™ [13]):

```
property prop_ADD_D_A;
  reg [p_instr_width-1:0] opcode;                    // variable for the opcode
  reg condition_ok;                                  // variable for conditional execution
    t ##0 AR_INSTR ==? 16'b001000XXXX00XXXX and                    // TRIGGERS
    ...
    t ##0 set_freeze(opcode, AR_INSTR) and                         // store opcode
    t ##1 set_freeze(condition_ok, check_condition(getField_i(opcode))) // store condition
implies                                                            // COMMITMENTS
    t ##1 AR_PC == ($past(CO_INTERRUPT_TRIG) ? $past(CO_INTERRUPT_ADDR)
                                             : $past(AR_PC) + 1'b1) and
    t ##0 CI_STK_OP == stack_nop and
    t ##2 AR_A[getField_a(opcode)] == (condition_ok)?
            $past(AR_A[getField_a(opcode)]) + $past(AR_D[getField_d(opcode)])
          : $past(AR_A[getField_a(opcode)]) and
    t ##2 (getField_a(opcode) == 'd0) || $stable(AR_A[0]) and // AR_A[0] modified or stable
    t ##2 (getField_a(opcode) == 'd1) || $stable(AR_A[1]) and // AR_A[1] modified or stable
    ...
    t ##1 right_hook;
endproperty
```

ISA-specific constraints and the completeness plan are also automatically derived from the tabular specification. The constraints are responsible for restricting execution traces to valid programs, among others, by limiting the output of program memory to sequences of valid opcodes. The completeness plan lists all signals that are supposed to be determined by the tabular specification (commitments), all signals treated as primary inputs (including cluster outputs as these signals are verified separately), the ISA-specific constraints, and the set of expected successor instructions for each instruction (property graph).

The properties of auxiliary clusters are developed manually. The informal cluster specification, together with the tabular specification defining the interface and handshaking, serves as a reference. Care is taken to make the cluster properties complete. In the simplest case, this is achieved by developing properties that unconditionally specify the cluster outputs as a function of cluster inputs. For instance, stacks are described with properties that contain a reference stack model which exhaustively covers the expected input-output behavior.

B. Early Completeness Checks

The completeness and soundness of tabular specifications can be verified even before the implementation begins. After the translation to properties, the methods described in Section II.A are used to find the following issues: (i) execution scenarios that do not match to any specified instruction and hence are uncovered by the specification (gaps in the triggers), (ii) execution traces in which some primary outputs or architectural registers are not specified or ambiguous (gaps in commitments), (iii) contradictions in the triggers and commitments which may reveal architectural issues, (iv) execution traces that violate the property graph specified in the completeness plan (instruction sequencing problems). The early completeness checks can considerably improve the specification quality and reduce the number of design iterations.

C. Verification Process

The resulting DSP verification framework consists of the translators for tabular specifications and the set of properties for auxiliary clusters. To verify any DSP architecture, the following input needs to be provided apart from the implementation: (i) the tabular specification of the target architecture (largely reused across DSP implementations), (ii) the mapping of signals in the verification framework to signals of the implementation (straightforward since the tabular specification serves as a reference to implementation), (iii) target architectural parameters (fixing variables in the verification framework). The work necessary to verify any new architecture of the DSP family is therefore reduced to a minimum.

To lower the effort of formal reasoning, auxiliary clusters can be black-boxed in the implementation model while verifying core functionality. To this end, signals corresponding to cluster outputs are replaced with primary inputs. This method, however, should be used with care if the outputs of a black-boxed cluster are used in any trigger of the tabular specification. In this case, black-boxing may mask a bug that makes the trigger unsatisfiable—an instruction that can never execute may go unnoticed.

Similarly, the verification of auxiliary clusters can be performed at module-level (i.e., just for the module that implements the auxiliary cluster) and/or with loosened environmental constraints. This does not compromise completeness but may result in spurious counterexamples. In such cases, interface assumptions are required, and potential circular dependencies need to be taken care of (assume-guarantee reasoning). In the verification of the DSP family, we resort to module-level verification just for the stacks which, however, require no additional assumptions.

D. Sign-off Review

The methodology of [1] specifies which elements of the verification framework should be reviewed for sign-off. These guidelines can be directly applied to the automatically generated operational properties, constraints, and the completeness plan of core functionality. With the tabular specification as a reference, the review process is fast and often straightforward. Since the generated results are examined, the automatic translators need not be reviewed.

The sign-off review of auxiliary clusters must make sure that the properties of auxiliary clusters adhere to the informal specifications and cover all cluster outputs. To assure completeness, it must be additionally verified that cluster outputs are determined unambiguously and unconditionally, i.e. for every possible execution trace. This holds trivially if the properties describe the cluster outputs as a combinational or sequential function of just the cluster inputs and/or primary inputs.

E. Implementation

For portability and accessibility, we store tabular specifications in a standard spreadsheet format. The automatic translators of the tabular representation are implemented using Java Emitter Templates (JET) [5]. The operational properties

of core functionality are expressed in SystemVerilog assertions using Timing Diagram Assertion Library (TIDAL™) [13]. The properties of auxiliary clusters are written in plain SystemVerilog assertions. Design verification and completeness checks are performed with OneSpin 360 Design Verifier.

To reduce the risk of flaws in the tool chain and to conform to ISO 26262-8 guidelines for confidence in the use of software tools [6], we take care that the software which generates the verification framework is sufficiently divergent from the software that produces DSP implementations. Ideally, the tabular specification and the specification of architectural parameters should be the only data common to the design and verification processes.

V. RESULTS

Three verification engineers were involved in the verification of the DSP family, two of whom had no previous experience with complete formal verification. At the beginning, just one typical DSP architecture was considered. Specification clustering made verification planning a straightforward task—two engineers were assigned to auxiliary clusters until this functionality was completely verified. Later, verification subtasks for core functionality were identified by decomposing the tabular specification row-wise (instruction-wise) and column-wise (signal-wise). After the typical DSP architecture was fully verified, the verification framework was applied to four further architectures, and remaining issues were resolved.

For a typical member of the DSP family, the tabular specification contains about 260 rows and 60 columns. The software for automatic translation of tabular specifications consists of only 3400 lines of code (LOC). The manually written properties of auxiliary clusters require 2500 LOC. Typically, the set of generated properties and ISA-specific constraints comprises 20k LOC, and the generated completeness plan consists of 600 LOC. The architecture-specific parameters and the signal mapping require about 600 LOC.

A. *Lessons Learned*

To make the automatic translation of tabular specifications reusable across architectures and projects, we strived to describe all design details in the tabular specification and keep the translators simple. This required several iterations with the designers and resulted in a custom syntax of the tabular specification.

Using one operational property per instruction turns out to be more favorable than describing each instruction cycle with a separate property. This makes the specification of trigger conditions simpler and significantly reduces the effort of property verification and completeness checking.

Specifying auxiliary clusters with manually coded properties works well as long as the properties fully describe cluster outputs and can be efficiently handled by the verification tool. In case of complex clusters, however, the functionality may need to be split into several properties in order to reduce the effort of formal reasoning and ease debugging. In this case, instead of using informal cluster specifications, recursive application of the tabular specification may increase productivity. Although this approach requires separate tabular specifications for each complex cluster, it facilitates automated completeness analysis and eases the sign-off review.

B. *Identified Issues*

The main strength of complete formal verifications lies in discovering specification ambiguities that may lead to misinterpretation in the design process and leave design bugs undetected. In total, we identified about 50 issues with the tabular specification and 10 problems with the informal specification of auxiliary clusters. The majority of identified issues were due to specification gaps such as incomplete instruction triggers or missing commitments.

Prior to formal verification, the DSP family was verified by the design team using traditional simulative techniques. Although the implementations were fairly mature, formal verification found nine bugs, eight of which affected all five architectures. While software workarounds could be applied to five of these bugs, the performance loss would be inacceptable for some target applications. The majority of the identified issues were corner-case bugs that posed a potential safety risk.

As a by-product of formal verification, the ISA-specific constraints are used as assertions in simulations to detect bugs in application software and in the DSP toolchain. For example, they verify limits for operand specifiers to make sure that only implemented registers are accessed. These assertions revealed a bug already in the very first application software.

C. *Verification Effort*

For a typical DSP architecture, a complete verification run takes about five hours with six formal engines running in parallel on a typical workstation with twelve cores and 100 GB memory. One hour is spent on the verification of core functionality, another hour on completeness checking, and three hours are consumed by the verification of auxiliary clusters.

The development of the verification framework took seven person months in total, including the verification and debugging effort for five DSP architectures. Most of this time was spent on the development of automatic translators for the tabular representation and on the verification of core functionality. The verification of auxiliary clusters took about one and a half person months.

Based on our previous experience with complete formal verification, we predict that with a traditional specification and manual property development, the verification of a single DSP architecture would take a comparable amount of time as we required for the full DSP family. Thanks to the semi-formal specification, just a few hours' work is sufficient to apply the verification framework to any future DSP architecture.

VI. CONCLUSION

While formal verification promises full functional coverage, the development and maintenance of complete formal specifications for configurable designs remains a challenge. The presented semi-formal specification approach is sufficiently clear and intuitive to be used early in the design process and precise enough to support completeness analysis. The tabular representation provides a good trade-off between precision and maintenance effort. Through clustering, the verification problem is effectively decomposed into manageable verification goals that ease verification planning. The application of this approach to a family of processor designs proves its effectiveness in finding specification ambiguities and design bugs, and it confirms great productivity gains through IP reuse.

ACKNOWLEDGMENTS

We thank all our colleagues for their contribution to the success of this project, especially Slava Bulach, Tilman Glökler, Jo Pletinckx, Alexander Emperle, Jan Scheuing, Jens Goldeck, and the OneSpin Support Team.

REFERENCES

[1] Bormann, J. and Busch, H. 2005. Method for determining the quality of a set of properties. European Patent Application, publication number EP1764715.

[2] Bormann, J., Beyer, S., Maggiore, A., Siegel, M., Skalberg, S., Blackmore, T., and Bruno, F. 2007. Complete formal verification of TriCore2 and other processors. In Design and Verification Conference (DVCon'07).

[3] Chattopadhyay, A., Sinha, A., Zhang, D., Leupers, R., Ascheid, G., and Meyr, H. 2009. Integrated verification approach during ADL-driven processor design. Microelectronics Journal, 40, no. 7, pp. 1111-1123.

[4] Claessen, K. 2007. A coverage analysis for safety property lists. In IEEE Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD'07), pp. 139-145.

[5] Eclipse Consortium. 2003. Java Emitter Templates (JET). https://eclipse.org/modeling/m2t/?project=jet (accessed March 30, 2016).

[6] ISO 26262-8:2011. Road vehicles — Functional safety — Part 8: Supporting processes. International Organization for Standardization, 2011.

[7] Kühne, U., Beyer, S., Bormann, J., and Barstow, J. 2010. Automated formal verification of processors based on architectural models. In IEEE Proc. Formal Methods in Computer-Aided Design (FMCAD'10), pp. 129-136.

[8] Loitz, S., Wedler, M., Stoffel, D., Brehm, C., Kunz, W., and Wehn, N. 2010. Complete verification of weakly programmable IPs against their operational ISA model. In IEEE Proc. Forum on Specification & Design Languages (FDL'10).

[9] McMillan, K.L. 2000. A methodology for hardware verification using compositional model checking. In Science of Computer Programming, Volume 37, Issues 1–3, pp. 279-309.

[10] Mishra, P. and Dutt, N. (Eds.). 2011. Processor description languages. Morgan Kaufmann, San Francisco, CA, USA.

[11] Nguyen, M., Thalmaier, M., Wedler, M., Bormann, J., Stoffel, D., and Kunz, W. 2008. Unbounded protocol compliance verification using interval property checking with invariants. IEEE Transactions on CAD, 27, no. 11, pp. 2068-2082.

[12] Pnueli, A. 1985. In transition from global to modular temporal reasoning about programs. In Logics and models of concurrent systems, Springer-Verlag, pp. 123–144.

[13] Siegel, M. and Bormann, J. 2008. Timing diagrams ease formal property development. In ChipDesignMag, http://chipdesignmag.com/display.php?articleId=2588 (accessed March 30, 2016).

[14] Soeken, M., Kühne, U., Freibothe, M., Fe, G., and Drechsler, R. 2011. Automatic property generation for the formal verification of bus bridges. In IEEE Proc. International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS'11), pp. 417-422.