Automatized High-Level Evaluation of Security Properties for RTL Hardware Designs

Andrea Höller, Christopher Preschern, Christian Steger, Christian Kreiner
Institute for Techn. Informatics
Graz University of Technology
Graz, Austria
{andrea.hoeller, christopher.preschern, steger, christian.kreiner}@tugraz.at

Armin Krieg, Holger Bock, Josef Haid
Infineon Technologies Austria
Design Center Graz
Graz, Austria
{krieg.external, holger.bock, josef.haid}@infineon.com

ABSTRACT

The ever increasing integration of embedded systems into our every lives created a strong demand for trustable software and hardware implementations. To provide such trust between manufacturer and customer of integrated systems, regulatory rules like the Common Criteria have been defined. While this international standard clearly prescribes the usage of formal methods at high assurance level, formal verification at code-level is not widespread in practice.
This work introduces a novel approach to verify the correct functionality of security critical hardware implementations under fault conditions. Generality is enabled by high-level evaluation using state machines extracted in an automatized way.

Keywords

Model checking, security properties, automatized evaluation

1. INTRODUCTION

Increased hardware integration and software complexity led to a well-known problem of strongly increasing verification effort. In the security domain a variety of rules have been introduced to provide correct system behaviour under fault conditions. Depending on the strictness level of these rules, they only concern the documentation of the targeted system, or on higher levels, exhaustive penetration testing is needed.

For complex implementations traditional simulation techniques to enable evaluation at earlier stages of the design flow cannot be applied. The reason for this are permissive performance and time requirements. Therefore, formal verification is increasingly introduced to conquer various verification challenges resulting from system complexity.

high, consistent set of standards

The Common Criteria (CC) is widely used to provide consistent evaluation standards for the development of high quality security products. Formal methods to verify the compliance with security policies are prescribed by the CC. However, while formal methods are mainly used during concept phase, the formal verification of the implementation regarding security policies is mostly neglected. Therefore, this work presents a novel methodology to formally analyze the implementation using a model checker. A tool-set to extract finite state machines (FSMs) from existing hardware implementations to enable model checking techniques is presented. Based on an established fault attack evaluation flow presented in [20], a complete link with physical attack and implementation evaluation can be created. The main contributions of this work are:

- A novel methodology for the efficient evaluation of RTL-implementations using a widely used model checker (see Figure 1)
- Model-based fault injection for robustness and fault attack evaluations of implemented state machines
- Case studies using an UART with authentication and a counter implementation for fault injection testing

This paper is structured as follows: Section 2 gives a brief overview over the state-of-the-art methods concerning formal verification of hardware implementations. Followed by Section 3 introducing a methodology for the automatized

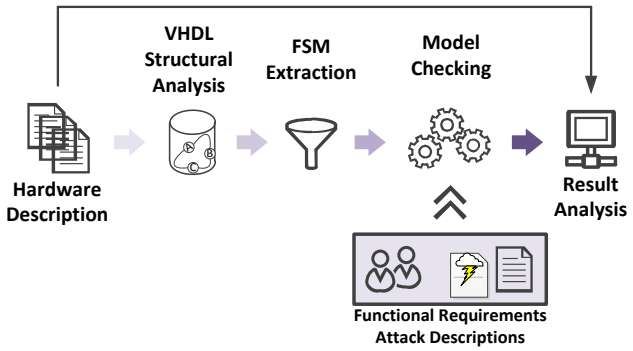

Figure 1: Global model-checking based security verification flow - Application of formal methods for certification and testing in an automated manner

evaluation of RTL implementations on model-checking level. Furthermore, the integration of model-based fault injection is described in Section 4. The applicability of the proposed approach is experimentally investigated in Section 5. Finally, our results are concluded and some details about our future work are given in Section 6.

2. RELATED WORK

Here we outline the related work by first giving an overview of formal hardware verification methods and then describing the state-of-the art concerning high-level verification.

2.1 Classic Application of Formal Methods in Hardware Verification

Checking the correctness of a device using verification has become a major task in the development of integrated circuits. There are two fundamental approaches to verification: dynamic and static verification.

For this work we will focus on static verification techniques to profit from the advantages concerning complex system designs. Static formal techniques provide a formal proof on an abstract mathematical model of the system. A commonly used way to model the behaviour of a system is to use Finite State Machines (FSMs).

Model checking is a formal verification technique that exhaustively and automatically checks whether a FSM meets a given specification. If the criterion is not fulfilled, a counterexample to support the debugging is generated. Model checkers operate on two inputs: a model of the system $\mathcal{M}$ and system properties ϕ. The model checker automatically checks whether $\mathcal{M}$, which is denoted by FSM transitions, meets the given specification ϕ. This specification is described by formulas in temporal logic [10]. The effectiveness of a model checker is limited by the fact that they can only verify models in their own language and thus $\mathcal{M}$ has to be abstracted manually.

Much research has been dedicated to the task of extracting control flow elements from RTL descriptions automatically. Early work incorporating not only FSM extraction but also symbolic model checking have been shown in [12, 15]. The proprietary implementation was mainly targeting basic implementation problems, like reachability analysis, but included no possibility to check for high-level rules. A similar approach has been taken in [7] also describing the transformation of simple synchronous circuits into state machines. A fully automated approach for the abstraction of Verilog code has been shown in [2].

Moundanos et al. used in [24] formal verification techniques to provide an unified framework for design validation and test generation. They abstracted hardware implementations to use the information about the control flow to assist Automatic Test Pattern Generation (ATPG).

The presented approaches mainly focus on bit-level evaluations and therefore, are not well scalable for very large systems. This fact resulted in techniques like *Predicate Abstraction* as used in [17, 3], or *Term-Level Abstraction* as practiced in [21]. In [19], High-Level Decision Diagrams (HLDDs) have been not only used to describe a system's behavior but also to correct wrong transitions automatically.

The New Symbolic Model Verifier (NuSMV) project, provides a well-tested and open source implementation of a symbolic model checker. For this work we will build on this infrastructure for the final verification of specifications given in linear temporal logic (LTL).

2.2 High-Level Verification for the Security Domain

Security engineering is highly regulated because of the implication of loss of trust. A common requirement of security-related products is that they have to be fault tolerant in order to be resistant against fault attacks. The development of systems that maintain correct operation even in the event of failures has been especially treated in the safety domain.

A very comprehensive approach using the NuSMV model checker and providing fault injection capability has been taken in [11]. Their approach nonetheless is limited to software implementations and classic verification properties like liveness or reachability. The authors of [16] showed an efficient application of model checking to verify fault tolerance by using mutated models of sub-systems.

In [23, 13] concepts for the application of formal methods to a secure software implementation flow are presented. Some of the presented techniques can also be applied to hardware certification, but abstraction is a challenge concerning RTL descriptions.

In [6] an industry-oriented tool to enhance the usability of formal hardware verification is presented. The tool uses the model checker SMV and supports hardware description languages such as VHDL and Verilog.

Love et al. proposed a framework for facilitating theorem proving of RTL hardware implementations written in Verilog. The authors showed how to translate Verilog code into the Coq theorem-proving language. This offers straightforward validations of security-related properties by the consumer of an IP module to build trust to unknown vendors. Their approach mainly targets the differentiation of Trojan-infested from trustworthy IP modules. However, we propose to use the power of formal model extraction from RTL designs also for the verification of CC security requirements for IP modules using an in-house design chain. While Love et al. applied theorem proving, we show that also model checking can be used for ensuring the compliance with formal security properties. The advantage of the model checking approach is that constructive refutations are provided, if the proof fails.

The authors of [9] showed that model checking is suitable for describing and verifying requirements for CC Evaluation Assurance Level 6 evaluation at a high level of abstraction. The same authors presented in [8] a development flow using model checking for the verification of security policies during the concept phase. As shown in Figure 2, we propose to extend this flow. Our approach not only provides the assurance of the high-quality of the functional specification, but also checks the implementation regarding formal security requirements.

In [5] the gap in the software domain between source code verification and CC certification is discussed. However, there is also a long existing gap between analysis of hardware RTL descriptions and checking of high-level rules and specifications.

In short, our approach contributes a step towards filling this gap by automatically extracting formal models out of RTL descriptions and using model checking to show the consistency to high-level security properties.

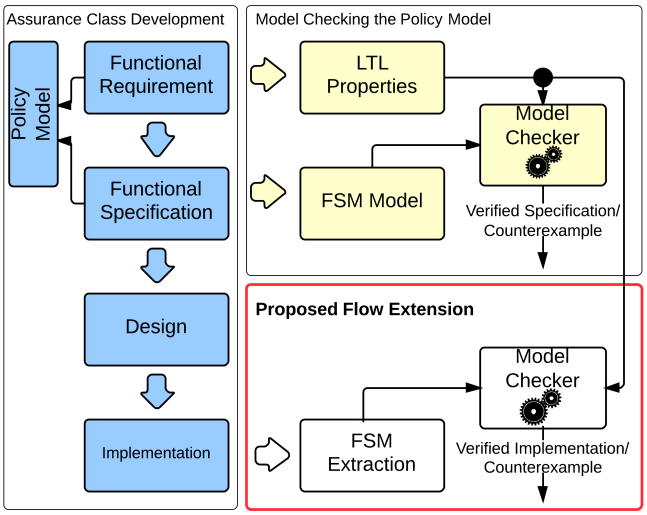

Figure 2: Flow of CC assurance class development and proposed usage of model checking for checking high-level security properties of the functional specification as proposed in [8]. Our approach extends the flow by also taking the implementation into account.

3. SYSTEM ANALYSIS AND FSM EXTRACTION METHODOLOGY

In this paper we address the verification of high-level features from a given hardware implementation at RT-level using the model checking approach.

Therefore the HDL implementation has to be represented as a model $\mathcal{M}$ readable by a model checker. In order to overcome the disadvantages of manual abstraction like the introduction of human errors and cost of time, we propose to generate this model automatically.

How such a generation could be achieved is described in this section. We implemented a framework that automatically generates NuSMV models of FSMs described in a given VHDL implementation. Therefore we developed a Java program that parses and analyzes a given VHDL implementation. The program identifies behavioural constructs describing an FSM and translates the found FSMs in a model that can be analyzed by the NuSMV model checker. The automatic identification of FSMs can support design and test engineers in getting a deeper understanding of the behavioural implementation. Furthermore, it highlights which parts of the implemented logic are suitable for a detailed analysis with a model checker.

Alternatively, other tools (e.g. adapted VHDL compilers) could be used for the translation of VHDL into a format suitable for model checkers. However, static analyzing the code with a parser offers a transparent, open and direct way of achieving a formal model.

3.1 Supported VHDL Subset

We address control-dominated synthesizable VHDL designs. Only sequential behaviour and no combinational logic is considered. Since the NuSMV language is supposed to describe FSMs, we limit our analysis to VHDL code realizing FSMs.

There are two ways for describing an FSM in VHDL: one-segment and multi-segment code style [14]. The first uses only one process to describe the logic and the state register. This style introduces additional latches during synthesis and thus, multi-segment code style using separate processes for state register and logic is often preferred. Our approach supports both coding styles.

3.2 Translation of VHDL for NuSMV

The translation consists of five steps: VHDL structure parsing (*1*), translation of VHDL processes into Control Flow Graphs (CFGs) (*2*), FSM identification inside the CFGs (*3*), FSM extraction (*4*), and conversion of these FSMs into the NuSMV input language (*5*). Finally, the NuSMV model representation can be used to check high-level assumptions about the expected behaviour of the system.

3.2.1 VHDL Structure Parsing

The first step is to parse the structure of the VHDL code by analysing entity and architecture descriptions holding behavioural processes. To accomplish that, we rely on a straight-forward implementation of an VHDL parser. This parser is able to extract system-internal structures and dependencies. Therefore, any other similar tool-chain providing this information, can be applied for this approach.

3.2.2 Translation of VHDL Processes into CFG Description

To find and extract FSMs, the logic described in VHDL is converted in an intermediate format. Therefore, we use the CFG representation as proposed in [22]. A CFG consists of nodes representing conditional branches like *if* and *case* statements. Each node holds a list of transitions, which are executed, if the condition of the node holds.

Every node has only one parent node but can have multiple child nodes. In our context, there are two types of child nodes: "true child nodes" that represent the further execution flow, if the condition of the actual node is true, and "false child nodes" for the case that the condition is false.

Transitions represent VHDL assignments and store information about the target signal, the source value and the type of assignment (variable or signal).

3.2.3 FSM Identification

Formally, an FSM is defined as a 5-tuple (Q, Σ, T, q_0, F), where Q is a finite set of states, Σ is a finite input alphabet, T is the transition function $T : Q \times \Sigma \rightarrow Q$, q_0 is the initial state, and F is the set of final states [18].

The realization of an FSM consists of a synchronized state register, a next-state logic implementing the transition function, and an output logic defining the outputs [14]. Because every FSM needs a state register, finding a state register equals finding an FSM. Thus, first all registers are identified and then it is decided which of them are state registers. Since synchronous registers react on an active clock edge, every CFG node with an expression indicating an active signal edge is marked as a register.

To identify state registers, the observation that state signals depend on themselves is used. To examine the relationships of signals, a dependency list is created for each signal. The list holds all signals to which the signal depends on and is determined by collecting all transitions targeting the signal. If a circular dependency is found, the register is regarded as a state register.

Table 1: VHDL to NuSMV data types translation examples

VHDL type	NuSMV type
`std_logic`	`boolean`
`std_logic_vector(<range>)`	`word[<size>]`
`type <identifier> is range <range>`	`<min>..<max>`
`type <identifier> is array`	`array <min>..<max>`
`<index_constrained>`	`of <element subtype>`
`type typename is (enum1, enum2, ...)`	`{enum1, enum2,...}`

3.2.4 FSM Extraction

Extracted properties are the next-state logic, the initial state, and the input variables as stated in the FSM definition. Furthermore, the output logic and the initial values of the output signals are extracted.

The next-state logic represents a collection of all node transitions and their conditions targeting the next-state signal. It depends on the code styling which signal holds the next state. In one-segment code styling there is only one signal holding both the state and the next-state information, whereas in two-segment code a separate signal to store the next-state is used.

The conditions of the state transitions are determined by combining conditions of the CFG node holding the next state transition and its predecessor nodes. More specifically, the condition of the regarded node and its parent node are linked with a logical *and*, if the current node belongs to the "true child nodes" of the parent node, or with a logical *and not* if it belongs to the "false child nodes".

For the determination of the initial values it is expected that the state register has a reset functionality. The values accessed to signals when a reset occurs are considered as initial values.

The FSM definition states that the transition functions of the next state depend on the input values. Hence, all signals that occur in the dependency list of the next-state signals are regarded as input signals.

All the signals that depend on the state are output signals. Thus, all signals having the state signal in their dependency list are output signals.

3.2.5 FSM to NuSMV Language Translation

The last step is to translate the structure of the FSM into the NuSMV language. An exemplary translated code segment is shown in Figure 7.

Variable Declarations.

The declaration of a NuSMV variable includes the indication of the data type. Since the language is supposed to describe FSMs, only finite data types are supported. Translating VHDL data types that occur in FSMs is basically a simple syntax rewriting as exemplified in Table 1.

A NuSMV variable can be a state, input, or frozen variable. State variables include variables that can take the values only from the domain of its type. Input variables cannot be instances of modules and frozen variables do not change their initial value. In the translation process every variable with no transition function targeting the variable is declared as a frozen variable.

Initial Values, Next-State and Output Logic.

The NuSMV language is well suited to represent transition relations. The next-state logic is converted to NuSMV switch-case statements, whereas every transition of the next-state signal in an CFG node is translated into a case branch condition. NuSMV supports the description of the next value of a variable using the expression `next(varname)`. The output logic is described in a similar way, with a differentiation between VHDL signal and variable assignments. A signal assignment means that the signal is updated after the completion of a process, whereas a variable assignment has an immediate effect. Consequently, the signal word `next` is only used to describe signal assignments.

For the determination of the init values it is exprected that the state register has a reset functionality. The transitions that are executed if a reset occurs represent the initial behaviour. For every variable with a noticed initial value a NuSMV statement in the form `init(varname):=init_val` is created.

Additionally, constants are converted from the VHDL to the NuSMV syntax. For example a logical expressions like `'1'` is converted to `TRUE`, numerical expressions that indicate the base of the number system like the VHDL expression `x"F"` are converted to the right NuSMV syntax like `0h_f`.

FSM Interconnection.

One VHDL entity can contain the description of multiple FSMs. Since NuSMV allows a hierarchical description of FSMs, it is possible to describe every FSM using its own module and connecting them at a higher level.

An FSM is declared in NuSMV using a statement like `MODULE FSM1(input_of_other_fsms)`. In the main module the described FSMs are instantiated and connected. The variables of an FSM instance can be accessed using an expression like `fsm1: FSM1(fsm2.var1)`. To find the inputs that are variables in other FSMs, it is examined whether a variable used in the FSM has a transition function targeting a variable in another FSM.

4. MODEL-BASED FAULT INJECTION

When developing hardware performing security operations, designer have to keep side channel attacks (SCA) and fault attacks in mind. In such a case the behaviour of an implementation is examined under the influence injection operational or data faults. Using fault induction, an attacker would try to reconstruct secret data by analysing faulty calculations. Fault analysis can also be used to influence state-machines by manipulating state transitions. Faults can be introduced by many means, like temperature effects, light, radiation, electromagnetic induction, power supply transients etc. A large compendium is given in [4].

Therefore, the CC also contains requirements concerning fault tolerance. Part two of the CC [1] describes security functional components, which are the basis for security functional requirements. The expected security behaviour of a Target of Evaluation (TOE) is specified by these requirements. One category of requirements deal with fault tolerance and ensure that the TOE operates correctly even if failures occur. One of these requirements is the following:
FRU_FLT.1.1: The TOE Security Functionality (TSF) shall ensure the operation of [assignment: list of TOE capabilities] when the following failures occur: [assignment: list of type of failures].

We propose to use the generated NuSMV model to perform the verification of this requirement. Exemplary failure types, which can be checked are given below.

Traditionally, model checking is used to proof the correct behaviour of a system. However, to analyse the vulnerability to fault induction attacks, it is also necessary to evaluate security systems under faulty conditions. To achieve this evaluation fault injection is a widely used technique.

Current approaches using physical penetration, gate-level, or RTL-level test platforms always come with the major problem of investigation completeness. Furthermore, it is often necessary to manipulate the original design to inject faults. To overcome these disadvantages, we show how an abstracted model of the implemented system can be used to evaluate the robustness of the derived state machines against external influences. Examples of common modes of failure are:

- *random faults*: the value of a bit changes to randomly
- *stuck-at faults*: a bit sticks to its current value
- *bit flips (inverted)*: the value of a bit is inverted [16]

The random occurrence of faults in a NuSMV model could be achieved by describing the activation of faulty behaviour using input variables. Figure 3 shows how these failure modes could be modelled with NuSMV. Such model extension for fault injection purposes can be generated in an automated way, but are limited to module internal variable.

We extended our Java program to support the automatic injection of faulty behaviour for given signals. The faulty behaviour of the given signals is modelled by adding fault-injection statements like those in Figure 3 to the transition functions of the signals when performing the VHDL to NuSMV translation.

```
MODULE main
VAR
  var : boolean;
  -- random occurrence of failures
  random_failure    : boolean;
  stuckat_failure   : boolean;
  inverted_failure  : boolean;
ASSIGN
  next(var): case
    random_failure    : {TRUE , FALSE};
    stuckat_failure   : var;
    inverted_failure  : !var
  esac;
```

Figure 3: Modelling several failure modes with the NuSMV language

5. EXPERIMENTAL RESULTS

To demonstrate our approach we first present an UART (Universal Asynchronous Receiver Transmitter) communication controller implementation and show how to verify a high-level security property. Then, we give a simple use case demonstrating the automatic injection of faults in the formal model.

5.1 Verification of Security Policies

We used a VHDL implementation of an UART control logic, automatically translated the design in a NuSMV model and checked a security policy derived from a CC requirement. An UART controller realizes serial communication properties to transmit and receive data asynchronously. This communication logic could be used to connect a microcontroller to external interfaces.

The transmission logic includes a parallel-to-serial converter, while the reception logic realizes a serial-to-parallel conversion. The data format consists of a low start bit triggering the transmission following by eight data bits, a parity bit and a trailing high stop bit.

The analyzed UART implementation offers an additional authentication feature. Meaning that data is only transmitted, if a certain password has been received. If the buffer used for the reception contains a predefined value the communication partner is regarded to be trustworthy. Only if this is the case, the UART controller forwards the data, which is transmitted by the microcontroller.

5.1.1 VHDL Design

The complexity of the analyzed RTL design is relatively low consisting of about 270 lines of VHDL code. An extract of the VHDL code is shown in Figure 8. An FSM with a state variable called `rx_fsm` and one input signal `rx` implements the reception logic. The FSM stays in idle state until a falling edge of rx indicating the start of a transmission is detected. Then the FSM goes into a state called *data* to receive data and to perform the serial-to-parallel conversion. After receiving eight bits, the FSM goes into a state called *parity* and checks the correctness of the parity bit. If the parity bit indicates that the transmission was correct, an output called `rx_ready` is set to signalize the microprocessor that the data is ready for reading. In addition to this standard UART reception logic, it is checked whether the received data stored in a buffer matches a predefined password. This password check is only performed if the parity bit indicates that the data was transmitted correctly.

The process of transmission is similar to the reception logic. A variable called `rx_fsm` stores the current state of the FSM for reception. An eight bit input `rx_data` is converted to a one bit output `rx`. Data is only transmitted if the communication partner successfully authenticated himself. Thus, the FSM implementing the transmission stays in idle state if no password was given, even if the microcontroller attempts to send data. This provides for the output to stay always in high state (`UART_IDLE`) and does no data information leaks to the outside.

5.1.2 Translation into the NuSMV Language

The VHDL implementation of Figure 8 was automatically converted into the NuSMV language as shown in Figure 7 using the process described in Section 3.

5.1.3 Verification of Security Requirements

An example of a CC property, which can be found in the specification of secrets (SOS) within the requirement family identification and authentication (FIA) is the following:

FIA_SOS.2.2: The TSF shall be able to enforce the use of TSF generated secrets for [assignment: list of TSF functions] [1].

With regard to the previous described UART implemen-

tation this requirement could be that the microcontroller shall not be able to transmit data until a correct password is given form the external communication partner. A LTL representation of the specification could state that it is impossible to get in a state, where `rx` is not `UART_IDLE` and no password has been provided (see Equation 1).

$$G\neg(\neg(rx = UART\_IDLE) \wedge \neg pwd\_given) \quad (1)$$

5.1.4 Result

Since the complexity of the analysed design was low, the NuSMV model checker achieved a verification within a few seconds. The result of the evaluation showed that the previous described implementation rules are fulfilled by the given VHDL design. This enhances the confidence that the analyzed module has been implemented correctly.

5.2 Model-based Fault Injection

In this section we show how an abstracted model of an implementation can be used to evaluate the robustness of the derived state machines against external influences.

5.2.1 VHDL Design

For demonstration, a simple counter implementation has been selected (Figure 6). It can be easily seen that this counter will count to eight and reset its value to zero afterwards. This implementation also incorporates a visible bug as such that if the variable cnt gets higher than eight for some reason, the counter will never reset.

5.2.2 Fault Injection

Single Event Transients (SETs) are modeled by extending our extracted FSM representation to enable flipping of a single bit inside the `cnt` variable for the length of one state as shown in Figure 4.

```
MODULE FSM_CNTVAR
  cnt : word[4];
  set : boolean; <--
  fi : boolean; <--
  errorpos : 0..3; <--
ASSIGN
  init(cnt) := 0h_0;
  next(cnt) := case
    set : cnt xor (0b4_0001 << errorpos); <--
    (cnt = 0h_8) : 0h_0;
    !((cnt = 0h_8)) : cnt + 0h_1;
    TRUE: cnt;
    esac;
  next(set) := seu ? FALSE : fi; <--
```

Figure 4: Augmented FSM representation of the given VHDL counter implementation - Fault injection extensions are marked

Now a very simple functional specification (shown in Equation 2) for this counter can be verified also under fault conditions. The specification says that if the value of the counter exceeds the maximum value (i.e. caused by a fault), it has to be valid again in the next state. While this example may seem trivial, especially such small system parts can play a critical role when certain access limitations are based on hardware counter values.

$$G((cnt >= 8) > X(cnt < 8)) \quad (2)$$

5.2.3 Results

It can be seen that the specification, will be violated under the influence of an injected fault. The NuSMV model checker provides a counterexample that shows that the specification is not fulfilled under faulty conditions (see Figure 5). This can be used to show that the given RTL implementation is not fault resistant and helps the designer making the implementation more robust.

5.3 Experimental Conclusion

Based on an open available implementation of a simple counter implementation, we have shown that high-level and fault injection assumptions can be applied to automatically extracted models in an intuitive way. This approach reduces the verification effort for state-machine based implementations significantly and opens the possibility to support a semi-formal development process as proposed by certification authorities. Furthermore, no specialized model checkers or verification tools, as long as the structure and dependencies of the RTL implementation are extracted, (like our straight-forward implementation of a simple VHDL parser and analyzer) are needed.

This flow can be further enhanced by linking the final synthesized implementation with the abstracted and RTL view. Such an application has been successfully shown in [20] and therefore, with this work a complete flow from high-level description to synthesized netlist can be established.

6. CONCLUSIONS

This work presented a novel methodology for the automatized extraction and verification of FSMs from hardware descriptions written in the VHDL language. The extracted model descriptions are fed into the NuSMV model checker to verify given security rules defined in an LTL format. This high-level representation of the hardware implementation also allows the principal evaluation of the fault-attack robustness by applying fault injection principles on FSM-level. The chosen approach aims to fill a verification gap between formal descriptions as demanded by certification authorities and the RTL implementation. Based on the described tools, a further integration with physical attack analysis flows can be provided.

Our future work includes the further investigation of high-level system security evaluations to help conquering the increasing verification problem in modern smart-card systems. Furthermore, we plan the extension of our tool-chain to other hardware description languages like Verilog.

7. ACKNOWLEDGMENTS

The authors would like to thank the Austrian Federal Ministry for Transport, Innovation, and Technology, which funded the Power-Modes Project under the FIT-IT contract FFG 825749. We would also like to thank our project partners Infineon Technologies Austria AG and Austria Card GmbH.

8. REFERENCES

[1] Common Criteria for Information Technology Security Evaluation Part 2 Version 3.1, 2012.

[2] Z. Andraus and K. Sakallah. Automatic abstraction and verification of Verilog models. In *Proceedings of the 41st annual Design Automation Conference*, 2004.

[3] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122. Springer-Verlag New York, Inc., 2001.

[4] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[5] B. Beckert, D. Bruns, and S. Grebing. Mind the gap: Formal verification and the Common Criteria. *International Verification Workshop*, 2010.

[6] I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: an industry-oriented formal verification tool. In *Proceedings of the 33rd annual Design Automation Conference*. ACM, 1996.

[7] J. Bei, H. Li, J. Bian, H. Xue, and X. Hong. Fsm modeling of synchronous vhdl design for symbolic model checking. In *Proceedings of the ASP-DAC'99*, 1999.

[8] G. Beuster and K. Greimel. Developing a Formal Security Policy Model for a Smart Card EAL6 Evaluation. Presentation, 2011. International Common Criteria Conference.

[9] G. Beuster and K. Greimel. Formal security policy models for smart card evaluations. *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, 2012.

[10] M. Boulé and Z. Zilic. *Generating hardware assertion checkers: for hardware verification, emulation, post-fabrication debugging and on-line monitoring*. Springer Verlag, 2008.

[11] M. Bozzano and A. Villafiorita. The FSAP/NuSMV-SA safety analysis platform. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(1):5–24, 2007.

[12] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, et al. VIS: A system for verification and synthesis. In *Computer Aided Verification*, pages 428–432. Springer, 1996.

[13] B. Chetali and Q. Nguyen. Industrial use of formal methods for a high-level security evaluation. *FM 2008: Formal Methods*, 2008.

[14] P. Chu. *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. Wiley-IEEE Press, 2006.

[15] D. Déharbe, S. Shankar, and E. Clarke. Model checking VHDL with CV. In *Formal Methods in Computer-Aided Design*. Springer, 1998.

[16] J. Ezekiel and A. Lomuscio. Combining fault injection and model checking to verify fault tolerance in multi-agent systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 113–120, 2009.

[17] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer aided verification*, pages 72–83. Springer, 1997.

[18] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2006.

[19] A. Karputkin, R. Ubar, M. Tombak, and J. Raik. Automated correction of design errors by edge redirection on High-Level Decision Diagrams. In *Quality Electronic Design (ISQED), 2012 13th International Symposium on*, pages 686–693. IEEE, 2012.

[20] A. Krieg, C. Bachmann, J. Grinschgl, C. Steger, R. Weiss, and J. Haid. Acceleration of fault attack emulation by consideration of fault propagation. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 239 – –242. IEEE, 2012.

[21] S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design*, pages 142–159. Springer, 2002.

[22] J. Lohse, J. Bormann, M. Payer, and G. Venzl. VHDL-translation for BDD-based formal verification. 1994.

[23] S. Morimoto, S. Shigematsu, Y. Goto, and J. Cheng. Formal verification of security specifications with common criteria. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1506–1512. ACM, 2007.

[24] D. Moundanos, J. A. Abraham, and Y. Heskote. A unified framework for design validation and manufacturing test. In *Test Conference*. IEEE, 1996.

APPENDIX

A. LISTINGS

```
Trace Description : LTL Counterexample
-> State : 1.1 <-
    fsm_cnt.cnt = 0ud4_0
    fsm_cnt.seu = FALSE
    fsm_cnt.finj = FALSE
    fsm_cnt.errorpos = 0
-> State : 1.2 <-
    fsm_cnt.cnt = 0ud4_1
    fsm_cnt.finj = TRUE
-> State : 1.3 <-
    fsm_cnt.cnt = 0ud4_2
    fsm_cnt.seu = TRUE
    fsm_cnt.finj = FALSE
    fsm_cnt.errorpos = 3
-> State : 1.4 <-
    fsm_cnt.cnt = 0ud4_10
    fsm_cnt.seu = FALSE
    fsm_cnt.finj = TRUE
    fsm_cnt.errorpos = 0
-> State : 1.5 <-
    fsm_cnt.cnt = 0ud4_11
```

Figure 5: NuSMV output showing that the VHDL implementation in Figure 6 does not fulfill Equation 2 by providing a counterexample

```
process(cnt)
begin
  if (cnt = 8) then
     next_cnt <= 0;
  else
     next_cnt <= cnt + 1;
  end if;
end process;
```

Figure 6: Simple counter process for exemplary fault injection evaluation

```
MODULE main
 VAR fsm_rx : RX_FSM;
     fsm_tx : TX_FSM(fsm_rx.pwd_given);
MODULE RX_FSM
 VAR rx_fsm : {idle, data, parity};
     rx_ready : boolean;
     rx_data : unsigned word[8];
     ...
IVAR rx : boolean;
ASSIGN
  init(rx_fsm) := idle;
  init(pwd_given) := FALSE;
  ...
  next(rx_fsm):= case
   rx_rcv_init & rx_fsm = idle : data;
   rx_rcv_init & rx_fsm = data &
          rx_data_cnt = 7 : idle;
    ... esac;
  next(rx_ready):= case
   rx_fsm=parity & rx_par_bit = rx: TRUE;
  esac;
  next(pwd_given):= case
   rx_fsm=parity & rx_par_bit = rx &
       rx_data=PWD : TRUE;
  esac;
  ...
MODULE TX_FSM(pwd_given)
VAR
 tx_fsm : {idle, data, parity, stop};
 tx : boolean;
 ...
IVAR
 tx_data : unsigned word[8];
ASSIGN
 -- several initialisations...
 next(tx_fsm):= case
  tx_fsm=idle & tx_init & pwd_given : data;
  tx_fsm=idle & !tx_init : idle;
  tx_fsm=stop : idle;
  ... esac;
  next(tx) := case
   tx_fsm=data : tx_data[0];
   TRUE : UART_IDLE;
 ...
```

Figure 7: Snipped of NuSMV description resulting from Figure 8

```
architecture Behavioral of uart is
 -- several signal definitions...
begin
 -- set rx_rcv_init if falling edge etc...
 rx_proc:process(clk)
 begin
  if clk'event and clk = '1' then
   rx_ready <=  '0';
   case rx_fsm is
    when IDLE => --Wait
     if rx_rcv_init = '1' then
       rx_fsm <= DATA;
         end if;
    when DATA => -- Receive data
    -- serial-to-parallel conversion...
    -- parity bit calculation etc...
     if rx_data_cnt = 7 then
       rx_fsm <= PARITY;
         end if;
    when PARITY =>
     rx_fsm <=  IDLE;
     if rx_par_bit = rx then --check parity
       rx_ready <= '1';
       if rx_data = PWD then --check password
         pwd_given = '1';
           end if;
     end if;
     ...
    end case;
  end if;
 end process;
 tx_proc:process(clk)
 begin
  if clk'event and clk = '1' then
   tx <=         UART_IDLE;
   case tx_fsm is
    when IDLE =>
     if tx_init = '1' then
      if pwd_given = '1' then --check pwd
        tx_fsm <= DATA;
       --initializations etc...
    when DATA =>
      --parallel-to-serial conv. etc...
      tx <= tx_data(0);
      tx_fsm <= STOP;
    when STOP =>
      --Send stop bit etc...
      tx_fsm = IDLE
   end case;
  end if;
 end process;
end Behavioral;
```

Figure 8: VHDL snipped of an UART logic with password functionality