SPARC: SPecification and Analysis of HardwaRe-Software InteraCtions

Kshitij Raj, *Member, IEEE,* Sandip Ray, *Senior Member, IEEE*
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA
kshitijraj@ufl.edu, sandip@ece.ufl.edu

***Abstract*—Recent years have seen a dramatic increase in the size and complexity of firmware (FW) in custom and SoC designs. In a commercial SoC design, it is common to find more than a dozen IPs implementing significant functionality through FW-FW in different IPs that can execute on different microcontrollers, e.g., a typical SoC may include microcontrollers implementing X86, AR, 8051, etc., as well as a proprietary custom instruction set. Given this extensive use, it is increasingly crucial to develop validation technologies for the analysis of functional and security properties of modern FW.**

A key challenge with FW validation is the need to comprehend its interaction with hardware (HW) as well as other FWs in an SoC. Furthermore, since FW is developed in a resource-constrained environment, a significant component of FW implementation entails managing these resources manually. We address this problem through a comprehensive infrastructure for systematic specification and analysis of hardware-firmware interactions. The infrastructure includes a specification frontend that enables the design of intuitive executable specification of hardware-firmware interactions as well as a synthesis backend that enables the automatic generation of various implementation and validation collateral. Specifications designed using the infrastructure are formal; however, the front end is carefully designed to use by architects having limited familiarity with formal methods. We show how to integrate off-the-shelf tools with the infrastructure to enable a formal analysis of such the interactions defined. We demonstrate the application of the framework in the specification and analysis of representative secure boot flows.

***Index Terms*—System-on-Chip, Specification, HW-SW Interaction, Formal Validation.**

I. Introduction

MODERN System-on-Chip (SoC) designs have seen an exponential rise in the size as well as the complexity of interacting HW-SW components, ranging from information flow, security-critical asset management, inter-IP communications, etc. Ensuring the correct implementation of such interactions is critical in ensuring a bug-free design, starting from the specification of such interactions. Unfortunately, a systematic mechanism for specifying such interactions does not exist apart from traditional and free-form diagrams and documents written in ambiguous English. In current industry practice, specifications are defined by multiple players, often without prior knowledge of how the HW or SW is visioned to be integrated into a system. For example, it is common practice for an architect to write a specification of multiple inter-operable systems on an SoC, which is then formalized into implementations by system integrators and hardware designers after a sequence of refinements. Such specifications are often presented in documents, charts, and diagrams in ambiguous language. The refinement process followed by system integrators and hardware designers, who often lack the full picture of specification, introduces "subtle" optimizations in the realization of the specification, leading to vulnerabilities being introduced into the system, and compromising the operation and reliability of the SoC. To further exacerbate the situation, if these vulnerabilities are detected later in the pipeline, e.g., post-fabrication or during deployment, the fix is expensive and frail, with potential butterfly effects on the entire SoC.

This paper introduces a novel language, SPARC (for "**SP**ecification and **A**nalysis of Hardwa**R**e-Software Intera**C**tions"), for specifying the functionality of hardware or software entities in modern SoCs, while also providing a platform to model their interactions and undergo validation and analysis. The goal of SPARC is to (1) enable an intuitive definition of HW-SW components in an SoC design, and (2) provide a platform for the validation and analysis of the specification. SPARC provides intuitive constructs for specifying various functions specifications of various aspects of SoC designs, while also enabling formal semantics.

While there has been some work on languages and frameworks for specifications for micro-architectures and SoC, there has been little adoption of formal methods for the analysis of such specifications. The fundamental reason for this can be attributed to a number of reasons, the notable ones being (1) lack of familiarity with formal semantics and paradigms, and (2) describing specifications accounting for low-level implementation heuristics. As Leslie Lamport stated: "*An algorithm without proof is a conjecture, its not a theorem*" [1], similarly, a specification without validation is a description, not a blueprint. The key idea of SPARC is to introduce formal semantics in the specification of HW/SW definition and interaction. SPARC does this by taking an orthogonal approach to the specification design. Most specification languages achieve the notion of formalism using some form of Temporal Logic. On the contrary, the objective of SPARC is *extensibility* and *usability*. The SPARC paradigm neither requires nor assumes any previous expertise in Temporal Logic or Formal Methods. Albeit unfamiliar with Program Semantics, Temporal Logic, or Formal Methods, are familiar with programming languages, performance modeling, and executable specifications, i.e., System-TLM. – To that extent, SPARC, built on top of an existing language like C++ and extending a subset of the language by providing constructs to capture and express HW/SW definition and interaction, is in our thinking, indeed a pragmatic approach in delivering SoC specifications.

II. Background and Related Work

The increasing complexity of the SoC specification design process acts as a butterfly effect, raising the complexity of subsequent processes as a result. As a result, the overall Verification and Validation (V&V) effort increases exponentially. The objective of (V&V) is to ensure that the system behaves according to the specification. To reduce this effort, a strategy adopted by the industry is to initiate the (V&V) efforts at the specification abstraction level to ensure that errors and design vulnerabilities are identified and fixed at the specification level. The traditional and standard mechanism of writing specifications is to use charts, figures, and documents, which although h capture the system specifications, are infeasible to (V&V) due to the informal and non-machine analyzable nature of such documents. A solution to this problem has been, in theory at least, the adoption of numerous languages and semantics which try to standardize and capture specifications in a formal description such as Unified Modeling Language(UML), Specification Description Language (SDL), etc.

Although there are numerous languages and paradigms for hardware-level specification and verification such as CTL [2], Sapper [3], etc., these languages either have a steep learning curve or fall short on modeling system interactions involving multiple entities in the system. Other approaches [4] [5] [6] [7] based on formal semantics and temporal logic are promising solutions to automated testing from specification synthesis. These solutions capture specifications in an unambiguous and precise format but the drawback is the prerequisite knowledge of formal methods. Other specification frameworks include linguistic-based approaches with textual descriptions [8] [9], using lexical and semantic analysis.

III. The SPARC Language

SPARC is built by extending a subset of C++ with additional constructs that enable an architect to specify the hardware or software design block at the top level and also the interaction of components in the SoC. SPARC's rationale is as follows: Instead of developing an entirely new language from scratch, the language provides an expansive set of constructs built on top of C++, which an architect can treat as *System Description APIs*. The goal is to enable architects to depart from the traditional practice of describing SoC specifications in informal charts and documents and transcribe those specifications using SPARC. An added benefit of having the infrastructure based on a language like C++ is the existing toolchains and compilers already developed for the language. This eradicates the need to develop new synthesis tools and formal validation paradigms to generate executable specifications.

A. Language Overview

The SPARC language is designed to abstract away the low-level implementation details of the HW or SW entity in the SoC. The focus here is to express system-level communication and interaction between components. Features to express the

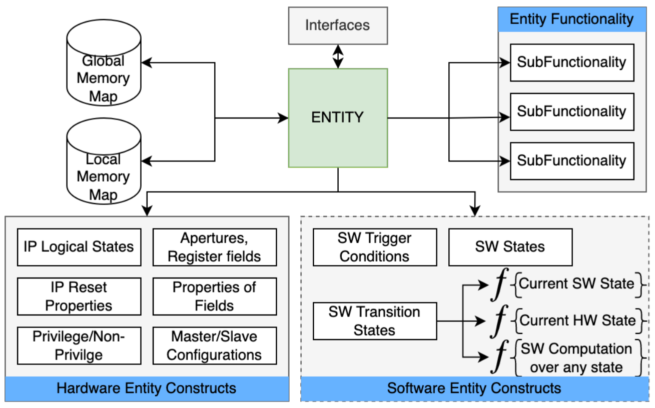


Fig. 1: Some SPARC Constructs for Modeling HW-SW System Specifications

same are baked into the language, and architects can be oblivious to their implementation details. SPARC constructs (Fig. 1) encapsulate the intrinsics of system modeling and specifications can be verified via dynamic or formal synthesis using existing validation tools. For each entity in the SoC (HW or SW), SPARC defines a new component with precise interfaces, a functional model, and the atomicity of the functional model, enabling the SPARC validation engine to model the precise semantics of the specification for synthesis and validation.

B. SPARC Grammar and Decorators

Fig. 1 shows an expansive SPARC feature set for specifying hardware-software entities and their interaction. The language provides succinct grammar to express such features. Note that in some cases, not all these features are required depending on the complexity of the specification and the granularity it dives into, and to that extent, each of these features can be explicitly turned off by the architect. Based on an initial configuration of the entity, SPARC automatically generates a template, populating each field shown in Fig. 1. The sole responsibility of the architect to complete the specification is to provide the definition of the low-level behavioral blocks of the entity. One key aspect to consider is the notion of atomicity while specifying the low-level behavior of such blocks. Each block should capture the lowest level in the functionality of the entity, consisting of one execution unit, and must not contain multiple atomic units. This is to ensure the sanctity of atomicity is preserved in the program semantics.

To establish a 1-1 correspondence, this can be thought of as 1 block in the flowchart of the specification should correspond to one atomic unit in the behavioral model of the entity. This also ensures that the architect has to do nothing more than what they are already doing while specifying specification behavior as per the current industry practice.

SPARC also provides constructs for writing assertions to prove key properties of the system specification (Fig. 2). While these assertions are not synthesized during the executable generation, SPARC automatically synthesizes all assertions during dynamic and formal validation.

```
// Events
@event_ bool __timerfinish  = false;

// History and Prophecy Variables
@history_  bool __globalresetAssert = false;
@prophecy_ bool __crcTriggerAssert  = false;

// Assertions
@assertion_ assert(Condition or Property);
```

Fig. 2: Some Decorators in SPARC

C. Entity Definition

The definition of a hardware or software control entity in an SoC environment is based on two basic pillars: (1) the interfaces of the entity, and (2) the functional model of the entity. Fig. 3 shows a representative instantiation of a Timer controller based on these two pillars using SPARC grammar. This is auto-generated using a basic configuration provided by an architect, and the description of the functionality of low-level behavioral units is the "only" requirement from an architect. One might observe a remarkable similarity of this grammar with C++. The resemblance has been intentionally preserved to enable architects to use this infrastructure without having to pick up a new language or paradigm. SPARC provides accurate semantics to specify the interfaces of entities, describing the IO, name, and default value of the interface using built-in interface registers in the language.

```
#include "SSEL_HEADER.h"
class TIMER_CONTROLLER : public entity{
// Interface definitions of the Timer Controller
   public:
        iReg duration   = {"duration",   "INPUT", 0};
        iReg startTimer = {"startTimer","INPUT", 0};
        iReg timerDone  = {"timerDone",  "OUTPUT",0};
// Low-level behavior functions of the Timer Controller
   private:
        @atomic_unit void timer_start();
        @atomic_unit void waiton_Reset();
};
```

Fig. 3: A Representative Timer Controller Instantiation using SPARC

System entities (HW or SW) should be specified in self-isolated instances of their respective class types, with explicit interface definition, memory arrangements, and timing constraints. To specify the functional model of the entity, SPARC provides a modified subset of features derived from C++ such as loops, functions, structures, etc. Multiple low-level behaviors can be coupled together to realize a higher-level behavior that aims to capture the entity's functionality. These sub-behaviors are executed using the principle of compositionality. These low-level behaviors must however be atomic in terms of expression, meaning this behavior is guaranteed to be executed as a single transaction. This can be achieved using the `@atomic_unit` construct in SPARC. Using SPARC-provided constructs guarantees the formal validation of the specification, as these constructs have been designed keeping formal validation and its scalability challenges in mind.

IV. Analysis of Specifications using SPARC

The analysis and V&V efforts can be segmented into two branches: (1) Synthesis under Dynamic Validation, and (2) Synthesis under Formal Validation. SoC specifications inherently possess the notion of concurrency, as by design, each entity in the SoC operates independently. To preserve such program semantics and the concurrent behavior of the specification, SPARC considers each entity to be its own process, working coherently with other entities in the SoC.

SPARC integrates formal validation of the specifications using the in-built engine which is seamlessly integrated with existing model checking and formal validation tool-flows, providing a mechanism to validate the specifications.

A. Synthesis under Dynamic Validation

Using this validation flow, SPARC generates executable specifications which are validated using off-the-shelf validation and debug tools for C++. This executable remains concurrent in nature and synthesizes all validation lemmas included by the architect in the original specification including assertions, properties, history and prophecy variables, etc. Structural vacuity checks are also integrated into this validation flow.

B. Synthesis under Formal Validation

One major challenge of extending the formal validation is providing support for multi-threaded sequences of the specification. To tackle this, we created a novel formalization of concurrency. This formalization enables the architect to simply write specifications in a multi-threaded program while enabling existing formal tools to be applied for analysis of the code (even though most formal software model-checking tools cannot consume multi-threaded implementations). The formalization (Fig. 4) works as follows: SPARC creates an interleaved sequence of all atomic operations in the specifications using the `@atomic_unit` decorator. Collateral specified by the architect including monitors, history and prophecy variables, assertions, and properties are then synthesized automatically and based on the syntax tree generated by SPARC, all such collateral are positioned within the instruction tree, ensuring that program semantics are preserved. Then, SPARC creates a test harness for formal validation by placing the first atomic operation of each entity into its separate queue, and symbolically picking a processID for each entity and executing the queue exhaustively. This permits the formalization of such semantics directly as multi-threaded code, while still permitting formal analysis using C++-based tools. SPARC also incorporates a property verification engine. This can be used both as a debugging aid and as a formal analysis tool to verify properties involving concurrent interleaving, particularly targeting invariants that involve multiple entities in the system.

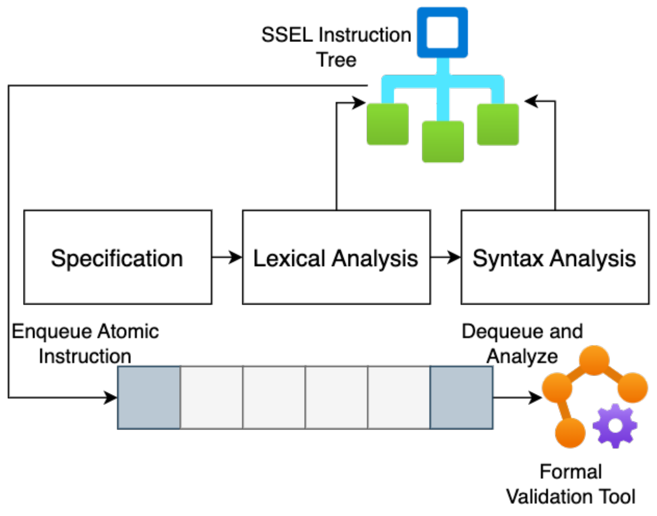


Fig. 4: SPARC Pre-Processing and Tree Construction

Formal Test Harness

Up to this point in the validation flow, all program transformations are independent of the underlying formal validation tool used for specification validation and analysis. The test harness generator (Fig. 5) integrated into SPARC then generates the test harness depending on the validation tool to be used. As of now, SPARC supports the KLEE Symbolic Execution Engine [10] for formal validation.

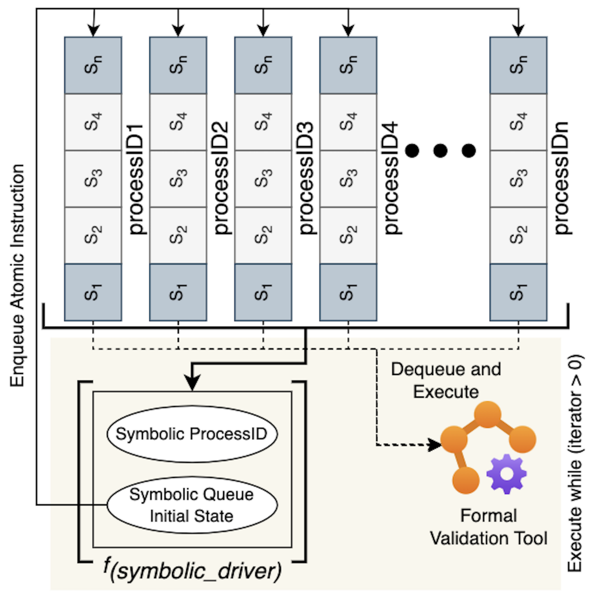


Fig. 5: SPARC Test Harness Generation and Test Execution

As shown in Fig. 5, the SPARC test generator randomly picks a processID (P_{id}) and its initial state (S_i), where $S_i \in \mathbb{A}$, where $\mathbb{A}$ is the set of atomic instructions in the queue of that P_{id}. This state is then dequeued and executed by KLEE, and the control flow of the driver of the entity ensures the next correct atomic instruction is enqueued for execution.

V. SPARC Case Studies

In this Section, we showcase certain specification mechanisms and validation using a case study based on Fig. 6. In this representation, the specification has four entities, the reset controller, the software controller, the CRC, and the Timer module. Each of these entities can be specified similarly as Fig. 3. In an SoC, each entity would have its own thread, and operate concurrently *w.r.t.* other entities in the system via interaction. When the system specification is being developed

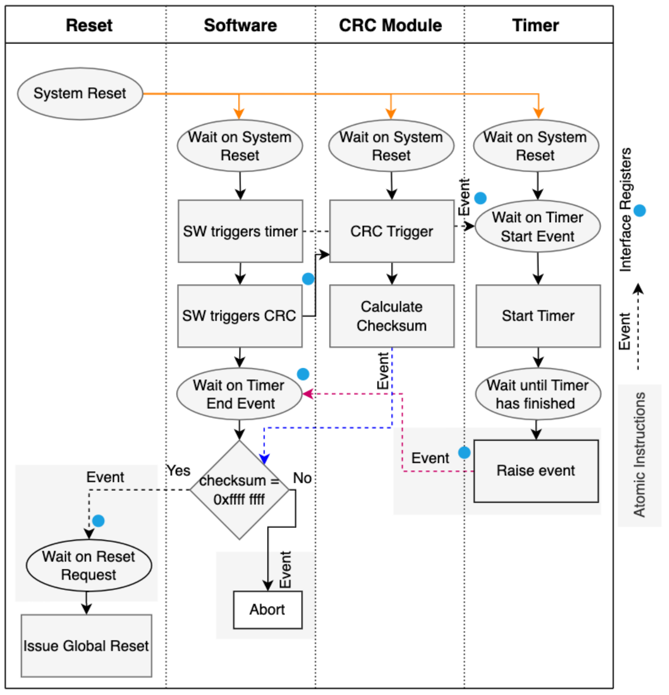
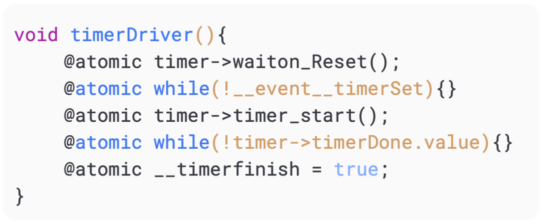


Fig. 6: SPARC Framework Workflow

and all individual entities are being integrated, the integrated specification would require events, assertions, history and prophecy variables, etc. The architect is free to add any such instruction anywhere in the specification, but the decoration provided by SPARC needs to be used for dynamic and formal validation to successfully execute. A brief snapshot of such decorators in the SPARC grammar is provided in Fig. 2.

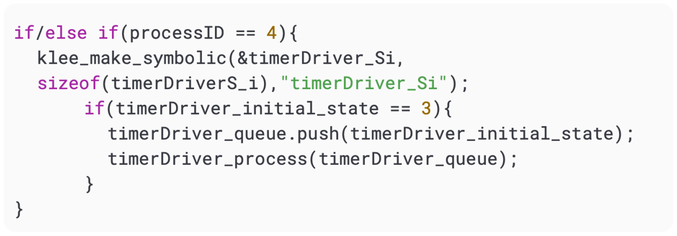
When creating the integrated specification, each entity is contained in its own driver function. The program control (main()) spins each driver in an independent thread, and communication & interaction between threads are facilitated via *"events"* provided by SPARC. As shown in Fig. 6, the Timer has five atomic operations, and the driver for the Timer in the integrated specification (Fig. 8) also has five corresponding atomic instructions. The procedure for the other entities in the system is similar to this definition.

The test harness is generated using the process described in Fig. 2. A processID (P_{id}) is first symbolically set, and the initial state of the P_{id} queue is set symbolically. This test harness is then executed by KLEE and the validation continues until all instructions in the P_{id} queue have been executed. Any assertions or monitors in the specification are also synthesized and checked during execution.



```
void timerDriver(){
    @atomic timer->waiton_Reset();
    @atomic while(!__event__timerSet){}
    @atomic timer->timer_start();
    @atomic while(!timer->timerDone.value){}
    @atomic __timerfinish = true;
}
```

Fig. 7: Wrapper Function for the Timer Module



```
if/else if(processID == 4){
  klee_make_symbolic(&timerDriver_Si,
  sizeof(timerDriverS_i),"timerDriver_Si");
      if(timerDriver_initial_state == 3){
        timerDriver_queue.push(timerDriver_initial_state);
        timerDriver_process(timerDriver_queue);
      }
}
```

Fig. 8: Symbolic Test Wrapper

VI. Conclusion

SPARC presents a systematic and standardized way of developing formal and analyzable SoC specifications, capable of defining a wide variety of use cases in modern SoC designs. Built on top of an established language like C++ helps this ecosystem leverage on existing tools for validation while keeping the learning curve as shallow as possible. This also reduces the design complexity of developing specifications. This design philosophy provides a generic yet scalable solution interoperable with modern-day architectures. Built-in SPARC constructs and features provide a succinct and exhaustive mechanism, at the same time, being extensible to incorporate new constructs in the language with relative ease. Using and existing programming language for the foundation of SPARC lends itself to interesting challenges for formal validation, one being concurrency. We explained how SPARC addresses those challenges. In future work, we aim to extend SPARC to be compatible with diverse formal validation tools as the current incarnation provides support for only KLEE. We also plan to develop intuitive constructs and features for information flow and access control and address scalability challenges in the formal validation of SoC specifications.

Acknowledgement

This research is being supported by Semiconductor Research Corporation (SRC) Task No. CADT 3112.001. The opinions and conclusions in this paper are those of the authors and do not reflect the views or opinions of the sponsoring agency, either implied or expressed.

References

[1] Q. Magazine. The man who revolutionized computer science with math. Youtube. [Online]. Available: https://www.youtube.com/watch?v=rkZzg7Vowao&ab_channel=QuantaMagazine

[2] R. Kapur, M. Lousberg, T. Taylor, B. Keller, P. Reuter, and D. Kay, “Ctl the language for describing core-based test,” 02 2001, pp. 131–139.

[3] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, “Sapper: A language for hardware-level security policy enforcement,” *SIGARCH Comput. Archit. News*, vol. 42, no. 1, p. 97–112, feb 2014. [Online]. Available: https://doi.org/10.1145/2654822.2541947

[4] J. Tretmans and E. Brinksma, “Torx: Automated model-based testing,” *First European Conference on Model-Driven Software Engineering*, 01 2003.

[5] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, “Requirements by contracts allow automated system testing,” in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, 2003, pp. 85–96.

[6] A. Rajan, “Automated requirements-based test case generation,” *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 6, p. 1–2, nov 2006. [Online]. Available: https://doi.org/10.1145/1218776.1218799

[7] J.-R. Abrial, *The B-Book: Assigning Programs to Meanings*. USA: Cambridge University Press, 1996.

[8] M. Riebisch and M. Hubner, “Traceability-driven model refinement for test case generation,” in *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS’05)*, 2005, pp. 113–120.

[9] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari, “Application of linguistic techniques for use case analysis,” in *Proceedings IEEE Joint International Conference on Requirements Engineering*, 2002, pp. 157–164.

[10] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf