Menhir: Generic High-Speed FPGA Model-Checker

Emilien Fournier
Lab-STICC, ENSTA Bretagne
Brest, France
emilien.fournier@ensta-bretagne.org

Ciprian Teodorov
Lab-STICC, ENSTA Bretagne
Brest, France
ciprian.teodorov@ensta-bretagne.fr

Loïc Lagadec
Lab-STICC, ENSTA Bretagne
Brest, France
loic.lagadec@ensta-bretagne.fr

***Abstract*—Among formal methods, model-checking offers a high-level of automation and can lower the cost of the verification process. Two preliminary studies on FPGA model-checking show a high-performance increase, thanks to the massive parallelism and precise memory control opportunities. However, these approaches rely on HDL-based ad-hoc model encoding, and miss the importance of decoupling the modeling language from the verification core, which greatly limits their usability.**

In this paper we propose Menhir, a new highly modular hardware model-checker, inspired by the architecture of software verification frameworks. Menhir is based on a generic language-verification interface which isolates the modeling-language semantics from the verification core, allowing their independent evolution. Menhir opens the architecture to the whole spectrum of modeling languages. Moreover, it proposes a polymorphic verification core, which offers a continuum between partial and exhaustive verification, with promising performances.

***Index Terms*—Model-checking, Verification Core, FPGA**

I. Introduction

Formal verification offers the promise of guaranteeing the system's correctness through mathematical proofs. Among formal verification methods, model-checking [1], [2] enacts a proof-by-counterexample approach. Model-checking is a generic and rather intuitive proof technique with good diagnosis capabilities, successfully applied to the verification of critical hardware and software [3]. However, model-checking is often challenged with the state-space explosion problem [4]. Numerous techniques, such as symbolic model-checking, partial-order reduction, and model abstraction, have been proposed to reduce the impact of this problem [3]. Complementary with these techniques, a research trend focuses on the decomposition of the verification problem [5]–[8], aiming at reducing the memory requirement at the expense of higher verification time. Moreover, the emergence of low cost parallel and distributed architectures have fueled, during the last 15 years, the emergence of a large number of model-checking algorithms targeting SIMD [9], shared memory [10], and distributed systems [11].

Furthermore, two approaches for accelerating model-checking using FPGAs have been proposed. The first approach [12], inspired by the Murϕ [13] tool, focuses on building a safety model-checker on FPGA and has shown a 200X speed improvement over Murϕ. The second approach [14], is a FPGA realisation of the swarm verification proposal by Holzmann [5]. It achieves 900X speedup while reducing by 10X the power consumption. Sadly, their evaluation is based on manually-coded ad-hoc VHDL models. No solution is provided for the integration of the commonly used modeling languages, like Murϕ [13], Promela [15] or DVE specification language [16]. Furthermore, their relative performances are very difficult to compare, precluding incremental algorithmic improvements. Consequently, despite the promising results, the research community highly disregarded the usage of specialized circuits for accelerating model-checking.

Our claim is that the root cause of this failure lies in the implicit interleaving of the modeling language semantics with the verification algorithms. This led to an opportunistic and unstructured research methodology which missed the extreme importance, and the industrial relevance of high-performance solutions for the model-checking problem.

In this paper we propose Menhir, a new modular hardware model-checker, inspired by the architecture of highly successful software model-checking frameworks [15], [17]. Firstly, Menhir decouples the verification core from the modeling language semantics by introducing a generic language-agnostic verification interface (**GLI**). This interface enables the seamless integration of off-the-shelf specification-language semantics with the verification core, thus opening the model-checking core to the whole spectrum of specification and modeling languages. Secondly, Menhir provides a polymorphic verification core, which offers a continuum between exhaustive and partial safety verification algorithms. Moreover, by isolating the language semantics, the proposed architecture eases the objective evaluation of the improvements on the verification core. Furthermore, Menhir is open to evolution. It is intended to be a platform for fostering collaborative research and development of hardware model-checking cores.

Current status: Menhir offers a pure hardware configuration as well as a system-on-chip (SoC) configuration. The pure hardware configuration relies on a VHDL-based **GLI**-specific modeling language for capturing the verification problem. We also propose a SoC solution, which relies on existing software implementation of the modeling language. Currently, Menhir integrates two existing system specification formalisms: *a)* DVE, an academic formal language, developed around Divine tool [16] and also supported by LTSmin [17]; *b)* EMI, a formal subset of the industrially used OMG UML standard [18] designed for embedded executable specifications, currently supported by the OBP2 software model-checker [19]. The verification core integrates six verification algorithms, ranging from exhaustive reachability, to bitstate-hashing and bounded model-checking.

Our approach is evaluated a Zynq ZedBoard using a generic benchmark encoded in all three modeling formalisms (GLI, DVE, EMI). The results show that the pure hardware configuration is 50 times faster than the equivalent software setup. The SoC configuration is slower but offers realistic modeling capabilities.

Sec. II introduces the background and related works before stating the problem addressed. Sec. III presents our contribution, the Menhir FPGA model-checker. Sec. IV discusses the experimental setup and shows our evaluation results. Sec. V concludes this paper giving some future research directions.

II. Background and Problem Statement

This section introduces the framework necessary for appreciating our contribution, overviews the related work and characterizes the problem addressed. The interested reader is directed to [20] for a detailed description of model-checking.

A. Background

Among formal verification techniques, model-checking provides a generic and powerful automated proof technique based on the analysis of the state-space underlying the execution of a model. The applicability of this technique is based on the hypothesis that the model, viewed as transition-system, induces a finite state-space that can be exhaustively enumerated. The underlying verification problem is to check if the model satisfies a specification. Both, the model and its specification, are viewed as transition systems. Model-checking verification consist in checking if the model includes paths of the complemented specification. If the specification is considered correct, its complement includes all unwanted behaviors. Thus obtaining an empty intersection *proves that the model does not contains unwanted behaviors* [20].

The specification language is typically based on *ω-automata*, which can recognize infinite-words in *ω-regular* languages. While this encoding is generic, in practice, however, *non-deterministic finite automata* (NFA) – recognizing finite words – are typically used for the significant, practically important, subclass of **regular safety** specifications. In this case the verification procedure simply requires the computation of the state-space, by any reachability procedure while looking for the unsafe states [20].

Based on the model-encoding, the model-checking can be classified as offline of **online**. Offline verification, requires the generation of the state-space before running the state-checking routine. For online verification, the model is *represented implicitly* (by a program) and the verification procedure interleaves the generation of the state-space with the state-checking routine.

According to the internal representation of the state-space, the model-checking algorithms are either **explicit-state** or symbolic. Explicit-state approaches analyse the state-space iteratively, one configuration at a time. Symbolic approaches, manipulate compressed sets of states via satisfiability solvers. Both approaches have their advantages and disadvantages [21], however explicit-state model-checking seems to allow easier binding of complex modeling languages [19], while still allowing the use of symbolic algorithms via explicit-to-symbolic conversions [17].

In this study we focus on **explicit state online** model checking for **regular safety** specifications [22].

B. The Verification Algorithm

In the following, when referring to the states in the state-space constructed during reachability, we will use the notion of **configuration**, to lower the risk of confusion with the notion of state in state-based modeling languages. Abstractly the set of configurations (the state-space) can be seen as a finite type $\mathcal{C}$ with each configuration an element of the type.

One key insight, that structures our approach, is that the next-state generator component and the invariant checker are intimately linked to the semantics of the input language. Hence, in the following, we group them together to build the **model frontend**. Without loosing generality, an implicit model built over an arbitrary configuration type $\mathcal{C}$ can be defined as the structure $\mathcal{M}(\mathcal{C})$, shown in Listing 1, where **initial** is a nullary function returning the set of initial configurations, **next** is a function returning the set of configurations reachable from a given configuration, and **is_safe** is a predicate over a configuration that encodes a safety assertion. This abstraction allows the verification algorithms to handle each configuration as an opaque binary word, with its interpretation delegated to the model frontend. Moreover, this representation abstracts away the semantics of the verification problem, which is encapsulated in the **next** function. Thus, the synchronous composition, between the model and the specification, needed for the verification is also hidden from the algorithmic backend.

```
structure 𝓜 (𝓒 : Type) :=
    (initial : set C)
    (next    : C → set C)
    (is_safe : C → bool)
```

Listing 1. Implicit representation of a model

The verification algorithm can be defined as the **safety_checker**(m : $\mathcal{M}$) predicate, presented in Listing 2. The algorithm progresses updating two sets: the known $\mathcal{K}$ and the frontier $\mathcal{F}$, both initially empty. The known set, stores the states encountered. The frontier set contains only the configurations that have been discovered recently, which are not yet analysed. A configuration x is considered analysed when its fanout has been computed (via the call to **m.next(x)**). The algorithm starts by initializing a neighbours set $\mathcal{N}$ with the initial states (line 4). The *do while* loop computes a fixed point on the known set using the elements in the frontier set. When the frontier set is empty all configurations have been analysed. The closure loop firstly asserts that all neighboring configurations are safe (line 6). If an unsafe configuration is found, the algorithm terminates returning false (the model m is not safe). If the configurations in $\mathcal{N}$ are safe, the known and the frontier sets are updated (the simultaneous assignment $\Leftarrow$ on line 8). The neighboring configurations are added to the known. The new configurations are added to the frontier set. Then a new neighboring set is computed for the next iteration of the loop. Please note that the algorithm in Listing 2 is explicitly kept abstract and simple (without any optimization) to illustrate the minimum set of requirements both in terms of data-structures and operations.

```
def safety_checker (m : $\mathcal{M}$) : bool :=
  $\mathcal{K} \leftarrow \emptyset$
  $\mathcal{F} \leftarrow \emptyset$
  $\mathcal{N} \leftarrow$ m.initial
  do
    if $\exists$ n $\in \mathcal{N}$, $\neg$ m.is_safe(n) then
      return false
    $\mathcal{K}, \mathcal{F} \Leftarrow \mathcal{K} \cup \mathcal{N}, \mathcal{N} \setminus \mathcal{K}$
    $\mathcal{N} \leftarrow$ { n | $\forall$ x $\in \mathcal{F}$, n $\in$ m.next(x) }
  while $\mathcal{F} \neq \emptyset$
  return true
```

Listing 2. Generic safety verification function

C. Related Work

Despite the simplicity, and the linear complexity (in the number of configurations) of the verification algorithm presented in the last section, in practice, model-checking is confronted with very large state-spaces [4]. Numerous research efforts addressed this problem [3]. Besides symbolic approaches [23], most of these techniques focus on **reducing the complexity** of the verification problems through symmetry [24] or partial-order reduction [25] and through model abstraction [26]. This study is focused on accelerating the core verification algorithm. Nevertheless the approach is fully compatible with these complexity reduction techniques, which can be applied without restrictions in the model frontend implementation.

Complementary research efforts focus on the decomposition of the verification problem. These approaches trade-off either the verification time [6]–[8] or the completeness [5] in exchange for smaller state-spaces during verification. In [6], [7] the authors use verification guides to split the hard verification instances in smaller problems, which can be discharged independently. In [8] the authors propose decomposing the specification into multiple smaller properties, which reduce the state-space. Swarm verification [5] arbitrarily dispatches the configurations to multiple instances of the verification algorithm. Each instance uses a randomized access to the frontier set to explore different parts of the state-space. To further reduce the memory requirements the known set is represented using a probabilistic data-structure, a bloom-filter, approach known in the literature as bitstate-hashing [27]. This approach trades-off the completeness proof, ensuring only a high probability of full coverage. Our approach supports bitstate-hashing and the verification core could be replicated to implement swarm verification.

Bounded model-checking (BMC) [28], [29] also trades-off completeness in exchange for manageable state-space. This approach relies on the observation that a breadth-first search on a tree-like interpretation of the state-space requires only the storage of 2 layers of the state-space (instead of d-layers, where d is the diameter of the state-space). However, this optimization comes at the cost of loosing termination, which is restored by limiting the number of layers analysed (typically an arbitrary bound). Thus, similarly to swarm and bitstate-hashing, BMC also looses the completeness, which can only be established by proving that the bound chosen is sufficient (at least equal to the diameter for reachability). Our verification core can be configured to perform an explicit-state BMC procedure.

The ever increasing cost of verification time along with the emergence of low cost parallel and distributed architectures has pushed the community to studying parallel model-checking solutions. Numerous works targeted SIMD architectures [9], multi-core shared memory [10], distributed systems [11]. Besides these, two approaches [12], [14] rely on building dedicated hardware verification cores on reconfigurable FPGA architectures.

The first approach [12], named PHAST, focuses on building a pipelined model-checker for safety specifications. Inspired by the Murpϕ software model-checker [13], this approach implements a probabilistic verification procedure based on hash-compaction [30]. The particularity of this approach is that the known set does not store the full configurations but only a hash signature of the them, which reduces the memory pressure. In the initial 2008 paper [12], the authors claim a 200X performance improved over Murϕ. However, in the master thesis of the first author, published in 2012 [31], the performance improvement decreased to 30X, which is very impressing still. Currently, the Menhir core does not yet support the hashcompaction algorithm, however the genericity of the core allows its implementation while benefiting from the integrated modeling languages and the existing pipeline structure.

More recently, in 2018, a new hardware core for safety model-checking was proposed [14]. This time inspired by the swarm verification procedure [5] of the SPIN model-checker [15]. Implemented using C-based High-level Synthesis (HLS), FPGASwarm dispatches randomized partitions of the state-space to a swarm of 40 verification cores. The results achieve 900X speedup and 10X lower power consumption against SPIN running on a 24 core supercomputer. In the context of this paper we focused on the creation of generic verification core, which could be replicated to realize the swarm approach.

D. Problem Statement

These research efforts on hardware model-checking traded-off the completeness of the verification for lower memory pressure, proposing probabilistic verification procedures. Thus, currently we observe the **lack of research on accelerating exhaustive safety model-checking**.

Moreover, both PHAST [12] and FPGASwarm [14] based their evaluation on manually-coded ad-hoc models (in VHDL), **without providing any solutions for the integration of commonly used modeling languages**, like Murϕ [13], Promela [15] or DVE specification language [16]. This emphasizes the

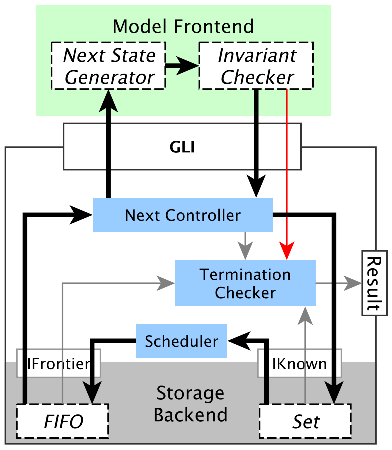

Fig. 1. Architecture overview. The dashed boxes represent placeholders for specific modules: language specific components on the model fronted, and Known and Frontier set implementations in the storage backend. The blue rectangles are internal components of the Menhir controller. The arrows show the flow of configurations (thick arrows) and control signals.

need for creating semantic bridges between these verification cores and the modeling languages used by the community.

Furthermore, while both approaches encapsulate the next-state generation and the state-validation in separate modules, they **lack the definition of a generic interface**, which could enable the independent evolution of the verification backend from the model frontend. Moreover, they **do not exploit at all the algorithmic variability**.

III. Menhir: A Modular Hardware Model-Checker

This section overviews the architecture of Menhir, presents the model-algorithm interface and the variability of the verification core.

A. Overview

The Menhir approach splits the modeling semantics from the verification engine and promotes a modular algorithmic backend. Exploiting the algorithmic variability enables morphing the verification core to implement multiple verification algorithms. The verification engine is implemented as a dedicated hardware design, which interacts through well specified interfaces with the executable models under verification.

The model-checker in organized in two parametric layers, Fig. 1. The *Model Fronted* layer encapsulates the *Next State Generator* and the *Invariant Checker* behind a *Generic Language Interface* (GLI), which mediates the dialog with the verification core. The verification core itself is decomposed in two parts: the controller and the storage backend. The Menhir controller itself has three modules: *a)* the *Next Controller*, which mediates the access to the GLI interface; *b)* the *Scheduler*, which forwards the newly discovered configurations to the frontier set; and *c)* the *Termination checker*, which monitors the progression of the algorithm deciding the termination conditions. The *Storage Backend* harbors the known and frontier set representations. The controller accesses these representations through two generic interfaces: IKnown and IFrontier.

The operations apply to the configurations in a pipelined fashion. Fig. 1 shows the path of a configuration (thick black arrows). A new configuration n is generated by the *next state generator*, which produces either an **initial** configuration (at the beginning) or the **next** from a source configuration x. The new configuration is fed through the *invariant checker* which asserts the **is_safe** predicate. If the configuration is safe, then it is forwarded to the *Known* set. If the configuration is not already included in the known, it is added and passed to the scheduler, which adds it to the *Frontier* set $\mathcal{F}$. Based on the order imposed by the frontier set implementation, a frontier configuration is selected as source and sent to the *next state generator* as soon as it finishes producing the neighbours of the previous source. To guarantee termination, each of the pipeline blocks can send a signal to the *termination* checker which propagates the results at the end of the algorithm. While the algorithm in 2 terminates only in two cases, namely if an unsafe state is found, or if the frontier set is empty, the end of the verification task is a bit more complex in hardware. The pipelined implementation involves several states being processed at the same time. Then the termination is occurring only when the *Frontier* set is empty, and no states are being processed by the *Model Frontend*, nor by the *Known* set.

B. Isolating The Model From The Verification Core

The conceptual organization presented in the Listing 1 (Section II) already poses a good basis for achieving the model isolation. It hides the modeling and specification language details as well as their synchronous composition behind a functional interface. Composed of three functions, this interface can directly be invoked by the verification algorithm (shown in Listing 2).

However, the high cost of invoking hardware functions pushes us to refine this interface. Each call of the **is_safe** predicate needs sending a full configuration from the verification core to the model frontend to retrieve one bit of information in exchange. Inlining this call, in the new model structure, reduces the communication cost to only one bit.

```
structure 𝓜_H (𝒞 : Type) :=
    (initial : set (𝒞 × bool))
    (next    : 𝒞 → set (𝒞 × bool))
```

Listing 3. Implicit representation of a model

The proposed refinement, shown in Listing 3, inlines the calls to the **is_safe** function in the **initial** and **next** calls. This results in new return types for these, a product type ($\mathcal{C} \times bool$) composed of the configuration ($\mathcal{C}$ as previously) and a is_safe bit (*bool* type).

Listing 4 shows the mathematical transformation, which can be applied to any model following the previous model structure (Listing 1) to obtain the new one Listing 3. The new **initial** set is obtained by adding the result of **m.is_safe** to each initial configuration i. For each source s, the new neighborhood set is the couple (n, **m.is_safe**(n)) for all n in the **m.next**(s).

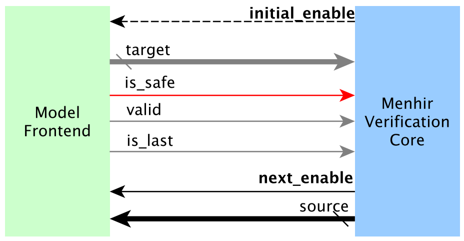

Fig. 2. The GLI signals between the model frontend and the verification core

```
def  $\mathcal{M}2\mathcal{M}_H$ (m : $\mathcal{M}$ $\mathcal{C}$) : $\mathcal{M}_H$ $\mathcal{C}$ :=
⟨initial ← { (i, m.is_safe(i)) |
  ∀ i ∈ m.initial },
 next ← λ s, { (n, m.is_safe(n)) |
  ∀ n ∈ m.next(s) }⟩
```

Listing 4. Conversion between $\mathcal{M}$ and $\mathcal{M}_H$

Fig. 1 already eluded to this transformation when showing the invariant checker on the configuration pipeline. Basically each produced state gets annotated with the **is_safe** bit. This bit is sent through the GLI interface to the termination checker (the red arrow in Fig. 1) which stops the analysis if the signal is not asserted (a violation of the property was found). Please note that for safety model-checking this bit is transient, and not stored in the state-space.

Fig. 2 shows the hardware signals of the GLI interface, which implement the model in Listing 3. Conceptually this interface establishes a communication channel between the model frontend and the verification core, which uses streams to implement the access to the initial and next sets. This channel transfers the new configurations (target bus) along with the **is_safe** bit. When the **initial_enable/next_enable** signal is asserted the model fronted produces a new configuration couple and asserts the **valid** signal. For the last configuration in the initial/next set the **is_last** signal is asserted. The source state s, needed for computing the next set, is sent along the **source** bus when the **next_enable** signal is asserted. Due to the exclusive access to the **initial** and the **next** sets (the initial set is produced before all the next sets), the GLI interface uses the same communication channel from the model frontend to the verification core for both transfers.

Furthermore, this structure of the GLI also signals the presence of deadlocks in the model, which can lead to termination if the *deadlock-checking* option is enabled in the *Termination Checker*. A deadlock exists in the model either trivially, if the model does not have any initial states (**m.initial** $= \emptyset$), or if the next set of a given configuration s is empty (**m.next**(s) $= \emptyset$). These conditions are signaled by the model frontend by asserting the **is_last** signal without (at the same time) asserting the **valid** signal.

To allow the use of existing modeling language, the model-frontend presented in Fig. 2 can encapsulates a processor on which the language runtime can be executed. In this SoC setup, the processor is bound with the Menhir core through an AXI4 bus, which exposes logic registers in processor's addressable space. The software running on the processor is reduced to a minimal driver mapped directly to the hardware GLI. This driver offers an adaptation layer, based on Listing 3, to wrap existing modeling language runtimes.

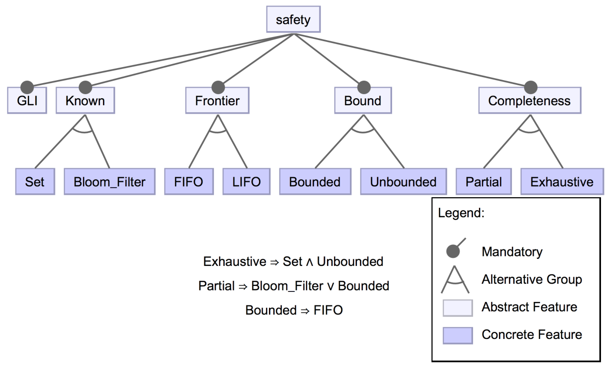

Fig. 3. The variability model of the Menhir core

C. Exploiting Algorithm Variability in the Verification Core

Besides the model variability fully exposed by our *generic language interface* the high-level safety verification algorithm, illustrated in Listing 2, exposes data structure variability. Historically, these have been exploited by software model-checking frameworks, such as SPIN [15], [17], [23], to achieve tremendous results in terms of verification scalability and speed. In this section we focus on four simple data structures that represent the backbone of more than 6 explicit-state model-checking algorithms.

Fig. 3 illustrates a variability diagram that characterizes the Menhir core implementation. Menhir has 3 main variability axes (the GLI, the Known set and the Frontier set) and two dependent features (the Bound and the Completeness). The algorithm performs an exhaustive search only if the known implements a set data-structure and the search is not bounded (the Exhaustive condition in the figure). A partial analysis is performed if either the known is represented using a probabilistic data-structure (a bloom filter) or if the analysis is bounded (typically depth bounded). Note that, in this study, we restrict the bounded analysis to a FIFO based depth-bounded algorithm (Bounded $\implies$ FIFO), leaving other approaches, such as [29], for future work.

1) Known Set Variants: The Known set is typically implemented using an associative memory (a hashtable), which offers an amortized constant time addition cost. To accommodate an iterative state-space traversal discipline, the Known exposes an **add_if_absent** function [10], which adds a configuration to the set if not already included and returns either the configuration added or null, if the configuration was already in the set. A probabilistic bloom filter implementation of the Known set interface morphs the verification algorithm to the bitstate-hashing approach [27], which stores only one bit per state in the state-space thus achieving high-scalabily for bug detection.

2) Frontier Set Variants: The Frontier set is implemented either as a stack (LIFO) or a first-in-first-out (FIFO) queue. According to the frontier set discipline, the verification algorithm analyses the state-space either in a depth-first or a breadth-first order. The *scheduler* adds a configuration c to the frontier only if it is new (**add_if_absent**$(c,\mathcal{K}) \neq$ null). The *Next Controller* removes an element from the frontier when querying the *Model Frontend* for its neighborhood.

3) Bounded Model-Checking: The four variants of the verification core are augmented with two supplementary variants, by introducing the bound parameter, a layered FIFO and the possibility to clear the contents of the known set. In this case the verification core performs an explicit-state BMC algorithm, with or without bitstate-hashing. In this configuration, the core stores only two layers of the state-space in memory, the current layer in the FIFO-frontier and the next layer in the Known set. This behavior is obtained by storing the *Frontier* in a dual-staged FIFO. While the first stage stores the remaining states from the current layer, the second stage stores the frontier states of the next layer. Once the current layer is fully explored (ie. the first stage is empty), the *Known* set is cleared, and the FIFO stages are swapped.

The current version of the Menhir model-checker exposes the GLI interface, which eases the integration of modeling frontends. Menhir implements all 6 variations of the safety algorithm described in this section, which shows the potential of creating an infrastructure for structuring the research efforts on hardware model-checking.

IV. Experimental Results

After presenting the evaluation setup, this section discusses the results obtained for six of the 18 possible variations (6 algorithms $\times$ 3 modeling languages).

A. Evaluation Setup

To evaluate our approach we have implemented the GLI-software interface for both the UML and DVE specification language. These two execution runtimes run on an ARM processor and are connected with the Menhir core through an AXI4 bus. In addition, we have used the VHDL GLI interface as a domain-specific language for implementing the full hardware configuration.

1) UML: has been chosen because it is a high-level specification language, commonly used in the industry. Out of the numerous research efforts that focus on model-checking of UML models, the EMI approach proposed in [19] offers an executable runtime, which can be executed on an embedded processor and is supported by the OBP2 model-checker. Moreover, the model interpreter features an controllable execution API that was used for the implementation of the GLI interface.

2) The DVE Specification Language: has been chosen as a typical model-checking language; being used by two high performance model-checking tools, Divine [32] and LTSmin [17]. From a modeling perspective Divine can be seen as a lower-level language when compared to UML. In this case we have based the implementation of the GLI interface on the Divine-to-C code generator used by Divine tool. This generator produces optimized C code compliant with the Divine CESMI interface, which is functionally similar to the GLI interface presented in the Listing 1.

3) The VHDL-based GLI: is the low-level interface exposed by the Menhir core. Implementing the verification model as a VHDL circuit targeting this interface eliminates any costly indirection that exists in the previous cases. However, from a modeling perspective, this interface is very low-level compared to UML and DVE languages.

a) Evaluation model.: The results presented are based on a representative parametric model, illustrated in Listing 5. The configuration of the model is an array of n bits. The initial configuration, shown by the constant function **initial**, assigns 0 to all configuration bits. The **next** function return the set of configurations obtained by flipping one bit in the source configuration s. While simple, this model exhibits an exponential state-space (2^n configurations), the worst case for a n-bit configuration. Moreover, the transition structure of the nbits model exposes a high-degree of non-determinism, for each configuration s there are n target configurations. Furthermore, for the purpose of the evaluation we assume that all states are safe, which induces the worst case execution time for model-checking (if no violation is found, the analysis continues for all the 2^n configuration).

```
def  nbits (w ∈ ℕ⁺) :  𝓜_H 𝓒  :=
⟨initial ← { (n,T)|∀ i∈[0,w), n_i=0 },
 next ← λs, { (n,T)| ∃ i∈[0,w), n_i=¬s_i }⟩
```

Listing 5. Conversion between $\mathcal{M}$ and $\mathcal{M}_H$

For the purpose of the evaluation, the nbits model has been implemented in all three specification formalisms supported by the Menhir core (UML, DVE, and VHDL-based GLI).

b) Evaluation Platform: The evaluations where performed on a Zynq XC7Z020-CLG484 platform. The SoC configuration used one core of the ARM processor running at 667 MHz. OBP2 EMI [19], used as baseline for UML, was executed using OpenJDK v1.8.0, on the Zynq platform using Debian 8. Divine v3.3.3 [32], used as baseline for DVE, was cross-compiled and executed on the Zynq platform using gcc 5.4. The Menhir core was synthesized using Vivado 2018.3 and runs at 100 MHz frequency.

B. Evaluation Results

In the following, we show the performance results obtained on 6 amongst the 18 variations of our platform. The Exhaustive BFS results illustrate the performance of the Menhir core with the three modeling languages against the two baselines. The BMC and bitstate hashing configurations show the results on the VHDL-based GLI implementation of the nbits model. Lastly, we show the FPGA resource utilisation of the verification core in these experiments.

1) Exhaustive BFS: Fig. 4 shows the performance gain obtained for exhaustive breadth-first search with respect to Divine and OBP2 EMI. Seven versions of the nbits model

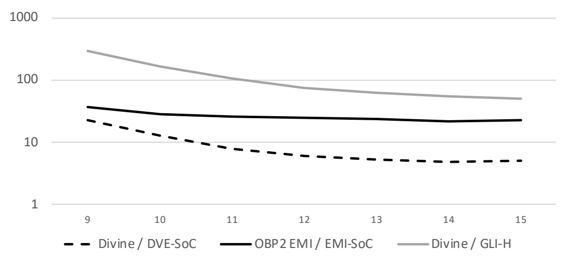

Fig. 4. SoC results vs Divine for DVE and OBP for UML

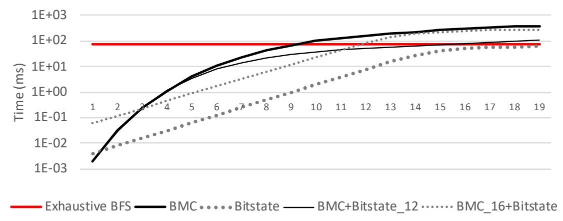

Fig. 5. Partial verification execution time.

where considered ranging from 9 bits to 15 bits configurations. The black line represents the ratio between the running times of OBP2 EMI versus the EMI-SoC configuration using the UML model. In this case the performance gain ranges between 36X to 22X. The dashed line represents the ratio between the running times of Divine versus the DVE-SoC configuration using the DVE model. In this case the performance gain ranges between 23X to 5X. The gray line show the performance gain obtained by the full hardware configuration (VHDL-based GLI model) versus Divine. In this case the performance gain ranges between 296X to 50X.

The high performance gains for smaller models, in Fig. 4 illustrate the faster initialization time of the Menhir verification core with respect to the respective software. As the model gets bigger, however, these differences are less visible, and the performance gain reaches a plateau. The results show the gain is strongly dependent on the performance of the model-execution engine, DVE engine is faster than UML engine on the CPU execution. The VHDL implementation of the nbits model is the fastest, achieving a throughput of one configuration per cycle. In our experiment, the UML verification (in the SoC configuration) is in average 14 times slower than the DVE SoC verification. However, note that in their respective software configurations the UML verification with OBP2 is 52 times slower than DVE with Divine.

2) *Non-exhaustive:* Fig. 5 illustrates the results on a 15 bits model in the context of three non-exhaustive algorithms. The red line references the analysis time of the exhaustive BFS analysis (as illustrated in Fig. 4).

a) Bounded model-checking: The thick black line, in Fig. 5, shows the analysis time while varying the exploration bound in a BMC configuration using a set for the known representation. Please note that when the bound reaches the diameter (16 in our case) this analysis is exhaustive. The results show that even though BMC re-explores nodes, for smaller diameters it can be faster than the unbounded analysis (the black line intersects the red one at the bound=9 in the figure). When the analysis proceeds to the reachability diameter BMC is 5.5 times slower in our experiment.

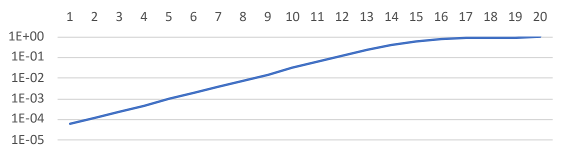

Fig. 6. State-space coverage in while varying the bloom filter size.

b) Bitstate Hashing: The thick dotted line, in Fig. 5, reports the execution time for the bitstate hashing algorithm while varying the address width of the bloom filter from 1 to 19. While this configuration divides the memory requirements by a factor proportional to the configuration width, it looses the exhaustivity. On the other hand we can observe that this partial analysis is faster than BMC, and reaches a similar runtime to BFS when the bloom filter is large enough. In this case the state-space coverage is improved, illustrated in Fig. 6, reaching 98% (0.98 in the figure) for a bloom filter with 2^{20} bits. Please note that the coverage can greatly be improved by using multiple hash-functions [27].

c) Hybrid – BMC & Bitstate Hashing: The thin black line, in Fig. 5, shows the execution time in a hybrid setting that combines BMC with bitstate hashing. In this case the bloomfilter size was fixes to 2^{12} bits and the bound was varied again between 1 and 19. In this case we can observe that the analysis almost matches the exhaustive BFS execution time when the bound approaches the reachability diameter. However, the state-space coverage is much smaller due to the insufficient size of the bloom filter.

The thin dotted line, in Fig. 5, illustrates the result in a similar hybrid configuration that fixes the bound of BMC at the diameter and varies the bloom filter size. It is not surprising that in this case, the analysis time follows the trend of the bitstate-hashing configuration (shown by the dotted thick line). What is interesting to note though, is that, in practice, these two hybrid configurations enable tuning the memory pressure and the analysis time for non-exhaustive analysis (ie. testing).

Table I shows the resources' utilisation on the FPGA for the Menhir core. The first line illustrates the resources needed for the exhaustive BFS analysis discussed in Fig. 4. The second line shows the resources needed for the bitstate hashing configuration. A noticeable point is the reduced number of BRAM cells being used, since a bloom filter needs to store only a bit per state. The third line (SoC bridge) shows the resources needed by the AXI interface to connect the Menhir core with the ARM processor on the Zynq platform.

These results mainly emphasize the high flexibility of our proposal. Menhir can be configured on demand to conform to many configurations. This study focused on establishing an open and extensible verification core architecture. As a consequence, the results are rather pessimistic, since we chose not

TABLE I
FPGA Resource Utilisation

	LUTs	**FFs**	**DSPs**	**BRAM**
Menhir core (Set)	951	741	13	130
Menhir core (Bloom Filter)	656	439	13	68
SoC bridge	881	1159	0	0

to exploit any optimisation opportunity (besides the pipelined architecture). Still, a preliminary study on accelerating the hash function (needed for the Known) shows an order of magnitude increase, making our results in-lined with the literature [12], [14].

V. Conclusion

Menhir is new highly modular hardware model-checker, inspired by the architecture of software verification frameworks. Menhir demonstrates up to 50X speedup vs software baseline. Compared to other hardware implementations found in the literature, Menhir decouples the model from the verification core. As a result, Menhir is compatible with many modelling formalisms and supports algorithmic customization, making it a perfect candidate to implement just-fit solutions. Based on the methodical approach proposed in this study, future evolutions will focus on optimizing key components of the framework to improve the performances. Furthermore, we plan to extend the core to ω-regular model-checking algorithms.

References

[1] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in cesar," in *Proceedings of the 5th Colloquium on International Symposium on Programming*. London, UK: Springer-Verlag, 1982, pp. 337–351.

[2] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*, ser. Lecture Notes in Computer Science, D. Kozen, Ed. Springer Berlin Heidelberg, 1982, vol. 131, pp. 52–71.

[3] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: Algorithmic verification and debugging," *Commun. ACM*, vol. 52, no. 11, p. 74–84, Nov. 2009.

[4] E. Clarke, E. Emerson, and A. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.

[5] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2008, pp. 1–6.

[6] C. Teodorov, L. Le Roux, Z. Drey, and P. Dhaussy, "Past-free[ze] reachability analysis: reaching further with dag-directed exhaustive state-space analysis," *Software Testing, Verification and Reliability*, vol. 26, no. 7, pp. 516–542, 2016.

[7] L. Le Roux and C. Teodorov, "Partially bounded context-aware verification," in *Software Engineering and Formal Methods*, P. C. Ölveczky and G. Salaün, Eds. Cham: Springer International Publishing, 2019, pp. 532–548.

[8] S. Apel, D. Beyer, V. Mordan, V. Mutilin, and A. Stahlbauer, "On-the-fly decomposition of specifications in software model checking," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 349–361.

[9] J. Barnat, L. Brim, M. Ceška, and T. Lamr, "Cuda accelerated ltl model checking," in *2009 15th International Conference on Parallel and Distributed Systems*, Dec 2009, pp. 34–41.

[10] A. Laarman, "Scalable multi-core model checking," Ph.D. dissertation, University of Twente, Netherlands, 5 2014, iPA Dissertation Series No. 2014-06.

[11] I. Černá and R. Pelánek, "Distributed explicit fair cycle detection (set based approach)," in *Model Checking Software*, T. Ball and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 49–73.

[12] M. E. Fuess, M. Leeser, and T. Leonard, "An fpga implementation of explicit-state model checking," in *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, April 2008, pp. 119–126.

[13] D. L. Dill, "The murϕ verification system," in *Computer Aided Verification*, R. Alur and T. A. Henzinger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 390–393.

[14] S. Cho, M. Ferdman, and P. Milder, "FPGASwarm: High Throughput Model Checking on FPGAs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2018, pp. 435–4357.

[15] G. J. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.

[16] J. Barnat, L. Brim, and P. Ročkai, "Scalable shared memory ltl model checking," *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 2, pp. 139–153, 2010.

[17] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk, "Ltsmin: High-performance language-independent model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 692–707.

[18] OMG, "Unified modeling language," Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/PDF

[19] V. Besnard, M. Brun, F. Jouault, C. Teodorov, and P. Dhaussy, "Unified ltl verification and embedded execution of uml models," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 112–122.

[20] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[21] M. Y. Vardi, "Automata-theoretic model checking revisited," in *Verification, Model Checking, and Abstract Interpretation*, B. Cook and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 137–150.

[22] G. J. Holzmann, *Explicit-State Model Checking*. Cham: Springer International Publishing, 2018, pp. 153–171.

[23] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 1020 states and beyond," *Inf. Comput.*, vol. 98, no. 2, p. 142–170, Jun. 1992.

[24] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla, "Symmetry reductions in model checking," in *Computer Aided Verification*, A. J. Hu and M. Y. Vardi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 147–158.

[25] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, 1999.

[26] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, E. A. Emerson and A. P. Sistla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169.

[27] G. J. Holzmann, "An analysis of bitstate hashing," *Formal Methods in System Design*, vol. 13, no. 3, pp. 289–307, 1998.

[28] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu *et al.*, "Bounded model checking." *Advances in computers*, vol. 58, no. 11, pp. 117–148, 2003.

[29] A. Udupa, A. Desai, and S. Rajamani, "Depth bounded explicit-state model checking," in *Model Checking Software*, A. Groce and M. Musuvathi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 57–74.

[30] U. Stern and D. L. Dill, "Improved probabilistic verification by hash compaction," in *Correct Hardware Design and Verification Methods*, P. E. Camurati and H. Eveking, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 206–224.

[31] M. E. Tie, "Accelerating explicit state model checking on an fpga : Phast," Master's thesis, Northeastern University, Boston, may 2012.

[32] V. Štill, P. Ročkai, and J. Barnat, "Divine: Explicit-state ltl model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 920–922.